Building Distributed OO Applications:

Modula-3 Objects at Work

Michel R. Dagenais

Draft Version, 14 January 1997

Contents

Preface

Beware, this book is still under development. Details, examples, references and polishing are missing, among other things.

This book is intented for students and practioners who want to learn the latest object oriented techniques to develop large interactive distributed applications. The techniques illustrated in the book include separate interfaces and implementations, multiple execution threads, user interface development tools and toolkits, persistent storage of data structures, and remote method invocation on network objects. Thus, practioners can deepen their knowledge in these areas.

For students, this book is useful in a senior undergraduate or entry level graduate course on object oriented programming, where the emphasis is on building robust interactive distributed applications.

To achieve these goals, the book contains a large number of sample programs that do useful things. The environment is based on the Modula-3 language and libraries. This environment offers advanced functionality while aiming at simplicity and consistency. Run time checks detect all errors before they corrupt the program; in other words no core dump! User interfaces and distributed applications can be constructed in relatively few lines.

Multi threaded and distributed applications are already very challenging by themselves. A simple and safe programming environment lets the reader concentrate on learning these new concepts rather than chasing dangling pointers.

The concepts learned in this environment are directly applicable to other languages and environments, as they evolve to support all these advanced programming techniques. For instance, Java, which has been described as Modula-3 in C++ clothes by Jim Waldo of SunSoft, approaches Modula-3 in many ways and is used throughout the book for comparing language constructs and libraries.

Organization

The chapters ordering reflects the needs of the classroom. Sections required to write simple programs are presented early, while more specialized material used in assignments at the end of term are presented last. For each topic, the main concepts are presented and then illustrated by examples. For a number of tools and libraries described in this book, the detailed description or reference manual is freely available separately, in electronic form.

The text starts with a brief review of existing object oriented programming languages used as examples later in the text. Then, the main concepts in object oriented design and programming are introduced. The implementation of these concepts in the languages Modula-3, C++ and Java is discussed.

The following sections study the decomposition of programs into documented interfaces and modules, and how compilation tools perform the dependency analysis to determine which modules need to be compiled. Then, run time error detection, debugging, coverage analysis, profiling, dynamic memory allocation, and multi-threading are discussed.

This is followed by a section concentrating on object oriented libraries and related tools. These libraries cover operating system access as well as user interface building tools.

Finally, tools and libraries for persistent and distributed objects, which enable distributed applications with persistent data, are presented.

In order to fully exploit the material presented in this book, the reader is encouraged to gain access to SRC Modula-3 Version 3.6, with associated tools and documentation, available on http://www.vlsi.polymtl.ca/m3/binaries. The electronically available documents include: Modula-3 language definition, SRC Modula-3 Version 3.6, Some useful Modula-3 Interfaces, Obliq A language with distributed scope, The FormsVBT Reference Manual.

Acknowledgments

This book hopefully helps the reader construct sophisticated object oriented interactive distributed applications using simple but flexible and efficient tools. Few other programming environments will enable students, within a one term course, to develop such advanced programs.

This would not be possible without the insight and dedication of researchers, and those supporting them, at the DEC Systems Research Center, and around the world, who are contributing to the development of the Modula-3 programming environment.

In particular, Marc H. Brown, Luca Cardelli, Bill Kalsow, Paul McJones, Marc Najork and Greg Nelson spent a lot of time discussing with me various issues related to object oriented programming and Modula-3.

Similarly, outside DEC SRC, Eliot Moss, Farshad Nayeri and Geoff Wyant contributed stimulating discussions.

Chapter 1

Introduction

Computer programmers, not so long ago, were expected to master the development of medium to large batch programs using structured programming. Newer applications require distributed interactive programs with graphical user interfaces. Fortunately, a number of techniques, tools and libraries have been developed to alleviate the increased complexity of these applications.

Graphical user interfaces benefit from libraries of reusable components such as buttons and menus, and tools that let the programmer visually position the elements composing the interface. The object oriented programming paradigm helps designing user interfaces because of the commonality between all the graphical elements involved, such as menus, buttons, type-in boxes, editors... The Smalltalk environment contains these elements and was highly successful in the area of user interface prototyping.

Distributed programming is often implemented using a layered approach. Ultimately, the programmer should not see much difference between interacting with local procedures and objects, and remote ones. Techniques such as Remote Procedure Call (RPC) and more recently Remote Method Invocation, or Network Objects, have simplified significantly the development of distributed applications. Of course, there is more to distributed programming. Indeed, when some of the remote objects or programs become unavailable (because of failures or simply unexpected communication delays), the difference between local and remote objects suddenly reappears.

A program receiving requests from several sources must somehow ensure that they each get a fair share of its attention. Similarly, a user interface should ideally be continually responsive to the user. However, some operations may require a long time to terminate. To remain responsive, any long operation must be interruptible. This can be achieved with the painful

cooperation of long operations which must regularly stop and let other operations proceed. The alternative is to have multiple threads of execution that are preemptively scheduled such that they each get a fair share of the processing time.

None of these tools and techniques make the software development process automatic and effortless. However, together, along with the classical tools for modular design, documentation, module dependency analysis, debugging and test coverage, they can help shorten the time required to develop robust and maintainable distributed interactive applications.

There are several ways to implement these concepts. Several hundred computer languages have been developed over the years and offer the basic capabilities required for programming. Even the good old *vi* editor on Unix platforms has been shown Turing Complete, meaning that any program could be replaced by an equivalent file containing only *vi* macros.

The environment selected for this book had to support modular programming, safety and simplicity, object oriented programming, graphical user interfaces, multiple threads of execution and network objects. Furthermore, the reader could not be expected to buy an expensive development environment just to perform the exercises suggested in this book. All these criterions fortunately shortened considerably the list of suitable languages. One clearly stood apart and was selected: Modula-3.

The reality of current software development practices must not be ignored however. The important concepts are presented using a clean and powerful language such as Modula-3. Then, other popular languages such as Java, C++, Delphi, or Visual Basic may be used when required. While the readers will probably not have C++ implementations that support multiple execution threads and automatic memory management, they will be ready when this functionality becomes available. Similarly, being used a clean language, they may maintain good design practices even when using a language which does not enforce any.

Of course, spending less time learning C++ is not as good for acquiring good reflexes for debugging C binaries corrupted by unsafe operations such as dangling pointers. With the current software development backlog, however, one may hope that future software development practices will quickly evolve towards safer environments, such as Java, and will relegate pointer chasing to folklore.

Chapter 2

Object Oriented Programming

2.1 The Evolution of Programming Languages

A number of languages were developed when computers started to become available to a significant user community, around 1960. Such languages include FORTRAN [?] and COBOL [?], still in widespread use today albeit with many extensions.

In the 1960s, researchers started to bring important ideas that were to affect the development of computer languages thereafter. Algol [?] is the ancestor of many classical structured programming languages. Simula [?], initially targeted at simulation tasks, brought many of the important concepts now behind object oriented programming languages. Lisp is representative of flexible interpreted programming environments and has the seeds of functional programming.

The 1970s brought Pascal [?], Modula-2 [?] and C [?]. Modula-2 was created by Niklaus Wirth, as an improvement over his earlier creation Pascal. It is the classical example of a structured modular compiled programming language. C was developed as a portable assembler to build the UNIX operating system. While it offers several high level constructions like structures, procedures and loops, it retains all the flexibility of assembly programming.

In the following decade, several programming languages tried to establish themselves as successors to the popular FORTRAN, COBOL, Pascal and C. Smalltalk [?] was probably the forerunner in the object oriented renaissance. Lisp and its object oriented version CLOS [?] gained in popularity. Ada was developed as the standard programming language for the

United States defense department, and was later followed by Ada95 [?]; it is strongly inspired from Modula-2. Functional programming took off with Scheme [?] and eventually ML [?].

The C language popularity grew with UNIX usage, and object oriented extensions, C++ [?], were developed by Bjarne Stroustrup. Similarly, Niklaus Wirth experimented with minimalism and produced a strikingly simple language Oberon [?], later followed by Oberon II which includes object oriented features. Luca Cardelli, Jim Donahue, Mick Jordan, Bill Kalsow and Greg Nelson developed Modula-3 [?], a modular object oriented programming language strongly inspired from Modula-2. Eiffel [?] was developed by Bertrand Meyer and centers everything around objects.

Most widely used programming languages were extended to support object oriented programming. Extending some of these old languages while retaining strict upward compatibility was a serious challenge. The result is often a general lack of consistency, incomplete upward compatibility, or a mix of both. These renovated languages include C++, Delphi [?] (Object Pascal), Ada95, Visual Basic version 4.0 [?], and recent work on an object oriented version of COBOL.

The earliest of these *compatible* object oriented extension, C++, was strongly criticised for its lack of modern features, (multi-threading, garbage collection, standard user interface library...), and sheer complexity. It prompted Sun Labs, a traditional C and C++ shop, to develop and promote Java [?, ?], a C/C++ like new and modern programming language. It has been described by Jim Waldo of Sun Soft as Modula-3 in C++ clothes [?].

Selecting a programming language among those available can be an arduous task. Criterions may include performance, language features, available libraries, potential for reuse, maintainability, cost and availability, programming support... While several hundred programming languages have been developed over the years, only a few tens are reasonably active and, for example, have a discussion group on the Internet Usenet News system.

In the following sections, a number of serious contenders for developing object oriented applications are examined.

2.1.1 Ada 95

Ada 95 is a powerful, albeit complex, compiled modular object oriented programming language. Its use is mandated for several US government projects. Ada did not succeed as expected in the past because of its complexity and the lack of lightweight, efficient and affordable implementations. Libraries availability was also problematic. Ada 95 was created with the

intention of correcting some of these deficiencies. Free implementations and more libraries are now available.

Ada95 is highly structured and supports all the important object oriented concepts. It is particularly strong in the area of tasks (multi-threading) and packages (separate interfaces and implementations). Several different ways of synchronizing threads, and performing other similar specialized tasks, are available. As a result, the language reference amounts to 500 pages (900 pages for the annotated reference manual). Probably because of its *embedded systems* bias, Ada95 lacks garbage collection in the mandated base implementation, as well as standardised libraries for graphical user interfaces.

C++

C++ builds on the popularity of C on UNIX and DOS/Windows platforms. It offers many object oriented features, including virtual and non virtual methods, function and operator overloading, single and multiple inheritance, templates and exceptions. It lacks garbage collection and support for multi-threading.

The biggest asset, and problem, of C++ is C compatibility. It enabled C++ to become the most widely used object oriented language. However, many C features, chiefly the unrestricted use of pointers and the lack of explicit interfaces, make encapsulation, garbage collection and run-time error detection difficult if not impossible to achieve. Free and low cost efficient implementations are available for almost every platform.

Unfortunately, very few C++ libraries were standardized. There are no agreed upon standards for using threads, graphical user interfaces, databases...

2.1.2 CLOS

This interpreted extension of LISP offers very powerful object oriented features. It supports all the usual features, including multiple inheritance and garbage collection, and adds generic functions which are selected at run time based on the actual type of all the arguments. Multi-threading support, however, is missing. Several libraries and free implementations are available.

The cost of this expressiveness and flexibility is in much higher memory and CPU usage as compared to most compiled object oriented languages.

2.1.3 COBOL 97

Object oriented extensions to COBOL are in the final phase of being developed and standardised. It includes objects, inheritance and methods, and is supported by all the major COBOL compiler vendors. Being a superset of COBOL, COBOL 97 will instantly become the most widely used object oriented programming language.

2.1.4 Delphi

Delphi is an object oriented dialect of Pascal integrated in a graphical user interface development environment. It is a clean, simple and modular language, and is strikingly similar to Modula-3 in terms of syntax, object model and exception handling. Its other distinctive features are fast compilation and execution times, and excellent facilities for graphical user interfaces and database access. It remains nonetheless a single vendor single architecture language, and lacks garbage collection and multi-threading.

2.1.5 Eiffel

Eiffel uses object types not only for inheritance but also as the unit of encapsulation and modularity. It supports multiple inheritance and garbage collection. It emphasizes assertions about properties that must hold for each object type before or after methods are called. Its main selling points have been true object orientation, and support for software engineering through assertions.

Eiffel does not yet offer any support for multi-threading. For a long time, the only implementations available were slow and buggy, and few libraries were available. Newer implementations are much better now, and a free implementation, SmallEiffel, was released on the Internet.

2.1.6 Java

In 1987, Sun Microsystems was very enthusiastic about C++ and even planned to write the newer version of its operating system using it. This project was however abandoned and only some smaller projects were implemented at Sun in C++. By the early 1990s, Sun was actively looking for a C++ replacement which would support modern programming concepts such as threads and garbage collection while offering safety, simplicity and portability.

Their objective was to create a language modeled after Modula-3, Mesa, Beta and Dylan while retaining a familiar C++ look and feel. They removed some of the complex and/or unsafe features of C++ like pointer

arithmetic, manual storage reclamation, multiple inheritance, and operator overloading. Then they added garbage collection, threads, safety and a large number of portable libraries. Java's growth was also due in large part to excellent timing and marketing.

It was presented as a mainstream language (C++ like) which opened a whole new world for Web programming. A key factor was its inclusion in the Netscape browser, making available a Java environment on the vast majority of network connected computers in a surprisingly short time. Java was subsequently pushed as not just a language for small web applications (applets) but also for general purpose programming. Java shirts, cups, conferences, computers and chips followed. While it does not have the simplicity and elegance of other languages such as Modula-3, and lacks a few features such as generic modules, it offers a viable programming alternative to C++.

2.1.7 ML

ML and its cousins enjoy wide recognition among computer scientists and mathematicians, and typifies functional programming languages. Functional programs have a number of interesting properties and are easy to transform in various ways, in particular to execute on parallel computers. Functional programming languages, however, are not well suited for many interactive programming tasks and often require much more memory and some more CPU than equivalent non functional programs. Several free implementations are available.

2.1.8 Modula-3

This language represents an excellent compromise between expressiveness and simplicity. Modula-2+ was built as an object oriented extension to Modula-2. Modula-3, instead, was built from scratch retaining only the best proven features of Modula-2+ and other similar languages. It does not support multiple inheritance and operator overloading but comes with garbage collection and multi-threading support.

A free implementation and numerous libraries are available. It is particularly well suited for distributed object oriented applications.

2.1.9 Oberon

Oberon is an experiment in minimalism. It offers a very limited set of features. It was developed for building an operating system used for teaching. Very efficient free implementations are available along with interest-

ing libraries. It uses a very limited garbage collection and tasking (multi-threading) model.

2.1.10 Smalltalk

Being one of the earliest widely available object oriented language, Smalltalk attracted much attention during the 1980s. The Smalltalk environment, with its graphical libraries, is very interesting for prototyping interactive applications. Free implementations and many libraries are available. It is a simple yet very expressive language. Smalltalk was initially interpreted and relatively slow; even when compiled, Smalltalk carries a significant run-time overhead. For large applications development, the lack of type declarations and static type checking is also a drawback.

The pure object oriented philosophy of Smalltalk was pursued further in the newer Self language [?].

2.1.11 Visual Basic 4.0

In its latest incarnation, Visual Basic 4.0 [?], Microsoft's most popular language now qualifies as an object oriented language. It has encapsulation, objects, properties and methods. The development environment, as in Delphi, is centered around the graphical user interface. Its main strength is the high level of integration with the Microsoft system libraries (dialogs, databases, DDE, OLE...) and applications (Word, Excel, Project...).

It does not support multiple inheritance, or even single inheritance in a general way. Garbage collection is performed in a limited way, presumably through reference counting. As implied by its name, BASIC, this language was forced into a number of compromises to attract non technical programmers, and maintain upward compatibility.

2.1.12 Discussion

In the computing world at large, Visual Basic, COBOL, Delphi and a few database access languages such as Access, Oracle and PowerBuilder, probably account for 90% of the programs being written [?]. Nonetheless, these languages are rarely mentioned at technical conferences such as OOPSLA (Object Oriented Programming Systems, Languages and Applications.

Indeed, the more technical areas of computing, not centered around user interfaces and databases, often use different tools and languages. This includes computer aided design, high performance graphics, embedded systems, distributed and/or real-time applications, compilers and operating systems. In this area, C++ is in widespread use, followed far behind by

Smalltalk, Ada, Modula-3 and Eiffel. Java may become a serious contender, and quickly surpass Smalltalk, although it is mostly confined to World Wide Web applets for the time being.

2.2 Object Oriented Concepts

Object oriented programming is a keyword used in numerous contexts. It may represent a set of similar but competing design methodologies, complete with philosophical and almost religious aspects. Nonetheless, this section attempts to identify and describe a number of well defined programming constructs present in most programming languages labeled as object oriented. Thus, the emplasis is on technical aspects rather than on terminology.

2.2.1 Inheritance

Most classical programming languages have data types composed of a number of fields and often called *structure* or *record* types. These are now called object types or *classes*; the individual records of a record type are thus called the objects, or instances, of the corresponding class.

Sometimes, a data type C will contain all the same fields as another one A plus a few additional ones. In that case, it is said that class C inherits from class A or that C is a child and A an ancestor. This serves two purposes. First, the declaration of C is simpler since it is sufficient to declare that it inherits the fields from A instead of repeating the fields in A in the declaration of C. Secondly, an instance of class C is also a valid object of class A since C it is a superset of A. Thus, when a procedure expects an object of class A, an object of class C or of any other child class of A can be sent; this is called polymorphism.

In most implementations, like in C++ or Modula-3, the fields common to A and C will be at the same offset within objects of type A and C, such that a procedure compiled to operate on objects of type A will still work correctly on objects of type C. A class may have several child classes inheriting from it. Child classes may in turn have childs. This produces an inheritance tree.

Some languages allow multiple inheritance. In that case, a class may inherit from two or more classes. The result is an acyclic directed inheritance graph instead of a tree. While this may lead to overly complex relationships between classes, there are cases where multiple inheritance is interesting. It poses, however, a number of implementation problems. The most significant problem may be the offsets of fields that now may vary between an

ancestor and its childs.

Indeed, if class C inherits from A and then from B. The fields in A are also present in C and at the same offset in the record. The fields of B, however, can only be placed in C after the fields inherited from A and will thus be at different offsets. A number of implementation techniques have been developed to cope with this problem but each carry an overhead in record size, access time and implementation complexity.

Some, mostly interpreted, languages may store fields in a table of name-content pairs. There is obviously a time penalty in replacing the access through a fixed offset by a table search. In such cases, interestingly, offsets are never used and multiple inheritance poses fewer implementation problems. Moreover, such languages may also enable adding and removing fields dynamically, while the program is running. When the list of fields changes dynamically, it can only be known at run time if a given field exists or not. Typing errors in a field name may thus go undetected until the statement containing the error is executed.

2.2.2 Class variables

Each instance of a class contains a number of fields, just like ordinary records. These are called instance variables. Class variables are global variables that are associated with a class. They can be accessed by specifying the class and variable name or by specifying an object of that class and the variable name.

Class variables are mostly a convenience. The syntax to access class variables may be the same as the syntax to access instance variables (fields in records). It also divides the name space for global variables along the classes. If no support for class variables exists in a language, one gets the same result by naming adequately the variables by prepending the class name to the variable name. Modular languages, with variables names scoped by module, obtain the same benefits through a more general mechanism.

The following example illustrates class variables in C++.

```
class A { public:
  int field1;   /* instance variable */
  int field2;   /* instance variable */
  static int global_count; /* static indicates class variable */
};

A  obj1, obj2;
```

```
/* class variables can be accessed as a global variable */

A::global_count = 0;

obj1.field1 = 0; obj1.field2 = 0;
obj2.field1 = 0; obj2.field2 = 0;

/* obj1.global_count and obj2.global_count are A::global_count */

obj1.field1++; obj1.field2++; obj1.global_count++;
obj2.field1++; obj2.field2++; obj2.global_count++;

/* Thus, A::global_count is now equal to 2 */
```

Class variables are inherited but the child class may use the parent variable or have its own. In C++, for example, if class A has a class variable v, a child class C will also have a class variable named v; however, A::v and C::v will be distinct variables. In other languages, such as CLOS, it is possible to specify if A::v and C::v are the same or are distinct.

2.2.3 Encapsulation

Encapsulation and data hiding are often mentioned in connection with object oriented programming. This is nothing else than keeping the internal details private to a module while only exporting in the interface what the clients need to know. In some languages, like Modula-3, it is possible to place in the interface only the desired subset of the type, variables and procedures declarations. Other languages, like C and C++, because of implementation limitations, require the full declaration of every exported type to appear in the interface. However, in C++, it is possible to flag portions of the type declarations as private; these still appear in the interface but the compiler ensures that the private portions are not used by clients.

2.2.4 Procedure name overloading

Procedure names used to be unique. There are cases however where several related procedures perform the same function but on arguments of different type. For instance, there could be print_integer, print_boolean... Some languages, like C++, allow several procedures to have the same name. The compiler then determines the desired procedure by looking at the argument types.

The downside of this is that if the wrong argument is specified, there may be another procedure of the same name matching the type of the wrong argument, leaving the error undetected at compile time. The advantages are shorter names for procedures and the possibility to simply change the type of a variable and have the compiler automatically find the appropriate procedures to call, given the new type.

The examples below illustrate the impact of allowing several procedures to have the same name.

```
void print(int), print(char *);

int identity, fame;

char *name;
...

/* if identity changes type, this statement needs no change */

  print(identity);

/* if the programmer types fame instead of name by mistake
   it will go undetected at compile time. */

  print(name);
...

int identity, fame;

char *name;
...

/* if identity changes type, this statement needs change */

  print_text(identity);

/* if the programmer types fame instead of name by mistake
   the compiler will catch it. */

  print_int(name);
...
```

Most programming tools, in particular linkers, are based on the assump-

tion that procedure names are unique. Thus, C++ compilers must often generate unique external names for the different procedures that have the same name but different arguments. This may simply mean using the concatenation of the procedure name and of the arguments type names as a unique procedure name supplied to the linker; this process is called name mangling. Other programming tools, such as the debugger, must understand mangled names and translate them to a more readable form for the programmer.

2.2.5 Operator overloading

Most programming languages support a number of arithmetic operators. The evaluation order within an arithmetic expression may or not be specified in the language. If the evaluation order changes, from one compiler to another for instance, rounding errors may propagate differently and produce slightly different results.

Apart from the evaluation order which may change, there is little difference between an operator which takes two arguments and produces a result and an equivalent procedure. A number of languages, striving for syntax regularity, like LISP, only support procedures. Arithmetic operations are simply part of the built-in procedures.

Another approach is to support user defined operators. Letting a program define new operators or change the number of arguments for operators would require a dynamically changing lexical analyzer; few languages allow that. It is possible, however, in C++ to associate procedures to operators based on the type of the associated arguments.

This is simply a syntaxic convenience. Indeed, a + b is simply replaced by the procedure call operator+(a,b). Programmers reading a program must then check carefully the declarations in order to know what procedure is associated with each operator in the program.

2.2.6 Methods

A number of procedures are often associated with a class. The methods of the class are fields that contain pointers to these procedures. Naturally, one of their arguments will be an instance of that class. Often, the argument of that class will be the first argument, and will be called the receiving object. The receiving object is thus used both to access the method, and as argument to the procedure. This may look as following in a non object oriented language.

In C++, some methods have the *static* qualifier. This identifies methods which only access class variables. Such methods do not require a receiving

object, and thus are not really methods and may therefore be called directly.

```
TYPE
  A = RECORD
    field1, field2: INTEGER;
    method1: PROCEDURE(arg1: A; arg2: REAL);
    method2: PROCEDURE(arg1: A; arg2, arg3: TEXT);
    END;

PROCEDURE MyProcedure(arg1: A; arg2: REAL) =
  BEGIN
  ...
  END MyProcedure;

VAR
  obj: A;
...
BEGIN
  obj.field1 := 5;
  obj.method1 := MyProcedure;
  obj.method1(obj,22.0);
...
END.
```

The advantage of storing a pointer to a procedure in each object is that each object may react in its own way when the method is activated. Indeed, the method must contain a pointer to a procedure of the declared type but each object may be associated with a different one, upon need. For example, each object may represent a printer model, and a PaperEmpty method may contain a procedure that can communicate with the printer and determine the number of pages left in the tray. Calling a method is commonly named *invoquing a method on an object* or *sending a message to the object*.

Very often, all the objects of a given class would use the same procedures as methods. Thus, having identical methods stored in every object of a class is wasteful. Instead, all the methods may be stored in a table and only a pointer to that shared table is stored in each object.

```
TYPE
  TableForA = RECORD
    method1: PROCEDURE(arg1: A; arg2: REAL);
    method2: PROCEDURE(arg1: A; arg2, arg3: TEXT);
    END;
```

```
  A = RECORD
    table: TableForA;
    field1, field2: INTEGER;
    END;

PROCEDURE MyProcedure(arg1: A; arg2: REAL) =
  BEGIN
  ...
  END MyProcedure;

VAR
  obj1, obj2: A;
  tableForA: TableForA;
...
BEGIN
  tableForA.method1 := MyProcedure;
  tableForA.method2 := OtherProc;
  obj1.table := tableForA;
  obj2.table := tableForA;

  obj1.field1 := 5;

  (* obj1 is used to find the method and is also the first argument *)

  obj1.table.method1(obj1,22.0);
...
END.
```

Since methods are a common construct, many languages automate the construction of the table. A table, and a field pointing to it, is automatically allocated for each class. Furthermore, since the object used to find the method is also the first argument to the method, it is automatically supplied. Thus, with this work handled automatically, the above example becomes:

```
TYPE
  A = OBJECT
      field1, field2: INTEGER;
    METHODS
      (* implicitly there is a first argument of type A *)
      method1(arg2: REAL) := MyProcedure;
```

```
      method2(arg2, arg3: TEXT) := OtherProc;
    END;

PROCEDURE MyProcedure(arg1: A; arg2: REAL) =
  BEGIN
  ...
  END MyProcedure;

VAR
  obj1, obj2: A;
...
BEGIN

  obj1.field1 := 5;

  (* obj1 is also supplied as argument automatically *)

  obj1.method1(22.0);
...
END.
```

Some interpreted languages store the fields and methods of each instance in a table of name-content pairs. In that case, the method invocation mechanism may be more elaborate than simply retrieving a pointer to procedure from a record. Indeed, the method is first searched in the instance. If not found, the class provides a default value. This way, a default method may be specified for all instances of a class but may be superseeded by another method in the instances where this is desired.

There is another dimension to method invocation, inheritance. Indeed, a procedure that expects an instance of class A as first argument (receiving object) may operate as well on an instance of its child class C, since C has a superset of the fields defined for A. Therefore, all the methods defined for class A are applicable to class C and need only be redefined when a different behavior is desired for C.

In most compiled object oriented languages like C++ or Modula-3, all the methods defined for class A (stored in its table) are used to initialize the methods for class C. However, C may override some of the methods inherited from A (store a different procedure at the corresponding location in the table) or may have additional methods (entries added at the end of the table).

```
TYPE
```

```
  A = OBJECT
      field1, field2: INTEGER;
    METHODS
      (* implicitly there is a first argument of type A *)
      method1(arg2: REAL) := MyProcedure;
      method2(arg2, arg3: TEXT) := OtherProc;
    END;

  C = A OBJECT (* inherit from A *)
    METHODS
      addMethod(arg2: INTEGER); (* additional method *)
    OVERRIDES
      method1 := MyProcForC; (* different implementation*)
    END;

PROCEDURE MyProcedure(arg1: A; arg2: REAL) =
  BEGIN
  ...
  END MyProcedure;

PROCEDURE MyProcForC(arg1: C; arg2: REAL) =
  BEGIN
  ...
  END MyProcedure;
...
END.
```

In some interpreted languages, where fields are stored in a table of name-content pairs, the method invocation looks if the method is defined in the instance, then in the corresponding class and then in the ancestor classes. Several table searches may be involved before finding the procedure to execute. If methods may be added and removed dynamically, it may also happen that the method cannot be found. The following example illustrates how this search may proceed.

```
TYPE
  Class = REF RECORD
      parents: REF ARRAY OF Class;
      methods: ProcTable; (* class methods *)
      name: TEXT;
    END;
```

```
  Obj = OBJECT
      class: Class;
      methods: ProcTable; (* instance methods *)
    END;

  A = Obj OBJECT
      field1, field2: INTEGER;
    END;

...

PROCEDURE InvoqueMethod(receiver: Obj; methodName: TEXT;
      arg: ArgList) =
  VAR
    proc: Proc := NIL;
    i := 0;
  BEGIN
    proc := Find(methodName,receiver.methods);

    IF proc = NIL THEN
      proc := FindMethod(methodName,receiver.class);
    END;

    IF proc = NIL THEN
      Wr.PutText(Stdio.stderr,"Method " & methodName &
          " not found in object of type " &
          receiver.class.name & "\n");
      RAISE MethodNotFound;
    END;
    proc(receiver,arg);
  END InvoqueMethod;

PROCEDURE FindMethod(name: TEXT; class: Class): Proc =
  VAR
    proc: Proc := NIL;
  BEGIN
    proc := Find(name,class.methods);
    IF proc # NIL THEN RETURN proc; END;

    FOR i := 0 TO LAST(class.parents^) DO
      proc := FindMethod(name,class.parents[i]);
      IF proc # NIL THEN RETURN proc; END;
```

```
    END;

    RETURN proc;
  END FindMethod;
```

This flexible mechanism easily handles even multiple inheritance. It is however somewhat inefficient. As a speedup, it is possible to store in the child classes a copy of the methods offered by the parents. This avoids searching in the parents upon method invocation but requires more operations during the, much less frequent, method add/remove operations; there is also a small memory overhead in storing the pointers to methods in all the child classes.

Efficient method invocation in the presence of multiple inheritance, using method tables, is more challenging. When multiple inheritance was added to C++, most C++ compilers did not handle all cases properly until several realeases. Consider the case of a class C, inheriting from A1 and A2. The fields of each ancestor A1 and A2 must be concatenated to form C. On the other hand, C must qualify as an object of type A1 and A2. In particular, the following operations must be supported and require special processing:

- Convert a reference of type C to a reference of type Ai.
- Convert back a reference of type Ai (to an object of type C), for storing in a reference of type C.
- Invoque on an object of type C a method inherited from type Ai.
- Invoque on an object of type C, stored in a reference of type Ai, a method specified in type Ai but overridden in type C.

A possible implementation is outlined below.

```
TYPE
  (* Each object contains a reference to the methods table
     followed by its fields. *)

  A1 = REF RECORD
    methods: TableA1 := tableA1forA1;
    a1Field1: TEXT;
    END;

  A2 = REF RECORD
```

```
  methods: TableA2 := tableA2forA2;
  a2Field1: TEXT;
  END;

(* It is not possible to have both ancestors A1 and A2 start
   at the beginning of C. Thus, A1 starts with C at offset 0,
   while A2 starts two fields away from C at offset 8 on a machine
   with 4 bytes per reference field. A2 starts with a reference
   to a methods table, even if it lies in the middle of C. *)

C = REF RECORD
  methods: TableC := tableCforC;   (* C and A1 start here *)
  a1Field1: TEXT;
  methods: TableA2 := tableA2forC; (* A2 starts here at offset 8 *)
  a2Field1: TEXT;
  cField1: TEXT;
  END;

(* The methods table contains references to methods
   as well as offset to add to the receiver. *)

TableA1 = REF RECORD
  a1Method1: PROCEDURE();
  a1Method1Offest: INTEGER;
  a1Method2: PROCEDURE();
  a1Method2Offest: INTEGER;
  END;

TableA2 = REF RECORD
  a2Method1: PROCEDURE();
  a2Method1Offest: INTEGER;
  a2Method2: PROCEDURE();
  a2Method2Offest: INTEGER;
  END;

(* The methods table for C contains entries for
   the methods of A1, A2 and C. *)

TableC = REF RECORD
  a1Method1: PROCEDURE();
  a1Method1Offest: INTEGER;
  a1Method2: PROCEDURE();
```

```
    a1Method2Offest: INTEGER;
    a2Method1: PROCEDURE();
    a2Method1Offest: INTEGER;
    a2Method2: PROCEDURE();
    a2Method2Offest: INTEGER;
    CMethod1: PROCEDURE();
    CMethod1Offset: INTEGER;
    CMethod2: PROCEDURE();
    CMethod2Offset: INTEGER;
    END;

...
  tableA1forA1^ := TableA1{
    ProcA1M1forA1, 0,
    ProcA1M2forA1, 0};

  tableA2forA2^ := TableA2{
    ProcA2M1forA2, 0,
    ProcA2M2forA2, 0};

  (* Methods table for a reference to C stored in a reference
     of type A2. *)

  tableA2forC^ := TableA2{
    ProcA2M1forC, -8,  (* overridden in C, C starts at A2 - 8 *)
    ProcA2M2forA2, 0};  (* inherited from A2 *)

  tableCforC^ := TableC{
    ProcA1M1forC, 0,    (* overridden in C *)
    ProcA1M2forA1, 0,   (* inherited from A1, A1 starts at C + 0 *)
    ProcA2M1forC, 0,    (* overridden in C *)
    ProcA2M2forA2, 8,  (* inherited from A2, A2 starts at C + 8 *)
    ProcCM1forC, 0,     (* defined in C *)
    ProcCM2forC, 0};    (* defined in C *)
...
  (* a1, a2 and c are references to an object of type C *)

  a1 := c;  (* convert C to A1 *)

  (* convert A1 to C, if indeed a1 points to an instance of C *)
  IF ISTYPE(C, a1) THEN c := a1; ELSE Error... END;
```

```
a2 := c + 8; (* convert C to A2, offset the reference by 8 *)

(* convert A2 to C, if indeed a2 points to an instance of C *)
IF ISTYPE(C, a1) THEN c := a2 - 8; ELSE Error... END;

(* invoque a method; a2 is set to c + 8. Whether a given method is
   invoqued from c or a2, the method called and the argument
   received should be the same. *)

(* c.methods is tableCforC, the argument is c + 0 *)
c.methods.a2Method1(c + c.methods.a2Method1Offset);

(* a2.methods is tableA2forC, the argument is (c + 8) - 8 *)
a2.methods.a2Method1(a2 + a2.methods.a2Method1Offset);

(* the argument is c + 8 *)
c.methods.a2Method2(c + c.methods.a2Method2Offset);

(* the argument is (c + 8) + 0 *)
a2.methods.a2Method2(a2 + a2.methods.a2Method2Offset);
```

The consequence of multiple inheritance is to double the size of the methods tables, in order to store the offset (to apply to the receiver argument) in addition to the reference to procedure, for each entry. Moreover, each type conversion incurs finding and adding the relevant offset. Finally, each method invocation incurs the additional cost of fetching the offset from the methods table and adding it to the receiver argument.

2.2.7 Prototype/Cloning based inheritance

Obliq is a simple yet powerful object oriented interpreted language representative of a class of languages such as Self [?]. While it is useful as an embedded interpreter and as a distributed language, it is an interesting, prototype based, programming language in its own rights. The language is described in the reference manual entitled *Obliq A language with distributed scope*.

In Obliq, each object contains a number of data fields and methods. There are no classes, methods table or inheritance defined in the language. Nonetheless, through cloning and delegation, it is possible to implement single and multiple inheritance, instance and class variables as well as dynamically changing the methods for an instance or for a class.

A class is implemented by defining a prototype object. This prototype object is then cloned in order to obtain instances. The memory usage drawback is that a full object is used to represent the class while a declaration of the attribute names and types would have been sufficient in a non *prototype based* language. Furthermore, each object stores a method table instead of simply pointing to a shared method table.

```
let CircleClass = {
  color => [0.0,0.5,0.0],
  origin => 0.0,
  radius => 0.0,
  draw => meth(self,window) ... end
};

var
  c1 = clone(CircleClass),
  c2 = clone(CircleClass);

  c1.radius := 1.0;
  c2.radius := 2.0;
```

Single or multiple inheritance is easily obtained by cloning one or more parent class prototypes to create a child class prototype.

```
let RootClass = {
  name => ok,
  store => meth(self,file) ... end
};

let ShapeClass = {
  color => [0.0,0.5,0.0],
  origin => 0.0
};

let CircleFields = {
  radius => 0.0,
  draw => meth(self,window) ... end
};

let CircleClass = clone(RootClass, ShapeClass, CircleFields);
```

It is possible to update the fields and methods of each instance. Updating a data field or method in a class prototype will not affect the existing instances of this class but only the instances created after the update.

Even though each object (instance) has its fields and methods, it is possible through delegation to have class variables and methods.

```
let RootFields = {
  name => ok,
  store => meth(self,file) ... end
};

let ShapeFields = {
  nbShapes => 0;
  color => Blue,
  origin => 0.0
};

let CircleFields = {
  radius => 0.0,
  draw => meth(self,window) ... end
};

let CircleClass = clone(RootFields, ShapeFields, CircleFields);

CircleClass.store := alias store of RootFields;
CircleClass.draw := alias draw of CircleFields;
CircleClass.nbShapes := alias nbShapes of ShapeFields;
```

In this example, the *store*, *draw* and *nbShapes* fields are delegated to the class fields holders. Therefore, all objects created by cloning *CircleClass* share these fields. Whenever the *store* method of *RootFields* is updated, for example, all the instances of *CircleClass* are affected and will access the updated method.

2.2.8 Interface inheritance

Inheritance may be divided into implementation inheritance and interface inheritance. Most languages offer a single form of inheritance which combines the two. Java however offers interface inheritance as a replacement for the more complex multiple inheritance supported by C++.

An interface declares a number of methods and may inherit from several interfaces. When a class is declared to *implement* a number of interfaces, it must implement all the methods declared in these interfaces. A reference declared for an interface type can only be assigned an object of a class implementing the corresponding interface.

This offers much of the benefits of multiple inheritance without the same complexity overhead. Accessing the methods of an object through an interface reference may be implemented as follows. Whenever an object is assigned to an interface reference, it must be checked if the object implements the interface. The table connecting the methods in the interface to corresponding methods in the object type must be retrieved.

```
TYPE
  Interfaces = {I, II};

  A = REF RECORD
    methods: TableForA := tableForA;
    field1: TEXT;
    field2: TEXT;
    END;

  TableForA = REF RECORD
    typecode: CARDINAL := typecodeForA;
    interfaces: ARRAY Interfaces OF REFANY;
    method1: PROCEDURE() := Method1ForA;
    method2: PROCEDURE() := Method2ForA;
    method3: PROCEDURE() := Method3ForA;
    END;

  InterfaceI = REF RECORD
    method1: PROCEDURE();
    method3: PROCEDURE();
    END;

  InterfaceII = REF RECORD
    method2: PROCEDURE();
    method3: PROCEDURE();
    END;

  InterfaceIReference = RECORD
    object: REFANY;
```

```
    interface: InterfaceI;
    END;

VAR
  interfaceIforA: InterfaceI;
  interfaceIIforA: InterfaceII;
  tableForA: TableForA;
  ri: InterfaceIReference;
  ra: A;
...
  interfaceIforA := NEW(InterfaceI,
      method1 := Method1ForA, method3 := Method3ForA);
  interfaceIIforA := NEW(InterfaceII,
      method2 := Method2ForA, method3 := Method3ForA);

  tableForA := NEW(TableForA);
  tableForA.interfaces[I] := interfaceIforA;
  tableForA.interfaces[II] := interfaceIIforA;
...
  (* Assign an object of type A to a reference to
     InterfaceI. Retrieve the method translation
     table which converts entries in InterfaceI to
     methods in type A. *)

  IF ra.methods.interfaces[I] = NIL THEN
    Error("InterfaceI not implemented for this object");
  END;

  ri.object := ra;
  ri.interface := ra.methods.interfaces[I];

  (* Invoque a method from the interface reference *)

  ri.interface.method2(ri.object);
```

2.2.9 Variations on inheritance

A number of object oriented programming languages bring a few minor options on top of the basic single or multiple inheritance concepts.

Some languages such as Java explicitely support the concept of abstract classes. An abstract class is a class defined for the sole purpose of inheri-

tance. For example, a GraphicsObject class could be defined as parent for Circle and Polygon classes. Several methods may be defined without implementation in an abstract class. Therefore, instances of an abstract class cannot be created, as their methods table would contain unfilled entries.

Another inheritance qualifier supported in Java is *final*. This qualifier may be applied to methods, indicating that they should not be overridden, and classes, indicating that they should not be inherited from and thus that all its methods are final. Preventing programmers from reusing a class through inheritance is questionable; a class designer may not realize a priori all the contexts in which the class could be useful. However, informing the compiler that some methods are not overridden enables significant optimizations. A simple procedure call may be used instead of the more costly method dispatch mechanism. Furthermore, procedure inlining may be used whenever appropriate and bring a significant speedup.

The same optimizations are possible without a *final* qualifier but require a whole-program optimizing compiler. Such compilers which determine automatically which methods are final through whole program analysis have been built by research groups but are seldom available in production environments.

The *final* qualifier in Java may also apply to object attributes. A final attribute is initialized once and remains constant thereafter. This insures that an attribute is not mistakenly modified, and allows the compiler to optimize the access to this attribute, since it cannot be modified.

A common problem with multiple inheritance is avoiding inheriting more than once a given class. The *virtual* qualifier in C++ is used to specify that the corresponding class should be inherited only once, as shown in the following example.

```
class Window {
  virtual void draw(), inner_draw();
};

class BorderWindow: public virtual Window {
  virtual void draw(), inner_draw();
}

class ShadowWindow: public virtual Window {
  virtual void draw(), inner_draw();
}

/* A window with a border and a shadow. The virtual
```

```
   qualifier insures that Window is inherited only once */

class BorderShadowWindow: public virtual Window,
    public BorderWindow, public ShadowWindow {
  virtual void draw(), inner_draw();
}

void Window::draw() { inner_draw(); }

void BorderWindow::draw() {
  Window::inner_draw(); // draw the window
  inner_draw();         // draw the border
}

void ShadowWindow::draw() {
  Window::inner_draw(); // draw the window
  inner_draw();         // draw the shadow
}

void BorderShadowWindow::draw() {
  Window::inner_draw(); // draw the window
  Border::inner_draw(); // draw the border
  Shadow::inner_draw(); // draw the shadow
}
```

Since the object oriented renaissance started by Smalltalk, a number of traditional languages have been extended with object oriented facilities, sometimes in unusual ways. Visual Basic 4.0, has objects, methods and attributes. It does not directly support inheritance, not unlike prototype based languages. The objects are divided into four groups: Forms, Controls, MDIForms, and user defined classes. References may be declared as having the specific type of an object (e.g. TextBox) or as having a generic type (Form, Control, MDIForm, or Object). Modules may add methods and attributes to a Form instance, which may then be further instantiated. This surprising mix of object oriented concepts with uncommon limitations and terminology is probably due to the following factors. This language targets a large audience uninitiated to the object oriented concepts. It carries the burden of compatibility with a long history. It must leave room for further revenue generating enhanced versions.

2.2.10 Before and after methods

Often, a child class overrides a method because it needs additional work to be performed, on top of what the ancestor method does. While it is possible to simply lazily copy the ancestor method and modify it to perform the added work, this creates maintenance problems. Indeed, when the ancestor method is modified, the portion of the child method copied from the ancestor will probably need to be modified in the same way.

Some languages allow the definition of procedures to be executed before or after a method is called. The same result, however, is obtained by overriding the parent method with a method that does the additional work and calls the parent method.

Sometimes, the additional work needs to be done in the middle of what the parent method does. In that case, the content of the parent method should be put in two procedures. These two procedures will be called in the parent method. They will also be called in the child method, but with some code between the two calls.

In classical programming languages, there are no parent and child versions of a procedure. One procedure does it all and tests are inserted to adapt to the needs of different types. The resulting code is full of IF/THEN/ELSE and hard to read but there are fewer incentives to lazily copy code.

In the example below, a parent method and a good and bad version of the corresponding method for the child is shown. The Child method needs only do more work before and after the parent method.

```
TYPE
  A = OBJECT METHODS
      method() := MethodInAncestor;
    END;

  C = A OBJECT METHODS OVERRIDES
      method := GoodMethodInChild;
    END;

PROCEDURE MethodInAncestor(self: A) =
  BEGIN
    do step i...
    do step ii...
    do step iii...
    do step iv...
  END MethodInAncestor;
```

```
PROCEDURE BadMethodInChild(self: C) =
  BEGIN
    do something before...
    do step i...
    do step ii...
    do step iii...
    do step iv...
    do something after...
  END BadMethodInChild;

PROCEDURE GoodMethodInChild(self: C) =
  BEGIN
    do something before...
    A.method(self); (* call the parent method *)
    do something after...
  END GoodMethodInChild;
```

In the example below, the child method needs to do something in between the steps performed by the parent method.

```
PROCEDURE Step1And2() =
  BEGIN
    do step i...
    do step ii...
  END Step1And2;

PROCEDURE Step3And4() =
  BEGIN
    do step iii...
    do step iv...
  END Step3And4;

PROCEDURE MethodInAncestor(self: A) =
  BEGIN
    Step1And2();
    Step3And4();
  END MethodInAncestor;

PROCEDURE GoodMethodInChild(self: C) =
  BEGIN
    do something before...
```

```
    Step1And2();
    do something in middle
    Step3And4();
    do something after...
  END GoodMethodInChild;
```

2.2.11 Method selection based on all arguments

In most object oriented languages, one argument is the receiving object and its type determines the method to execute, through its method table. In the CLOS language, the method selection process has been generalized to account for the type of each of the arguments.

Many versions of a given procedure may be defined, all with the same name but each with different types for the arguments, as a *generic* procedure. When a procedure call with that name is issued, the actual type of each argument is verified and the suitable version of the generic procedure is found. If the argument supplied in position 1 is of type C, it is suitable for a version of the procedure that expects an argument of its ancestor type A at the same position. However, if there is a version of the procedure defined for an argument of type C in that position, it is a better match since it was defined specifically for that type.

There are cases, however, where the best match is not obvious. For instance, if arguments in position 1 and 2 are of type C, it is difficult to select between a version that expects types A and C in position 1 and 2 and a version that expects types C and A.

This flexibility and expressive power comes at the cost of searching the best match. To alleviate that cost, a cache may be attached to each generic procedure. This cache is a hash table that associates types of arguments to the best matching version of the generic procedure. When a new version of the generic procedure is added, which is infrequent, the cache content is emptied. When a match is found, the type of the arguments is used as key to insert the matched version in the cache. The next time a call is made with the same argument types, the matching version is readily found in the cache instead of searching through all the available versions. The cost of searching in the cache, especially given the length of the key formed by all the arguments type names, is still much higher that accessing a method in a method table.

In C++, many procedures may have the same name. The suitable procedure to call is selected based on the arguments type, but at compile time. The declared argument type is thus used instead of the actual argument type. Indeed, a variable of declared type A may hold an object of type A or of any of its descendents.

```
class A {...} *pa1, *pa2;

class C : A {...} c1, c2;

/* In C++ this version will get called */

void TheFunction(A arg1, A arg2) {
  do something general...
}

/* In CLOS this version would get called */

void TheFunction(C arg1, C arg2) {
  do something more appropriate...
}

main() {
  pa1 = &c1;
  pa2 = &c2;

  TheFunction(pa1,pa2);
}
```

2.2.12 Non virtual methods

In C++, only the methods declared as virtual are effectively placed in the method table. The other *non virtual* methods are simply ordinary procedures using the methods syntax. While the version of a virtual method called depends on the actual type of the receiving object, the version of a non virtual method depends on its declared type. It is thus mostly a syntactic convenience.

```
class A {
  public:
    void theMethod();
    virtual void theVirtualMethod();
} *pa1;

class C : A {
  public:
    void theMethod();
    virtual void theVirtualMethod();
```

```
} c1;

main() {
  pa1 = &c1;

  /* The version from A is executed */

  pa1->theMethod();

  /* The version from C is executed */

  pa1->theVirtualMethod();
  c1.theMethod();
  c1.theVirtualMethod();
}
```

Because virtual methods are selected through the method table at run time, this is often called dynamic binding. Non virtual methods are selected at compile time and thus represent static binding. It is virtual methods that enable objects of different types, stored in a variable declared as some common ancestor type, to produce different results. For example, a list of GraphicsObject could store Circles, Rectangles and Polygons, each having an appropriate draw method. This is called polymorphism.

2.2.13 Constructors and destructors

In some languages, like C++, it is possible to define a method called the constructor that will automatically be called each time a new object of the corresponding type is created. When an object is created, the constructors of its type and of all its ancestor types are called. The order in which all these constructors are called and mechanisms to pass arguments to all these constructors need to specified appropriately. In C++, new objects are created for global and local variables, when dynamically allocated, or when arguments and return values are passed by value (copied). In languages without constructors, the same effect is obtained through initialization methods called explicitly when new objects are created. The constructors initialize the objects, setting default values and allocating internal structures.

Destructors are methods automatically called each time an object ceases to exist. In C++, local variables and temporary objects cease to exist when they fall out of scope, global variables cease to exist when the program ends and dynamically allocated objects cease to exist when *free* is called. The

destructors usually deallocate internal structures and may write permanent state to disk.

The notion of destructor does not work well in a garbage collected environment. Indeed, the garbage collector, once in a while, finds unreachable objects and deallocates them. Calling a destructor method may change the reachability of the object (store a pointer to the object in a global variable somewhere), in which case the object should not be collected now and the destructor should not have been called.

In garbage collected environments, objects may be registered for finalization. The registered procedure is eventually called when the object is unreachable. If the registered procedure makes the object reachable again, it will not be garbage collected and may be registered again for finalization if needed.

In the example below, both constructors and explicit initializations are illustrated. Constructors need to relay some of the received arguments to the constructors of their parent classes while no special mechanism is required in explicit initialization procedures.

```
class employee {
  char *name;
  Address home;
public:
  employee(char *n, Address adr) : name(n), home(adr) {}
}

class manager: public employee {
  Group group;
public
  manager(char *n, Address adr, Group grp):
       employee(n,adr), group(grp) {}
}

manager boss("John Smith","1 Main Street CA",Finance);


TYPE
  Employee = OBJECT
       name: TEXT;
       home: Address;
    METHODS
       init(name: TEXT; home: Address) := InitE;
    END;
```

```
  Manager = Employee OBJECT
      group: Group;
    METHODS
      init(name: TEXT; home: Address; group: Group) := InitM;
    END;

PROCEDURE InitE(self: Employee; name: TEXT; address: Address) =
  BEGIN
    self.name := name;
    self.address := address;
  END InitE;

PROCEDURE InitM(self: Manager; name: TEXT; address: Address;
    group: Group) =
  BEGIN
    Employee.init(self,name,address);
    self.group := group;
  END InitM;

VAR
  boss := NEW(Manager).init("John Smith","1 Main Street CA",Finance);
```

2.2.14 Metaclasses

It is often useful to obtain information about a *class* at run time. This information may include the class name, the list of fields, with their name and type, and the ancestor type. The class of objects that contain information about classes is sometimes called *metaclass*. These objects may contain the class methods and class fields, common to all objects of the corresponding class.

Adding a method to a class thus becomes calling the *addMethod* method upon the corresponding metaclass object.

2.2.15 Run time type identification

It is sometimes useful to determine at run time the actual type of an object referenced from a variable. This should only be used when calling a method, that will perform the desired work for each possible subtype, is not possible or convenient. This requires each object to carry a type identifier. Furthermore, an object of type C should qualify in a run time test

as member of type C, and as member of its ancestor type A.

Each object already points to a method table unique to its type. The type information may thus be stored in the methods table. It usually includes a type code and a reference to its parent type. To verify if an object O is of a certain type T, the program can access the type code in O's method table and compare it with T's typecode. If it fails, the parent's typecode may be checked, the parent's parent... This simple minded organization would require O(d) operations where d is the inheritance depth of O.

Fortunately, an O(1) algorithm exists. If typecodes are assigned to inheritance trees in a depth first search manner, it is possible for each parent class to remember the smallest and largest typecode used for its childs. Any typecode between these two values must belong to a subtype of this parent. An example follows.

```
TYPE
  A = REF RECORD
    methods: TableA;
    field1: TEXT;
    END;

  C1 = REF RECORD
    methods: TableC1;
    field2: TEXT;
    END;

  C2 = REF RECORD
    methods: TableC2;
    field3: TEXT;
    END;

  TableA = REF RECORD
    typecode: CARDINAL := 1;
    maxCode: CARDINAL := 3;
    method1: PROCEDURE();
    END;

  TableC1 = REF RECORD
    typecode: CARDINAL := 2;
    maxCode: CARDINAL := 2;
    method1: PROCEDURE();
    END;
```

```
  TableC2 = REF RECORD
    typecode: CARDINAL := 3;
    maxCode: CARDINAL := 3;
    method1: PROCEDURE();
    END;
...
  (* r contains a reference to an object of type C2 *)

  (* IF ISTYPE(A, r) THEN ... *)

  IF r.methods.typecode <= tableA.maxCode AND
     r.methods.typecode >= tableA.typecode THEN ...
```

Run time type tests become more complex in the presence of multiple inheritance or dynamically changing inheritance trees (new types being added at run time).

2.2.16 Exception Handling

Many library functions return an error code whenever something goes wrong. The calling function may be able to deal with the error or may need to propagate the error code to its calling function. The result is that eventually almost every function call is followed by a test and a return. This affects the program readability by adding a lot of code only useful in exceptional cases.

Exceptions are a mechanism to automatically return from procedures, recursively, until the error condition can be handled appropriately. The number of places where exceptional conditions must be checked and handled is thus greatly reduced. The downside is that procedures may be interrupted abruptly by exceptions at almost every nested procedure call. The programmer must insure that the data structures are in a consistent state at these points. Alternatively he may intercept exceptions and put the structures in a consistent state before sending the exception to outermost procedures.

Exceptions are best used for really exceptional, unpredictable, conditions such as disk full, no memory left or user input syntax errors.

```
EXCEPTION BadFormat(TEXT);

PROCEDURE WriteInfo(msg: TEXT)
  RAISES{BadFormat,IOError} =
```

```
  BEGIN
    TRY
      OpenMessageFile();
      IF BadMsg(msg) THEN RAISE BadFormat(msg); END;
      WriteMessage(msg);
    END;
  END WriteInfo;

PROCEDURE CallWrite() RAISES ANY =
  BEGIN
    TRY
      WriteInfo("Is this message well formatted?");
    EXCEPT
    | IOError => ...;
    | BadFormat(m) => Wr.PutText(Stdio.stderr,m);
    ELSE
      Wr.PutText(Stdio.stderr,"Unknown Error");
    END;
  END CallWrite;
```

Exception handling involves an exception type, an argument describing the exception, exception handlers, and lists of accepted exception types for a procedure. In C++, any class may be used as an exception type, while in Java exception types must inherit from class *throwable*; the exception argument is an instance of the exception type. In Modula-3, exceptions are a separate type space with an optional argument; there are no inheritance relationships between exception types.

Handlers are attached to a code block. Any exception raised within the block transfers the control to the corresponding handlers section. Then, the exception type is checked, taking into account inheritance in the case of C++ and Java, and the first suitable handler found is executed. If no handler is appropriate, the exception is propagated further.

Several languages such as Modula-3, Java and Delphi offer a TRY FINALLY statement. This is used to perform cleanup operations (freeing allocated space, closing files...) whenever an exception exits a code block prematurely. This is equivalent to a handler accepting any exception, performing the cleanup, and raising the exception again.

```
/* without try finally */

try {
  /* allocate memory, open files */
```

```
  ...
  /* free memory, close files */
}
catch(Throwable e) {
  /* free memory, close files */
  throw e; /* propagate the exception */
}

/* with try finally */

try {
  /* allocate memory, open files */
 ...
}
finally {
  /* free memory, close files */
}
```

It is important to know the exceptions that may arise from called procedures. Suitable handlers may then be provided. However, when a procedure is declared to only raise a specific set of exceptions, it is extremely difficult to verify at compile time. Indeed, procedure A may call procedure B which raises FileNotFound and DeviceNotReady. The compiler could assume that A may thus propagate these two exceptions. However, if B is always called in a way that can only raise FileNotFound, declaring A as raising only FileNotFound would be correct. Since the compiler cannot be expected take into account in what context procedures are called, the verification must be performed at run time. This can be implemented as exception handlers.

```
(* A should not raise anything else than
   FileNotFound: RAISES {FileNotFound} *)
PROCEDURE A() =
  BEGIN
    TRY
      ...
      B();
      ...
    EXCEPT
      (* Let FileNotFound propagate *)
    | FileNotFound => RAISES FileNotFound;
    ELSE
```

```
      Wr.PutText("Unhandled exception in A\n");
      Process.Exit(1);
    END;
  END A;
```

In C++, the destructors of local variables must be called upon exiting a block or procedure. The compiler inserts a TRY FINALLY block to insure that the destructors are called even if an exception is raised.

Checking the exceptions propagated by a procedure, cleaning up upon exiting a procedure or a TRY FINALLY block, or ordinary handlers, all rely on a basic mechanism to jump to a handler section. Establishing a handler block, raising an exception, and selecting the appropriate handler can be implemented as follows.

```
  TRY
    DoSomething();
  EXCEPT
  | ErrorType1 => Handle1();
  | ErrorType2 => Handle2();
  END;
...
PROCEDURE DoSomething()
...
  RAISE ErrorType1;
...
```

```
  VAR
    previousContext: jmpbuf := currentContext;
    exceptionRaised: BOOLEAN;
  BEGIN
    exceptionRaised = (setjmp(currentContext) = 0);

    IF NOT exceptionRaised THEN
      DoSomething();
    END;

    currentContext := previousContext;

    IF exceptionRaised THEN
      CASE currentError OF
      | ErrorType1 => Handle1();
      | ErrorType2 => Handle2();
      ELSE
```

```
        longjmp(currentContext);
      END;
    END;
  END;
...
PROCEDURE DoSomething()
...
  currentError = ErrorType1;
  longjmp(currentContext);
...
```

The setjmp system call saves all the execution context, typically the processor registers including the program counter and the stack pointer. It then returns 0. The saved context may later be used with the longjmp system call to return at the point where the setjmp was issued. This time however, to distinguish the longjmp call from a normal setjmp return, 1 is returned.

To establish a handler block, the previous handler context is stored and the context of the new handler block is saved in *currentContext*. Then, anything called within the handler block can issue a longjmp to return directly to the handler block; the context of the previous handler block is then restored in *currentContext*. The error encountered and stored in *currentError* is checked against the handlers available. If no suitable handler is found, the control is passed to the next handler.

In a multi-threaded environment, where several stacks are simultaneously active, some precautions are necessary. One *currentContext* and one *currentError* variable must exist for each thread, and therefore cannot be global variables. Fortunately, multi-threaded environments such as POSIX threads and Win32 do offer per thread properties. The *previousContext* variable is local to the handler block and thus already specific to the thread/stack executing it.

Exception handling was a late addition to the C++ language. As a consequence, few libraries take advantage of it, which is unfortunate.

2.2.17 Generic Modules

There are cases where a similar functionality is needed in several different contexts. In some cases this can be achieved through objects and methods. In other cases, it is not practical for simplicity or performance reasons. A typical example is container classes (lists, stacks, trees...). It is possible to have a ListAny type which accepts any type (an ancestor of all other types). Tests, however, will be required when taking out objects to verify

that they are of the expected type. A single implementation would suffice but it would be less efficient and more cumbersome to use. Moreover, in many programming languages there is no type which can accept every type, including characters, integers and floating point values. In such cases, it may be more convenient and efficient. to have distinct IntegerList, GeometryList, BooleanList types.

Another example is when some functionality should be added to several otherwise unrelated classes. A field and associated methods to store property lists may be wanted. While the same functionality is required in each case, the offset in the structure where the field will be added and other similar parameters may change from one class involved to another.

While distinct executable code will be required in most cases to handle these variants (the size of contained objects changes from GeometryList to BooleanList), a single copy of the source code should exist for maintainability purposes. This *generic* module is then parameterized. Each set of parameters defines a context in which the generic module can be compiled. Below is a simple generic interface and module followed by an instantiation for integer elements.

```
GENERIC INTERFACE ArrayOps(Element);

PROCEDURE Max(READONLY anArray:ARRAY OF Element.T):
  Element.T;

END ArrayOps;


GENERIC MODULE ArrayOps(Element);

PROCEDURE Max(READONLY anArray:ARRAY OF Element.T):
    Element.T =
  VAR largest := Element.First();
  BEGIN
    FOR i:= FIRST(anArray) TO LAST(anArray) DO
      IF Element.Greater(anArray[i],largest)
        THEN largest := anArray[i];
      END;
    END;
    RETURN largest;
  END Max;

END ArrayOps;
```

```
INTERFACE IntegerElement;

TYPE T = INTEGER;
PROCEDURE Greater(a,b:T):BOOLEAN;
PROCEDURE First():T;

END IntegerElement;


INTERFACE IntegerArrayOps = ArrayOps(IntegerElement)
END IntegerArrayOps

MODULE IntegerArrayOps = ArrayOps(IntegerElement)
END IntegerArrayOps
```

2.2.18 Substitutes for multiple inheritance

Multiple inheritance is an interesting mechanism that comes at a relatively high cost in conceptual and implementation complexity. It also incurs a performance penalty. Its benefits are, in some cases, easier code reuse.

For example, a number of classes may need to store properties. A *Propertyclass* may provide a data structure and access methods to store and access properties. Setting the Property class as a common ancestor, using single inheritance, may not be feasible if other classes with common ancestors have no need for properties. While multiple inheritance solves the problem, other approaches may be used to obtain the same result without too much difficulty.

```
/* Class C contains the fields and methods of A plus those of
   PropertyClass plus some of its own. */

class C : A PropertyClass {
...
}

/* The methods of PropertyClass, when inherited from C,
   need to know the offset of the PropertyClass fields
   within C. This involves an overhead in space and
   execution time. */

C objc;
```

```
main() {
...
  objc.addProperty("name",value);
...
}
```

The first way is through aggregation. A PropertyClass object is allocated in each object of classes that need to store properties.

```
(* Class Child contains the fields and methods of Ancestor.T, plus a
   field of type Property.T, plus others as needed. *)

IMPORT Ancestor, Property;

TYPE
  Child = Ancestor.T OBJECT
      properties: Property.T;
      ...
    METHODS
      ...
    END;

VAR
  objc := NEW(Child).init();

(* The same result is achieved simply through accessing the "properties"
   field. A PropertyClass object is contained instead of inherited. *)

BEGIN
...
  objc.properties.addProperty("name",value);
...
```

A second way is to provide a template that adds a property list to a class, producing a child class with properties. In that case, a copy of the methods required to process properties is created in the context of class Child. These copied methods use some memory but avoid the space and time overhead associated with multiple inheritance.

```
GENERIC INTERFACE Property(BaseType);
```

```
TYPE
  Child = BaseType.T OBJECT
      ...fields required to store properties...
    METHODS
      addProperty(name: TEXT; value: INTEGER);
      ...
    END;

END Property.


MODULE DefineChild = Property(Ancestor);
END DefineChild;

...

VAR
  objc: NEW(DefineChild.Child).init();

BEGIN
...
  objc.addProperty("name",value);
...
```

Multiple implementation or interface inheritance may also be useful when a new object type must qualify as two unrelated types. For example, a text buffer may need to appear as a text editor window to the user and as an evolving source code file for an incremental compiler. The window system expects a *Window* descendant, which receives mouse and key clicks and paints text by overriding the *Window* methods. The incremental compiler expects notification about modifications to the buffer from a descendant of the type *ModifiableText*. The intuitive solution may be to inherit from both these object types. The absence of multiple inheritance is only a minor inconvenience. Indeed, it is easy to inherit from one type and provide an adapter to the other type.

For example, CM, a descendant of *ModifiableText*, could be interfaced to the incremental compiler and contain the text. Another type, CW, could be derived from the *Window* type to connect to CM. An instance of CM would contain the text, connect to the compiler, and offer suitable methods to modify the contained text and receive notification. An instance of CW

would connect to an instance of CM and to the window system. It would translate mouse and key clicks into modifications to the text in the CM, and text change notification into painting commands, to update the associated text editor window on screen. Interestingly, this decomposition follows the Model View Controler paradigm and produces more easily reused types.

2.2.19 Discussion

In classical structured programs, each procedure must account for the differences in data types it must handle. For exemple, the draw procedure would need to test and execute different sections to handle Circles and Polygons. In object oriented languages, the common fields and methods are stored in parent classes, while the specifics of the different types handled are defined in their respective class.

When some operation is modified, the changes are localized in the corresponding procedure for classical structured programs. In an object oriented program, the corresponding method, in all the types that override it, must be examined.

Upon adding a new type to handle, all the procedures must be modified in a classical structured program. In an object oriented program, a new class is simply added and the methods specific to that class are defined.

Thus, object oriented programming often requires more time to set up properly the inheritance hierarchy and insure that the common parts are implemented in the parent types. However, adding new types is much easier thereafter.

Interpreted object oriented languages allow more features than compiled ones. Indeed, most interpreted languages will allow dynamically adding new types or adding and removing fields and methods of existing types. Some even allow dynamically changing the type of an existing object or the ancestors of a class. On the other hand, interpreted languages usually execute much slower (2 to 20 times) and cannot determine before the program is executed if a non-existent method will be called or if the arguments types are incorrect. Therefore, interpreted languages are often more interesting for rapid prototyping while compiled languages are better for large, efficient and robust programs.

Encapsulation often means hiding some operations is methods calls. Similarly, decomposing a method in a common part, implemented in the ancestor type, and a specific part, implemented in the child type may also cause extra methods calls. Furthermore, some compilers cannot optimize as much programs when procedures are called through tables, as are methods. For all these reasons, some object oriented programs may run slower than

their classical counterpart. However, a quicker development may justify this overhead. It may even allow spending more time to find efficient algorithms.

The overhead of these extra calls belongs to the low level, constant factor, optimizations. Once a program runs well, it is often easy to identify the component which uses up most of the execution time. Then, some methods calls in this small component may be replaced by non object oriented procedure calls.

Chapter 3

Modular Programming

3.1 Interfaces

In this section, the decomposition of program portions into modules and interfaces is examined. Different organizations are studied.

A large program or library is usually divided into modules. Each module offers a number of variables, procedures and type definitions to be used from other modules; this information is called the interface exported by the module. A module needing information provided by the interface of another module will be called the *client* of this other module. Some languages, like Modula-3, allow modules to divide their exported information into several interfaces; for example, one interface might contain the information required by almost every client while a second interface may contain more specialized information useful to only a few clients. Large libraries of modules usually contain private interfaces, to be used from other modules within the library, and public interfaces accessible to modules outside the library.

The interface should contain only the pieces of information required by client modules. The type definitions and procedures only used locally within the module should be kept out of the interface, to avoid diluting the useful information present in the interface. This is a form of what is often called *information hiding* or *encapsulation*.

The programmer needs to know from the interface the name and arguments of procedures, and the name and public portions of data types and variables. Indeed, private fields within exported data types are only used from within the exporting module and need not be known by clients. Similarly, the exact address of procedures within a module are not useful to the

programmer. The compiler and linker, however, need to know the offset of each field within a data type and the address of each exported procedure and variable.

On virtually every system, the addresses of procedures and variables are stored by the compiler in the object file and used by the linker. The offset of data fields, however, must often be known to the compiler. In many languages, like in C or C++, the private portions of type definitions must be included in the interface to let the compiler compute the offset of the public data fields. Other languages, like Modula-3, keep the interfaces clean by having the compiler output the field offsets into a separate file, used when compiling client modules.

There may be redundancy in declaring exported procedures in the interface and then implementing them in the module. Some programming environments let the programmer flag the exported items in the module and offer tools that automatically extract these items and produce the exported interface.

While most structured languages, (Modula, ADA), have explicit mechanisms to export and import interfaces, some have none and obtain similar functionality through a macro pre-processor. Indeed, in C or C++, a declaration is shared by several modules simply by putting it in a separate file included by several modules. While using an existing tool (macro pre-processor) instead of adding to the language, is interesting, there are some very serious problems with this.

The macro pre-processor has its own language and does not understand the program semantics. For example, a missing semi-colon in an included file will not be noticed until the following statement in the including file and the compiler may have a hard time reporting the error location in a meaningful way. More importantly, the effect of the textual inclusion depends on the pre-processor state. Therefore, if file A includes files B and C and file B also includes C, the pre-processor cannot decide that including C the second time is useless because the macro pre-processor state may have changed and macros within C may produce a different result the second time.

The example below shows portions of a C++ program with associated include files serving as exported interfaces. The private portions of data types must be found by the compiler in the interface. Some of the interfaces will be included several times needlessly.

```
File Main.c

#include <stdio.h>
#include <iostream.h>
```

```
#include "Second.h"

main() {
...
  routine(a,b);
}
```

File Second.c

```
#include "Second.h"

void routine(int a, Obj b) {
...
}

void local_routine() {
...
}
```

File Second.h

```
#include <iostream.h>

class Obj {
  private:
    int a;
    char *b;
  public:
    int access();
    void set(int i);
  ...
};

void routine(int a, Obj b);
...
```

In the second example, a tool automatically exports all items marked as *export*. The compiler can be told to ignore the export keyword by defining it as a macro that produces the empty string. Thus, the programmer only writes the *Second.c* file and the *Second.h* file is generated automatically.

File Main.c

```
#include <stdio.h>
#include <iostream.h>
#include "Second.h"

main() {
...
  routine(a,b);
}
```

File Second.c

```
export #include <iostream.h>

export class Obj {
  private:
    int a;
    char *b;
  public:
    int access();
    void set(int i);
  ...
};

export void routine(int a, Obj b) {
...
}

void local_routine() {
...
}
```

In Java, the interface is automatically extracted from the source file by the compiler. Any declaration carrying the *public* attribute is available to the outside. This scheme is easier to manage since fewer files are required (i.e. no separate interface file). However, one cannot have several interfaces for a module and finding the relevant public declarations requires looking up the complete module.

In the third example, an equivalent Modula-3 program portion is shown. Interfaces and Modules are defined in the language and it is possible to declare in the interface only the public portion of data types.

File Main.m3

```
MODULE Main;

IMPORT Stdio, Second;

BEGIN
...
  Second.Routine(a,b);
END Main.
```

File Second.m3

```
MODULE Second;

IMPORT Stdio;

REVEAL
  Obj = Public OBJECT
      a: INTEGER;
      b: TEXT;
    METHODS
    END;

PROCEDURE Routine(a: INTEGER; b: Obj) =
  BEGIN
...
  END Routine;

PROCEDURE Local_routine() =
  BEGIN
...
  END Local_routine;

BEGIN
END Second.
```

File Second.i3

```
INTERFACE Second;

IMPORT Stdio;

TYPE
```

```
  Obj <: Public;
  Public = OBJECT METHODS
    access(): INTEGER;
    set(i: INTEGER);
    ...
  END;

PROCEDURE Routine(a: INTEGER; b: Obj);
...
```

3.2 Language Barriers

Interfacing modules written in different computer languages has always been problematic, yet unavoidable. For example, most available libraries on POSIX systems are written in C. This includes system calls, graphics libraries, and possibly other specialized libraries. On Win32 systems, most libraries are now written in C but are descendants of libraries once written in Pascal. For this reason, most of the Win32 API libraries retain the Pascal calling convention and use the WINAPI keyword in their declaration.

In order to call a procedure written in a different language, several parameters need to be adjusted. The arguments are usually passed on the stack, but the first few may be put in registers. The arguments may be pushed on the stack from left to right or right to left. The return value may added to the left or to the right of the argument list, or placed in a register. The parameters may be passed by reference or by value. Sometimes large values such as structures are not passed on the stack but copied elsewhere in memory.

The procedure name itself is often mangled in some way. C compilers often add an underscore in front, C++ compilers add the list or arguments plus a few funny characters at the end, and WINAPI procedures have the total size of their arguments appended with an @.

Thus, when a foreign procedure must be used, the compiler must be told the corresponding type (i.e. the calling convention such as C, WINAPI...). If the compiler supports the desired calling convention it will then generate the procedure name and process the arguments appropriately.

An even more serious problem remains. The data types of the arguments must be compatible. Usually, exchanging arguments of primitive types (integer, float, char) does not pose any problem. Exchanging structures (records) require that the two languages use the same layout in memory. Failing that, the calling language needs to have enough control over the memory layout of structures to adapt to the called language. In most

cases, the memory layout dictated by the dominant language on a platform will be followed by all the other languages.

There are however more advanced constructs and facilities which are harder to make interoperable between languages. Objects in C++ support multiple inheritance and it would be wasteful in Modula-3 or Java to use the same layout for objects and methods tables as C++. Similarly, exception handling and synchronization primitives have appeared fairly recently. They are not implemented in dominant languages (i.e. C) and will most often not be interoperable between C++, Java and Modula-3.

Garbage collection adds another twist to interoperability. Copying garbage collectors may change the location of objects during the execution as long as references to these objects are updated appropriately. In the presence of preemptive multi-threading, the garbage collector may be called in at any time, including in the middle of a call to a foreign procedure. Unless the foreign language interfaces to the garbage collector and provides information about references to update, this may cause problems. Thus, only fixed objects need to be passed to the foreign procedures. This may be achieved by temporarily disabling the garbage collector, pinning these objects in memory (asking the garbage collector not to move these objects), or passing objects allocated from a special memory area outside the reach of the garbage collector.

In SRC Modula-3, foreign procedures are declared as EXTERNAL with a keyword to identify the calling convention (i.e. C, WINAPI). Primitive types and records use the same layout as C structures. Objects referenced from the stack are pinned and may thus be passed to foreign procedures; it is also possible to allocate UNTRACED objects (i.e. not managed by the garbage collector). In Java, foreign C procedures are declared as *native* methods. A non-copying garbage collector is used, which may be less efficient but avoids the problem of moving objects.

3.3 Litterate Programming

There are software development environments where each activity (analysis, design, coding, documentation, testing...) is supported and sometimes enforced. Such environments manage the relationships between the modules documents and test files and insure the that they are kept in synchrony. Nonetheless, they often lack the flexibility of simpler organizations.

The term literate programming is associated with an organization where the documentation is part of the program; a subset of the comments and code in the program are automatically extracted and included in the documentation. Thus, whenever the program is modified, the relevant comments

should also be updated. An updated documentation can then be extracted automatically.

Donald Knuth included documentation in the TeX program. Tools would use the documented program to produce a printed book with the typeset annotated program listing and to produce the Pascal source code. Greg Nelson developed a tool to extract the documentation from Modula-3 interfaces. The extracted interface documentation may then be included as sections of a larger document describing a whole library or program. Linking the extracted documentation to the main document may be achieved through inclusion or reference mechanisms in document preparation systems.

For instance, TeX allows files to be included with the *input* command. Similarly, it is possible in HTML to define hypertext references to other files. The HTML browser or server may even be configured to extract on the fly the documentation whenever a program file is referenced. This way the most up to date documentation is always available, without the need to extract the documentation and reformat the document each time. Most of the Java and Modula-3 modules are documented in this way using tools like *m3browser*.

3.4 Dependency Analysis

In this section are investigated ways to specify which modules form a program and, when some modules or interfaces are modified, to determine which modules need be recompiled.

Each module in a program or library may *export* public procedures, variables and data types in one or more interfaces. Client modules, which need the exported functionality, will *import* the corresponding interface.

A number of modules linked together may form a library. In that case, the list of modules to group into the library must be supplied, as well as the list of interfaces. Some interfaces are for internal use, while others are to be accessible from modules outside the library and are said to be *exported by the library*. Modules, and perhaps some libraries, linked together may form a program. The starting point of the program must be identified, usually by the reserved name *Main*.

The list of modules forming a program might be inferred by looking recursively at the interfaces imported by the main module and imported by the modules associated with these interfaces. Additional information may be required however. Indeed, some of the required modules may reside in separate directories or in libraries. Therefore, the list of directories and libraries to search needs to be specified.

There are further cases where this information is not sufficient. Other modules not imported by any of the modules reached from the *Main* module may still affect the program. Indeed, the initialization code associated with a module may call procedures in modules reached from the *Main* module and affect the value of some global variables used elsewhere in the program. Finally, some programming environments allow connecting to procedures at run time. For instance, some environments allow reading persistent objects from disk and will reconnect any pointer to procedure to the needed procedures in the program. In such cases, the complete list of modules required for a program cannot be deduced automatically but needs to be specified by the programmer.

Program development usually requires numerous recompilations, each time after corrections are made to a small subset of the modules. Dependency analysis must be performed to determine which modules need to be recompiled. Indeed, recompiling everything each time is wasteful. Leaving the programmer decide which modules need recompilation is error prone and leads to frustrating situations when the compiled program does not correspond to the source code.

A simple form of dependency analysis is performed in *makefiles*. There, the files used for a program are listed along with their dependencies and the actions needed to produce the derived files. For example, the executable program depends on all the object files and is produced by linking them together. Each object file depends on the corresponding source file, and the imported interfaces, and is produced by the compiler. Some versions of *makefiles* also account for the compilation options used. To rebuild a program, it is checked if the options changed or if any file is older than files on which it depends, in which case the associated action is triggered. Thus, if a source file is newer than the corresponding object file, it gets recompiled and the program gets relinked because the new object file is more recent than the executable than depends on it.

For large programs, however, finer dependency analysis may reduce the number of modules to recompile each time. Furthermore, it is interesting if the list of dependencies, especially between a module and the imported interfaces, are deduced automatically. Some systems, like m3build for Modula-3, will store with each compiled module the list of elements (type declaration, variable, procedure...) on which they depend, and a fingerprint of their content. When an interface that a module imports changes, this list can be used to determine if the change indeed impacts on this module. In many cases, this finer grained dependency analysis can reduce by a large amount the number of modules recompiled.

In C or C++, a class A may use another class B as field. The field is an instance of class B instead of simply a pointer to it. Thus, the size of

an instance of class A, and the offsets of its fields, depend on the size of B. Therefore, whenever a field is added in B, any module accessing fields in A need to be recompiled. In Modula-3 or Java, an object type A may only contain a reference to an object type B as field. When a field is added to object type B, the size of the reference to it does not change, which reduces the dependencies and amount of recompilation required. The downside is that two seperate objects reside in memory, which increases the overhead.

Below is a sample *makefile*. The *make* program is a fairly general dependency analysis tool, not limited to building C or C++ programs. For each component, the dependents are listed after the colon and the associated action is supplied on the following line.

```
Main.o: Main.c Second.h
        cc -c Main.c

Second.o: Second.h
        cc -c Second.c

Main: Main.o Second.o
        cc -o Main Main.o Second.o
```

The following is a *m3makefile* which is used by *m3build*, a tool developed to perform fine grained dependency analysis of Modula-3 programs. Each component is identified as a module (implementation), an interface (interface), or a module and an interface (module) and the name of the produced executable is supplied (program). The first letter is capitalized if the corresponding component is to be exported (e.g. Interface("Foo") if Foo is to be exported by a library).

```
implementation("Main")
module("Second")
Program("Main")
```

Most large software projects developed in C or C++ suffer from the coarse grained dependency analysis and the error-prone manual entry of dependencies between modules (.c) and interfaces (.h). The common remedy is often to make a clean recompilation each night. Whoever introduces a fatal error which prevents the program from running the following day is in trouble. In some cases, for very large projects where included interface files were not carefully designed to avoid repeated inclusions, the night may not be long enough to recompile everything.

3.5 Modules in Modula-3

In this section, the use of tools to assemble C files, Modula-3 files, and libraries, into libraries and programs is illustrated. The selected example involves C and Modula-3 files and libraries, and remains simple while being a useful application. The tools involved are `m3build`, `m3ship`, `m3where` and `m3tohtmlf`. These tools are available both from the command line and from menus within Emacs.

Each library or program is a separate package with its own directory tree and `m3makefile`. The example detailed in this section consists in two packages:

- *database*: a library `Database` which interfaces to the C database dbm library.
- *query*: a program `QueryDbase` using this library.

Assuming that all the packages are stored in, for instance, the `pkg` subdirectory of the home directory, the following directory structure will exist:

- ~ /pkg/database/src, source code for the Database library.
- ~ /pkg/database/src/m3makefile, description of the Database library for `m3build`.
- ~ /pkg/database/LINUXELF, directory created by `m3build` to store the object files for the Database library on a LINUXELF computer. Storing the object files separate from the source code keeps the `src` directory cleaner and allows compiled versions for several architectures to coexist.
- ~ /pkg/query/src, source code for the QueryDbase program.
- ~ /pkg/query/src/m3makefile, description of the QueryDbase program for `m3build`.
- ~ /pkg/query/LINUXELF, directory created by `m3build` to store the object files for the QueryDbase program on a LINUXELF computer.

The Database library is composed of an interface to the dbm C library, `Dbm.i3`, a C module `Dbm.c`, a module `Dbase.m3` and an interface `Dbase.i3`. The dbm library, `/usr/lib/libdbm.a`, and the Modula-3 standard library, `libm3`, are used. Thus, the `m3makefile` for the `database` package is as follows:

```
import("libm3")           % Modula-3 library
import_lib("dbm","/usr/lib") % Foreign library

c_source("Dbm")

interface("Dbm")          % non exported

Interface("Dbase")        % Dbase.i3 is exported

implementation("Dbase")   % non exported

Library("Database")       % libDatabase.a is exported
```

The interfaces to be available outside of the package are listed with `Interface`, and usually document the package, while the interfaces used internally are listed with `interface`.

The `Dbm.i3` interface only redirects the calls to the appropriate C functions in dbm and provides Modula-3 declarations for the C structures and constants. The Modula-3 compiler uses the same primitive types and the same memory layout for struct/RECORD as the C compiler.

```
UNSAFE INTERFACE Dbm;

(* The functions and types offered by the dbm library are declared
   in Modula-3 terms to make them accessible to Modula-3 procedures. *)

IMPORT Ctypes;

TYPE
  Datum = RECORD
      dptr: Ctypes.char_star;
      dsize: Ctypes.int;
    END;

  T = UNTRACED ROOT;

CONST
  DBM_INSERT = 0;
  DBM_REPLACE = 1;
  O_RDONLY = 0;
```

```
  O_WRONLY = 1;
  O_RDWR = 2;
  O_CREAT = 16_0200;

<*EXTERNAL dbm_open*>
PROCEDURE Open(file: Ctypes.char_star; flags, mode: INTEGER): T;

<*EXTERNAL dbm_close*>
PROCEDURE Close(db: T);

<*EXTERNAL dbm2_fetch*>
PROCEDURE Fetch(db: T; key: Datum): UNTRACED REF Datum;

<*EXTERNAL dbm_store*>
PROCEDURE Store(db: T; key, content: Datum; flags: INTEGER): INTEGER;

<*EXTERNAL dbm_delete*>
PROCEDURE Delete(db: T; key: Datum): INTEGER;

<*EXTERNAL dbm2_firstkey*>
PROCEDURE FirstKey(db: T): UNTRACED REF Datum;

<*EXTERNAL dbm2_nextkey*>
PROCEDURE NextKey(db: T): UNTRACED REF Datum;

END Dbm.
```

A few lines in C are provided in `Dbm.c` to adapt the values returned by some functions.

```
#include <ndbm.h>

/* Functions returning structures are often handled in non portable
   ways. Instead, a pointer to structure is returned here. */

datum tmp_datum;

datum *dbm2_fetch(DBM *db, datum key)
{
  tmp_datum = dbm_fetch(db,key);
```

```
  return &tmp_datum;
}

datum *dbm2_firstkey(DBM *db)
{
  tmp_datum = dbm_firstkey(db);
  return &tmp_datum;
}

datum *dbm2_nextkey(DBM *db)
{
  tmp_datum = dbm_nextkey(db);
  return &tmp_datum;
}
```

The `Dbase.i3` interface, with the corresponding implementation `Dbase.m3`, offers a cleaner access to the dbm library.

```
INTERFACE Dbase;

(* A Dbase.T is a handle on a dbm database. *)

TYPE
  T <: REFANY;

(* Create opens the named file as a new database. *)

PROCEDURE Create(file: TEXT): T;

(* Open opens the existing named file as a database. *)

PROCEDURE Open(file: TEXT): T;

(* Close releases the program resources associated with the database. *)

PROCEDURE Close(db: T);

(* Fetch retrieves an item in the database given its key. NIL is returned
   if not found. *)

PROCEDURE Fetch(db: T; key: TEXT): TEXT;
```

```
(* Store puts a content under the given key in the database. *)

PROCEDURE Store(db: T; key, content: TEXT);

(* Delete removes the specified key and its content from the database. *)

PROCEDURE Delete(db: T; key: TEXT);

(* FirstKey and NextKey may be used to iterate through all the
   keys of the database entries. NIL is returned when there are no more
   entries. *)

PROCEDURE FirstKey(db: T): TEXT;

PROCEDURE NextKey(db: T): TEXT;

END Dbase.
```

The `Dbase.i3` interface is the only visible part of the database package and contains the relevant documentation. This documentation may be extracted and formatted with `m3tohtmlf`.

```
cassis>m3tohtmlf Dbase.i3 >Dbase.html
```

The implementation in `Dbase.m3`, for sake of simplicity, does not perform much error checking. Fortunately most errors will be caught by the Modula-3 run time checks.

The dbm C library reuses the datum strings returned. Thus, they must be copied before calling again the dbm library. This behavior is not uncommon in C, in the absence of garbage collection, but could create problems in a multi-threaded environment.

```
UNSAFE MODULE Dbase;

(* This module offers a cleaner interface to the ndbm functions.
   It converts arguments and results between Modula-3 TEXT and
   C character pointers. All the pointers to structures returned by
   C functions (DBM and datum) are UNTRACED REFs because they should
   not be managed by the Modula-3 garbage collector. *)
```

```
IMPORT Text, Dbm, Ctypes, M3toC, Cstring, TextF;

REVEAL
  T = BRANDED REF RECORD
      d: Dbm.T;
    END;

CONST
  Mode = 8_77777;

PROCEDURE Create(file: TEXT): T =
  VAR
    db := NEW(T);
  BEGIN
    db.d := Dbm.Open(M3toC.TtoS(file), Dbm.O_RDWR + Dbm.O_CREAT, Mode);
    IF db.d = NIL THEN RETURN NIL; END;
    RETURN db;
  END Create;

PROCEDURE Open(file: TEXT): T =
  VAR
    db := NEW(T);
  BEGIN
    db.d := Dbm.Open(M3toC.TtoS(file), Dbm.O_RDWR, Mode);
    IF db.d = NIL THEN RETURN NIL; END;
    RETURN db;
  END Open;

PROCEDURE Close(db: T) =
  BEGIN
    Dbm.Close(db.d);
  END Close;

PROCEDURE Fetch(db: T; key: TEXT): TEXT =
  VAR
    in: Dbm.Datum;
    out: UNTRACED REF Dbm.Datum;
  BEGIN
    in.dptr := M3toC.TtoS(key);
    in.dsize := Text.Length(key);
    out := Dbm.Fetch(db.d,in);
    IF out.dptr = NIL THEN RETURN NIL; END;
```

```
    RETURN CopyStrNtoT(out.dptr,out.dsize);
  END Fetch;

PROCEDURE Store(db: T; key, content: TEXT) =
  VAR
    k, c: Dbm.Datum;
  BEGIN
    k.dptr := M3toC.TtoS(key);
    k.dsize := Text.Length(key);
    c.dptr := M3toC.TtoS(content);
    c.dsize := Text.Length(content);
    EVAL Dbm.Store(db.d, k, c, Dbm.DBM_REPLACE);
  END Store;

PROCEDURE Delete(db: T; key: TEXT) =
  VAR
    k: Dbm.Datum;
  BEGIN
    k.dptr := M3toC.TtoS(key);
    k.dsize := Text.Length(key);
    EVAL Dbm.Delete(db.d, k);
  END Delete;

PROCEDURE FirstKey(db: T): TEXT =
  VAR
    out: UNTRACED REF Dbm.Datum;
  BEGIN
    out := Dbm.FirstKey(db.d);
    IF out.dptr = NIL THEN RETURN NIL; END;
    RETURN CopyStrNtoT(out.dptr,out.dsize);
  END FirstKey;

PROCEDURE NextKey(db: T): TEXT =
  VAR
    out: UNTRACED REF Dbm.Datum;
  BEGIN
    out := Dbm.NextKey(db.d);
    IF out.dptr = NIL THEN RETURN NIL; END;
    RETURN CopyStrNtoT(out.dptr,out.dsize);
  END NextKey;

(* This procedure relies on the internal representation of TEXT
```

```
   elements. *)

PROCEDURE CopyStrNtoT (s: Ctypes.char_star; n: INTEGER): TEXT =
  VAR
    t := NEW (TEXT, n + 1);
  BEGIN
    EVAL Cstring.memcpy (ADR (t[0]), s, n);
    t[n] := '\000';
    RETURN t;
  END CopyStrNtoT;

BEGIN
END Dbase.
```

The `QueryDbase` program consists in a single module which exercises the `Database` library. Its m3makefile and implementation, `Query.m3`, are as follows. It contains time and memory performance bugs which will be used later to illustrate how such bugs can be uncovered.

```
% database is located in our private package collection
override("database","../..")

import("database")

implementation("Query")

% link statically to ease debugging
build_standalone()

Program("QueryDbase")


MODULE Query EXPORTS Main;

(* This program accepts commands to create or open ndbm databases
   and to store, fetch or delete elements. A sorted list of keys
   in the database may also be printed. *)

IMPORT Stdio, Dbase, Wr, Rd, Lex, Text, Thread;

(* Exceptions are not caught and will stop the program execution. *)

<*FATAL Wr.Failure*>
```

```
<*FATAL Rd.Failure*>
<*FATAL Thread.Alerted*>

(* Read all the keys in the database into an array. If the array
   is too small, double its size. *)

PROCEDURE PrintDbase(db: Dbase.T) =
  VAR
    key, tmp: TEXT;
    cursor: CARDINAL := 0;
  BEGIN
    key := Dbase.FirstKey(db);
    WHILE key # NIL DO
      IF cursor > LAST(keyArray^) THEN
        oldKeyArray := keyArray;
        keyArray := NEW(REF ARRAY OF TEXT,2 * NUMBER(oldKeyArray^));
        SUBARRAY(keyArray^,0,NUMBER(oldKeyArray^)) := oldKeyArray^;
      END;
      keyArray[cursor] := key;
      INC(cursor);
      key := Dbase.NextKey(db);
    END;

    (* Use an inefficient sorting algorithm *)

    FOR i := cursor - 1 TO 0 BY -1 DO
      FOR j := 0 TO i - 1 DO
        IF Text.Compare(keyArray[j], keyArray[i]) = 1 THEN
          tmp := keyArray[i];
          keyArray[i] := keyArray[j];
          keyArray[j] := tmp;
        END;
      END;
    END;

    (* Print all the keys *)

    FOR i := 0 TO cursor - 1 DO
      Wr.PutText(Stdio.stdout,keyArray[i] & "\n");
    END;
  END PrintDbase;
```

```
(* The array for printing the keys is kept from one invocation to the next
   instead of starting from scratch each time the PrintDbase procedure
   is called. *)

VAR
  keyArray := NEW(REF ARRAY OF TEXT,8);
  oldKeyArray : REF ARRAY OF TEXT;
  db: Dbase.T;
  command, name, key, content: TEXT;

BEGIN

  (* Commands are read. For each possible command, the
     appropriate few lines are executed. *)

  LOOP
    Wr.PutText(Stdio.stdout,"Enter command\n");
    Wr.Flush(Stdio.stdout);

    Lex.Skip(Stdio.stdin);
    command := Lex.Scan(Stdio.stdin);

    IF Text.Equal(command,"exit") THEN EXIT;

    ELSIF Text.Equal(command,"create") THEN
      Lex.Skip(Stdio.stdin);
      name := Lex.Scan(Stdio.stdin);
      db := Dbase.Create(name);

    ELSIF Text.Equal(command,"open") THEN
      Lex.Skip(Stdio.stdin);
      name := Lex.Scan(Stdio.stdin);
      db := Dbase.Open(name);

    ELSIF Text.Equal(command,"close") THEN
      Dbase.Close(db);

    ELSIF Text.Equal(command,"fetch") THEN
      Lex.Skip(Stdio.stdin);
      key := Lex.Scan(Stdio.stdin);
      Wr.PutText(Stdio.stdout,Dbase.Fetch(db,key) & "\n");
```

```
    ELSIF Text.Equal(command,"store") THEN
      Lex.Skip(Stdio.stdin);
      key := Lex.Scan(Stdio.stdin);
      Lex.Skip(Stdio.stdin);
      content := Lex.Scan(Stdio.stdin);
      Dbase.Store(db,key,content);

    ELSIF Text.Equal(command,"delete") THEN
      Lex.Skip(Stdio.stdin);
      key := Lex.Scan(Stdio.stdin);
      Dbase.Delete(db,key);

    ELSIF Text.Equal(command,"print") THEN
      PrintDbase(db);

    ELSE
      Wr.PutText(Stdio.stdout,"Incorrect command\n");
    END;
  END;
END Query.
```

Whenever the *database* source files (Dbase.i3, Dbase.m3, Dbm.i3 or Dbm.c) or one of the imported libraries (ndbm or libm3) are modified, the library may be rebuilt (recompiling only the needed modules) with `m3build`. Similarly, when `query` or its imported libraries are modified (Query.m3 or database), it may be rebuilt with `m3build`.

```
cassis>cd ~/pkg/database
cassis>ls
src
cassis>m3build
mkdir LINUXELF
--- building in LINUXELF ---
m3 -w1 -why -g -a libDatabase.a -F/usr/tmp/qk13734aaa
new source -> compiling ../src/Dbm.c
new source -> compiling ../src/Dbm.i3
new source -> compiling ../src/Dbase.i3
new source -> compiling ../src/Dbase.m3
 -> archiving libDatabase.a
cassis>cd ..
cassis>ls
```

```
database/  query/
cassis>cd query
cassis>ls
src/
cassis>m3build
mkdir LINUXELF
--- building in LINUXELF ---
m3 -w1 -why -g -o QueryDbase -F/usr/tmp/qk13793aaa
new source -> compiling ../src/Query.m3
 -> linking QueryDbase
```

Often, a common repository is used to make the packages available for general use. The **m3ship** command is used to copy all the exported interfaces and binaries produced by a package to the common repository. Once a package is installed in the common repository, there is no need to use the **override** command in the m3makefile to specify its location.

```
cassis>cd ~/pkg/database
cassis>m3ship
m3mkdir /usr/local/lib/m3/pkg/database/src
cp -p Dbase.i3 /usr/local/lib/m3/pkg/database/src
m3mkdir /usr/local/lib/m3/pkg/database/LINUX
cp -p libDatabase.a /usr/local/lib/m3/pkg/database/LINUXELF
--- shipping from LINUXELF ---
```

To know exactly where all the included modules, interfaces and libraries were found, the **m3where** command may be used. This is useful, among other things, to specify the paths when using a debugger.

```
cassis>cd ~/pkg/query
cassis>m3where -h
--- searching in LINUXELF ---
/usr/local/lib/m3/pkg/libm3/src/random/Common/RandomPerm.i3
/usr/local/lib/m3/pkg/libm3/src/formatter/Formatter.i3
/usr/local/lib/m3/pkg/libm3/LINUX/SortedTextRefTbl.i3

...

../../database/src/Dbase.i3
```

The executable program is produced in the architecture specific directory, LINUXELF on an intel architecture running the Linux operating system.

```
cassis>LINUXELF/QueryDbase
Enter command
create example
Enter command
store key1 content1
Enter command
store key3 content3
Enter command
store key2 content2
Enter command
print
key1
key2
key3
Enter command
close
Enter command
exit
cassis>
```

Chapter 4

Run time Libraries and Tools

4.1 Safe Language Features

Programming safety means that programs maintain a minimum coherency and fail in a graceful way. In other words, no core dumps. This forbids accessing unitialized pointers or discarded objects, unrestricted pointer arithmetic, and unchecked array indexing (thus accessing outside an array). Indeed, each of these errors may lead to memory corruption and unpredictible behavior. The constraints and checks required for safety may be summarized as follows.

- All variables and object fields must be initialized to avoid later finding there a value inconsistent with their type. An unitialized integer range may contain a value outside the prescribed range, while an unitialized pointer may point to a random location in memory. In Modula-3 this is achieved by garanteeing that each variable and field is initialized with a value coherent with the declared type. Typically references are initialized to NIL (0) and references to NIL are trapped by the memory protection hardware. In Java, the default constructors for objects provide coherent values, and the compiler has a complex set of rules to insure that each variable is initialized before used.

- Any value assigned to an object field or variable must be consistent with its type. The compiler can readily accept assignments where the right value domain is a subset of the left value domain (i.e. a := b where a is [0..25] and b is [10..15], or a is of type A and b of

type C where C inherits from A). Similarly, the compiler can reject assignments where the right and left values domains do not intersect (i.e. a := b where a is [0..15] and b is [20..35]). When the right value domain is not a proper subset but intersects the left value domain, a run time check is required. In C++, a type cast may be used but is not checked. In Java and Modula-3, the compiler adds run time checks to the generated code; an exception is raised during the execution if the check fails.

```
VAR
  a: [0..15];
  b: [10..20];

BEGIN
...
  (* check: IF b < 0 OR b > 15 THEN RAISE NARROW_FAILED END *)
  a := b;
END
```

- No pointer arithmetic is allowed. Memory addresses are only obtained from the dynamic allocator (i.e. NEW) or when indexing arrays. References accesses are checked for unitialized pointers (i.e. NIL). Array accesses are checked against the array bounds. This way, it is only possible to reference objects in memory of the expected type, as per the reference of array type.

- Manual deallocation of objects is not permitted. Indeed, any reference to the object would not any more point to a valid object. An object can only be deallocated when no reference to it remains, which is what a garbage collector is all about: finding and reclaiming unreferenced objects. Unlike the other safety checks which are relatively simple to implement efficiently, garbage collection is a major component and is discussed separately. Ada does not mandate garbage collection; it simply forbids manual deallocation when safety is required, thus greatly limiting the programming flexibility allowed.

Fortunately, on an adequate operating system, the damages caused by unsafe programs are usually limited to crashing the faulty program, using up all available memory and CPU time, and messing up with the program owner's files.

Program development, testing, and debugging benefit from programming safety. Indeed, without such coherency checks, many errors may

remain undetected for some time. Even when an error is detected, its symptoms may appear far from the original cause, making it very hard to locate. There is however a price to pay, in performance and memory, for such checks. Each pointer and array access may require a test. Garbage collection is also mandatory. This may amount to something like a 20% execution time overhead. However, promising work is under way to detect when such checks are unnecessary and can be optimized out. Most programming environments also offer a compilation option to keep or remove such checks.

Nevertheless, there are languages where the semantics are based on pointer arithmetic and such checks are close to impossible in the general case. In C++, for instance, a safe subset has been defined for which checks may be added to provide safety. Some C and C++ environments offer debugging tools that will catch a number of such errors; checks are added to all pointer accesses in order to verify that the address belongs to a live object, which may detect using discarded objects or accessing uninitialized pointers.

There are however cases where unsafe operations are required. A possible example is the core of the run time libraries that implement garbage collection. Another example is interacting with library procedures implemented in an unsafe programming language. Modula-3 offers an interesting solution to this problem. When a module is prominently marked as UNSAFE, which should discourage careless use, some unsafe operations such as pointer arithmetic are allowed. Experience shows that, outside the low level libraries, few Modula-3 programs ever need unsafe operations. This allows Modula-3 to remain a safe programming language while not forcing programmers to use another language for exceptional cases of unsafe low level programming. In Java, unsafe operations must be coded in C as *native* methods. It does require using another language but Java being very close to C, it is only a minor annoyance.

4.2 Debugging

More often than not, a program does not produce the expected result upon its first execution. A debugger is a tool used to trace the execution and examine the content of variables. This is sufficient to determine what is going on in the program. In some cases, the debugger may enable changing the content of variables, changing the execution flow and, in rare cases, changing the program, all this while it is executing. This is useful to see the effect of proposed changes or to better test specific parts of a program.

Since it is tedious to trace the program execution through millions of

executed statements, until it reaches the program section of interest, it is possible to specify breakpoints, statements where the execution will be interrupted. The program is then started and, once the specified breakpoint is reached, its execution may be traced step by step. Similarly, it is possible to set a watchpoint on some variable such that the program is interrupted each time this variable is modified. This is useful for determining why some variable does not have the expected content.

To achieve this, the debugger must be able to control the execution of a program. Most operating systems offer such facilities. They allow the debugger to execute a program one instruction at a time or until some specified addresses, and to read or write the program memory. On Posix systems, this is achieved through the *ptrace* system call. It is even possible to attach to an already running process and start debugging it.

Nonetheless, the debugger is responsible for interpreting the program memory content. Indeed, a program is simply memory containing machine code and binary data. The user, on the other hand, understands procedures, variables, types and source code line numbers. Debugging information is stored in the executable file by the compiler and linker. This information gives the name and address of each variable and procedure as well as the address of each module line number. It also describes the type and offset of each field, for each type.

This information is used by the debugger to interact with the user. The debugger may also offer language modes to better support the syntax and calling conventions of each supported language. Effectively, the built-in types, syntax and naming changes from one language to another (e.g. char * and my_function in C versus TEXT and Module.MyFonction in Modula-3).

In some cases, the debugging information is not available or this level or detail is not needed. Instead, it may be sufficient to trace all the interactions between the program and the operating system. Indeed, all file accesses, for instance, are performed through system calls and may be used to determine what the program is doing. Many operating systems allow tracing system calls in a program, for example strace or trace on Unix systems.

4.3 Coverage Analysis

Computer engineers often need to know how frequently each instruction in a program is executed. This information may be used to determine which portions of the program have been executed and tested by the test data or which portions of the program consume the most CPU time. It may also be used to determine which instruction types are used more often and design

better computers accordingly, more efficient for the frequent instruction types.

This information may be obtained by running programs on a computer while a logic analyzer connects to the instruction address bus and determines the number of times each instruction in the program is executed. Using the debugging information, which tells to which line number corresponds each address, it is also possible to determine the number of times each line number (statement) is executed. The debugger may be used to generate an execution trace. The trace can thereafter be processed to count the number of times each instruction or statement is executed. However, this is usually quite slow.

Another method is to have the compiler, or assembler, insert instructions that count the number of times each program section is executed. Since sequences of instructions without intervening branch or entry point are always executed together, a single counter is sufficient for each such sequence. At the end of the program execution, the final counts may be printed in a file. The time overhead of this method is small since it only requires one increment operation per instruction sequence.

There are cases where the assembly or high level source code is not available and using special purpose hardware like a logic analyzer is not convenient. Fortunately, there are some tools that read an executable file and output a modified executable file with instructions inserted to count the number of times each instruction is executed.

While frequency counts are an indication of CPU time usage, they are not very accurate. Indeed, they do not tell the number of cycles spent on each instruction (which may be hard to compute on a superscalar pipelined processor) or the effect of waiting for the cache or virtual memory. The frequency counts may be sufficient to detect obvious performance bugs but other tools may offer greater accuracy.

4.4 Performance Analysis

Performance tuning is an important aspect of program development, once the program runs correctly. Ideally, tools should be able to tell exactly how much time is spent on each instruction or in each procedure in a program. The time spent in a procedure may or not include the time spent in nested procedures.

A first problem is to obtain the timing information. Measuring the exact time spent on each instruction is not possible. Instead, statistical measures are frequently used. At a regular interval, the program is interrupted and the stack is examined to determine in which procedures the program is.

The number of times a procedure appears on the top of stack, when the program is interrupted, is a good indication of the CPU time spent in that procedure. Often, the number of times each procedure is called is also computed; the compiler inserts instructions to count the calls.

This information produces a list of call stacks that can be merged together to form a call tree. Each node in the tree is annotated with the number of times it appeared on the top of stack. This tree can be huge since each procedure (child) may be called from several others (parents) and even be called recursively. From there, for each procedure, the following information may be extracted and presented to the user:

- The number of times it was called.
- The time spent in the procedure, including and excluding the nested calls, from each possible parent and in total.
- The time spent in each procedure called from this one.

This information should help determining which sections of a program need modifications to improve the performance. It may not be sufficient for a number of reasons:

- The time spent in a procedure may depend largely on the arguments provided or even the value of global variables.
- The effect of the program structure, and data access pattern, on the cache memory and virtual memory hit rate is not measured.
- The time spent waiting for the disk or network may not be measured.

This is particularly important since the bottleneck is rapidly moving from the processor, to the memory hierarchy and disks, as newer speedy processors are appearing. Fortunately, other tools may help to gather statistics about the cache memory, virtual memory and disk accesses.

4.5 Dynamic Memory Allocation

Most modern programming languages support the dynamic creation and deletion of data structures (also called objects). A chunk of memory is allocated for a new structure when requested. When a structure is no longer needed, the corresponding chunk of memory may be reused for other purposes.

4.5.1 Memory allocation errors

When new objects are repeatedly created but unused ones are not discarded, long running programs eventually run out of memory. This is called a memory leak. Fortunately, this error is relatively easy to detect by monitoring memory usage and printing statistics about allocated and deallocated objects types.

The other type of error is discarding objects still in use, causing memory corruption. For example, a *Color* object may be created and used by two *Window* objects. When discarding one of the *Window* objects, the program may also discard erroneously the referenced *Color* object; when the remaining *Window* object accesses the discarded *Color* object, unpredictable results are obtained.

Indeed, discarded objects may not be reused immediately and may be accessed a number of times without apparent problem. Eventually, the associated memory is reallocated to another object. Subsequent accesses will return incorrect values, which may cause considerable damage if used as pointers, for example.

The only way to prevent such errors is to have the system automatically detect unused objects. This is called garbage collection.

4.5.2 Memory allocation algorithms

Objects may be allocated consecutively in memory. Since each object may have a different size, the size must be attached to the object. Often, when an object of n bytes is requested, n+4 bytes are reserved, the first 4 bytes are used to store the size and the address of the remaining n bytes is returned.

When an object is discarded, the size information is checked. A list of discarded objects is maintained to make them available for reuse. At the next allocation, the list of discarded objects is examined to find an object of the correct size. The first object of the correct or greater size may be selected. The object with the closest matching size (still greater or equal) may alternatively be selected. In either case, allocation takes quite some time and the memory becomes more and more fragmented as discarded objects are split to allocate slightly smaller objects. Checking for contiguous free memory among the discarded objects, in order to regroup them into larger less fragmented objects, is a possibility but adds to the allocation time.

More elaborate algorithms achieve better results. For example, objects may be allocated in a limited number of fixed sizes corresponding to the powers of 2 (8, 16, 32, 64..., 1GB, 2GB, 4GB bytes). Furthermore, these objects may be aligned on addresses that are a multiple of their size and be

created in contiguous pairs. Finding unused objects of the right size is easy as the number of sizes is limited and a pair of smaller objects is obtained by splitting in two an object of the next larger size. Finding contiguous discarded objects to merge also becomes easier since only contiguous pairs of the same size exist.

Even with these more elaborate algorithms, dynamic memory allocation is somewhat expensive. Several programs using dynamically allocated linked lists spend almost half of their processing time in the memory allocation routines. Moreover, at least 4 bytes of information is added to each allocated object and the size is rounded to the next power of two. The resulting size is thus between n+4 and 2(n+4)-1 for an average overhead of (n+4-n + 2n+8-1-n) / 2 = 0.5n + 5.5. In other words, on average, a program requiring 1MB of dynamically allocated objects will actually use 1.5MB of heap memory with this popular algorithm.

4.5.3 Garbage collection

The objective of the garbage collector is to identify and reclaim objects unreachable from the program. Any object which cannot be reached can safely be reclaimed since it cannot be used any further. It is not necessary to discard the objects immediately when they become unreachable; it may be done somewhat later.

Objects may be reached by the program through the global variables (global roots), through the local variables on the stack of each thread (stack roots), or recursively through objects accessible from these global and stack roots.

While garbage collection prevents memory corruption (i.e. reclaiming objects still in use), memory leaks are still possible. Indeed, pointers to unneeded objects should be cleared. Failing that, the object remains reachable and cannot safely be discarded. Fortunately, memory leaks are relatively easy to track down.

The excellent book [**?**] presents a detailed survey of all the important garbage collection algorithms. The algorithms are divided in three main categories: reference counting, mark and sweep, and copying collectors. In each category, different design objectives (simplicity, memory overhead, time overhead, pause length, pause frequency) lead to different algorithms.

Reference counting

Each object may store a count of the number of references to itself. Whenever a reference variable is assigned, the previously referenced object has

its count decremented, and the newly referenced object has its count incremented. If an object has its count decremented to zero, no reference to it remain and it may be deallocated. Before the object is deallocated, any reference stored in the object must be zeroed to release the references to other objects. These objects may in turn get a zero count and be deallocated.

This simple scheme is widely known and used for simple applications. Indeed, it has a number of problems which may not be readily apparent.

- The compiler must generate additional instructions to increment and decrement reference counts, and to check if the reference count reaches zero. Thus, a modified compiler is required, and the additional instructions at each assignment add up to a significant execution overhead.

  The compiler may be able to optimize out a few count increment/decrement. For instance, when a reference is removed from a variable and stored into another, the reference count is unchanged and the redundant decrement-increment pair may be removed.

- In some cases, removing a reference may take a long time, perhaps causing a noticable delay during the execution of an interactive application. Indeed, if the last reference to a long linked list is removed, each link in the chain will get freed in cascade. To prevent this, defered cleanup is often used. The list is set aside when its reference count goes to zero. Then, a few of the objects set aside have their references cleared and are deallocated. The cleared references may cause other objects to be unreferenced and set aside but they may be processed later.

- Each object must contain a reference count and information about where embedded references are stored. The object size and location of the embedded references is typically stored along with the method table. The reference count may add 4 bytes to the memory allocation overhead.

- Cyclic references prevent reference counts of unreachable objects to reach zero. Indeed, a circular linked list may be unreachable while each link references one other link and is referenced by one link. Various schemes have been proposed to detect and discard these cycles. One scheme is to periodically perform a mark and sweep garbage collection, to reclaim the unreachable cycles which could not be handled by reference counting.

Reference counting is most useful for objects without embedded references. It has been used in some C++ graphics libraries for *resources*.

These *resources* did not contain embedded references and their reference count was automatically updated by overloading the assignment operator. Reference counting is also used in simple scripting languages.

Mark and sweep

In this scheme, a certain amount of memory is initially allocated to the heap. Each object allocation takes up some of the available memory until the heap fills up. Then, when the object allocation cannot proceed, the garbage collector is activated. If enough space is reclaimed, the object allocation proceeds. Otherwise, the heap is enlarged. If the available virtual memory is exhausted and the heap cannot be expanded, the allocation fails.

The garbage collection proceeds in two steps: mark and sweep. Each allocated object has a mark bit initially false. In the mark phase, all the objects accessible from the roots are visited. If a visited object is not marked (mark bit false), it is marked and the objects referenced from it are recursively visited. At the end of the mark phase, all reachable objects have been marked. The complexity of the mark phase is proportional to the number of reachable objects plus the number of objects they reference.

The sweep phase checks each object in the heap, reclaims unmarked objects, and resets the mark bit on the others in preparation for the next garbage collection. The complexity of this phase is proportional to the number of objects in the heap.

The mark and sweep algorithm is relatively simple and does not require compiler assistance besides providing information about the location and type of roots within the program and the stack, and of references within each object. At runtime, the allocation routines are replaced with a version performing the mark and sweep, once the heap is filled. It is much more efficient than reference counting, since it does not require code at each reference assignment, and does handle correctly circular references.

Mark and sweep retains the overhead and memory fragmentation problems of the underlying allocation algorithms which manage the objects on the heap. Furthermore, the time required to perform a complete mark and sweep may cause severe distractions to interactive applications; scanning the complete heap may require several seconds during which the application is unresponsive to user input. The sweep phase may be delayed and performed in smaller chunks, if objects allocated after the sweep may be differentiated from unreachable objects through a special mark bit.

Conservative mark and sweep

In some cases, it is not possible to determine where are the references in each object, or the type of the stack roots. Indeed, C compilers do not produce any runtime information about the location of fields in objects or even about the type of an object pointed to. Conservative mark and sweep garbage collectors can handle these situations. Each root is presumed containing a heap reference if the address that it would contain is within the heap address range. Similarly, each field in referenced objects is presumed to contain references to other objects, as the objects size but not their fields types are known.

The result is that any reachable object will be marked, but some unreachable objects may be marked as well. Indeed, if a global variable, local variable, or reachable object field contains a simple integer which corresponds to the address of an unreachable object, it will be falsely interpreted as a reference and the object will be presumed reachable. Fortunately, the heap address range usually does not contain address 0, a fairly common value for non reference fields.

Copying collectors

In Copying garbage collection, a *FromSpace* is initially empty and is filled sequentially as objects are allocated. Allocating a new object simply involves updating the pointer to the last address used within *FromSpace*; no need to scan a free list and to split or merge free objects.

Once the *FromSpace* is filled, a similarly sized area, the *ToSpace* is allocated. Then, the objects in *FromSpace* are scanned, starting from the roots and recursively following referenced objects. When an object is first visited, it gets copied to the *ToSpace*, it is marked, and the address of the copy in *ToSpace* is stored using the space of now copied fields. The address of the copy is then returned, such that the root or object field from which the visited object was referenced can be updated to reference the copy. When an already visited node is scanned, the address of the copy is fetched and returned; indeed, even if an object is referenced several times, it should be copied only once.

Once the scanning is performed, all the reachable objects in *FromSpace* have been copied to *ToSpace*, and the references in roots and objects fields have been updated. The unreachable objects are never visited nor copied and remain in *FromSpace*. The *ToSpace* then becomes the *FromScape* and should contain some free space. The old *FromSpace* can then be recycled into the *ToSpace* for the next garbage collection.

Copying garbage collectors run in time proportional to the number of

reachable objects. This is particularly efficient in environments where most objects are freed soon after being allocated. The overhead of copying objects is offset by the much simplified allocation and the elimination of memory fragmentation. The problem of long garbage collection pauses for interactive or real-time applications remains.

When the type of stack roots cannot be determined, a conservative approach is required. Indeed, if the value found in a stack root is a reference, the object is reachable and the reference would need to be updated if the object is copied. However, if the value in a stack root is simply an integer which happens to be within the heap address range, this integer value should not be updated. These constraints are met by not moving nor reclaiming any object which may be referenced from a stack root of unknown type; this cannot be easily accomodated in the *FromSpace* and *ToSpace* approach.

Advanced garbage collection algorithms

An ideal garbage collector should support ambiguous roots (i.e. stack roots of unknown type), be efficient, have a low memory overhead, and perform the collection in a number of short non-disruptive steps. The Appel-Ellis-Li garbage collector, implemented in SRC Modula-3, is an interesting compromise between these conflicting goals. It is a concurrent, incremental, generational, conservative mostly-copying collector which uses virtual memory protection as read/write barriers on heap pages.

A concurrent collector handles correctly multiple threads of execution and suspends threads as needed at critical steps. Incremental collection divides the work in small chunks to reduce the longest time during which the program is unresponsive. A generational collector divides the objects into newly created objects, those which survived one collection, two collections... Since most objects are used either for a very short time or for a relatively long time, it makes sense to perform more frequent collections among recently created objects; the collection time is proportional to the number of reachable objects, which should be low among newly created objects.

The heap is split in pages of equal size which may be protected against read or write. Objects as large as a page or larger occupy a set of contiguous pages not shared with any other object and thus never need to be copied. Several small objects may share a page but do not cross page boundaries. A list of free pages is maintained from which sets of one or more contiguous pages may be allocated using a best-fit algorithm.

The pages in use are divided into several lists (typically three) based on their generation (i.e. number of garbage collections they survived). Objects

are allocated linearly in pages from the free list added to the first generation until the first generation list reaches a certain size. Then, all program roots are scanned and the first generation objects reached are copied into second generation pages. Pages storing large objects are simply promoted to the next generation without copy. The roots are updated accordingly but the references within the copied objects are not scanned yet. Instead, these pages are protected against read since their references have not been scanned and updated.

Whenever one of the protected pages is accessed, the protection violation generates an interrupt. The interrupt routine disables all the other threads, unprotects the page, scans and copies the referenced objects, and updates the references. Then, the garbage collection for this page is through, and the threads may be resumed and now have full access to the page. Thus, the garbage collection proceeds one page at a time as objects on uncollected pages are accessed.

When the second generation fills up, it is garbage collected in a similar way and objects are copied to pages in the third generation. When the last generation fills up (3 or 4 generations is typical), objects are copied into a new list which then replaces the last generation; this is similar to the *ToSpace* becoming the new *FromSpace* in the simple copying collector.

Objects that may be reachable from the ambiguous stack roots cannot be moved. Thus, a page containing such an object is promoted to the next generation. It may become fragmented because of the holes created by unreachable objects which remain between unmovable objects.

After a collection, the first generation is empty and there can be no reference from an older generation to objects in the first generation. As new objects are created, the first generation fills up and references to these new objects may be stored into older generation objects. Therefore, scanning the roots for references to first generation objects is not sufficient. To determine reachable objects within the first generation, either the references stored into older generations must be remembered, or all the reachable objects within older generations must be scanned (but need not be copied).

The solution adopted is to write protect all the pages in older generations. When one of these pages is modified, the protection violation interrupt function remembers the modified page and removes the protection. When collecting a younger generation, all the modified pages are added as roots. The objects on the modified pages are scanned for references to first generation objects.

Objects without any embedded references (pure objects) are often copied into pages containing only such pure objects. In that case, no reference updating is required and the page need not be read protected. In programs with several threads, the root scanning phase may be unduly long. It is

possible to see the stack as unprocessed pages; the stack pages may be read protected and are processed as they are accessed. This is only possible however if the protection violation handler uses a different unprotected stack.

This garbage collection scheme combines very efficient allocation (sequential allocation in free pages), efficient collection (large objects are never copied and younger generations are collected more often), and relatively short pauses. The longest pause is when scanning the roots at the beginning of a collection; subsequent pauses collect one page at a time. These shorter pauses come at the cost of using the operating system memory protection mechanisms. Since these mechanisms were initially offered to detect fatal conditions, their frequent use was not anticipated and their implementation is relatively slow on some systems.

Discussion

Garbage collection is mandated to prevent memory corruption errors. It also simplifies the decoupling between libraries and applications; it avoids the debate about which of the calling program or the library allocates and discards the objects used to return results. It also relieves the programmer of memory management tasks such as keeping track of how many Window objects reference a Color object to know when the Color object may be discarded.

The price to pay for this may be increased memory consumption, although traditional dynamic memory allocation routines have a similar overhead. More important is the garbage collecting time, which is usually larger than the traditional dynamic allocation time. While this overhead in the past was measured around 10 to 20%, recent implementations claim an almost negligible overhead, possibly below 5%. Nevertheless, this time overhead may be concentrated and disrupt the execution. Incremental collectors are in most cases adequate for interactive applications but cannot be used for hard real time applications.

4.6 Multi-threading

Multiple asynchronous processes or threads may be used for several purposes. On a multi-processor, they split the computation such that more than one processor may work in parallel. Even on a single processor, time sharing insures that when a task takes a long time to complete, other tasks may still proceed. Finally, the program structure may benefit from such a decomposition. Different processes may work in pipeline, each performing a

small part of the work and sending its results to the next stage. Several processes may each be responsible for reading events from a single source, when asynchronous events from several sources are expected, instead of having an event loop which polls each possible source repeatedly and dispatches the events to different routines.

Such asynchronous processes may reside on multiple networked computers and communicate through messages. However, sending messages between processes, especially over the network, incurs a relatively large overhead. Multiple processes or threads on a single computer can communicate through shared memory. The message passing overhead is avoided but proper synchronization through locking must be insured. Indeed, each data structure accessible from more than one process must have an associated lock. Before accessing such a structure, a process must acquire the associated lock and only release it after the access, once the data structure is back to a consistent state.

Processes may share a portion of or all their address space, with the exception of the execution stack, and may share other resources such as file pointers. The more is shared, the lower is the overhead associated with creating a new process. In particular, lightweight processes where everything is shared, except the execution stack, are often called threads within a process. The process is the address space, and associated resources such as file pointers, while each thread is a separate execution stack.

It has been claimed that threads belong to the operating system and should not be a programming language issue. In practice, however, relatively few operating systems offer threads and they generally have different application programming interfaces. Furthermore, most available libraries are not thread safe, in the sense that structures that would be accessed by multiple threads are not protected by locks.

Therefore, there are significant advantages to have threads defined in the standard libraries associated with a language. Portability is gained and it encourages library developers to insure that their code is thread safe. Few languages, apart from Modula-3, Ada and Java, support threads in this manner.

4.6.1 Synchronization primitives

In multi-threaded applications, each piece of data accessed asynchronously from more than one thread should be protected by a lock. There are several programming constructs which have been proposed to achieve this. The mutual exclusion (mutex) and condition variables are conceptually simple and are used in POSIX threads libraries, Modula-3 and Java.

Before accessing shared data, a thread should acquire the associated

mutex. If another thread attempts to acquire the same mutex, it will block waiting for the first thread to release the mutex. Condition variables complement mutexes by allowing notification among threads. One thread may wait for a certain condition which will later be signaled by another thread.

In Modula-3, Thread.Mutex and Thread.Condition are objects which may be inherited from, or may be created and stored in variables and objects. The `LOCK mutex DO ... END` statement is used for protecting the access to shared data. The procedures Thread.Wait, Thread.Signal, and Thread.Broadcast are used to wait on a condition and signal that the condition has arrived.

In Java, mutex and condition variables are not directly accessible. Each class and each object with at least one *synchronized* method contains an associated mutex and condition variable. A complete method or a block of statements may be labeled *synchronized* { ... }. When entering a synchronized block, the mutex associated with the current object is locked. Similarly, the wait, notify and notifyAll methods inherited from class *Object* are used to wait and notify the condition associated with an object. The mutex in Java are reentrant. A thread enters immediately a *synchronized* block if it already holds the associated mutex.

4.6.2 Synchronization errors

What if shared data is not suitably protected by mutexes. When some shared data is not protected, one thread may be modifying the data while another thread reads it while it is being modified and possibly in an inconsistent state. Such races between two threads accessing unprotected data are difficult to detect as they may be intermittent with a low frequency of occurence. Often races remain undetected in tests and start showing up when used intensively for long periods of time or under heavy load. Furthermore, debugging code added to detect the cause of the problem may alter the timing and make it happen less frequently or under different conditions. For these reasons, it is important to devise and follow a simple and robust locking protocol for protecting all the shared data.

The other possible error is to over protect shared data, making it unaccessible to another thread, or causing a deadlock. Deadlocks happen when threads acquire several locks. For instance, thread 1 may acquire mutex A and wait for mutex B, while thread 2 may acquire mutex B and then wait for mutex A. The symptoms of a deadlock is that some threads become irremediably blocked and thus some work does not get done. When all threads are involved in a deadlock, the runtime system may detect it and print an error message.

Deadlocks are therefore relatively easy to detect. To avoid deadlocks, a proper locking order must be defined in the locking protocol. If all threads always follow the same order when acquiring multiple mutexes, cyclic dependencies between threads waiting on mutexes cannot happen and deadlocks are avoided.

4.6.3 Implementation

A growing number of operating systems offer thread support. Then, threads are managed much like processes except that they share their address space and file pointers. The overhead associated with operating system level threads varies greatly from one system to another but may be significant.

On operating systems where threads are not supported or incur a high overhead, user level threads may be available. User level threads imply several mechanisms which have to be implemented in the runtime thread library.

- At regular time intervals, the program execution must be interrupted to let the scheduler procedure decide if it is time to switch to another thread. This may be accomplished with timers (e.g. setitimer on POSIX) which generate an interrupt at specified intervals (e.g. 0.01s).

- Multiple stacks must be created, one for each thread, and the scheduler must be capable to transfer the control from one to another. Each stack is simply a chunk of memory which may be obtained by dynamic memory allocation. Transfering the control from one thread to another involves changing the context (program counter and stack pointer) by saving the context of the current thread and switching to the previously saved context of another thread. This is easily achieved on POSIX through setjmp (to record the current context) and longjmp (to transfer to a previously recorded context).

- Stack overflows must be detected. This may be achieved by checking if the stack pointer is within bounds at each thread switch, or at each procedure activation. A more reliable and efficient mechanism is to reserve a read/write protected guard page at the end of each stack.

- Non blocking system calls must be used. Indeed, the user level timers used to switch between threads are inactive when the process is blocked waiting for a system call. Thus, when a thread performs a blocking read system call, the whole process would block even though other threads could continue. The solution is to use non blocking

system calls. Whenever a thread attempts a read, the library function issues a non blocking asynchronous read system call and sets the thread state as waiting for a system call. Each time the scheduler is awakened, the status of pending system calls is verified to see if they completed and the associated thread is runnable. When all threads are waiting for system calls, the scheduler could wait until the next timer event, or wait until any of the system calls finish. On POSIX, it is possible to wait for several file descriptors expecting asynchronous input/output completion using *select*.

4.6.4 Discussion

Threads are an excellent vehicle to effectively use shared memory multiprocessors, and may greatly simplify the structure of distributed and interactive applications. In return the require a strong discipline from the programmers when defining and using locking protocols to protect shared data. This is becoming less and less an issue but on some systems many libraries and programming tools may not support threads.

4.7 Embedded Languages

Languages are sometimes categorized as compiled or interpreted. Of course C interpreters and SmallTalk compilers have been written but in most cases there are important differences between languages designed to be compiled or to be interpreted. Compiled languages lead to more efficient executables and strong static type checking while interpreted languages are more flexible and offer a richer runtime environment.

In particular, an interpreted language often lets the user add new code at runtime. This new code might be used for user specified behavior (.emacs initialization/customization file in the emacs editor) or for testing and prototyping. For example, mathematical operations may be called by the user to realize more complex operations, like in spreadsheet packages.

Several interpreted languages, such as SmallTalk and Self, thus include an interpreter/compiler at runtime. More traditional compiled languages such as Ada95 or Modula-3 could conceivably offer a runtime library supporting on the fly compilation and dynamic linking. However, the semantics of adding/replacing code dynamically, (including changing type declarations), are often not well defined. Moreover, the extensive type checking performed by these compilers would make dynamic compilation a fairly high overhead process.

A good compromise is to have a compiled language with an embedded scripting language. This scripting language is a simple interpreted language which can call the compiled procedures (for prototyping) and be called from the compiled procedures (for initialization/customization). A well known example is the emacs editor, written in C and with an emacs LISP interpreter embedded. A common problem with this approach is that the scripting language is often very different from the compiled language, which makes it more difficult to learn. There are also several cases where the scripting language is unfit for program development and is hard to use for anything but fairly trivial procedures (lack of expressive power, dynamic scoping, no type checking, no procedures...). The Bourne shell, C shell, Perl and Postscript scripting languages exhibit these deficiencies at various levels and are often qualified of write-only (i.e. the code is unreadable and unmaintainable).

4.8 Run Time Analysis of Modula-3 Programs

In this section, a number of run time analysis tools are applied to the `QueryDbase` program presented earlier.

4.8.1 Run time checks

Although the QueryDbase program performs little error checking, most problems will be detected immediately by the run time checks (NIL pointers, incorrect array indexing...). The program then stops and the problem location may be identified quickly using the debugger.

For instance, trying to fetch an entry without first opening a database will cause an access through a NIL database object.

```
cassis>LINUXELF/QueryDbase
Enter command
fetch key10


***
*** runtime error:
***    attempt to dereference NIL
***

IOT trap
cassis>
```

4.8.2 Coverage Analysis

Modula-3 programs may be compiled to extract line frequency execution data. This information may then be used to determine which portions were tested adequately or where most of the execution time is spent. The proper option has to be specified in the m3makefile and the program recompiled. The program is then run and creates a file named coverage.out with the desired information.

```
cassis>head -5 src/m3makefile
m3_option("-Z")

% database is located in our private package collection
override("database","../..")
import("database")
cassis>rm -R LINUXELF
cassis>m3build
...
cassis>QueryDbase <example2.in >/dev/null
cassis>ls
LINUXELF/    coverage.out    example2.in    src/
```

This coverage information is then displayed using `analyze_coverage`.

```
cassis>analyze_coverage -S src -L
*************** COVERAGE OF Query.m3
...
       PROCEDURE PrintDbase(db: Dbase.T) =
     1    VAR
            key, tmp: TEXT;
            cursor: CARDINAL := 0;
          BEGIN
     1      key := Dbase.FirstKey(db);
     1      WHILE key # NIL DO
   100        IF cursor > LAST(keyArray^) THEN
     4          oldKeyArray := keyArray;
     4          keyArray := NEW(REF ARRAY OF TEXT,2 * NUMBER(oldKeyArray^));
     4          SUBARRAY(keyArray^,0,NUMBER(oldKeyArray^)) := oldKeyArray^;
              END;
   100        keyArray[cursor] := key;
   100        INC(cursor);
   100        key := Dbase.NextKey(db);
            END;
```

```
           (* Use an inefficient sorting algorithm *)

     1      FOR i := cursor - 1 TO 0 BY -1 DO
   100        FOR j := 0 TO i - 1 DO
  4950          IF Text.Compare(keyArray[j], keyArray[i]) = 1 THEN
   890            tmp := keyArray[i];
   890            keyArray[i] := keyArray[j];
   890            keyArray[j] := tmp;
               END;
             END;
           END;

           (* Print all the keys *)

     1      FOR i := 0 TO cursor - 1 DO
   100        Wr.PutText(Stdio.stdout,keyArray[i] & "\n");
           END;
         END PrintDbase;

...
     2    LOOP
   105      Wr.PutText(Stdio.stdout,"Enter command\n");
   105      Wr.Flush(Stdio.stdout);

   105      Lex.Skip(Stdio.stdin);
   105      command := Lex.Scan(Stdio.stdin);

   105      IF Text.Equal(command,"exit") THEN EXIT;

           ELSIF Text.Equal(command,"create") THEN
     0        Lex.Skip(Stdio.stdin);
     0        name := Lex.Scan(Stdio.stdin);
     0        db := Dbase.Create(name);

           ELSIF Text.Equal(command,"open") THEN
     1        Lex.Skip(Stdio.stdin);
     1        name := Lex.Scan(Stdio.stdin);
     1        db := Dbase.Open(name);

           ELSIF Text.Equal(command,"close") THEN
     1        Dbase.Close(db);
```

```
        ELSIF Text.Equal(command,"fetch") THEN
  0        Lex.Skip(Stdio.stdin);
  0        key := Lex.Scan(Stdio.stdin);
  0        Wr.PutText(Stdio.stdout,Dbase.Fetch(db,key) & "\n");

        ELSIF Text.Equal(command,"store") THEN
100        Lex.Skip(Stdio.stdin);
100        key := Lex.Scan(Stdio.stdin);
100        Lex.Skip(Stdio.stdin);
100        content := Lex.Scan(Stdio.stdin);
100        Dbase.Store(db,key,content);

        ELSIF Text.Equal(command,"delete") THEN
  0        Lex.Skip(Stdio.stdin);
  0        key := Lex.Scan(Stdio.stdin);
  0        Dbase.Delete(db,key);

        ELSIF Text.Equal(command,"print") THEN
  1        PrintDbase(db);

        ELSE
  0        Wr.PutText(Stdio.stdout,"Incorrect command\n");
        END;
      END;
    END Query.
```

This data reveals that some portions (fetching and deleting entries) were not tested (executed frequency is 0) and that most of the time is spent in the inefficient sorting performed in the PrintDbase procedure.

4.8.3 Profiling

The usual profiling tools, for instance gprof, may also be used for finding where the execution time is spent in the call tree of a large program. These tools produce more complete information when all the modules and libraries involved have been compiled with the profiling flag. This may be achieved with the correct `m3_option` entry in the m3makefile or with a special m3build template which builds the program in a different directory. A special template is interesting because this way the profiled and ordinary versions of programs and libraries may coexist in separate directories.

The procedure is similar to coverage analysis. The program is recompiled and then run creating a file named gmon.out. This data is then processed and displayed using `gprof`. The explanations have been annotated directly on the output listing portions which follow.

```
cassis>m3build -b LINUXELF-PROF
...
cassis>LINUXELF-PROF/QueryDbase <example2.in >/dev/null
cassis>gprof LINUXELF-PROF/QueryDbase
...
granularity: each sample hit covers 2 byte(s) for 0.09% of 11.49 seconds

                                  called/total       parents
index  %time    self descendents  called+self    name           index
                                  called/total       children


...

# Each module initialization code is called from RunMainBodies.
# __INITM_Query is the body of the main module (Query.m3).

[4]     97.0    0.00       11.14       1         _RTLinker__RunMainBodies
                0.01       11.13       1/1           __INITM_Query [5]
                0.00        0.00       1/1           __INITM_ProcessPosix [112]
                0.00        0.00       1/1           __INITM_Stdio [123]
                0.00        0.00       1/1           __INITM_TimePosix [129]
...

-----------------------------------------------

# __INITM_Query is called by RunMainBodies and calls a number
# of procedures. On a total execution time of 11.13s, PrintDbase
# uses 7.34s.
                0.01       11.13       1/1           _RTLinker__RunMainBodies
[5]     97.0    0.01       11.13       1         __INITM_Query [5]
                1.14        7.34       1/1           _Query__PrintDbase [6]
                0.06        0.97    1004/1004        _Wr__Flush [9]
                0.03        0.71    3005/3005        _Lex__Scan [13]
                0.02        0.63    1000/1000        _Dbase__Store [15]
                0.01        0.13    3005/3005        _Lex__Skip [24]
                0.01        0.04    1004/2004        _Wr__PutText [31]
                0.04        0.00    6015/6015        _Text__Equal [48]
```

```
                0.00        0.00       1/1           _Dbase__Create [119]
                0.00        0.00       1/5044        _RTHooks__AllocateOpenArray
                0.00        0.00       1/1           _Dbase__Close [584]

-----------------------------------------------

# The bulk of the time spent in PrintDbase is consumed by Text__Compare.
# Speeding up Text__Compare would help but in this case changing
# the sorting algorithm to reduce the number of Text__Compare
# would have a much greater impact.

                1.14        7.34       1/1           __INITM_Query [5]
[6]     73.8    1.14        7.34       1          _Query__PrintDbase [6]
                7.06        0.00  499500/499500      _Text__Compare [7]
                0.00        0.22    1000/1000        _Dbase__NextKey [22]
                0.00        0.04    1000/2004        _Wr__PutText [31]
                0.00        0.02    1000/4005        _RTHooks__Concat [33]
                0.00        0.00       7/5044        _RTHooks__AllocateOpenArray
                0.00        0.00       1/1           _Dbase__FirstKey [130]
                0.00        0.00       7/7016        _memmove [61]

-----------------------------------------------

# Text__Compare consumes the execution time itself, it does not
# call other procedures.

                7.06        0.00  499500/499500      _Query__PrintDbase [6]
[7]     61.4    7.06        0.00  499500          _Text__Compare [7]

-----------------------------------------------

# _syscall is called from a number of procedures, _write and
# _fcntl being the most frequent. It does not however consume
# too much execution time.

                0.00        0.00       2/5993        _gettimeofday [128]
                0.00        0.00       3/5993        _open [122]
                0.00        0.00       5/5993        _fstat [118]
                0.00        0.00      16/5993        _getrusage [101]
                0.19        0.00     863/5993        _read [23]
                0.45        0.00    2089/5993        _write [17]
                0.65        0.00    3015/5993        _fcntl [14]
```

```
[8]     11.3    1.30        0.00    5993         _syscall [8]

...

# Summary of the time spent per procedure.
#

   %  cumulative    self              self    total
 time   seconds   seconds    calls  ms/call  ms/call name
 55.2       7.06     7.06   499500     0.01     0.01  _Text__Compare [7]
 10.2       8.36     1.30     5993     0.22     0.22  _syscall [8]
 10.2       9.66     1.30                             mcount (425)
  8.9      10.80     1.14        1  1140.01  8481.51  _Query__PrintDbase [6]
  2.1      11.07     0.27    29821     0.01     0.01  _Rd__GetChar [18]
  1.0      11.20     0.13     1000     0.13     0.14  _dbm_do_nextkey [26]
  0.8      11.30     0.10    38843     0.00     0.00  _RTHooks__LockMutex [30]
...
```

The profiling information is useful to see which procedure, called from where, consumes most of the execution time. These measures are nevertheless somewhat imprecise since a statistical sampling of the stack is used to evaluate the CPU usage. Furthermore, the profiling information does not tell much about the influence of a number of important factors such as cache memory misses, virtual memory page faults and input-output delays.

4.8.4 Debugging

Although any debugger may be used to trace the execution through a Modula-3 program, it is preferable to have a Modula-3 aware debugger. This way, the Modula-3 syntax for specifying modules, procedures, variables, types and references may be used.

The debugger is used to understand the behavior of a program, thus finding errors, by tracing the execution and the content of variables. It may also be used to call data gathering procedures at certain points in the execution. One application of this is to study the memory allocation.

A session under the Modula-3 aware debugger `m3gdb` is presented and annotated below.

```
cassis>m3gdb LINUXELF/QueryDbase
GDB is free software and you are welcome to distribute copies of it
 under certain conditions; type "show copying" to see the conditions.
There is absolutely no warranty for GDB; type "show warranty" for details.
```

```
GDB 940920 (EPC-M2 & SRC-M3) (i486-unknown-linux),
Copyright 1994 Free Software Foundation, Inc...

# The path to locate the source code is specified.
# An initial break point is specified to examine the PrintDbase procedure

(gdb) dir src
(gdb) dir ../database/src
(gdb) list Query.m3:0
1       MODULE Query EXPORTS Main;
2
3       (* This program accepts commands to create or open ndbm databases
4          and to store, fetch or delete elements. A sorted list of keys
(gdb) break Query.PrintDbase
Breakpoint 1 at 0xddf: file Query.m3, line 22.

# The @M3novm option is used to avoid using memory protection
# for the garbage collector. It is slightly slower but
# simpler to debug

(gdb) run @M3novm <example2.in >/dev/null
Starting program: LINUXELF/QueryDbase @M3novm <example2.in >/dev/null

Breakpoint 1, Query.PrintDbase (db=16_e50f4) at Query.m3:22
22          key, tmp: TEXT;
Current language:  auto; currently m3
(gdb) where
#0  Query.PrintDbase (db=16_e50f4) at Query.m3:22
#1  16_1945 in Query.m3.MAIN (phase=3) at Query.m3:111
#2  16_6799 in RTLinker.RunInitPhase (phase=3) at RTLinker.m3:25
#3  16_6969 in RTLinker.m3.MAIN (phase=3) at RTLinker.m3:67
#4  16_d85 in main (argc=1, argv=16_bffff960, envp=16_bffff968)
    at _m3main.c:2305

# RTutils.Heap prints the number of each object type currently allocated.
# Pathname.T which is the same as TEXT (because of structural equivalence)
# is the most common type here.

(gdb) set lang c
(gdb) print RTutils__Heap()
Code   Count   TotalSize  AvgSize  Name
---- --------- --------- --------- -------------------------
```

```
   1      2235     38736        17 Pathname.T
   8         2        24        12 RegularFile.T
  10         1        12        12 Terminal.T
  18         2        32        16 RTutils.Visitor
  26         1        16        16 RefSeq.T
  41         1        28        28 Atom.NewAtomTbl
  50         3        36        12 Thread.Mutex
  54         3       120        40 FileWr.T
  60         1        40        40 FileRd.T
  61         1       148       148 Thread.T
  62         1         8         8 Thread.Condition
...
 145         1        40        40 TextSeq.RefArray
     --------- ---------
          2276     60372
$1 = void

# The PrintDbase function is run to completion and the
# Heap usage examined again.

(gdb) finish
Run till exit from #0  Query.PrintDbase () at Query.m3:22
0x1945 in Query.m3.MAIN () at Query.m3:111
111           PrintDbase(db);
(gdb) print RTutils__Heap()
Code   Count   TotalSize  AvgSize  Name
---- --------- --------- --------- --------------------------
   1      1929     31484        16 Pathname.T
   8         2        24        12 RegularFile.T
  10         1        12        12 Terminal.T
  18         3        48        16 RTutils.Visitor
  26         1        16        16 RefSeq.T
  41         1        28        28 Atom.NewAtomTbl
  50         3        36        12 Thread.Mutex
  54         3       120        40 FileWr.T
  60         1        40        40 FileRd.T
  61         1       148       148 Thread.T
  62         1         8         8 Thread.Condition
 145         8      8224      1028 TextSeq.RefArray
     --------- ---------
          1980     63716
```

```
# RTHeapStats.ReportReachable is called to determine the number
# of active, reachable, objects (i.e. those which would not be
# garbage collected). Of the 1980 objects on the heap, only slightly
# more than half are still alive.

(gdb) print RTHeapStats__ReportReachable()

HEAP: 0xde000 .. 0x120000 => 0.2 Mbytes

Module globals:
 # objects   # bytes  unit
 ---------  --------  ------------------
      1007     26260  Query.m3
        11     16656  Stdio.i3
         2      2012  RTutils.m3
         3       440  RTHeapPosix.m3
...

Global variable roots:
 # objects   # bytes         ref type                location
 ---------  --------  ---------- -----------------   ------------------------
      1001     24108  0x11e004 TextSeq.RefArray    Query.m3 + 32
       513     12300  0xe2004 TextSeq.RefArray    Query.m3 + 36
         3      4168  0xdea10 FileRd.T            Stdio.i3 + 28
         3      4168  0xdfa48 FileWr.T            Stdio.i3 + 32
...
         1        24  0xf2aec Pathname.T          Query.m3 + 56
         1        20  0xf2b04 Pathname.T          Query.m3 + 44
         1        20  0xe50e0 Pathname.T          Query.m3 + 48
         1        20  0xf2ad8 Pathname.T          Query.m3 + 52
         1        16  0xde09c Thread.Mutex        RTHeapPosix.m3 + 464
...

# KeyArray and oldKeyArray point to 1001 and 513 objects respectively
#
# The program is rerun. PrintDbase is executed one line at a time

(gdb) run
Starting program: LINUXELF/QueryDbase @M3novm <example2.in >/dev/null

Breakpoint 1, Query.PrintDbase () at Query.m3:22
22          key, tmp: TEXT;
```

```
(gdb) set lang m3
(gdb) n
23          cursor: CARDINAL := 0;
(gdb) n
25          key := Dbase.FirstKey(db);
(gdb) n
26          WHILE key # NIL DO
(gdb) n
27            IF cursor > LAST(keyArray^) THEN
(gdb) n
32            keyArray[cursor] := key;
(gdb) n
33            INC(cursor);

# The type and content of variables may be displayed.

(gdb) print key
$5 =  "key515"
(gdb) ptype key
type = BRANDED "Text 1.0" REF Convert.Buffer
(gdb) ptype key^
type = ARRAY OF CHAR
(gdb)

# This text object (key) is noted to check if it is free later on.

(gdb) print RTHeapDebug.Free(key)
$6 = 0
(gdb) finish
Run till exit from #0  Query.PrintDbase (db=16_e50f4) at Query.m3:33
16_1945 in Query.m3.MAIN (phase=3) at Query.m3:111
111           PrintDbase(db);
(gdb) print RTHeapDebug__CheckHeap()
Path to 'free' object:
   Ref in root at address 0x48644...
   Object of type TextSeq.RefArray at address 11e34c...
   Free object of type Pathname.T at address 11ea90...
$9 = void
(gdb) set lang m3
(gdb) print keyArray
$10 = 16_11e34c
```

```
# Even though the PrintDbase procedure is finished, keyArray still
# exists and maintains alive the keys printed.
```

In the above example, two allocation problems are identified. For this, the number of expected active objects may be checked against the information produced by RTHeapStats.ReportReachable. Another lead is to assert that a certain object is free and let RTHeapDebug.CheckHeap determine if a path still reaches the object and keeps it from being garbage collected.

A first problem is that the oldKeyArray should be released with `oldKeyArray := NIL;` once it has been used to initialize the new keyArray. A second problem is that the keyArray is not released once PrintDbase() is finished. It may be set to NIL, or if it is to be reused, its content should be set to NIL (keyArray[0] := NIL, keyArray[1]...).

It is interesting to note that keyArray reaches 1001 objects and oldKeyArray 513 even though the Query module as a whole reaches only 1007 objects. The explanation is that the 512 keys stored in oldKeyArray are also stored in keyArray and thus are not distinct objects.

4.8.5 Graphical tools

Three tools are available to graphically display statistics about Modula-3 programs.

- `shownew` displays the number and size of newly allocated objects of each type. This may help pinpoint object types where allocations are too numerous and optimizations such as recycling would help. It does not account for garbage collected or unreachable objects. It is accessed with, for the above example, `LINUXELF/QueryDbase @M3shownew`.

- `showthread` displays the status of each thread (running, stopped, waiting...) along the time axis. It is useful to see how much and when each thread is consuming execution time.

- `showheap` is a more specialized tool to examine the efficiency of the memory allocation and garbage collection. It displays the heap memory pages along with their status (free, allocated...). It is accessed with `LINUXELF/QueryDbase @M3showheap`.

4.8.6 Embedded language

The `QueryDbase` example accepts simple commands from the user. Storing intermediate results for further queries, defining new commands and other

similar enhancements are not supported and would require a significant programming effort to incorporate.

This is where using an existing embeddable interpreter provides a much enhanced functionality with little efforts. An embeddable interpreter for the simple yet powerful Obliq language is available. It is written in Modula-3 and is easy to interface to Modula-3 programs.

The `QueryDbase` program may thus be replaced by an Obliq interpreter where access functions to the `database` package have been added. The new program is named `OblQuery`. Its `m3makefile` and implementation `Main.m3` follow.

```
import("synloc")
import("obliqrt")
import("obliqlibm3")
import("obliq")

override("database","../..")
import("database")

implementation("Main")

Program("OblQuery")
```

```
MODULE Main;
IMPORT ObliqOnline;

IMPORT ObLib, ObLibM3, ObLibM3Help, Text, SynWr, SynLocation,
    ObCommand, ObValue, Dbase;

(* Obliq objects are defined for the Database objects and commands *)

TYPE
  ValDbase = ObValue.ValAnything BRANDED OBJECT
      value: Dbase.T;
    OVERRIDES Is := IsDbase; Copy := CopyDbase;
    END;

  OpCodes = ARRAY OF ObLib.OpCode;

  QueryCode = {Error, Create, Open, Close, Fetch, Store, Delete, First, Next};

  QueryOpCode = ObLib.OpCode OBJECT
```

```
        code: QueryCode;
      END;

  PackageQuery = ObLib.T OBJECT
      OVERRIDES
        Eval:=EvalQuery;
      END;

(* Support for Help messages *)

CONST
  Greetings =
      "  OblQuery (obliq with database libraries) (say \'help;\' for help)";
  HelpSummary =
      "  db                (the built-in database library)\n";
  HelpText =
      "  create(file: TEXT): Dbase.T\n" &
      "  open(file: TEXT): Dbase.T\n" &
      "  close(db: Dbase.T)\n" &
      "  fetch(db: Dbase.T; key: TEXT): TEXT\n" &
      "  store(db: Dbase.T; key, content: TEXT)\n" &
      "  delete(db: Dbase.T; key: TEXT)\n" &
      "  first(db: Dbase.T): TEXT\n" &
      "  next(db: Dbase.T): TEXT\n";

VAR queryException: ObValue.ValException;

PROCEDURE HelpQuery(self: ObCommand.T; arg: TEXT;
      <*UNUSED*>data: REFANY:=NIL)  =
  BEGIN
    IF Text.Equal(arg, "!") THEN
      SynWr.Text(SynWr.out,HelpSummary);
    ELSIF Text.Equal(arg, "?") THEN
      SynWr.Text(SynWr.out, HelpText);
      SynWr.NewLine(SynWr.out);
    ELSE
      SynWr.Text(SynWr.out, "Command " & self.name & ": bad argument: " & arg);
      SynWr.NewLine(SynWr.out);
    END;
  END HelpQuery;

PROCEDURE IsDbase(self: ValDbase; other: ObValue.ValAnything): BOOLEAN =
```

```
  BEGIN
    TYPECASE other OF ValDbase(oth)=> RETURN self.value = oth.value;
    ELSE RETURN FALSE END;
  END IsDbase;

PROCEDURE CopyDbase(<*UNUSED*>self: ObValue.ValAnything;
    <*UNUSED*>tbl: ObValue.Tbl;
    loc: SynLocation.T): ObValue.ValAnything RAISES {ObValue.Error} =
  BEGIN
    ObValue.RaiseError("Cannot copy db database", loc);
  END CopyDbase;

PROCEDURE NewQueryOC(name: TEXT; arity: INTEGER; code: QueryCode)
      : QueryOpCode =
  BEGIN
    RETURN NEW(QueryOpCode, name:=name, arity:=arity, code:=code);
  END NewQueryOC;

(* Define and register the new commands in the db package *)

PROCEDURE PackageSetup() =
  VAR opCodes: REF OpCodes;
  BEGIN
    opCodes := NEW(REF OpCodes, NUMBER(QueryCode));
    opCodes^ :=
      OpCodes{
        NewQueryOC("failure", -1, QueryCode.Error),
        NewQueryOC("create", 1, QueryCode.Create),
        NewQueryOC("open", 1, QueryCode.Open),
        NewQueryOC("close", 1, QueryCode.Close),
        NewQueryOC("fetch", 2, QueryCode.Fetch),
        NewQueryOC("store", 3, QueryCode.Store),
        NewQueryOC("delete", 2, QueryCode.Delete),
        NewQueryOC("first", 1, QueryCode.First),
        NewQueryOC("next",1, QueryCode.Next)
      };
    ObLib.Register(
      NEW(PackageQuery, name:="db", opCodes:=opCodes));
    queryException := NEW(ObValue.ValException, name:="db_failure");
    ObLib.RegisterHelp("db", HelpQuery);
  END PackageSetup;
```

```
(* For each database command, check the arguments, call the
   relevant procedure and wrap the result in Obliq objects. *)

PROCEDURE EvalQuery(self: PackageQuery; opCode: ObLib.OpCode;
    <*UNUSED*>arity: ObLib.OpArity; READONLY args: ObValue.ArgArray;
    loc: SynLocation.T)
    : ObValue.Val RAISES {ObValue.Error, ObValue.Exception} =
  VAR
    text1, text2: TEXT;
    dbase: Dbase.T;
  BEGIN
    CASE NARROW(opCode, QueryOpCode).code OF
    | QueryCode.Error =>
        RETURN queryException;

    | QueryCode.Create =>
        TYPECASE args[1] OF | ObValue.ValText(node) => text1:=node.text;
        ELSE ObValue.BadArgType(1, "text", self.name, opCode.name, loc); END;
        dbase := Dbase.Create(text1);
        IF dbase = NIL THEN
          ObValue.RaiseException(queryException, opCode.name, loc);
        END;
        RETURN NEW(ValDbase, what:="<a database>", picklable:=FALSE,
            value:=dbase);

    | QueryCode.Open =>
        TYPECASE args[1] OF | ObValue.ValText(node) => text1:=node.text;
        ELSE ObValue.BadArgType(1, "text", self.name, opCode.name, loc); END;
        dbase := Dbase.Open(text1);
        IF dbase = NIL THEN
          ObValue.RaiseException(queryException, opCode.name, loc);
        END;
        RETURN NEW(ValDbase, what:="<a database>", picklable:=FALSE,
            value:=dbase);

    | QueryCode.Close =>
        TYPECASE args[1] OF | ValDbase(node) => dbase:=node.value;
        ELSE ObValue.BadArgType(1, "database", self.name, opCode.name, loc);
        END;
        Dbase.Close(dbase);
        RETURN ObValue.valOk;
```

```
| QueryCode.Fetch =>
    TYPECASE args[1] OF | ValDbase(node) => dbase:=node.value;
    ELSE ObValue.BadArgType(1, "database", self.name, opCode.name, loc);
    END;
    TYPECASE args[2] OF | ObValue.ValText(node) => text1:=node.text;
    ELSE ObValue.BadArgType(2, "text", self.name, opCode.name, loc); END;
    text2 := Dbase.Fetch(dbase,text1);
    IF text2 = NIL THEN
      ObValue.RaiseException(queryException, opCode.name, loc);
    END;
    RETURN NEW(ObValue.ValText, text:=text2);

| QueryCode.Store =>
    TYPECASE args[1] OF | ValDbase(node) => dbase:=node.value;
    ELSE ObValue.BadArgType(1, "database", self.name, opCode.name, loc);
    END;
    TYPECASE args[2] OF | ObValue.ValText(node) => text1:=node.text;
    ELSE ObValue.BadArgType(2, "text", self.name, opCode.name, loc); END;
    TYPECASE args[3] OF | ObValue.ValText(node) => text2:=node.text;
    ELSE ObValue.BadArgType(3, "text", self.name, opCode.name, loc); END;
    Dbase.Store(dbase,text1,text2);
    RETURN ObValue.valOk;

| QueryCode.Delete =>
    TYPECASE args[1] OF | ValDbase(node) => dbase:=node.value;
    ELSE ObValue.BadArgType(1, "database", self.name, opCode.name, loc);
    END;
    TYPECASE args[2] OF | ObValue.ValText(node) => text1:=node.text;
    ELSE ObValue.BadArgType(2, "text", self.name, opCode.name, loc); END;
    Dbase.Delete(dbase,text1);
    RETURN ObValue.valOk;

| QueryCode.First =>
    TYPECASE args[1] OF | ValDbase(node) => dbase:=node.value;
    ELSE ObValue.BadArgType(1, "database", self.name, opCode.name, loc);
    END;
    text1 := Dbase.FirstKey(dbase);
    IF text1 = NIL THEN
      ObValue.RaiseException(queryException, opCode.name, loc);
    END;
    RETURN NEW(ObValue.ValText, text:=text1);
```

```
    | QueryCode.Next =>
        TYPECASE args[1] OF | ValDbase(node) => dbase:=node.value;
        ELSE ObValue.BadArgType(1, "database", self.name, opCode.name, loc);
        END;
        text1 := Dbase.NextKey(dbase);
        IF text1 = NIL THEN
          ObValue.RaiseException(queryException, opCode.name, loc);
        END;
        RETURN NEW(ObValue.ValText, text:=text1);

      ELSE
        ObValue.BadOp(self.name, opCode.name, loc);
      END;
  END EvalQuery;

BEGIN
  ObliqOnline.Setup();
  ObLibM3.PackageSetup();
  ObLibM3Help.Setup();
  PackageSetup();

  ObliqOnline.Interact(ObliqOnline.New(Greetings));
END Main.
```

The full Obliq language now becomes accessible to interact with the database procedures, and to store and process the results.

```
cassis>LINUXELF/OblQuery

  OblQuery (obliq with database libraries) (say 'help;' for help)

- help db;
  create(file: TEXT): Dbase.T
  open(file: TEXT): Dbase.T
  close(db: Dbase.T)
  fetch(db: Dbase.T; key: TEXT): TEXT
  store(db: Dbase.T; key, content: TEXT)
  delete(db: Dbase.T; key: TEXT)
  first(db: Dbase.T): TEXT
  next(db: Dbase.T): TEXT
- let d = db_create("testdb");
let d = <a database>
```

```
- db_store(d,"key1","content1");
ok
- db_store(d,"key2","content2");
ok
- db_store(d,"key3","content3");
ok
- db_fetch(d,"key2");
"content2"
- db_fetch(d,"bad key");
Uncaught exception (input line 8, char 10)
db_failure (fetch)
Error detected (last input line, char 10)
- db_first(d);
"key1"
- ^D
cassis>
```

Chapter 5

Object Oriented Libraries

Code reuse is often the most effective way to enhance productivity. Libraries are repositories of reusable code and often are one of the most important ingredient in a programming environment. While libraries have existed for a long time, this section examines how they evolved with the advent of newer programming techniques, including object oriented programming.

Reporting cleanly errors, encountered deep in a library, to the calling program has been a challenging problem in library design. With *exceptions*, it is much easier to construct simple to use libraries with good error handling and reporting facilities.

Similarly, *generic* modules help build general, easily reusable, library modules. In the past, libraries used techniques such as macro pre-processing or blind type conversions to achieve similar benefits. These techniques, however, were inconvenient and often inefficient or error-prone.

An increased emphasis on encapsulation, through well defined public interfaces, helps reduce the effort required to understand how to use a library module. Furthermore, it reduces greatly the dependence of the calling programs on the internal composition of library data structures. The resulting libraries have a simpler interface which does not change, even when the internal data structures are modified as newer versions are released.

Objects and inheritance may help reduce the number of procedures required in a library to achive a given functionality. Indeed, some procedures may operate at a high level in the inheritance hierarchy and thus be shared by a large number of child classes.

5.1 Standard Libraries

With each programming environment, a *standard* library is provided. This library implements most of the procedures commonly required and isolates the calling program from the platform dependent system libraries. It usually contains procedures to operate on the primitive types (integer, real, characters, text), support for common data structures (vectors, lists, trees), access to input/output facilities, access to other system services (memory allocation, creating new processes) and access to run time support (garbage collection, performance statistics).

A well defined and standardized language without a corresponding *standard* library will not achieve the portability goals since any useful program must interact with the underlying operating system, at least to perform input/output operations. The difficulty lies in coming up with a complete and easy to use library running efficiently on a large number of platforms. Indeed, when many different platforms are supported, a least common denominator must often be selected.

5.2 Comparative Study of the Modula-3 and Java Standard Libraries

This section examines two well designed object oriented libraries for accessing standard functionality (data structures, input output, operating system services) in a portable way. The Modula-3 standard library *libm3* offers a large number of interfaces and runs on several platforms (AIX386, ALPHA_OSF, AOSF, AP3000, ARM, DS3100, FreeBSD, HP300, HPPA, IBMR2, IBMRT, IRIX5, LINUX, NEXT, OKI, SEQUENT, SOLsun, SPARC, SUN3, SUN386, UMAX, VAX, WIN32). The Java standard packages are implemented on all Java supported platforms (Sun supports Solaris and Win32, and Netscape supports Macintosh, Windows 3.1, Win32, AIX, BSD/386, HP-UX, IRIX, Linux, OSF/1, Solaris and SunOS).

Most of these platforms are variants of Unix with nevertheless some small differences in their system calls. Some of them however differ more significantly; NEXT is based on the Mach operating system and Win32 is available on Windows NT and Windows95 from Microsoft. This way, Modula-3 and Java programs may be compiled and run unmodified on a large number of platforms.

Standard libraries differ in their scope and extent, and in the underlying organization and philosophy brought by its architects. A comparative study is useful in outlining common fundamental characteristics and alternative viable organizations.

The Modula-3 library is rather extensive but follows the same guidelines as the Modula-3 language design, in particular striving for simplicity. For example, a small number of simple and general data structures are available: Sequence, List, Table, and SortedTable. These offer all the functionality found in the dozen or so data structures often available in other libraries such as: stacks, queues, dequeues, priority queues, extensible vectors, sets, bags, dictionaries, lists, hash tables, balanced trees...

The *standard interfaces* section of the library supports TEXT elements, Threads, Integers, Reals and conversions between TEXT and Integers/Reals. The *data structures* section implements sequences (which subsume stacks, queues and extensible vectors), Atoms (an efficient representation for unique TEXT objects), Lists, Symbolic expressions (LISP like textual representations of lists useful for interacting with users), and Tables (which subsume sets, bags, dictionaries, hash tables and balanced trees). This section also implements Bundles, a convenient facility for incorporating files needed at run time (e.g. data files or user interface descriptions) into the executable file.

A section on *input/output* provides extensive facilities to open files or other devices. All the reading procedures operate on Readers and the writing procedures on Writers. A lot of flexibility is gained by the possibility of connecting Readers or Writers to files, other devices or even to TEXT objects.

The *Operating System* section handles the access to the system date and time, the files, the pipes and terminals, the file systems, the process creation facilities, the command line parameters and the environment variables.

The *runtime* section offers control over the garbage collector (weak references for finalization purposes, tuning parameters for performance), run time type information and heap usage statistics.

All the above mentioned interfaces are documented in the corresponding `.i3` files as well as in the technical report *Some Useful Modula-3 Interfaces*. A number of additional interfaces are also available but are not discussed in the report being either too specialized, not platform independent or subject to change. Their documentation must be found directly in the `.i3` files.

The Java standard packages are included in the java.lang, java.util and java.io hierarchies. Each class or interface is in a separate file. In java.lang are found the classes for supporting primitive object types (Object, Boolean, Character, Integer, Float, String, Process, Thread) and miscellaneous related operations (Math functions) and error types. The java.util package offers convenient data structures and container classes such as Enumeration, BitSet, Date, Dictionary, Random numbers and Vector. In package java.io are found input and output streams (for files, pipes, byte arrays, filters...), operations to input/output primitive types

from streams (Boolean, Integer, Long, Float...), a stream tokenizer, and file manipulation facilities.

5.2.1 Runtime support

The Modula-3 statement ISTYPE is used to verify the type of objects at runtime. TYPECODE returns a unique identifier for the type, which may be used with interface RTType to determine super types and other information.

In Java, class Object defines a number of methods inherited by all other classes. Method getClass returns information about the class. The other methods are used to compare objects, obtain a hash code, wait and notify on the condition variable associated with the object, or finalize the object.

In Modula-3, condition variables are allocated independently of objects and are found in the Thread interface. Finalization is also a separate service offered in the WeakRef interface. Comparison and hash code procedures are offered in each relevant interface (Text, Integer, Float...). They must be defined independently instead of inherited and overridden. However, they are better adapted to each context; the Text.Compare procedure expects two TEXT arguments while the equals method expects an argument of type Object.

5.2.2 Text and characters

The Modula-3 characters are from an enumeration containing at least the 256 ISO-8859-1 (latin 1) characters. They may be categorized as letters, digits... using the ASCII interface. Arrays of characters offer the usual array manipulation functions. Texts are immutable sequences of characters. There are no restrictions on the characters which may be contained (e.g. it can contain null characters) or on the length (i.e. much larger than 65000).

The Text interface offers procedures for concatenation, comparison, extracting a substring or a character, converting to/from arrays of characters, finding a character, computing a hash code, and retrieving the length.

Often several strings are identical. When reading a program file one token at a time, many keywords (e.g. FOR, WHILE) or identifiers appear several times. It is interesting to store each unique string only once. The Atom.T object is created from a Text. When several equal texts are used to create Atoms, the same Atom.T object is returned each time. Thus, each different text is stored in only one Atom, eliminating storage redundancy. Moreover, checking two tokens for equality simply involves checking if they reference the same Aton.

In Java, character, string and stringBuffer classes offer the same functionality. Characters use the Unicode standard and are presumably using a 16 bits representation. The character class offers methods to categorize characters as letters, digits... The string class represents immutable sequences of characters and can convert from character arrays, stringBuffer and byte arrays, and to character arrays. The methods offered for string manupulation are similar to the Modula-3 Text interface but with more variants.

Several methods named valueOf exist for different types of arguments (boolean, char, integer...) and produce a textual string representation for these types. The intern method returns a unique string object equal to the string argument. Indeed, the string class maintains a pool of unique strings, much like the Atom interface in Modula-3. All the string constants defined in the program are interned during the program initialization. Dynamically created strings must be interned explicitly if so desired. Interned strings may be compared more efficiently for equality (== versus equal()) and avoid storing more than once each unique text.

The stringBuffer class is an extensible array of characters. It contains several methods to insert or append strings. At each method call, the size of the internal storage buffer is checked and enlarged as needed. Similar functionality in Modula-3 is obtained through instantiating a generic Sequence for Characters.

5.2.3 Numeral types, Booleans and Enumerations

Modula-3 offers booleans, integers, integer ranges, sets, and floats (REAL, LONGREAL, EXTENDED). The Integer and Float interfaces provide comparison and hash code computation. A number of builtin operators allow operations on these types, including set manipulations. The Word interface allows bitwise manipulation of memory words (integers). The Real interface provides information about the numerical accuracy available. The FloatMode interface controls some of the parameters affecting floating point computations, such as the rounding mode. The RealFloat interface offers a number of operations on floating point numbers, and the Math interface provides the usual trigonometric functions.

In Java, booleans, integers (int, long) and floats (float, double) are offered. These primitive types have an associated class in case they need to be wrapped into objects. For each, conversion methods to/from several other types are available. In particular, conversions to/from strings are available. The constructors expecting a string as argument offer the exact same functionality as the corresponding valueOf methods in the string class.

It is often difficult to decide where to put operations involving more than one type. The conversions between strings and other types in Java could be placed in the string class, in the other types, or in both places. The architecture selected was to put the conversions to strings in both places and the conversions from strings only in the other types. Presumably the architects decided that the conversions to strings were more commonly used.

In Modula-3, the conversions to/from Text are found in the Lex and Fmt interfaces. This decomposition follows the functionality instead of the types involved.

5.2.4 Processes and Threads

The Process Modula-3 interface controls the current process (Exit, RegisterExitor, GetStandardFileHandles, Get/SetWorkingDirectory) as well as the creation of subprocesses (Create, Wait). The Params and Env interfaces are used to access the command line arguments and the environment variables.

The Thread interface controls the creation and manipulation of threads (Fork, Join, Wait, Signal, Broadcast, Pause, Alert), mutexes and condition variables. The Fork procedure creates a new thread from a Closure. A Closure is an object with a method named apply. A user defines a thread to fork by inheriting from Closure, adding fields for any data to store, and overriding apply to perform the thread duty.

In Java, the Process class is used to interact with subprocesses (get Output/Input/Error stream, Wait, Destroy). The runtime class interacts with the current process and creates subprocesses (exit, exec, freeMemory, gc, traceInstructions, traceCalls, load, getLocalized input/output stream). FreeMemory reports the available unused memory, Gc initiates a garbage collection in order to reclaim memory if possible, Trace methods produce debugging information if this facility is available, and GetLocalized produces input or ouput streams which convert data from the local format to Unicode. The Load methods dynamically load and initialize the Java code from the specified files.

The System class offers methods to interact with the Java virtual machine (in/out/err output stream, get/set SecurityManager, currentTimeMillis, get/set Property, exit, gc, load, arraycopy). The exit, gc, and load methods are equivalent to their runtime class counterpart. The arraycopy method efficiently copies arrays, and currentTimeMillis returns the current time since epoch (January 1 1970). The security manager determines which actions are allowed (reading/writing files...). Getproperty gives access to an open ended list of properties describing the system. The list includes at least java.version, user.name, user.home, path.separator...

The SecurityManager objects have a check method for each type of action which may or not be permitted (Access, Exit, Exec, PropertyAccess, Link, Read, Write, Delete, Connect, Listen...). The check methods raises an exception if the action is not permitted, and return normally otherwise.

The Thread class allows the creation and control of threads. Threads are created from a Runnable object (descendant of class Runnable with a run method). A created thread may be started, suspended and resumed several times, and stopped. A thread may sleep, wait for another thread to complete, or yield to another thread (on platforms where threads are non preemptive). It is possible to change the priority of each thread. Some schedulers will reserve more execution time for high priority threads. The interrupt method flags the thread as interrupted (which may be checked with isInterrupted) and generates an InterruptedException if the thread is currently waiting.

The ThreadGroup class objects regroup together several threads and ThreadGroup which may be operated on together (suspend, resume...). Thus, the ThreadGroup objects are nodes and the Thread objects are leaves of a tree representing the thread hierarchy.

The runtime and system classes overlap to some extent, and together with the process class offer some of the same functionality as the Modula-3 Process interface. The garbage collection control in Modula-3 is in the RT-Collector interface. The main difference however resides in the Java support for dynamically loading new classes and for security checks. This functionality is particularly needed for executing Java programs downloaded from the network and thus untrusted. These programs may need to load locally available classes and should be restricted in their resource access.

The Java Thread class and the Modula-3 Thread interface are very similar. Runnable objects in one correspond to Closures in the other. The Java thread interface offers additional methods to suspend and resume threads and to change their priority. Furthermore, Java brings the notion of ThreadGroup for conveniently manipulating several threads at once.

5.2.5 Data Structures and Algorithms

The Modula-3 standard library offers the Sequence, Table, SortedTable, List and ListSort generic modules. They can be instantiated for any desired type (integer, user defined objects...). The Sequence is an extensible circular buffer and may be used as a stack, queue or extensible vector. The Table is implemented as a hash table and maintains a mapping between keys and objects. The SortedTable inherits from Table, is implemented as a balanced tree, and can efficiently access the contained elements in sorted order. The Iterator object may be used to iterate through the objects stored in a Table

or SortedTable. The List modules implement linked lists.

The Sx interface provides input/output and manipulation services for symbolic expressions (i.e. LISP like expressions). The Random interface provides random number generators.

The java.util package contains several classes of interest. The Enumeration interface is used to iterate through objects in the various container classes defined. The BitSet class stores arrays of bits; these may be stored in Modula-3 as ARRAY OF [0..1] or as the builtin SET type. The Dictionary class defines the interface for the HashTable class. The contained object type is Object and their hash and equal methods are used to properly store are retrieve the key value pairs in the HashTable. The Properties class extends HashTable by allowing its content to be read/written to a file. Properties tables may be chained to provide default values. Properties tables, however, are restricted to keys and values of type String.

The Vector class implements an extensible vector and serves as a base to the Stack class. It is similar to the Modula-3 Sequence generic module. The Observable class and the Observer interface are used to provide a notification service. The Random class provides random number generators; it is an extension of the random methods available in the Math class. The StringTokenizer class divides an input string into individual tokens. It is similar to, yet less elaborate than the StreamTokenizer class from the java.io package.

5.2.6 Date and Time

The Modula-3 Date, Time and Tick interfaces provide the necessary support to manipulate dates, timestamps and low-level clock ticks. The same functionality is offered by the Java Date class in the java.util package which can extract the Date information (year, month, date...) from a timestamp (number of seconds since epoch, e.g. January 1 1970).

5.2.7 Input and Output

The Modula-3 standard library has two layers to represent input/output objects. At the lower level is the File object from which are derived the Pipe, Terminal, and RegularFile objects. These provide a platform independent access to raw unbuffered files and devices. Unlike Terminal objects, RegularFile objects offer methods for random access (i.e. they are seekable).

The second layer provides buffered readers and writers in the Rd and Wr interfaces. These readers and writers may be created from Text (TextRd, TextWr) or File (FileRd, FileWr) objects. Text and character arrays may be read/written to readers and writers. The Readers and Writers objects are

accessed through procedures instead of methods. This may be surprising since it is convenient to create Wr/Rd derived classes for several useful applications like formatting the output while it is being written, copying the output to two other writers...

This is not an oversight from the library designers. Instead, they placed the functionality which would not be overridden in procedures, since these can be inlined, and defined methods for the low level operations. Thus, the Wr interface defines several variants to send characters, arrays of characters and TEXT to a writer. The WrClass interface defines the method to place an array of characters into the associated buffer and flush the buffer when full. This way, the readers and writers are more efficient and easier to derive from.

The Lex and Fmt interfaces are used to convert the various primitive types (boolean, integer, real, longreal...) to/from textual representations on readers and writers. Converting from one of these types to a textual representation stored in a TEXT is achieved by writing to a TextWr.

The Java java.io package uses a different layering to achieve similar results. The DataInput and DataOutput interfaces are used to read/write the primitive types (byte, byte array, boolean, integer, float...). In many libraries, only bytes are supported; the other types are converted to an array of bytes. These interfaces support the other types for convenience and to insure that they are stored in a platform independent way. The RandomAccessFile class implements these interfaces and allows random access to the underlying file.

The InputStream and OutputStream classes are a separate layer of abstraction. From there are derived FileInputStream, PipedInputStream, ByteArrayInputStream, and StringBufferInputStream, as well as the corresponding output streams. These are used to read from, respectively, files, pipes, byte arrays and StringBuffer.

The SequenceInputStream concatenates the input from two InputStream objects into one. The overridden read method simply relays whatever is read from the first InputStream until the end of file is reached. It then continues with the second Input Stream.

The FilterInputStream class is the base for several useful derived classes. A filter is initialized with another InputStream object and performs some processing on read requests. The BufferedInputStream is a filter providing buffering. The DataInputStream class is a filter implementing the DataInput interface. LineNumberInputStream computes the number of end of line characters read to provide line number information. PushBackInputStream allows reading some characters to look ahead and then pushing these back as if they had not been read. The StreamTokenizer class allows defining which characters are in alphabetic, white space, comment, and number to-

kens, and can parse and return accordingly EndOfFile, EndOfLine, Word, Number or String tokens.

The FilterOutputStream, BufferedOutputStream, and DataOutput Stream are the counterpart to the corresponding InputStream derived types. The PrintStream class OutputFilter may be used to write different types of objects onto an output stream. The formatting is similar to the valueOf methods provided by the String class.

5.2.8 Files

The Modula-3 Pathname interface provides all the required procedures to manipulate file names in a platform independent way. A pathname may be seen as a Text string or may be decomposed into a sequence of arcs. The FS interface is used to create, delete and rename files and directories, list directories, and query the status of files and directories.

The Java File class in package java.io offers the same functionality. It does not support decomposing paths into sequences of arcs. However, it is possible to query the path separator and use string manipulation methods to achieve a similar result.

5.2.9 Discussion

The Modula-3 standard library and the core Java packages (lang, util, io) have a common objective: provide a convenient set of platform independent functions. It is reassuring to find many similarities between the products of these two teams of architects. The differences are also noteworthy and instructive.

In Java, each class or interface resides in a separate file. Thus, related classes and interfaces such as java.util.Observer and java.util.Observable are stored separately. Moreover, except for the primitive types, everything is an object and procedures can only exist as methods. In Modula-3, interfaces, objects and records, and procedures and methods are orthogonal concepts. For example, the Text primitive type is not an object type; this has the downside that you cannot inherit from Text but the advantage of being more efficient.

Modula-3 supports generic modules. Most of the container classes such as Sequence and Table are generics. Each type to be used as parameter to the Table generic module must provide a Compare and a Hash procedure in its interface. In Java, the Object superclass defines the hashCode and equal methods; the equal method expects an argument of type Object. These methods are overidden in any class to be put in a Dictionary. The Java HashTable is therefore equivalent to the Modula-3 Table except that

arguments and return values for the methods can only be checked at run-time. Furthermore, these container classes in Java cannot be used directly for primitive types. There is no compare method defined in Object and thus no sorted table is offered in the core java classes.

Java classes in the core packages offer numerous methods even if it sometimes leads to redundancy, such as the toString method of class Object, versus the valueOf methods of class string, and even the print methods in class PrintStream. In Modula-3, instead of having a method redefined in every class (toString), and having several (redundant) valueOf methods in String, all the conversions to/from Text are provided in the separate Lex and Fmt interfaces.

The Modula-3 interfaces strive for simplicity. An extreme example is the Sequence where the designers left out any insert or remove method; these cannot be implemented efficiently with arrays, and in most cases it is possible to use instead the efficient prepend or append methods. The Java Vector class offers all these methods and then some. However, many of these methods probably have an O(n) complexity, which is not documented and may surprise the unwary.

Another example is the concept of ThreadGroup. This notion is both useful and interesting. For example, a program running animations as separate threads may allow each animation to have its own ThreadGroup containing several threads. Whenever the user wants to terminate an animation, all the related threads are killed through the ThreadGroup. However, only a few applications are multithreaded, and even fewer have several groups of related threads which can be terminated independently. For those few applications, storing the threads in a Sequence and looping over the sequence to terminate the threads is not much more difficult.

Thus, both approaches have their merits. The quest for simplicity is exemplified by Modula-3, Oberon, reduced instruction set computers and to some extent microkernel operating systems. Somewhere between extensive functionality and creeping featurisms stand Ada, C++, Java and several commercial software packages which engage in features lists wars.

5.3 User Interface Libraries

Several applications use a graphical user interface. Numerous sophisticated tools and libraries exist for constructing user interfaces. While these libraries differ significantly, the underlying functionality is often divided into a number of layers to make it more manageable.

At the lowest level is the display hardware. Display adapters (graphics card) greatly vary in how they are programmed, their resolution, capabil-

ity and performance. At the top level is the user interface description in terms of buttons, menus, interactors, text and graphics editors and similar meaningful high level constructs.

5.3.1 Device independence

A layer on top of the display hardware is used to abstract the device specifics in terms of functions and capabilities found on most adapters. This layer offers conventional bit block transfer operations and primitive graphics operations. It may be queried for the resolution and the number of bits per pixel. It may also manage the input devices (mouse and keyboard). A good example is the Sun Pixrect library.

Given the very large number of different display adapters, it is essential to isolate these differences in a library such that the rest of the functionality can be developed independently of the hardware.

5.3.2 Low level graphics

A second layer can be used to offer the primitive building blocks such as rectangles, polygons, fonts and text, overlapping windows and input management. Indeed, this functionality is needed in almost all applications. The X windows server and the associated X library operates at this level.

Some systems go further and offer resolution independent graphics operations with transformation matrices, scalable fonts and filled regions in real units (e.g. millimeters) instead of pixels. The now defunct Sun Network Extensible Windowing System offered such functionality. Display Postscript also operates at this level.

5.3.3 Building blocks for toolkits

The next level is to provide building blocks for higher level constructs. This may include horizontal or vertical stacking of elements, overlaying elements, and similar constructs to build menus, frames, drop shadows... These elements form a tree. The root is the window, the leaves are the primitive elements, such as text or rectangles, and the internal nodes are groupings such as piles or overlays.

When an element changes appearance, it must send paint operations to its father in the tree. The father may clip or translate these paint operations before sending them up in the tree. When an element changes size, it notifies its father which may decide to re divide its available screen estate among the childs or ask its own father for more (or less) space. Similarly, when the window shape is modified by the user, the space allocated to each

node in the tree may be modified. If the window is moved or part of its content erased by another window, the elements in the tree associated with the damaged region will be asked to send their paint operations.

Mouse events (moves or clicks) are also propagated along the tree. A father will normally determine in which of its childs the mouse cursor is located and relay the mouse event to that child. Keyboard input may be directed to the component containing the mouse pointer or owning the *keyboard focus*.

This well structured organization is mostly found in object oriented libraries built on top of X windows. Interviews in C++, or its successor Fresco, and Trestle in Modula-3 are prime examples.

5.3.4 Toolkits

There is an important gap between the lower layer tightly bound to the screen appearance, and the toolkits high level *widgets*. Each toolkit widget corresponds to a conceptual element presented or queried from the user such as choice, file name, text...

It is interesting to note that the basic elements offered by most toolkits do not vary much: menus, buttons, check boxes, file choosers, text and graphics editors. Their appearance, however, varies widely in small details such as key bindings, mouse actions, colors, 3D look with drop shadows, default fonts...

Such toolkits include Motif, the higher level portion of Interviews and the Java AWT package, and VBTkit. A number of toolkits make the same application programming interface available on a number of different platforms (X windows, MS Windows, MacIntosh...) and appearance (Open Look or Motif look and feel). This way, application developers using such a toolkit can, with the appropriate version of the toolkit, get their application running on several platforms without change.

5.3.5 Graphical interface builders

Sometimes, tools are available above the toolkit layer. A graphical editor may be used to position the user interface components by direct manipulation (graphical interface builder). However, complex relationships are often easier to represent in a programming language than through direct manipulation. Indeed, positioning a file chooser at pixel 230 within a 500 pixels wide window is different from specifying that the file chooser must be centered within the window.

A graphical interface builder may produce source code calling the toolkit. This code must be compiled and linked with the application. Some interface

builders produce an interface description that is read by the application at run time. A small performance penalty is paid to interpret the interface description but this enables changing the user interface without recompiling the application. The FormsVBT interface builder produces such run time descriptions.

5.3.6 User interface and Object Orientation

User interfaces are obviously distinct from object orientation but they are often mentioned in books on object oriented programming, if not used as examples. Similarly, the father of the modern object oriented languages, Smalltalk, was mostly used in the context of graphical programs.

Indeed, user interfaces and the underlying layers are an excellent example of object oriented applications. Before C++ became popular, the Pixrect library used a function table to describe each device. The first function in the table would initialize the corresponding device, the second would copy a bitmap and so on; this is nothing else than a method table.

Each element, in the graphics tree associated with a window, must negotiate with its childs to allocate the available screen space. The childs may be of different types but must all support the space allocation negotiation, common to all graphics tree elements; this is easily achieved with inheritance and methods.

At the toolkit level, many different interactors (widgets) may be used to query a character string (file chooser, menu, type-in, editor). In each case, a method called `getText` may be used to obtain the string.

Finally, many user interfaces claim to be object oriented, irrespective of their underlying implementation. They often mean that a small set of actions is applicable to all graphical elements (select, move, resize, delete, change property).

5.3.7 The Modula-3 User Interface Libraries

Trestle is the basic graphics library which interfaces Modula-3 programs and higher level graphics libraries to bit mapped displays. It is concerned with receiving events from the displays (mouse or keyboard events, or window redraw and reshape events), dispatching them to screen regions (sub windows, or Virtual Bitmap Terminal, or simply VBT) and receiving painting commands for VBTs and sending them to the displays. A distinctive feature of Trestle is that one or more separate threads are responsible for delivering events. Thus, a program using Trestle is not forced into executing an event waiting loop. Trestle currently runs on X windows, the Firefly

experimental workstation and on the Win32 interface (Windows NT and Windows95 from Microsoft).

For most applications, higher level libraries such as VBTkit will offer the desired functionality (e.g. menus, editors, scrollbars...). There are cases however where Trestle should be used. Trestle is documented in the *Trestle Reference Manual*. Its main components are outlined below.

The *Trestle* object is used to establish connections with displays, install new windows (VBTs), enumerate the available screens and obtain information such as their color capabilities, the number of bits per pixel and the number of pixels per millimeter.

A number of resource objects are available for displaying on screens. They include painting operations *PaintOp*, cursors *Cursor*, bitmaps *Pixmap*, fonts *Font* and colors *Palette*. These resources have screen independent and screen dependent versions. Normally a screen independent specification is issued which is then resolved into the best matching screen dependent resource available for the target screen type.

Regions of the screen *VBTs*, receive input events from the connected display (mouse and keyboard events, and redraw and reshape events), and receive output commands (painting operations) from the program. A number of simple VBTs are already defined: TextVBT (display a text string), TextureVBT (display a regular background), HVBar (display a rectangle such as an horizontal or vertical bar) and Buttons (display a TextVBT and activate a procedure upon receiving a mouse click). Additional VBTs are easily constructed by inheriting from an existing VBT and overriding a few methods to obtain the desired functionality.

Splits are VBTs with childs. They usually divide their domain (screen region) into sub domains allocated to each child. Similarly, they dispatch the incoming input events to their childs; mouse events are usually dispatched according to the sub domain where the cursor is located and keyboard events are sent to the VBT which requested the keyboard focus.

The Splits available include ZSplit (overlapping childs such as pop up menus), HVSplit (horizontal or vertical stacking of childs such as items in a menu), PackSplit (horizontal or vertical stacking of childs on multiple lines such as buttons which may overflow on two or more lines) and TSplit (displays one of its childs at a time like a deck of cards where only the top card is visible).

VBTkit is a high level library for constructing user interfaces described in *vbtkit: A toolkit for Trestle, Reference Manual*. It offers customizable text editors, typescripts, viewports, buttons, switches, choices, drag and drop, menus, lists, file browsers, text and numeric type in, scrollbars hilighters and miscellaneous other objects.

If none of the objects available fulfills your needs, it is easy to inherit

from one of the existing objects and override a few methods.

User interface design often requires a lot of experimentation to obtain nice looking, intuitive and efficient displays. The FormsVBT library wraps up the objects available in VBTkit to make them available through an interface builder. Thus, most user interfaces are built through FormsVBT instead of using VBTkit directly.

The *FormsVBT* library and the associated *formsedit* user interface builder are used to develop the user interface for most Modula-3 programs. They are described in *The FormsVBT Reference Manual* which includes a tutorial and several examples.

FormsVBT builds upon the user interface components implemented in the VBTkit library. In FormsVBT, symbolic expressions are used to describe user interfaces. These LISP like expressions describe the user interface components (menus, editors, file choosers...), their properties (color, font...) and name. When a program using FormsVBT runs, it reads a symbolic expression and calls FormsVBT to display the corresponding user interface. It then registers call back procedures for a number of named components or queries the named components for their current value.

This organization has several advantages. The user interface only connects with the program through symbolic names and may be developed independently. It may be modified without recompiling the program, for rapid prototyping, or as a mechanism to allow customization by the user. A program may even create user interfaces on the fly, for example as new types are defined in an interpreter or database schema editor.

The drawback is possibly that the compiler and linker cannot determine a priori which user interface components will be used at run time and thus cannot selectively include the required components from the VBTkit library.

5.3.8 The Java User Interface Package

The AWT (Abstract Windowing Toolkit) package is used in Java to construct user interfaces. The Component class defines the methods common to all components: graphical properties (font, color), size negociation (minimum/preferred/maximum size, reshape, resize), painting (repaint, update, paint), and events (handleEvent, mouse Enter Exit Move Drag Up Down, key Up Down, got Focus).

Subclasses of Component include: Button, CheckBox, Choice, Label, List, Scrollbar, TextArea and TextField. A Canvas is a subclass of Component from which custom components should be derived. A Container is a subclass of Component which contains and layout contained components.

The Container class adds methods to add and remove contained components, and to associate a Layout Manager. The subclasses of container are Window (a top level window without frame or menu), Frame (a top level window with a frame and menu), and Panel (a container component somewhere within a Window/Frame). Subclasses of Window include Dialog and FileDialog (a file chooser).

The menus are a distinct class hierarchy. The MenuComponent is the super class of MenuBar and MenuItem. MenuItem is the super class of Menu and MenuCheckBox. Menus are attached to components which implement the MenuContainer interface. For instance, the MenuBar components are usually attached to Frames using the setMenuBar method.

The LayoutManager interface is implemented by a number of separate classes: BorderLayout, CardLayout, FlowLayout, GridBagLayout, and GridLayout. Layout Managers are assigned to container objects and compute their contained components location.

A number of additional classes are used to define objects used by the components: Color, Dimension, Event, Font, FontMetric, Image, Graphics, GridBagConstraints, Insets, Point, Polygon, Rectangle.

Event delivery proceeds as follows. The postEvent method is called, which causes AWT to call the deliverEvent method of a component. DeliverEvent then calls handleEvent and depending on the return value may call deliverEvent on one of the contained components if applicable. HandleEvent usually categorizes the event and accordingly calls one of mouseUp, mouseDown, keyUp, keyDown... to make life easier on the programmer.

Repainting is handled similarly. When some portion of the user interface is damaged or has its appearance changing, the repaint method is called with the damaged region as argument. This causes AWT to call the update method of the corresponding component. When the component needs to be painted from scratch, the paint method is called. Often update will call paint or vice-versa such that only one of the two contains the painting code.

5.3.9 Discussion

The VBT and Split objects in Trestle closely resemble the Component and Container classes in AWT. Many of the components are found in both libraries while VBTkit contains more Filter components, adding shadows and highlighting, and AWT has the Grid Layout managers.

The most striking difference is the fact that menus and layout managers are separate class hierarchies in AWT. Indeed, menus and menu items should be useable anywhere just like the other components. The layout managers could be subclasses of Panel instead of being separate objects connected to Panel objects. This design was dictated by the desire to reuse

the native components available on the different platforms. Each platform already has libraries for menus, dialogs, buttons and the like. Thus, AWT provides the glue code between the Java objects and the platform dependent peer toolkit objects. Thus, menu items need little more than storing the handle to the corresponding native library object.

The event handling machinery is also affected by this relationship with the platform dependent toolkits. On some platforms the handleEvent method of the top level window and each intervening container object will be called as the event propagates to the correct component. On other platforms, the native library propagates the event directly to the correct native component which then calls the peer AWT component.

Trestle does not rely on high level platform dependent toolkits. The benefit is that appearance and event delivery is the same on all platforms. This may also be a disadvantage if users prefer to have the native look and feel on each platform. There is no good solution to this dilemma since native toolkits differ significantly, and the differences will affect the code portability.

5.4 A Complete Graphical Application Example

In this section, a complete graphical example is examined. It is a simple diagram editor but is representative of much larger applications. It exercises the mouse access and painting operations from the Trestle windowing library, and higher level user interaction operations (menus and dialogs) from the FormsVBT library. It also uses network and persistent objects in an interesting way but this will be examined in a later chapter.

The diagram editor is built in a number of layers. Lower level layers could easily be reused in the context of a different graphical application. The lowest level is a *Diagram* containing *Items* which know how to paint themselves on a Trestle window. The next level is a *DiagramVBT* displaying a *Diagram* in a window. The top level is the *Draw* program which adds menus and editing operations to interact with a DiagramVBT.

5.4.1 Graphical items

The simple graphical editor handles two types of items: rectangle (Rectangle) and text (Label). Items are defined using a graphics context (color, text) and a list of vertices. They offer a number of methods, as defined in Item.i3.

```
(* This interface defines the ancestor type for all geometric primitives. *)

INTERFACE Item;

IMPORT Color, VBT, RealRect, RealPoint, RealTransform, Rect, Rd, Wr, Lex;

TYPE

(* The graphics context holds all the current attribute values which
   may be used in geometric primitives (e.g. current color, text...). *)

  GraphicsContext = RECORD
      color: Color.T;
      text: TEXT;
    END;

  T <: Public;
  Public = OBJECT
    METHODS
      init(c: GraphicsContext): T;
      read(rd: Rd.T)
          RAISES{Rd.Failure, Lex.Error};
      write(wr: Wr.T)
          RAISES{Wr.Failure};
      nbVertex(VAR min, nb, max: CARDINAL);
      setVertex(i: CARDINAL; p: RealPoint.T);
      getVertex(i: CARDINAL): RealPoint.T;
      boundingBox(): RealRect.T;
      paint(v: VBT.T; region: Rect.T; m: RealTransform.T)
    END;

(* The call "item.init(gc)" initializes "item" using the relevant
   attributes from the graphics context "gc".

   The call "item.read(rd)" initializes "item" from the description
   available from "rd". The call "item.write(wr)" produces on "wr"
   a description of "item" suitable for a later call to the "read" method.

   The call "item.nbVertex(min,nb,max)" sets min to the minimum number of
   vertices required (e.g. two for a line), "nb" to the number of vertices
   present (e.g. five for a polygon built from five vertices), and "max"
```

```
   to the maximum number of vertices allowed (e.g. LAST(CARDINAL) for
   a polygon).

   The call "item.setVertex(i,pt)" sets vertex "i" to "pt" while the
   call "item.getVertex(i)" returns vertex "i".

   The call "item.boundingBox()" returns the bounding box completely
   containing the item.

   When "item.paint(v,r,m)" is called, the item is drawn on the Virtual
   Bitmap Terminal "v", clipped to region "r", using "m" as the transformation
   matrix between the real coordinates used within item and the integer
   coordinates of "v".
*)

  TwoVertex <: T;
  (* Squeleton of a two vertex item to inherit from. The two vertices are
     the NorthWest and SouthEast corners and thus define the bounding box.
     It provides suitable default methods for everything except paint.
     The read and write methods simply read the two vertices and the color.  *)

  CreateProc = PROCEDURE(): T;
  (* A procedure to create an item. *)

PROCEDURE RegisterSubType(name: TEXT; createProc: CreateProc);
(* Register "createProc" as the procedure to create items of type "name" *)

PROCEDURE Create(subTypeName: TEXT): T;
(* Create an item of type "subTypeName" using one of the
   procedures registered with "RegisterSubType". If no such subtype
   exists, return NIL. *)

END Item.
```

The fields used to store the information common to all items (color) are defined in another interface only used by subtypes of Item.T.

```
(* This interface reveals the internal fields of Item.T to subtypes. *)

INTERFACE ItemRep;
```

```
IMPORT Item, Color, RealPoint;

REVEAL
  Item.T <: Suffix;

  Item.TwoVertex <: TwoVertexSuffix;

TYPE

(* The default init method of Item.T sets the color. *)

  Suffix = Item.Public OBJECT
      color: Color.T;
    END;

  TwoVertexSuffix = Item.T OBJECT
      v1, v2: RealPoint.T;
    END;

END ItemRep.
```

A default implementation of the Item.T methods is provided in Item.m3 for two vertex items. Indeed, several geometric primitives may be defined using two vertices: rectangle, circle, ellipse...

```
MODULE Item EXPORTS Item, ItemRep;

IMPORT Rd, Wr, Lex, Fmt, RealRect, RealPoint, Thread, FloatMode, TextRefTbl;

REVEAL
  T = Suffix BRANDED OBJECT OVERRIDES
      init := Init;
    END;

  TwoVertex = TwoVertexSuffix BRANDED OBJECT OVERRIDES
      read := Read;
      write := Write;
      nbVertex := NbVertex;
      setVertex := SetVertex;
      getVertex := GetVertex;
      boundingBox := BoundingBox;
```

```
    END;

<*FATAL Thread.Alerted*>
<*FATAL FloatMode.Trap*>

(* All items must have at least a color. *)

PROCEDURE Init(self: T; c: GraphicsContext): T =
  BEGIN
    self.color := c.color;
    RETURN self;
  END Init;

(* Read the two vertices and the color *)

PROCEDURE Read(self: TwoVertex; rd: Rd.T)
    RAISES{Rd.Failure, Lex.Error} =
  BEGIN
    Lex.Skip(rd); self.v1[0] := Lex.Real(rd);
    Lex.Skip(rd); self.v1[1] := Lex.Real(rd);
    Lex.Skip(rd); self.v2[0] := Lex.Real(rd);
    Lex.Skip(rd); self.v2[1] := Lex.Real(rd);
    Lex.Skip(rd); self.color.r := Lex.Real(rd);
    Lex.Skip(rd); self.color.g := Lex.Real(rd);
    Lex.Skip(rd); self.color.b := Lex.Real(rd);
  END Read;

(* Write the two vertices and the color *)

PROCEDURE Write(self: TwoVertex; wr: Wr.T)
    RAISES{Wr.Failure} =
  BEGIN
    Wr.PutText(wr, Fmt.Real(self.v1[0]) & " " &
                Fmt.Real(self.v1[1]) & " " &
                Fmt.Real(self.v2[0]) & " " &
                Fmt.Real(self.v2[1]) & " " &
                Fmt.Real(self.color.r) & " " &
                Fmt.Real(self.color.g) & " " &
                Fmt.Real(self.color.b));
  END Write;

(* There are exactly two vertices *)
```

```
PROCEDURE NbVertex(<*UNUSED*>self: T; VAR min, nb, max: CARDINAL) =
  BEGIN
    min := 2;
    nb := 2;
    max := 2;
  END NbVertex;

PROCEDURE SetVertex(self: TwoVertex; i: CARDINAL; p: RealPoint.T) =
  BEGIN
    IF i = 0 THEN self.v1 := p;
    ELSIF i = 1 THEN self.v2 := p;
    ELSE <*ASSERT FALSE*>
    END;
  END SetVertex;

PROCEDURE GetVertex(self: TwoVertex; i: CARDINAL): RealPoint.T =
  BEGIN
    IF i = 0 THEN RETURN self.v1;
    ELSIF i = 1 THEN RETURN self.v2;
    ELSE <*ASSERT FALSE*>
    END;
  END GetVertex;

(* The bounding box is defined by the two vertices *)

PROCEDURE BoundingBox(self: TwoVertex): RealRect.T =
  BEGIN
    RETURN RealRect.FromCorners(self.v1,self.v2);
  END BoundingBox;

TYPE
  CreateRecord = REF RECORD proc: CreateProc; END;

(* New geometric item types may be added without modifying this file.
   The new types only register their name and a procedure to create an
   instance of the type. *)

PROCEDURE RegisterSubType(name: TEXT; createProc: CreateProc) =
  VAR
    proc: REFANY := NEW(CreateRecord, proc := createProc);
  BEGIN
```

```
    EVAL subtypes.put(name,proc);
  END RegisterSubType;

(* Lookup the name in the table and create a new instance if found *)

PROCEDURE Create(subTypeName: TEXT): T =
  VAR
    proc: REFANY;
  BEGIN
    IF subTypeName = NIL THEN RETURN NIL; END;
    IF subtypes.get(subTypeName,proc) THEN
      RETURN NARROW(proc,CreateRecord).proc();
    END;
    RETURN NIL;
  END Create;

VAR
  subtypes := NEW(TextRefTbl.Default).init();

BEGIN
END Item.
```

Adding new geometric primitives is relatively simple. New types inherit from Item.T or Item.TwoVertex, redefining a few methods. This is where object oriented programming provides important savings through reuse.

```
(* This interface defines the Rectangle geometric item. A rectangle is defined
   by its NorthWest and SouthEast corners. *)

INTERFACE Rectangle;

IMPORT Item;

TYPE
  T <: Item.T;

END Rectangle.
```

```
MODULE Rectangle;
```

```
IMPORT Wr, VBT, RealTransform, Rect, PaintOp, Conv, Thread,
       Item, ItemRep;

REVEAL

  T = Item.TwoVertex BRANDED OBJECT OVERRIDES
      write := Write;
      paint := Paint;
    END;

<*FATAL Thread.Alerted*>

PROCEDURE New(): Item.T =
  BEGIN
    RETURN NEW(T);
  END New;

(* Write the item type. Everything else is managed by the ancestor
   "Item.TwoVertex". *)

PROCEDURE Write(self: T; wr: Wr.T)
    RAISES{Wr.Failure} =
  BEGIN
    Wr.PutText(wr, "Rectangle ");
    Item.TwoVertex.write(self,wr);
    Wr.PutText(wr,"\n");
  END Write;

PROCEDURE Paint(self: T; v: VBT.T; region: Rect.T; m: RealTransform.T) =
  VAR
    (* Convert the item rectangle to VBT coordinates and clip to the
       region needing refresh. *)

    rect := Rect.Meet(region,Rect.FromCorners(
        Conv.ToPoint(RealTransform.Transform(m,self.getVertex(0))),
        Conv.ToPoint(RealTransform.Transform(m,self.getVertex(1)))));
  BEGIN
    VBT.PaintTint(v,rect,
        PaintOp.FromRGB(self.color.r,self.color.g,self.color.b));
  END Paint;

BEGIN
```

```
  Item.RegisterSubType("Rectangle", New);

END Rectangle.


(* This interface defines a "Label" geometric primitive. A label is a
   text string positioned using two vertices. The string is centered in,
   and clipped to, the corresponding box. *)

INTERFACE Label;

IMPORT Item;

TYPE
  T <: Item.T;

END Label.

MODULE Label;

IMPORT Rd, Wr, VBT, Lex, RealTransform, Rect,
       Point, Font, PaintOp, Conv, Thread, Item, ItemRep;

REVEAL
  T = Item.TwoVertex BRANDED OBJECT
      text: TEXT;
    OVERRIDES
      read := Read;
      write := Write;
      init := Init;
      paint := Paint;
    END;

<*FATAL Thread.Alerted*>

PROCEDURE New(): Item.T =
  BEGIN
    RETURN NEW(T);
  END New;

CONST
  NoQuote: SET OF CHAR = SET OF CHAR{FIRST(CHAR) .. LAST(CHAR)} -
```

```
      SET OF CHAR{'"'};

(* Use the ancestor read method to read the two vertices and the color.
   Then read the quoted string. Currently there is no provision for
   putting quotes within the string. *)

PROCEDURE Read(self: T; rd: Rd.T)
    RAISES{Rd.Failure, Lex.Error} =
  BEGIN
    Item.TwoVertex.read(self,rd);
    Lex.Skip(rd);
    Lex.Match(rd,"\"");
    self.text := Lex.Scan(rd,NoQuote);
    Lex.Match(rd,"\"");
  END Read;

(* Write the type name "Label", then use the ancestor method to write the
   two vertices and color, and finally print out the string. *)

PROCEDURE Write(self: T; wr: Wr.T)
    RAISES{Wr.Failure} =
  BEGIN
    Wr.PutText(wr, "Label ");
    Item.TwoVertex.write(self,wr);
    Wr.PutText(wr," \"" & self.text & "\"\n");
  END Write;

(* Call the parent method for retrieving the color, and take the current
   text from the graphics context. *)

PROCEDURE Init(self: T; c: Item.GraphicsContext): Item.T =
  BEGIN
    EVAL Item.TwoVertex.init(self,c);
    self.text := c.text;
    RETURN self;
  END Init;

PROCEDURE Paint(self: T; v: VBT.T; region: Rect.T; m: RealTransform.T) =
  VAR
    (* Determine the rectangle for the item in VBT coordinates, as well
       as the center of the rectangle. *)
```

```
    itemRect := Rect.FromCorners(
        Conv.ToPoint(RealTransform.Transform(m,self.getVertex(0))),
        Conv.ToPoint(RealTransform.Transform(m,self.getVertex(1))));
    itemCenter := Rect.Middle(itemRect);

    (* Determine the bounding box of the string as well as its center.*)

    textRect := VBT.BoundingBox(v,self.text,Font.BuiltIn);
    textCenter := Rect.Middle(textRect);

    textStart: Point.T;
  BEGIN

    (* The text starting position must be such that the string center and
       the item rectangle centers coincide. *)

    textStart.h := itemCenter.h - textCenter.h;
    textStart.v := itemCenter.v - textCenter.v;

    (* Paint the text clipping to the item bounding box as well as the region
       to repaint. *)

    VBT.PaintText(v := v,
        clip := Rect.Meet(region,itemRect),
        pt := textStart, t := self.text, op := PaintOp.FromRGB(self.color.r,
        self.color.g, self.color.b));
  END Paint;

BEGIN
  Item.RegisterSubType("Label",New);
END Label.
```

5.4.2 Diagrams of items

A diagram manages a list of items. It inherits from NetObj.T to allow remote access for client/server applications. The display machinery needs to be notified whenever the diagram is modified. Thus, a notification service is offered.

```
(* This interface defines "Diagram.T" a sequence of geometric primitives.
   It inherits from NetObj.T to enable remote network object diagrams.
```

```
   Modification notification is available, which is useful to maintain
   one or more graphical views of the diagram. Accesses to a diagram
   and unmonitored, thus external locking is required for multi-threaded
   access. *)

INTERFACE Diagram;

IMPORT Item, Rd, Wr, NetObj, RealPoint, Lex, Thread;

CONST
  Brand = "Diagram";

TYPE
  T <: Public;
  Public = NetObj.T OBJECT METHODS
      init(): T
          RAISES{NetObj.Error, Thread.Alerted};
      initS()
          RAISES{NetObj.Error, Thread.Alerted};
      initT()
          RAISES{NetObj.Error, Thread.Alerted};
      read(rd: Rd.T; binary := FALSE)
          RAISES{NetObj.Error, Thread.Alerted, Rd.Failure, Lex.Error};
      write(wr: Wr.T; binary := FALSE)
          RAISES{NetObj.Error, Thread.Alerted, Wr.Failure};
      size(): CARDINAL
          RAISES{NetObj.Error, Thread.Alerted};
      get(i: CARDINAL): Item.T
          RAISES{NetObj.Error, Thread.Alerted};
      add(item: Item.T)
          RAISES{NetObj.Error, Thread.Alerted};
      rem(i: CARDINAL)
          RAISES{NetObj.Error, Thread.Alerted};
      remAll()
          RAISES{NetObj.Error, Thread.Alerted};
      select(pt: RealPoint.T; nth: CARDINAL; VAR i: CARDINAL): BOOLEAN
          RAISES{NetObj.Error, Thread.Alerted};
      attach(cl: Closure)
          RAISES{NetObj.Error, Thread.Alerted};
      detach(cl: Closure)
          RAISES{NetObj.Error, Thread.Alerted};
    END;
```

```
    <*PRAGMA STABLE*>
    <* STABLE UPDATE METHODS initS, add, rem, remAll *>

(* All the methods may raise NetObj.Error when used remotely.

   The call "d.init()" initializes "d" by calling "initS", for
   initializing the stable state, and "initT", for initializing the
   transient state. This is useful when the stable state may be recovered
   from a file using "stable objects". The methods which update the
   stable state are initS, add, rem, and remAll. The list of attached
   closures, for notification, is not part of the stable state.

   The calls "d.write(wr,binary)" and "d.read(rd,binary)" may be used to write
   the stable state to "wr" and later recover it from "rd". The file
   format is based on the Pickle module if "binary" is true, and is human
   readable otherwise.

   The "size" method returns the number of items in the diagram.

   The call "d.get(i)" returns the item at the corresponding position.

   The call "d.add(item)" adds "item" to the diagram.

   The call "rem(i)" removes the corresponding item from "d".

   Method "remAll" empties the diagram.

   The call "d.select(pt,nth,i)" returns whether an item was selected
   and sets "i" to the position of the item. This position remains
   valid until items are added or removed from the diagram. The
   selection looks for the "nth" item for which "pt" lies inside the
   bounding box.

   The call "d.attach(cl)" registers "cl" for notification whenever items
   are added or removed from the diagram. Several closures may be
   registered at any time.

   The call "d.detach(cl)" unregisters "cl" for notification.
*)

  Closure = OBJECT METHODS
      add(item: Item.T);
```

```
      rem(i: CARDINAL; item: Item.T);
      remAll();
    END;

(* The methods of a closure registered for notification are called
   appropriately whenever items are added or removed from the diagram. *)

END Diagram.

MODULE Diagram;

IMPORT Rd, Wr, RefSeq, RealPoint, Pickle, Thread, Item,
       RealRect, Lex, NetObj;

REVEAL
  T = Public BRANDED OBJECT
      items: RefSeq.T;
      closures: RefSeq.T;
    OVERRIDES
      init := Init;
      initS := InitS;
      initT := InitT;
      read := Read;
      write := Write;
      size := Size;
      get := Get;
      add := Add;
      rem := Rem;
      remAll := RemAll;
      select := Select;
      attach := Attach;
      detach := Detach;
    END;

<*FATAL Thread.Alerted*>
(*<*FATAL NetObj.Error*>*)

PROCEDURE Init(self: T): T =
  BEGIN
    InitS(self);
    InitT(self);
    RETURN self;
```

```
  END Init;

(* Initialize the sequence of items *)

PROCEDURE InitS(self: T) =
  BEGIN
    self.items := NEW(RefSeq.T).init();
  END InitS;

(* Initialize the sequence of notification closures *)

PROCEDURE InitT(self: T) =
  BEGIN
    self.closures := NEW(RefSeq.T).init();
  END InitT;

PROCEDURE Read(self: T; rd: Rd.T; binary := FALSE)
    RAISES{NetObj.Error, Rd.Failure, Lex.Error} =
  VAR
    name: TEXT;
    item: Item.T;
  BEGIN
    (* Read the items in "binary" using Pickle *)

    IF binary THEN
      TRY
        self.items := Pickle.Read(rd);
      EXCEPT
      | Pickle.Error, Rd.EndOfFile => RAISE Rd.Failure(NIL);
      END;
    ELSE

      (* Read the item type, create an item of that type,
         ask the item to read its content from the input stream,
         add the item to the diagram. *)

      WHILE(NOT Rd.EOF(rd)) DO
        Lex.Skip(rd);
        name := Lex.Scan(rd);
        item := Item.Create(name);
        IF item # NIL THEN
          item.read(rd);
```

```
          self.add(item);
        END;
      END;
    END;
  END Read;

PROCEDURE Write(self: T; wr: Wr.T; binary := FALSE)
    RAISES{NetObj.Error, Wr.Failure} =
  BEGIN
    (* Write all the items at once in binary using Pickle *)

    IF binary THEN
      TRY
        Pickle.Write(wr,self.items);
      EXCEPT
      | Pickle.Error => RAISE Wr.Failure(NIL);
      END;
    ELSE
      (* Write each item *)

      FOR i := 0 TO self.size() - 1 DO
        self.get(i).write(wr);
      END;
    END;
  END Write;

PROCEDURE Size(self: T): CARDINAL =
  BEGIN
    RETURN self.items.size();
  END Size;

PROCEDURE Get(self: T; i: CARDINAL): Item.T =
  BEGIN
    RETURN self.items.get(i);
  END Get;

(* Add the item and notify the closures *)

PROCEDURE Add(self: T; item: Item.T) =
  VAR
    cl: Closure;
  BEGIN
```

```
    self.items.addhi(item);
    FOR i := 0 TO self.closures.size() - 1 DO
      cl := self.closures.get(i);
      cl.add(item);
    END;
  END Add;

(* Remove the item, replacing it with the last item. This way, items
   are removed efficiently from the end of the sequence. Notify the
   closures of the removal. *)

PROCEDURE Rem(self: T; i: CARDINAL) =
  VAR
    item: Item.T;
    cl: Closure;
  BEGIN
    WITH items = self.items DO
      item := items.get(i);
      items.put(i,items.get(items.size() - 1));
      EVAL items.remhi();
    END;
    FOR ii := 0 TO self.closures.size() - 1 DO
      cl := self.closures.get(ii);
      cl.rem(i,item);
    END;
  END Rem;

(* Empty the items and notify the colsures. *)

PROCEDURE RemAll(self: T) =
  VAR
    cl: Closure;
  BEGIN
    EVAL self.items.init();
    FOR i := 0 TO self.closures.size() - 1 DO
      cl := self.closures.get(i);
      cl.remAll();
    END;
  END RemAll;

(* For each item check if the point lies within the bounding box. Retain
   the nth such item. *)
```

```
PROCEDURE Select(self: T; pt: RealPoint.T; nth: CARDINAL; VAR i: CARDINAL):
    BOOLEAN =
  VAR
    item: Item.T;
  BEGIN
    FOR ii := 0 TO self.items.size() - 1 DO
      item := self.items.get(ii);
      IF RealRect.Member(pt,item.boundingBox()) THEN
        DEC(nth);
        IF nth = 0 THEN
          i := ii;
          RETURN TRUE;
        END;
      END;
    END;
    RETURN FALSE;
  END Select;

(* Add the closure to the sequence of closures to notify. *)

PROCEDURE Attach(self: T; cl: Closure) =
  BEGIN
    self.closures.addhi(cl);
  END Attach;

(* Remove the closure from the sequence, replacing it with the
   last closure to efficiently remove from the end of the sequence. *)

PROCEDURE Detach(self: T; cl: Closure) =
  BEGIN
    WITH closures = self.closures DO
      FOR i := 0 TO self.closures.size() - 1 DO
        IF cl = self.closures.get(i) THEN
          closures.put(i,closures.get(closures.size() - 1));
          EVAL closures.remhi();
        END;
      END;
    END;
  END Detach;

BEGIN
```

```
END Diagram.
```

5.4.3 A Diagram display

The DiagramVBT.T continuously displays the content of a diagram in a trestle VBT. Whenever an item is added or removed, the corresponding screen region is repainted. Similarly, if the window display is *damaged* because another window overlapped the diagram, a repaint method is provided to repaint the screen. All the items are stored in real user coordinates. The normalized (0.0,0.0) to (1.0,1.0) square is mapped into the available VBT. When the reshape method is activated, because the available VBT has been resized, the mapping between user and screen coordinates is recomputed and the screen repainted accordingly.

```
(* A DiagramVBT.T continuously displays the area from (0.0,0.0) to
   (1.0,1.0) of a diagram within a VBT. *)

INTERFACE DiagramVBT;

IMPORT VBT, Diagram, RealPoint;

TYPE
  T <: Public;
  Public = VBT.Leaf OBJECT METHODS
      init(l: Diagram.T): T;
      attachProc(p: MouseProc);
    END;

(* The call "d.init(l)" initializes the diagram VBT with diagram "l".

   The call "d.attachProc(p)" insures that "p" is called each time
   a mouse button is pressed. Only one procedure may be attached
   and attaching NIL removes the currently attached procedure.

*)

  MouseProc = PROCEDURE(v: T; button: [0..2]; position: RealPoint.T);

(* The procedure is called with the mouse button (left, middle, right)
   and the position in user coordinates ([0.0..1.0] in both dimensions). *)
```

```
END DiagramVBT.


MODULE DiagramVBT;

IMPORT VBT, Diagram, Region, RealRect, RealTransform, Rect, Item,
       Conv, PaintOp;

REVEAL
  T = Public BRANDED OBJECT
      m, im: RealTransform.T;
      mouseProc: MouseProc;
      l: Diagram.T;
      cl: Closure;
    OVERRIDES
      init := Init;
      attachProc := AttachProc;
      mouse := Mouse;
      repaint := Repaint;
      reshape := Reshape;
    END;

TYPE
  Closure = Diagram.Closure OBJECT
      vbt: T;
    OVERRIDES
      add := Add;
      rem := Rem;
      remAll := RemAll;
    END;

(* Register with the diagram for notification and "mark" the VBT for
   the initial display. *)

PROCEDURE Init(self: T; l: Diagram.T): T =
  BEGIN
    self.l := l;
    self.cl := NEW(Closure, vbt := self);
    self.l.attach(self.cl) <*NOWARN*>;
    VBT.Mark(self);
    RETURN self;
  END Init;
```

```
(* Store the procedure to call upon a mouse click. *)

PROCEDURE AttachProc(self: T; p: MouseProc) =
  BEGIN
    self.mouseProc := p;
  END AttachProc;

(* Each time the VBT changes position or shape on screen, compute
   the new transformation matrices between world and screen
   coordinates, and repaint the whole VBT. *)

PROCEDURE Reshape(self: T; READONLY cd: VBT.ReshapeRec) =
  BEGIN
    self.m := RealTransform.Identity;
    self.m[0] := FLOAT(cd.new.east - cd.new.west);
    self.m[3] := FLOAT(cd.new.south - cd.new.north);
    self.m[4] := FLOAT(cd.new.west);
    self.m[5] := FLOAT(cd.new.north);
    self.im := RealTransform.Inverse(self.m);
    Repaint(self,Region.Full);
  END Reshape;

(* When a Down mouse click is received, call the registered procedure
   if any. The mouse position is converted to user coordinates. *)

PROCEDURE Mouse(self: T; READONLY cd: VBT.MouseRec) =
  BEGIN
    IF self.mouseProc # NIL AND cd.clickType = VBT.ClickType.FirstDown THEN
      self.mouseProc(self, ORD(cd.whatChanged) - ORD(VBT.Modifier.MouseL),
          RealTransform.Transform(self.im,Conv.ToRealPoint(cd.cp.pt)));
    END;
  END Mouse;

(* Compute the region to repaint in user coordinates and call Repaint3 to
   do the work. *)

PROCEDURE Repaint(self: T; READONLY rgn: Region.T) =
  BEGIN
    Repaint3(self,rgn.r,RealRect.FromCorners(
        RealTransform.Transform(self.im,
          Conv.ToRealPoint(Rect.NorthWest(rgn.r))),
```

```
        RealTransform.Transform(self.im,
          Conv.ToRealPoint(Rect.SouthEast(rgn.r)))));
  END Repaint;

(* Compute the region modified in screen coordinates and call Repaint3
   to do the work. *)

PROCEDURE Repaint2(self: T; rr: RealRect.T) =
  BEGIN
    Repaint3(self,Rect.FromCorners(
        Conv.ToPoint(RealTransform.Transform(self.m,RealRect.NorthWest(rr))),
        Conv.ToPoint(RealTransform.Transform(self.m,RealRect.SouthEast(rr)))),
        rr);
  END Repaint2;

(* Repaint each item intersecting the region to repaint, after
   repainting the background. *)

PROCEDURE Repaint3(self: T; r: Rect.T; rr: RealRect.T) =
  VAR
    item: Item.T;
  BEGIN
    VBT.PaintTint(self,r,bg);
    FOR i := 0 TO self.l.size() - 1 <*NOWARN*> DO
      item := self.l.get(i) <*NOWARN*>;
      IF RealRect.Overlap(rr,item.boundingBox()) THEN
        item.paint(self,r,self.m);
      END;
    END;
  END Repaint3;

(* An item was added to the diagram, repaint the corresponding region. *)

PROCEDURE Add(self: Closure; item: Item.T) =
  BEGIN
    Repaint2(self.vbt,item.boundingBox());
  END Add;

(* An item was removed from the diagram, repaint the corresponding region. *)

PROCEDURE Rem(self: Closure; <*UNUSED*>i: CARDINAL;
    item: Item.T) =
```

```
  BEGIN
    Repaint2(self.vbt, item.boundingBox());
  END Rem;

(* Everything was removed, repaint the whole thing. *)

PROCEDURE RemAll(self: Closure) =
  BEGIN
    Repaint2(self.vbt,FullRealRect);
  END RemAll;

CONST
  FullRealRect = RealRect.T{-1.0, 2.0, -1.0, 2.0};

VAR
  (* Background color *)
  bg := PaintOp.FromRGB(0.0,0.0,0.0);

BEGIN
END DiagramVBT.
```

5.4.4 The Diagram editor

The diagram editor consists in a user interface description, implemented using FormsVBT, and a module implementing the interaction between the menu buttons and mouse clicks, and the diagrams. The user interface description is stored in file draw.fv.

```
;
; Menu for a simple drawing program.
;
; The top level ZSplit enables pop up childs for showing error messages and
; browsing file names.
;
(ZSplit
  (ZBackground

    ; Provide an initial size.
    (Shape (Width 400 + Inf - 200) (Height 400 + Inf - 200)

        (VBox
          (Glue 5)
```

```
; The menu bar. It contains three top level entries:
; File, Color, and Connect.
(HBox
  (Glue 5)
  (Menu "File"
    (VBox
      ; In each case, pop the file browser
      (PopMButton %new (For selectFile) "New")
      (PopMButton %open (For selectFile) "Open")
      (PopMButton %save (For selectFile) "Save")
      (MButton %quit "Quit")
    )
  )
  (Glue 5)
  (Menu "Color"
    (VBox
      ; Select the desired color
      (MButton (Color "red") %red "Red")
      (MButton (Color "green") %green "Green")
      (MButton (Color "blue") %blue "Blue")
    )
  )
  (Glue 5)
  (Menu "Connect"
    (VBox
      ; Select the "server" command.
      (PopMButton (For connectServer) "Connect")
      (MButton %disconnect "Disconnect")
      (MButton %startMirror "Start Mirroring")
      (MButton %stopMirror "Stop Mirroring")
      (MButton %recover "Recover")
    )
  )
  Fill
)
; Dividing bar between the menu bar and the rest.
(Glue 5)(Bar 3)
;
; The Left side of the draw area contains buttons to select the
; current action, and displays the current color and text.
(HBox
```

```
(Glue 5)
(VBox
  Fill
  (Button %delete "Delete")
  Fill
  (Button %Rectangle "Rectangle")
  Fill
  (Button %Label "Label")
  Fill
  (Text "Current text:")
  (Frame Lowered (TypeIn %labelText))
  Fill
  (VBox
    (Text "Current color: ")(Glue 5)
    (Text %defaultColor "red")
  )
  (Frame Lowered (Shape (Height 10 + 50)
    (Texture %color (Color "red")))
  )
  Fill
)
; Dividing bar between the action selection and
; the draw area.
(Glue 5)(Bar 3)
(VBox
  (Glue 5)
  (HBox
    (Glue 5)
    (Frame Lowered
      (Shape (Width + inf) (Height + inf)
        (Generic %drawArea)
      )
    )
    (Glue 5)
  )
  ; A small message line appears just below the draw area
  ; separated by a dividing bar.
  (Glue 5)(Bar 3)(Glue 5)
  (Shape (Height 20)
    (HBox
      (Glue 5)
      (Text LeftAlign "Message:")
```

```
                (Text LeftAlign %message "")
                Fill
              )
            )
          )
        )
      )

  )
)
; A popup file browser
(ZChassis %selectFile (Title "Select a file")
  (Shape (Height 150 + inf) (Width 200 + inf)
    (VBox
      (FileBrowser %browseFile)
      (Glue 10)
      (Frame Lowered (Helper (For browseFile)))
      (Glue 10)
    )
  )
)
; A popup error message display
(ZChassis %errorChild (Title "Error message")
  (Shape (Height 75 + inf) (Width 300 + inf)
    (TextEdit ReadOnly %error)
  )
)
; A popup "host" and "port" selection window
(ZChassis %connectServer (Title "Select a server")
  (Shape (Height 75 + inf)(Width 300 + inf - 100)
    (VBox Fill
      (HBox
        (Glue 5)(Text LeftAlign "Connect to: ")
        (Frame Lowered (TypeIn %server))
        (Glue 5)(Text LeftAlign "Port: ")
        (Frame Lowered (Numeric (Min 2000) (Max 16000) %port =2288))
        (Glue 5)
      )
      Fill
      (HBox Fill
        (Button %confirmConnect "Confirm")
        Fill
```

```
          (Button %cancelConnect "Cancel")
          Fill
        )
        Fill
      )
    )
  )
)
```

The core of the diagram editor which connects all the rest is in Draw.m3.

```
MODULE Draw EXPORTS Draw, Main;

IMPORT FormsVBT, Diagram, DiagramVBT, Item,
       Rectangle <*NOWARN*>, Label <*NOWARN*>,
       VBT, Trestle, TrestleComm, Rsrc,
       DrawBundle, RealPoint, Text, ColorName, Rd, FileRd, Wr, FileWr, Thread,
       DrawClient;

<*FATAL Thread.Alerted*>
<*FATAL Rsrc.NotFound*>
<*FATAL FormsVBT.Error*>
<*FATAL Rd.Failure*>
<*FATAL FormsVBT.Unimplemented*>
<*FATAL TrestleComm.Failure*>
<*FATAL ColorName.NotFound*>

TYPE
  State = {None, EnterItemVertex, Delete};

  Form = FormsVBT.T OBJECT
      list: Diagram.T;
      client: DrawClient.T;
      vbt: DiagramVBT.T;
      gc: Item.GraphicsContext;
      itemType: TEXT;
      state := State.None;
      currentItem: Item.T;
      currentItemVertex: CARDINAL;
      currentItemMinVertex: CARDINAL;
      currentItemMaxVertex: CARDINAL;
      fileName: TEXT;
```

```
      open: TEXT;
    END;

  DVBT = DiagramVBT.T OBJECT
      form: Form;
    END;

  Cleaner = Thread.Closure OBJECT
      fv: FormsVBT.T;
    OVERRIDES
      apply := Clean;
    END;

(* Wait a few seconds before popping down an error message shown to the user *)

PROCEDURE Clean(self: Cleaner): REFANY =
  BEGIN
    Thread.Pause(10.0d0);
    LOCK VBT.mu DO FormsVBT.PopDown(self.fv,"errorChild"); END;
    RETURN NIL;
  END Clean;

(* Popup a window to show for a few seconds the error message *)

PROCEDURE Error(fv: FormsVBT.T; m: TEXT) =
  BEGIN
    FormsVBT.PutText(fv,"error",m);
    FormsVBT.PopUp(fv,"errorChild");
    EVAL Thread.Fork(NEW(Cleaner, fv := fv));
  END Error;

(* Prepare a new "current item". This is done each time a new item
   type or any component of the graphics context is changed. *)

PROCEDURE NewItem(fv: Form) =
  VAR
    i: CARDINAL;
  BEGIN
    fv.state := State.EnterItemVertex;
    fv.currentItemVertex := 0;
    fv.gc.text := FormsVBT.GetText(fv,"labelText");
    fv.gc.color := FormsVBT.GetColorProperty(fv,"color","Color");
```

```
    fv.currentItem := Item.Create(fv.itemType).init(fv.gc);
    fv.currentItem.nbVertex(fv.currentItemMinVertex,i,fv.currentItemMaxVertex);
  END NewItem;

(* Most of the action happens when mouse clicks are received. Depending
   on the current command and state the appropriate action is performed. *)

PROCEDURE Mouse(v: DiagramVBT.T; button: [0..2]; position: RealPoint.T) =
  VAR
    f: Form := NARROW(v,DVBT).form;
    selected: CARDINAL;
  BEGIN
    CASE f.state OF

    (* Nothing to do with this mouse click *)

    | State.None =>
        FormsVBT.PutText(f,"message","Select an action first");

    (* The object pointed to, if any, should be deleted *)

    | State.Delete =>
        IF f.list.select(position,1,selected) <*NOWARN*> THEN
          f.list.rem(selected) <*NOWARN*>;
        END;

    (* We are defining vertices for the current item *)

    | State.EnterItemVertex =>
        f.currentItem.setVertex(f.currentItemVertex,position);
        INC(f.currentItemVertex);

        (* All the vertices were entered, add the new item to the diagram.
           We have all the vertices when the maximum allowed is reached,
           or when the minimum is reached and the middle/right mouse button
           is used. *)

        IF ((button = 0 AND f.currentItemVertex >= f.currentItemMaxVertex)
             OR button # 0) AND
            f.currentItemVertex >= f.currentItemMinVertex THEN
          f.list.add(f.currentItem) <*NOWARN*>;
          NewItem(f);
```

```
        END;
    END;
  END Mouse;

PROCEDURE Quit(fv: FormsVBT.T; <*UNUSED*>name : Text.T;
    <* UNUSED *> cl   : REFANY; <* UNUSED *> time : VBT.TimeStamp) =
  BEGIN
    Trestle.Delete(fv);
  END Quit;

(* Remember if "new", "open", or "save" was entered as a menu item *)

PROCEDURE Open(fv: FormsVBT.T; name : Text.T;
    <* UNUSED *> cl   : REFANY; <* UNUSED *> time : VBT.TimeStamp) =
  VAR
    f: Form := fv;
  BEGIN
    f.open := name;
  END Open;

(* Get the selected file name and perform the selected operation
   (new, open, save). *)

PROCEDURE BrowseFile(fv: FormsVBT.T; <*UNUSED*>name : Text.T;
    <* UNUSED *> cl   : REFANY; <* UNUSED *> time : VBT.TimeStamp) =
  VAR
    f: Form := fv;
    rd: Rd.T;
    wr: Wr.T;
  BEGIN
    FormsVBT.PopDown(fv,"selectFile");
    f.fileName := FormsVBT.GetText(fv,"browseFile");

    (* New: empty the drawing and remember the file name for a later save *)

    IF Text.Equal(f.open,"new") THEN
      f.list.remAll() <*NOWARN*>;

    (* Open: read the diagram content from an existing file. *)

    ELSIF Text.Equal(f.open,"open") THEN
      TRY
```

```
        rd := FileRd.Open(f.fileName);
        f.list.remAll();
        f.list.read(rd);
        Rd.Close(rd);
      EXCEPT ELSE
        Error(fv,"Error opening and reading file " & f.fileName);
      END;

    (* Save the diagram content to a file. *)

    ELSIF Text.Equal(f.open,"save") THEN
      TRY
        wr := FileWr.Open(f.fileName);
        f.list.write(wr);
        Wr.Close(wr);
      EXCEPT ELSE
        Error(fv,"Error opening and writing file " & f.fileName);
      END;
    END;
  END BrowseFile;

(* Change the current color in the graphics context. *)

PROCEDURE SelectColor(fv: FormsVBT.T; name : Text.T;
    cl: REFANY; time: VBT.TimeStamp) =
  BEGIN
    FormsVBT.PutText(fv,"defaultColor",name);
    FormsVBT.PutColorProperty(fv,"color","Color",ColorName.ToRGB(name));
    UpdateGraphicsContext(fv,name,cl,time);
  END SelectColor;

(* Set the current action (state) to "Delete" *)

PROCEDURE DeleteItem(fv: FormsVBT.T; <*UNUSED*>name : Text.T;
    <* UNUSED *> cl   : REFANY; <* UNUSED *> time : VBT.TimeStamp) =
  VAR
    f: Form := fv;
  BEGIN
    f.state := State.Delete;
    FormsVBT.PutText(fv,"message","Click in item to delete");
  END DeleteItem;
```

```
(* Set the current action (state) to entering the item type specified
   as name of the menu button (Rectangle, Label...). *)

PROCEDURE SelectItemType(fv: FormsVBT.T; name : Text.T;
    <* UNUSED *> cl   : REFANY; <* UNUSED *> time : VBT.TimeStamp) =
  VAR
    f: Form := fv;
  BEGIN
    f.itemType := name;
    NewItem(f);
    IF f.currentItemMinVertex = f.currentItemMaxVertex THEN
      FormsVBT.PutText(fv,"message","Click to enter each vertex");
    ELSE
      FormsVBT.PutText(fv,"message",
          "Left click to enter vertex, right click for last");
    END;
  END SelectItemType;

(* Something changed in the graphics context. A new "current item" is
   created thus reading the new values for all the graphics context
   attributes. *)

PROCEDURE UpdateGraphicsContext(fv: FormsVBT.T; <*UNUSED*>name : Text.T;
    <* UNUSED *> cl   : REFANY; <* UNUSED *> time : VBT.TimeStamp) =
  VAR
    f: Form := fv;
  BEGIN
    IF f.state = State.EnterItemVertex THEN NewItem(f); END;
  END UpdateGraphicsContext;

VAR
  (* Read the user interface description. *)

  form: Form := NEW(Form).initFromRsrc("draw.fv",Rsrc.BuildPath("$DRAWPATH",
      DrawBundle.Get()));

BEGIN
  (* Attach the procedures defined above to items in the user interface *)

  FormsVBT.AttachProc(form,"red",SelectColor);
  FormsVBT.AttachProc(form,"green",SelectColor);
  FormsVBT.AttachProc(form,"blue",SelectColor);
```

```
  FormsVBT.AttachProc(form,"delete",DeleteItem);
  FormsVBT.AttachProc(form,"Rectangle",SelectItemType);
  FormsVBT.AttachProc(form,"Label",SelectItemType);
  FormsVBT.AttachProc(form,"labelText",UpdateGraphicsContext);
  FormsVBT.AttachProc(form,"new",Open);
  FormsVBT.AttachProc(form,"open",Open);
  FormsVBT.AttachProc(form,"save",Open);
  FormsVBT.AttachProc(form,"quit",Quit);
  FormsVBT.AttachProc(form,"browseFile",BrowseFile);

  (* Generate dummy events to initialize the current text and the
     current color in the graphics context. *)

  FormsVBT.MakeEvent(form,"labelText",0);
  FormsVBT.MakeEvent(form,FormsVBT.GetText(form,"defaultColor"),0);

  (* Create an initially empty Diagram and a DiagramVBT to display it. *)

  form.list := NEW(Diagram.T).init() <*NOWARN*>;
  form.vbt := NEW(DVBT, form := form).init(form.list);
  form.vbt.attachProc(Mouse);
  FormsVBT.PutGeneric(form,"drawArea",form.vbt);

  (* Enable remote mirroring of the diagram *)

  form.client := NEW(DrawClient.T).init(form,form.list);

  (* Show the user interface. *)

  Trestle.Install(form);
  Trestle.AwaitDelete(form);
END Draw.
```

5.4.5 Discussion

The example presented here is not a complete diagram editor. More attributes are required in the graphics context, several other graphical items should be added, and a richer interaction should be available (rubber banding, dragging vertices of existing items, changing the attributes of existing items, undo/redo log...). All these, however, would make the code longer and more difficult to understand.

The modules in the diagram editor make extensive use of callbacks (or

closures). This way, the different modules interact through a well defined mechanism instead of being interdependent. For instance, the DiagramVBT.T can be notified of any modification to a Diagram.T without special support besides the notification closures.

The objects in the editor store all the relevant data instead of using global variables. This way, very few modifications would be required to edit the same or different diagrams in several editing windows.

Little or no locking is performed in the diagram editor. This is correct because all the events are triggered by trestle events (mouse clicks in menu buttons or in the drawing area, key clicks...) which are internally serialized through the VBT.mu mutex. The only asynchronous event is the error window which is popped down 10 seconds after being displayed using a separate thread, and there the editor locks VBT.mu. A more elaborate locking strategy would be required if several asynchronous threads (remote users, background computations...) would modify the diagram.

Chapter 6

Distributed and Persistent Objects

6.1 Network Objects

Distributed programming involves programs running on a number of networked computers. The reasons for running on several computers may be:

- needed resources such as files, printers, databases, humans may be dispersed on several computers.
- using several computers in parallel allows more work to be done in a given amount of time.
- when a single computer fails, the other computers may be able to continue the task.

It is possible to have several programs cooperating in this way using the low level network communication primitives, for instance the Berkeley Sockets. The X windows graphics server accepts socket connections from remote programs, receives graphics commands and sends out events (key or mouse input) on these connections. Nonetheless, this is relatively painful to implement.

6.1.1 Remote procedure calls

To alleviate the difficulty of developing client server distributed applications, tools and libraries such as the Sun Remote Procedure Calls (RPC)

have been developed. It is then relatively easy to separate a number of procedures from a program and have them transparently running on a remote computer; the procedures in the remote program are accessed almost as easily as any procedure in the local program.

To achieve this, each procedure is assigned a unique identifier, and procedures to write and read the arguments and return value to/from the network must be supplied. A tool named *rpcgen* reads a file describing the remote procedure calls (their arguments and return values) and automatically generates the procedures that cross the network boundary. Each interface (set of remote procedures) is registered at a central coordinator which allocates a unique identifier for it.

The *rpcgen* tool also creates a main program for the remote procedure server, which registers the server to the portmap daemon, accepts connections from remote programs, and receives and executes remote calls. The programmer then simply has to implement these calls. Finally, rpcgen creates procedure stubs that relay their arguments to the procedure server across the network.

A common problem when passing arguments to remote procedures is deciding if a copy of the argument must be transmitted or if a reference (pointer) is sufficient. When a reference is transmitted, the remote procedure must not try to access it as a local reference. Furthermore, in a garbage collected environment, it is dangerous to freely transmit references. The transmitted reference may become invalid if the garbage collector moves the object to another address. Moreover, the garbage collector must not destroy the object while references are still being held in remote programs.

To make a remote call, a program gets a connection to the procedure server; the portmap daemon is queried and returns the address of the server, given the interface number and procedure name. The remote procedures are then accessed through the stubs, almost like local procedures, with the difference that the handle to the procedure server must be added to the list of arguments. The stubs relay the arguments to the procedure server, which executes the remote procedure call, and come back with the return value received.

Building distributed programs in this way is somewhat simpler than using network connections directly. However, it is still more complex than local calls. The connection to the remote procedure server must be established. Remote calls are likely to fail from time to time because of network failures and each call must be checked for errors. Finally, it is relatively difficult to build programs which do more than just wait for incoming requests. Indeed, Sun Remote Procedure Calls were developed for an environment without multi-threading. With threads, you may have one or more threads awaiting asynchronous requests while another thread carries its own com-

putation.

6.1.2 Remote Method Invocation on Network objects

Other libraries available for building distributed applications include DCE RPC, CORBA and OLE. They are more recent and more sophisticated than Sun's RPC. However, they are not as well integrated and easy to use as systems based on the concept of Remote Method Invocation (RMI).

Indeed, in remote procedure calls, a server handle must be supplied to the call. With objects, however, it is possible to have two derived types from a base type, a concrete type which provides data members and implements the methods, and a proxy type which contains the server handle and has methods which simply relay the arguments to the concrete object in a remote, server, process. The interesting property is that both types of objects are accessed in the same way, through their common ancestor type. Moreover, the file describing remote procedures, that must be written by the programmer, is not required if more powerful tools generate automatically the procedures to write and read objects to/from the network, directly from their declarations.

The DEC Systems Research Center came up with a clever design for network objects along these lines, as described in research report 115 *Network Objects*. They build upon existing Modula-3 functionality: threads, garbage collection, weak references, run time type information, procedures to read/write objects to streams (disk, network connection...) and Modula-3 code parsing tools.

The group of Jim Waldo at SunLabs in Massachusetts used Modula-3 network objects for some time and developed a Network Object Activation Daemon. When Java came out from SunLabs in California, they implemented the java.rmi package. Java already supported CORBA but the Java designers recommend CORBA when required by cross-language constraints, and RMI for functionality and ease of use. The java.rmi package, along with the object serialization support, is almost identical to the Modula-3 Network Objects and Pkl serialization module. The main difference lies in the use of dynamic loading and multiple interface inheritance in Java.

In Modula-3 Network Objects, objects for which proxys are needed, because they should be accessible remotely, must inherit from the NetObj type. The base type must be a pure object type: only methods, no data members. From there, a concrete type may be derived which contains data members and provides methods to operate on the data members. A tool named *stubgen*, based on the Modula-3 code parsing library M3tk, parses the base type declaration and automatically generates the derived proxy

type.

The NetObj runtime module then takes care of all network objects communications. Whenever arguments are sent to remote processes, ordinary objects are copied (along with the objects they are refering to, recursively). Network objects, however, are not copied but replaced by network references (object identifier within the process, process identifier within a computer and computer identifier within the Internet). Network references are created for concrete objects. Proxys only store the network reference of the corresponding concrete object. A table is maintained to store all the concrete objects for which network references were exported. A concrete object should not be garbage collected until all exported network object references have been dropped by the remote processes.

Whenever arguments are received by the NetObj runtime, network references must be replaced by network objects. If the network reference corresponds to a concrete object within this process, it is readily found in the table of exported concrete objects. Otherwise, a proxy must be created for the remote object, if it does not already exist. A table of weak references to proxys created is thus maintained. The first time a network reference for a given concrete object in a remote process is received, a proxy is created for it. When a network reference to the same object is received later, the proxy is simply found in the table and reused.

When a proxy becomes unused in a process, it gets garbage collected. The weak reference in the proxy table gets cleared and its cleaning procedure is called. This way, a message to the remote process can be sent to inform the concrete object that its network reference is not used any more in this process.

Often, the set of types defined in each process is not the same. In such a case, when a network reference is received, a proxy of the most specialized type known to this process is created. Suppose that in process A Rectangle inherits from Shape and NetObj but process B only knows Shape which inherits from NetObj. When a network reference to a Rectangle is received by B, a proxy of type Shape will be created. However, if that network reference is further sent from B to process C which knows about Rectangle, a proxy of type Rectangle in nevertheless created.

Each process may be both a client and a server. Indeed, a process may carry its own computation and do remote method calls while at the same time asynchronously receiving remote method calls. This is possible because the NetObj runtime starts its own execution thread. Furthermore, it usually maintains a pool of available threads such that a number of remote method calls may be handled simultaneously by several threads. This is important because otherwise a remote method call involving a long computation would block all the other remote method calls, if a single

thread was used.

The remaining problem is how are network objects exported in the first place. A special process, the *netobjd* daemon, maintains a table of exported network objects and their names. A process may then export one of its concrete objects to the daemon and give it a name. Another process may then retrieve by name this object (network reference) from the daemon. The current implementation does not include access control.

6.2 Persistent Objects

Many programs read information from persistent storage, perform some processing and then write the result back to persistent storage. Defining the format of these input and output files on the persistent storage medium is tedious and cumbersome.

Proponents of the persistent programming paradigm view programs as having volatile objects, which exist only during the current execution, and persistent objects which continue to exist between program executions. The persistent objects are automatically retrieved from the disk when actually accessed in the program, and modifications to these objects are automatically written back to disk as they happen or before the program terminates. This relieves the programmer from a number of low level considerations but involves numerous interesting technical issues.

6.2.1 Translation between memory and disk representations

The automatic translation from the memory to the disk representation implies that each object has a type tag and that the run time libraries have information about the type. This run time information must list the fields, their position and their type. Simple fields like integers, floats or character strings are simply converted to a suitable disk representation (binary or ASCII with a neutral byte ordering). Pointers must be replaced by an adequate disk representation, usually some unique integer identifier.

Several objects written together to disk are often called a pickle. If objects in a pickle contain pointers to other objects, not in the pickle, these pointers will have no meaning when the objects are read back. That is unless each object is manually assigned a unique identifier, unique among all pickles and program invocations. For this reason, many languages, like Modula-3 and Eiffel, provide pickling procedures that receive an object as argument. This object and all those recursively reachable from that object will be included in the pickle. This way, there are no pointers to objects

not in the pickle. Each object in the pickle is assigned a unique identifier. When reloading a pickle, all the pointers are converted to the new objects addresses.

Pointers to procedures are usually treated differently. The source code may not be available and the executable code cannot run on other architectures. Instead, the procedure name and arguments types are stored (perhaps as a compressed hashing code). Upon reloading, if the same procedure exists in the reading program, its address is used.

6.2.2 Lazy loading and unloading

Writing and reading objects explicitly, through the pickling mechanisms, is easier than defining input and output file formats. The next step in this direction is transparent loading on demand. Pointer variables are initialized with the desired globally unique object identifiers. Then, when (and if) these variables are accessed, the corresponding object is loaded from disk. This object in turn references other objects but these will be loaded only if accessed.

The advantage of this is that objects are accessed just as if they were all in memory although only the accessed objects are actually loaded. The problem is to implement this loading on demand efficiently and to assign unique global identifiers to all objects.

Among the possible implementations, there are mainly three in use. Each pointer to an object that may be on disk (often called smart pointer) contains the unique identifier, the memory address and a flag indicating if the memory address is known. Upon accessing the smart pointer, it is checked if the memory address is known, in which case the access proceeds. Otherwise, a table is searched to find if the object has been loaded and its memory address; the memory address field of the smart pointer can then be set and used. If the object is not loaded, it gets loaded and inserted in the table.

The overhead of this implementation is one test for each access plus searching through the table and loading the object upon the first access. It is relatively simple to implement and the overhead is small. To unload an object, all the smart pointers refering to it must have their flag reset to indicate that the memory address is invalid. Thus, each object must keep a list of the associated smart pointers.

The second possible implementation is to use a virtual address as object identifier within the program. This virtual address is the position where the object will be loaded, when (and if) required. Upon accessing a pointer, if the object is not there, a page fault occurs. The page fault interrupt service routine then loads the needed object at that address.

The overhead of this implementation is to service the interrupt and load the object upon the first access, and nothing for the subsequent accesses. Interrupt service routines are difficult to write and often non portable. Furthermore, interrupt service routines are costly in CPU time on many operating systems. Therefore, even though this implementation has no overhead once the object is loaded, the higher loading cost often makes it no more effective than the preceeding one. Unloading objects does not involve special actions. The virtual address remains reserved for the object in case it gets reloaded.

A third possible implementation is to only store the unique object identifier in the smart pointers. Each time an access is made, the table is searched to find the object in memory and load it if required. The overhead is one hash table access for each smart pointer access. Unloading objects only requires removing them from the table. This method is very simple to implement and very flexible but carries a relatively large overhead upon each access.

6.2.3 Persistence through reachability

In most implementations, persistent and volatile objects are different. Persistent objects have unique identifiers and are only accessed through smart pointers. Other implementations are more ambitious and objects become persistent simply by being reacheable from the persistent *roots*. The persistent roots are variables declared as leading to persistent objects.

Thus, whenever a pointer to volatile objects is stored in a persistent object, these become persistents. The mechanisms involved are refinements over those presented above.

6.2.4 Updating the persistent storage

When the program ends, all persistent objects may be written back to persistent storage. Similarly, when an object is unloaded, it may be written back to disk. Often, most objects are unmodified and there is no need to update them in the persistent store. In some implementations, every object has a flag indicating if they were modified. This flag is set by all the *state modifying* methods of the object. Then, only modified objects are written back, which saves some time.

In other implementations, a flag is maintained for each memory page. Memory pages are initially set read only. When an object on a page is modified, a memory protection violation interrupt occurs. The interrupt service routine sets the flag for the page and removes the read only protection. For this, the persistent store must work at the granularity of memory

pages.

An important problem is the update frequency. If modified objects are only written back at a later time, a lot of modifications may be lost if the program suddenly crashes. Thus, in some cases, every time an object is modified its state is written back to the persistent store.

As an optimization, it is possible to write the complete object state once in a while (checkpoint) and only write the state changes (in a change log) when an object is updated through *state modifying* methods. If the change log becomes too long, a new checkpoint should be written and the change log emptied. When a crash occurs, the last checkpoint and the change log are used to get back to the exact state prior to the crash.

6.3 Object Oriented Databases

The relational model is well established in the database world. Its simplicity makes it easy to understand, and powerful tools exist to support relational databases. There are, however, many applications where the limitations of the relational model make the mapping of the data model to the database model difficult, and even severely affect the performance.

Object oriented databases are appearing mainly from two sources. A number of relational databases are being modified into object oriented databases by using an extended relational model. These usually perform well and offer all the traditional database services such as recovery, transactions and query optimizations; however, they do not necessarily interface well to object oriented languages.

In other cases, an object oriented language gets persistent objects to which database services such as indexing and querying are added. The integration with the language is good but the database services are usually limited.

6.3.1 Fized size limitations

Most relational databases only support fixed size objects. Each field must have a fixed length. Therefore, if the family name field size is too short, several names will not fit, but if it is too long a lot of space is wasted. Furthermore, the number of fields is fixed. Since, for instance, in some cases co-owners own a home, the ownership information must be stored in a separate database from the homes. One database would store the homes, another the persons and a third would store the ownership relations (records with a person identifier and a home identifier). Object oriented databases usually dont have this fixed size limitation.

The effect of fixed field length limitation is wasted space and a larger database, hence a smaller fraction of it remains in high speed memory. The fixed number of fields results in spreading the information across several database items, hence requiring several disk accesses instead of one. In applications with complex modeling requirements, keeping in a single database item the information about a home, a medical patient, or a mechanical part can speed up by 10 or more the retrieval time.

Variable length fields and repeatable fields make the database support tools, including indexing, somewhat more complex but much more flexible.

6.3.2 Unique object identifiers

The basis of the relational model is using the primary key to relate items between databases. For example, each entry in a home database would be assigned a unique identifier (name or number). This identifier can then be used in other databases to specify a particular home. These identifiers are defined and managed explicitly. Several object oriented databases automatically assign a unique object identifier to each entry; these can then be used to refer to objects in the database.

The advantages of unique object identifiers are that they are generated automatically and that they are designed to be accessed very quickly, often through memory based hash tables. The disadvantage of automatically generated identifiers is when exceptional conditions require humans to examine these identifiers, for example if the database is corrupted or when two databases need to be merged. Indeed, when two separate databases are merged together, the same identifiers may be been used in each. Something must be done to translate object identifiers such that the new identifiers are unique with respect to the merged database.

6.3.3 The programming language versus database gap

With variable length fields, repeatable fields and object identifiers, it becomes feasible to model not only structures but also lists, arrays and pointers. This richer database model makes it possible to have a single model used both for the database and for the in memory representation of objects, when these are manipulated by regular programs. Having a single data model, conversion routines between the database and the in memory representation may be avoided.

To match object oriented languages, single or multiple inheritance of fields is easily added. Methods, although there are technical problems with this, may also be added to the database model.

6.3.4 Storing procedures in the database

Pointers to procedures, for instance in a method table, may in principle be stored in the database. If these procedures are in an interpreted language, this is fairly easy. The procedure source code is stored somewhere in the database and may be retrieved to be interpreted.

Source code in a safe programming language may also be stored in the database and be compiled and dynamically linked upon retrieval; not all operating systems and languages support such dynamic linking. Unsafe source code, which may destroy the application through incorrect pointer arithmetic, may be loaded in the same way only if trusted.

When dynamic compilation and linking is not feasible or too slow, it is also possible to store only procedure identifiers and hope that these procedures are available in the program retrieving the procedure identifiers. The program, for this purpose, must maintain a table of addresses of available procedures, accessed using the procedure identifiers.

6.4 Distributed Persistent Objects in Modula-3

The diagram editor examined in earlier sections uses distributed and persistent objects. Network objects allow remote access to a diagram while persistent objects insure that the up to date diagram is always available on persistent storage even if the diagram server crashes.

6.4.1 Network objects

The diagram inherits from NetObj.T. This way, it may be remotely accessible, as demonstrated by the DrawClient module and the DrawServer program. Indeed, with network objects, it is possible to call the methods of objects which are in another process, possibly on a remote server. The *stubgen* tool and *netobj* library take care of relaying the methods and arguments between programs.

The DrawServer *exports* a diagram which may be accessed by the diagram editor. The remote diagram is used to mirror the currently edited diagram. The connection to DrawServer and the mirroring is performed by the DrawClient module using the Diagram.T notification services.

```
(* This interface adds the necessary support to the Draw program for
   remote mirroring. It is possible to connect to a remote "DrawServer",
```

```
   mirror the current drawing to the server, and later recover the
   drawing from the server. *)

INTERFACE DrawClient;

IMPORT FormsVBT, Diagram;

TYPE
  T <: Public;
  Public = OBJECT METHODS
      init(fv: FormsVBT.T; list: Diagram.T): T;
    END;

(* The call "client.init(fv,list)" registers procedures to "server"
   related menu items in "fv". These procedures can connect to a remote
   server, and mirror/recover "list" from the server. It has limited
   functionality and is more intended as a simple example of using stable
   and network objects. *)

END DrawClient.

MODULE DrawClient;

IMPORT FormsVBT, VBT, Text, Diagram, IP, NetObj, TCPNetObj,
       Thread, Item, Draw, Fmt;

REVEAL
  T = Public BRANDED OBJECT
      fv: FormsVBT.T;
      list: Diagram.T;
      cl: DiagramClosure;
      serverName: TEXT;
      port: CARDINAL;
      server: Diagram.Public;
      mirror := FALSE;
    OVERRIDES
      init := Init;
    END;

TYPE
  DiagramClosure = Diagram.Closure OBJECT
      client: T;
```

```
    OVERRIDES
      add := Add;
      rem := Rem;
      remAll := RemAll;
    END;

  FVClosure = FormsVBT.Closure OBJECT
      client: T;
    END;

<*FATAL FormsVBT.Error*>
<*FATAL FormsVBT.Unimplemented*>
<*FATAL Thread.Alerted*>

(* Register with "list" to be notified of any modifications. Attach
   procedures to the "connect", "disconnect", "startMirror", "stopMirror",
   and "recover" menu buttons. *)

PROCEDURE Init(self: T; fv: FormsVBT.T; list: Diagram.T): T =
  BEGIN
    self.fv := fv;
    self.list := list;
    self.cl := NEW(DiagramClosure, client := self);
    self.list.attach(self.cl) <*NOWARN*>;
    FormsVBT.Attach(fv,"confirmConnect",
        NEW(FVClosure, client := self, apply := Connect));
    FormsVBT.Attach(fv,"cancelConnect",
        NEW(FVClosure, client := self, apply := CancelConnect));
    FormsVBT.Attach(fv,"disconnect",
        NEW(FVClosure, client := self, apply := Disconnect));
    FormsVBT.Attach(fv,"startMirror",
        NEW(FVClosure, client := self, apply := StartMirror));
    FormsVBT.Attach(fv,"stopMirror",
        NEW(FVClosure, client := self, apply := StopMirror));
    FormsVBT.Attach(fv,"recover",
        NEW(FVClosure, client := self, apply := Recover));
    RETURN self;
  END Init;

(* An item was added to the list. If mirroring, send it to the remote
   server. *)
```

```
PROCEDURE Add(self: DiagramClosure; item: Item.T) =
  BEGIN
    IF self.client.server # NIL AND self.client.mirror THEN
      TRY
        self.client.server.add(item);
      EXCEPT ELSE
        Draw.Error(self.client.fv,
            "Error while adding to mirror, disconnecting");
        self.client.server := NIL;
      END;
    END;
  END Add;

(* An item was deleted. If mirroring, send it to the remote server. *)

PROCEDURE Rem(self: DiagramClosure; i: CARDINAL; <*UNUSED*>item: Item.T) =
  BEGIN
    IF self.client.server # NIL AND self.client.mirror THEN
      TRY
        self.client.server.rem(i);
      EXCEPT ELSE
        Draw.Error(self.client.fv,
            "Error while removing from mirror, disconnecting");
        self.client.server := NIL;
      END;
    END;
  END Rem;

(* The list was emptied. If mirroring, notify the remote server. *)

PROCEDURE RemAll(self: DiagramClosure) =
  BEGIN
    IF self.client.server # NIL AND self.client.mirror THEN
      TRY
        self.client.server.remAll();
      EXCEPT ELSE
        Draw.Error(self.client.fv,
            "Error while removing all from mirror, disconnecting");
        self.client.server := NIL;
      END;
    END;
  END RemAll;
```

```
(* Connect to the server specified. Import the corresponding network
   object. Mirroring is not initially enabled. *)

PROCEDURE Connect(self: FVClosure; fv: FormsVBT.T; <*UNUSED*>name: Text.T;
    <*UNUSED*>time: VBT.TimeStamp) =
  VAR
    addr: IP.Endpoint;
  BEGIN
    WITH client = self.client DO
      FormsVBT.PopDown(fv,"connectServer");
      client.serverName := FormsVBT.GetText(fv,"server");
      client.port := FormsVBT.GetInteger(fv,"port");
      addr.port := client.port;
      TRY
        IF IP.GetHostByName(client.serverName, addr.addr) THEN
          client.server := NetObj.Import("DrawServer",TCPNetObj.Locate(addr));
        ELSE
          Draw.Error(fv,"Cannot locate host " & client.serverName);
        END;
      EXCEPT ELSE
        Draw.Error(fv,"Cannot connect to " & client.serverName &
            " at port " & Fmt.Int(client.port) & ", daemon not running?");
        client.server := NIL;
      END;
    END; (* WITH *)
  END Connect;

(* After all we did not want to connect *)

PROCEDURE CancelConnect(<*UNUSED*>self: FVClosure; fv: FormsVBT.T;
    <*UNUSED*>name: Text.T; <* UNUSED *>time: VBT.TimeStamp) =
  BEGIN
    FormsVBT.PopDown(fv,"connectServer");
  END CancelConnect;

(* Cut the connection. Stop mirroring first. *)

PROCEDURE Disconnect(self: FVClosure; fv: FormsVBT.T; name: Text.T;
    time: VBT.TimeStamp) =
  BEGIN
    IF self.client.mirror THEN StopMirror(self,fv,name,time); END;
```

```
    self.client.server := NIL;
  END Disconnect;

(* Start mirroring (must be already connected). The content of the
   diagram is sent to the remote server. *)

PROCEDURE StartMirror(self: FVClosure; fv: FormsVBT.T;
    <*UNUSED*>name: Text.T; <* UNUSED *>time: VBT.TimeStamp) =
  BEGIN
    WITH client = self.client DO
      IF client.server = NIL THEN
        Draw.Error(client.fv,"Not connected!");
        RETURN;
      END;

      IF client.mirror THEN RETURN END;
      client.mirror := TRUE;

      TRY
        client.server.remAll();
        FOR i := 0 TO client.list.size() - 1 DO
          client.server.add(client.list.get(i));
        END;
      EXCEPT ELSE
        Draw.Error(fv,"Error while starting to mirror, disconnecting");
        client.server := NIL;
      END;
    END;
  END StartMirror;

(* Stop mirroring. The remote server will continue to store the current
   content but will not receive further updates. *)

PROCEDURE StopMirror(self: FVClosure; <*UNUSED*>fv: FormsVBT.T;
    <*UNUSED*>name: Text.T; <* UNUSED *>time: VBT.TimeStamp) =
  BEGIN
    self.client.mirror := FALSE;
  END StopMirror;

(* Empty the list and copy the content of the list on the remote server
   to it. *)
```

```
PROCEDURE Recover(self: FVClosure; fv: FormsVBT.T;
    <*UNUSED*>name: Text.T; <* UNUSED *>time: VBT.TimeStamp) =
  BEGIN
    WITH client = self.client DO
      IF client.server = NIL THEN
        Draw.Error(fv,"Not connected !");
        RETURN;
      END;

      IF client.mirror THEN
        Draw.Error(fv,"Cannot recover while mirroring");
        RETURN;
      END;

      TRY
        client.list.remAll();
        FOR i := 0 TO client.server.size() - 1 DO
          client.list.add(client.server.get(i));
        END;
      EXCEPT ELSE
        Draw.Error(fv,"Error while recovering from server, disconnecting");
        client.server := NIL;
      END;
    END;
  END Recover;

BEGIN
END DrawClient.
```

6.4.2 Stable persistent objects

Migrating objects between the disk and memory, using a library such as Pickle, is a first step towards offering object oriented services similar to traditional databases. The next step is insuring that all modifications to the persistent objects are automatically propagated to disk even if the program or computer suddenly stops. Recovering from disk failures is another problem usually dealt with through disk mirroring or frequent backups to tape archives.

The *SmallDB* and *Stable* libraries, along with the *StableGen* tool may be used to provide persistent objects with little efforts. To add persistence to an object type, its declaration is passed to StableGen which generates a new derived type with the following methods and procedures added:

- Init: create and initialize a persistent object. If the object is new, a file is created, otherwise the object state is recovered from the file.
- Dispose: stop keeping a persistent state for the object.
- Checkpoint: write the object state to disk. Empty the log of changes since the last checkpoint.

Furthermore, all the methods which affect the object state are overridden to first write the method name and arguments to a log, then call the original method. Thus, when a persistent object is initialized after a crash, its state is restored from the last checkpoint and all the method calls stored in the log are redone to get back to the exact state prior to the crash.

In the diagram editor example shown in earlier sections, the DrawServer derives a persistent diagram from Diagram.T and exports such a persistent diagram. Each diagram modification mirrored from the drawing editor to the DrawServer program is thus automatically stored in persistent storage. It can thus be recovered even if the DrawServer program is killed or crashes and is later restarted.

```
(* This module exports a Diagram network object. Items may be added/removed
   remotely from the diagram. Furthermore, this diagram is a persistent
   object which retains its state across program invocations and crashes. *)

MODULE Main;

IMPORT NetObj, TCPNetObj, Wr, Rd, Stdio, Item,
       Rectangle <*NOWARN*>, Label <*NOWARN*>, Params, StableDiagram,
       Process, Text, TextRd, Lex, Fmt, Thread, FloatMode, StableError;

TYPE
  StableList = StableDiagram.T OBJECT
      nbLogItems: CARDINAL := 0;
    OVERRIDES
      writeCheckpoint := WriteCheckpoint;
      readCheckpoint := ReadCheckpoint;
      add := Add;
      rem := Rem;
      remAll := RemAll;
    END;

<*FATAL Thread.Alerted*>
```

```
<*FATAL Rd.Failure*>
<*FATAL Wr.Failure*>
<*FATAL FloatMode.Trap*>
<*FATAL NetObj.Error*>
<*FATAL TCPNetObj.Failed*>
<*FATAL Lex.Error*>
<*FATAL StableError.E*>

(* Show a message *)

PROCEDURE Msg(wr: Wr.T; m: TEXT) =
  BEGIN
    Wr.PutText(wr,m); Wr.Flush(wr);
  END Msg;

(* Report the command line usage *)

PROCEDURE Usage() =
  BEGIN
    Msg(Stdio.stderr,"DrawServer [-help] [-p port]\n");
  END Usage;

(* Complain about missing argument *)

PROCEDURE MissingArgument(option: TEXT) =
  BEGIN
    Msg(Stdio.stderr,"DrawServer: missing argument for option " &
        option & "\n");
    Process.Exit(1);
  END MissingArgument;

(* Read the command line options. *)

PROCEDURE ReadOptions() =
  VAR
    option: TEXT;
    i := 1;
  BEGIN
    WHILE i < Params.Count DO
      option := Params.Get(i);
      INC(i);
      IF Text.Equal(option,"-help") THEN
```

```
        Usage();
        Process.Exit();
      ELSIF Text.Equal(option,"-p") THEN
        IF i >= Params.Count THEN MissingArgument("-p"); END;
        port := Lex.Int(TextRd.New(Params.Get(i)));
        INC(i);
      ELSE
        Usage();
        Process.Exit(1);
      END;
    END;
  END ReadOptions;

(* An item is added, write information about it. *)

PROCEDURE Add(self: StableList; item: Item.T) =
  BEGIN
    Msg(out,"Item added: ");
    item.write(out); Wr.Flush(out);
    StableDiagram.T.add(self,item);
    IncNbLogItems(self);
  END Add;

(* An item is removed, write information about it. *)

PROCEDURE Rem(self: StableList; i: CARDINAL) =
  BEGIN
    Msg(out,"Item removed: ");
    self.get(i).write(out); Wr.Flush(out);
    StableDiagram.T.rem(self,i);
    IncNbLogItems(self);
  END Rem;

(* All items were removed, write information about it. *)

PROCEDURE RemAll(self: StableList) =
  BEGIN
    Msg(out,"List emptied\n\n");
    StableDiagram.T.remAll(self);
    IncNbLogItems(self);
  END RemAll;
```

```
(* Every so often write a checkpoint. This prevents the log from
   growing indefinitely by starting with a new "checkpoint" and an
   empty log. *)

PROCEDURE IncNbLogItems(self: StableList) =
  BEGIN
    INC(self.nbLogItems);
    IF self.nbLogItems > 20 THEN
      StableDiagram.Checkpoint(self);
      self.nbLogItems := 0;
    END;
  END IncNbLogItems;

(* The checkpoint is performed with the write method in binary. *)

PROCEDURE WriteCheckpoint(self: StableList; wr: Wr.T) =
  BEGIN
    self.write(wr, binary := TRUE);
  END WriteCheckpoint;

(* Reading back the checkpoint uses the corresponding read method. *)

PROCEDURE ReadCheckpoint(self: StableList; rd: Rd.T): StableDiagram.T =
  BEGIN
    self.read(rd, binary := TRUE);

    (* We need to initialize the transient state because it is
       accessed while the log is processed. *)
    self.initT();
    RETURN self;
  END ReadCheckpoint;

CONST
  name = "StableDrawList";

VAR
  port: CARDINAL := 2288;
  server: StableList;
  recovered: BOOLEAN;
  out := Stdio.stdout;

BEGIN
```

```
  ReadOptions();

  (* The diagram state is stored in the "name" file. The old state is
     reread from disk if any. *)

  server := NEW(StableList).init(name,recovered);

  IF recovered THEN
    Msg(out,"Recovered Stable Draw list from " & name & "\n\n");
  ELSE
    Msg(out,"New Stable Draw list created in " & name & "\n\n");
    (* Initialize the stable state from scratch. *)
    server.initS();
  END;
  server.initT();

  (* Export the diagram as a network object. *)

  NetObj.Export("DrawServer", server, TCPNetObj.Listen(port));
  Msg(out,"Draw list exported on port " & Fmt.Int(port) & "\n\n");

  (* Wait while remote connections are processed by the network object
     runtime. A loop over the Pause would be in order for a permanently
     running daemon. Stopping after a few hours is perhaps better for
     a daemon which is not expected to be needed for longer than that. *)

  Thread.Pause(50000.0D0);
END Main.
```

6.4.3 Indexing

For small databases that fit into memory, indexes are easily built using the Table (hash table) and SortedTable (sorted tree) interfaces. In the diagram example, indexes for the X and Y upper left coordinates of the items could be implemented using sorted tables that map the coordinate (REAL) to a cardinal item identifier. These indexes could be added as follows.

```
REVEAL T = Public BRANDED OBJECT
    items: RefSeq.T;
    indexX: RealItemIDSortedTable.T;
    indexY: RealItemIDSortedTable.T;
```

Then, each time new items are added in Diagram.Add, they must be entered in the indexes.

```
self.items.addhi(item);
EVAL self.indexX.put(item.boundingBox().west,item.id);
EVAL self.indexY.put(item.boundingBox().north,item.id);
```

Similarly, when an item is deleted, the index entries are removed in Diagram.Rem.

```
self.items.remhi();
EVAL self.indexX.delete(item.boundingBox().west,it);
EVAL self.indexY.delete(item.boundingBox().north,it);
```

The Queries on these indexes would return the sorted sequence of items in a speficied coordinate range.

```
PROCEDURE FindItems(tbl: RealItemIDSortedTable.T; min, max: REAL):
    IDSequence.T =
  VAR
    seq := NEW(IDSequence.T).init();
    iterator := tbl.iterateOrdered();
    cond := TRUE;
    key: REAL;
    val: Item.ID;
  BEGIN
    cond := iterator.seek(min);
    LOOP
      IF iterator.next(key,val) THEN
        IF key <= max THEN seq.addhi(val);
        ELSE EXIT;
        END;
      ELSE EXIT;
      END;
    END;
    SortIDSequence(seq);
    RETURN seq;
  END FindItems;
```

The intersection and union of such sequences may then be used to construct more complex queries.

```
PROCEDURE Intersect(seq1, seq2: IDSequence.T): IDSequence.T =
```

```
  VAR
    outSeq: NEW(IDSequence.T).init();
    i1 := 0;
    i2 := 0;
    end1 := seq1.size();
    end2 := seq2.size();
  BEGIN
    WHILE i1 < end1 AND i2 < end2 DO
      WITH id1 = seq1.get(i1), id2 = seq2.get(i2) DO
        IF id1 < id2 THEN
          INC(i1);
        ELSIF id2 < id1 THEN
          INC(i2);
        ELSE
          outSeq.addhi(id1);
          INC(i1);
          INC(i2);
        END;
      END;
    END;
  END Intersect;

PROCEDURE Union(seq1, seq2: IDSequence.T): IDSequence.T =
  VAR
    outSeq: NEW(IDSequence.T).init();
    i1 := 0;
    i2 := 0;
    end1 := seq1.size();
    end2 := seq2.size();
  BEGIN
    WHILE i1 < end1 AND i2 < end2 DO
      WITH id1 = seq1.get(i1), id2 = seq2.get(i2) DO
        IF id1 < id2 THEN
          outSeq.addhi(id1);
          INC(i1);
        ELSIF id2 < id1
          outSeq.addhi(id2);
          INC(i2);
        ELSE
          outSeq.addhi(id1);
          INC(i1);
          INC(i2);
```

```
      END;
    END;
  END;

  WHILE i1 < end1 DO
    outSeq.addhi(seq1.get(i1));
    INC(i1);
  END;

  WHILE i2 < end2 DO
    outSeq.addhi(seq2.get(i2));
    INC(i2);
  END;
END Union;
```

6.4.4 Query Processing

Database systems often offer a specialized query language. Queries are parsed and optimized in order to translate them into efficient low level database operations. For example, when looking for all items with an X coordinate between 0.1 and 0.11 and a Y coordinate between 0 and 1, at least three possible paths may lead to the desired result.

- Find the items between 0.1 and 0.11 in the X index, then the items between 0 and 1 in the Y index and find the intersection between these two lists of items.

- Find the items between 0.10 and 0.11 in the X index and for each of these retain only the ones where Y is between 0 and 1.

- Find the items between 0 and 1 in the Y index and for each of these retain only the ones where X is between 0.10 and 0.11.

If the database contains 1000 items uniformly distributed in the square defined by X and Y between 0 and 1, there will be approximately 10 items with X between 0.10 and 0.11 and 1000 items with Y between 0 and 1. Thus, the first solution finds a list of 10 and a list of 1000 and performs the intersection. The third solution is no better since it finds a lists of 1000 and checks each item for its X coordinate. The second solution in this case is clearly superior since it obtains a list of 10 items through the X index and checks the Y coordinate of each.

6.4.5 Transactions

Traditional sophisticated databases include support for atomic transactions. A transaction may be initiated, then a sequence of updating commands are issued and finally the transaction is ended with a *commit* or *abort* command. This way, the sequence of updating commands becomes atomic, they are all performed together if the *commit* is successfull or none of them is performed if the database crashes, the *commit* is not successfull or *abort* is issued.

In stable persistent objects, methods are atomic (logged atomically) but there is no explicit support for transactions. The same effect however may be obtained in different ways. For instance, all updating actions which need to be performed together may be grouped in a single atomic method.

Another possibility is to keep an undo list. The undo list starts recording when the `begin_transaction` method is called and is emptied after a successfull `commit`. However, when recovering from a crash or when the transaction is aborted the undo list would be used to recover the state prior to the `begin_transaction`.

6.4.6 Optimized loading and unloading

In the SmallDB and Stable libraries, no special mechanism is required to fit all the database in memory. The database must fit in virtual memory (less than 100MB or so) and the usual operating system swapping mechanisms are used to control which pages remain in main memory and which pages are left on disk. The performance is excellent since all the database is already converted to the in memory format, most queries are handled directly in main memory, and a single disk write is needed for each database update.

For larger databases (hundreds of megabytes to tens of gigabytes), however, special implementation tricks must be used. The loading, caching and unloading of database pages, from disk files to virtual memory, is managed directly by the database server. Objects are laid out on pages in such a way that little or no translation is required when they are loaded or unloaded (no pointers are used). The placement of objects on pages is optimized to reduce the number of pages to load for typical queries. For example, indexes are organized as balanced trees where each tree node fits exactly on a page.

In many cases, the database server accesses directly the raw disk device to avoid going through the operating system input-output buffer cache and to optimize the layout on disk. The result is some performance increase for very large databases at a huge cost in implementation complexity and non portability.

Chapter 7

Conclusion

In this book, the concepts underlying object oriented distributed programming were presented. They were illustrated through SRC Modula-3, a modern, clean and efficient programming environment which supports nicely graphical distributed object oriented applications. A distinctive feature of this environment is the emphasis on simplicity and the pervasive use of multi threading. Multi threading brings simple and efficient solutions to applications such as network objects receiving asynchronous requests from clients, or graphical objects receiving keyboard or mouse events.

A number of new developments may be expected in the coming years. Time will tell which ones will actually be widely used. One could hope that safety and garbage collection, offered by almost all newer languages, will be taken for granted. The current evolution from C++ to Java is an important step in this direction.

It is possible that new tools such as LARCH may be able to statically verify safety properties and thus obviate the need for some of the run time checks. Similarly, some synchronization errors between threads may be detected statically. Assertions, to be verified by such tools, are incorporated on an experimental basis in some of the modules in the SRC Modula-3 programming environment.

Distributed applications are inherently difficult to build. Facilities such as network objects greatly facilitate the development of such applications. Further developments in this area need to consider the problems of authentication and access control.

Object oriented databases, persistent objects, query languages, embedded interpreters and distributed objects have a lot in common. Eventually these features may be available in a coherent framework, using a few orthogonal mechanisms instead of the current proliferation of systems and

libraries offering mostly overlapping services.