Navigating the MeldC

The MeldC User's Manual

Howard Gershen
Erik Hilsdale

Technical Report
CUCS-031-91

Columbia University
Department of Computer Science
New York, NY 10027

Revised on October 6, 1992

The Programming Systems Laboratory is supported by National Science Foundation grants CCR-9106368 and CCR-8858029, by grants and fellowships from AT&T, BNR, DEC, IBM and SRA, by the New York State Center for Advanced Technology in Computers and Information Systems and by the NSF Engineering Research Center for Telecommunications Research.

Contents

1 Object-Oriented Programming **7**

1.1 The World According to Objects 7

1.2 Classes 9

1.2.1 Defining A Class 11

1.3 Objects 12

1.3.1 Object Creation and Destruction 13

1.3.2 The `init()` and `dest()` Special Methods 14

1.3.3 The MeldC Function Call 14

1.3.4 Special Objects 15

1.4 Features 15

1.4.1 Multiple Features and Interfacing 17

1.4.2 Declaration and Scope Rules 18

2 Inheritance **21**

2.1 Creating Classes using Inheritance 22

2.2 How Objects with Inheritance Behave 24

2.3 Conflicts and Override Inheritance 24

2.3.1 Inheritance of Methods 25

2.3.2 Masking and the `::` Operator 25

2.3.3 Inheritance of Instance Variables 27

2.3.4 Type Conflicts 27

2.3.5 Virtual parent versus Non-virtual parent 27

2.3.6 $self versus $vself 28

2.4 Conflicts and Merge Inheritance 29

2.4.1 Multiple Methods 30

2.5 The `-merge` Compiler Switch 32

3 Concurrency **33**

3.1 Inter-Object Communication 33

3.2 Message Passing in MeldC 36

3.2.1 Synchronous Message Passing 36

3.2.2 Asynchronous Message Passing 37

3.3 Pseudo-Parallelism 39

3.4 Synchronization Issues in Concurrency 40

3.4.1 Race Conditions and Mutual Exclusion 41

3.4.2 Atomic Blocks 42

3.4.3 `delayuntil` and `respond` 43

3.4.4 A Warning 44

4 Distributed Programming **45**

4.1 The General MeldC Model 45

4.1.1 The Protocol Object 46

4.2 The Nameserver 46
4.2.1 Getting and Putting 46
4.3 The Low-Level Approach 48
4.3.1 Putting an Object 48
4.3.2 Getting an Object 48
4.4 Transparency 50
4.4.1 Passing Objects to Methods 51
4.4.2 Network Failure 51
4.4.3 Synchronization and Atomic Blocks 52
4.5 Garbage Collection 52

5 Persistent Programming 53

5.1 The General MELDC Model 53
5.1.1 The Persistent Protocol Object 53
5.1.2 Getting and Putting 54

6 Dynamic Composition of Object's Behavior 57

6.1 Writing a Shadow Object 58
6.2 Attaching and Detaching 59
6.3 Multiple Shadow Objects 60
6.4 Shadow Object Examples 60
6.4.1 Object Composition 60
6.4.2 An Example of Object Composition 61
6.4.3 A Tracing Example 61
6.4.4 A Persistent Object Example 61

6.4.5 A Remote Object Example 62

7 MeldC Library Objects 63

7.1 The System Object 63

7.1.1 The `exit()` Method 65

7.2 The Shell Object 65

7.3 Memory Objects 66

8 Software Tools 69

8.1 The MeldC Debugger 69

8.1.1 Scope and the $Separator 69

8.1.2 `mcgdb` Commands 70

9 Advanced Topics 73

9.1 MeldC Optional Parameters 73

9.1.1 Syntax 73

9.1.2 Declaring a method with optional parameter 73

9.1.3 Referencing a particular optional parameter 74

9.1.4 Referencing All Optional Parameters Passed in as a Whole 75

9.1.5 Mapping Optional Parameters 75

9.1.6 Referencing individual optional parameters with Macros 76

9.2 Delegation 77

9.3 Active Values 78

9.3.1 Active Value Syntax 78

9.3.2 Assignment vs. Change 80

9.3.3 Active Values and Complex Data-Types 81

9.3.4 Active Values and Inheritance 82

9.4 The MetaClass 82

9.4.1 Destroy() 84

9.5 String Selectors 85

9.5.1 Syntax for Selectors and Messages 86

9.5.2 The $selector Keyword 87

9.5.3 Limitations 87

9.6 The MELDC Scheduler 88

9.7 Dynamic Linking of Classes 88

9.7.1 Syntax 89

9.7.2 Using the Dynamically Linked Class 90

9.7.3 Preemptive vs. Non-Preemptive Scheduling 90

9.8 Preventative Debugging 92

10 Examples **95**

10.1 Inheritance: Clock Radios 95

10.2 Concurrency: The Producer-Consumer Problem 98

10.3 Shadow Objects: Another Producer-Consumer 101

10.4 Dynamic Linking: Yet Another Producer-Consumer 106

A The MELDC Program File **109**

A.1 Feature Name 109

A.2 Interface 110

A.2.1 Exports 110

A.2.2 Imports 111
A.3 Implementation 111
A.3.1 Global Variable Declarations 111
A.3.2 Class Definitions 112

B Backus-Naur Form **115**
B.1 A Short Introduction to BNF Concepts 115
B.2 The MeldC BNF 116

C MeldC Keywords **123**

D The UNIX System Object **125**

Credits

Gail Kaiser (Associate Professor, Columbia University)
   Project leader

Wenwey Hseush (Doctoral Candidate)
   Principal designer of language kernel
   Co-designer of compiler

James Lee (Graduate Research Assistant and Staff)
   Project manager
   Principal designer of runtime system

Esther Woo (Graduate Research Assistant)
   Principal designer of compiler

Felix Wu (Doctoral Candidate)
   Contributor to designs of kernel and runtime system
   Implemented complex packages under MeldC

Steve Popovich (Doctoral Candidate)
   General design suggestions and criticism

Project Students:
   Howard Gershen (Manual)
   Erik Hilsdale (Manual, Persistent Objects)
   Larry Katzman (Active Values)
   James Show (Generic Objects, Debugger)
   David Worenklein (Mar)

Special Thanks to Travis Winfrey

Introduction

The MeldC language was developed at Columbia University to bring together several important strains of computer theory into a single programming language. In its present incarnation (October 1992), the language is a "meld" of conventional C and support for object-oriented programming, parallel (or concurrent) programming, and distributed programming (to allow access to objects and data stored in remote locations).

We can refer to MeldC's "present incarnation" because the implementation of the language and the language itself are both not yet finalized. This manual, therefore, can be thought of as a progress report on the current state of the research and programming that has gone into the creation of this language.

This manual covers not only MeldC keywords, syntax, and currently-implemented features, but also a simple (and, one hopes, painless) introduction to some areas of computer science research that spurred creation of the language. This approach was taken so that readers who are not familiar with the peculiarities of operating systems theory, the object-oriented programming paradigm, and network communication might gain a quick overview of those subjects and be able to better appreciate MeldC's features. For the sake of style, these overviews are knitted into the discussions of keywords and syntax, so more educated readers be forewarned.

For clarity's sake, we have devised certain conventions in the presentation of the material herein. New terms that are *not* keywords of the MeldC language appear in **boldface**. Keywords of MeldC found within discussions and example programs appear in a `different typefont`. The wide outer columns on each page can be used for notes. Each of the first three chapters introduces a key aspect of MeldC and concludes with a large example program. This example program can be typed into a terminal, compiled, and run. Explanatory comments are found with the example code, to detail the use of constructs described previously in the chapter.

The intent of these conventions is to make the material easy to follow and easy to refer back to during creation of MeldC programs. A series of appendices in the back of the manual detail areas not covered in the main chapters, including an annotated sample program, the BNF grammar for MeldC, and discussion of MeldC in relation to the UNIX operating system[1].

The manual was set by the LaTeXword-processing program, illustrations were created via the idraw graphics program, and the final product was printed as a group of Postscript-formatted files.

Our work on this manual was aided by those computer scientists who created the language: Professor Gail E. Kaiser; research staff associate Travis Winfrey; doctoral students Wenwey Hseush, Steven Popovich, and Felix Wu; and graduate students James Lee and Esther Woo, all of Columbia University; plus Larry Katzman, a visiting undergraduate student from the University of Pennsylvania. In addition, Chung Lin, Stephen Mauldin, James Show and Carolyn Philippe participated as project students.

Contributions to the authors can be made in the form of small, unmarked bills deposited in a hollow tree stump in New Jersey.

Howard Gershen

Erik Hilsdale

October 6, 1992

[1]UNIX is a trademark of Bell Laboratories.

Chapter 1

Object-Oriented Programming

1.1 The World According to Objects

Good programming style dictates that frequently-used sections of code should become separate functions or subroutines, called by a main function sending the appropriate parameters out to be processed, like so many dirty shirts sent out to the local dry cleaners. These subroutines are smaller versions of functions, and may call yet smaller functions to do their own dirty work. The subroutines and functions are generally written to process data in a linear manner, and a particular programmer may have to recode the same basic functions, in slightly different ways, in several different individual programs.

What's wrong with this picture?

The problem is that writing computer programs in the 20th Century shouldn't be dictated by methods from the 15th Century, before the era of movable type and interchangeable parts. Gutenberg could reuse pieces of type to create different verses of his Bible, but most programmers must recode each algorithm wholesale whenever a new piece of code is written.

Through **Object-Oriented Programming (OOP)** the same code can be used over and over again in identical or slightly different contexts. To be more precise, it is the *interface* which exhibits **reusability**, while the underlying code may or may not be changed at some time in the future.

Programmers who use C have already experienced a very simple type of reusable code: the routines found in the library of header files. Rather than have each programmer create a new set of I/O routines, for example, a library of functions is available for common use. The printf() function in a given implementation of C may have been based on a different algorithm than the printf() in one's own version of C, but both functions still require the same arguments in the same order.

Object-oriented programming takes this idea a step further. Instead of having a set of separate functions to be called from within a main program, OOP allows the creation of little program-units executing on their own, with their own set of private variables and executable code, distinct from a main program. These little program-units can send messages to and receive messages from other little program-units, which are the *objects* of *object*-oriented programming.

Each object is an abstract data type, much like simpler abstract data types such as integers, characters, and other very low-level elements of a programming language. Like spies who deal with important information on a "need-to-know" basis, programmers don't need to know the underlying basis of abstract data types. This is the concept of **encapsulation**, also known as **information-hiding**: an object can be used in a program even though the inner code and variables are a complete mystery.

What *is* important is the interface the object presents to the world. Other objects or programs can only access an object's innards through a specified set of openings. Imagine a bank where the tellers sit behind frosted glass. Each teller has only one opening in the glass, through which you can pass withdrawal slips and from which you can get cash. The bank may have completely renovated that part of the building behind the glass, or hired new tellers, or adopted a new method of processing withdrawals, *but you don't have to know that as long as you can get your money.*

There is a certain amount of trust involved here. A programmer must trust that the object will do whatever is necessary to produce a specified result. Likewise, the creator of the object must trust others to preserve the interface when they recode the insides of an object for better efficiency. The end result is that debugging is eased because the code within the objects is isolated and assumed to be error-free.

Before we pass through that all-important interface and see what lies inside an object that makes it so special, let's consider a broad overview of how to organize the fruits of our object-oriented programming labors.

1.2 Classes

Any set of objects that can be classified together into a group consisting of some common characteristics can be called a "class." For example, the animal and plant worlds were organized according to kingdoms, families, phyla, and so on by Linneas; atoms were organized by Mendeleev into a chart of the elements; and baseball cards have been organized and reorganized by kids in a number of different ways.

Classifying can be broad ("All the boys on the left; all the girls on the right!") or specific ("SWFNMNS, 115 lbs., 5'5", blue eyes, brown hair, newspaper professional, seeks SWMNMNS, over 5'10", college-educated, professional, able to bend steel in his bare hands, leap tall buildings in a single bound, and run to the altar faster than a speeding locomotive."), but to make a classification scheme useful it's important to establish some sort of hierarchy. Darwin's system of evolution is one familiar hierarchical device; a family tree is another.

We might set up a classification scheme for bagels. The class known as `bagel` could consist of: plain bagels, raisin bagels, salted bagels, pumpernickel bagels, sesameseed bagels, poppyseed bagels, etc. We could make a slight reorganization of this system by including poppyseed and sesameseed bagels under a single category: `seeded_bagel`. Now sesameseed bagels are in the class of `seeded_bagel`, which is itself in the **superclass** `bagel`. This shows a simple hierarchy at work.

Now let's make things a little more complicated: where do doughnuts fit in here?

Oh no, one might say, doughnuts aren't bagels.

But doughnuts share some of the same characteristics of bagels: they're circular breakfast foods made from dough and most of them have a hole in the center. So, if we look at doughnuts in this way, then they *could* be considered a member of the `bagel` class.

On the other hand, we might turn the view around and agree with the

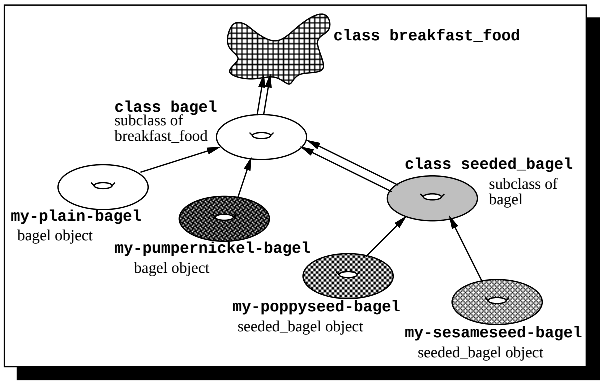

Figure 1.1: The `bagel` Family Tree

comedian who once joked, "A bagel is a doughnut dipped in cement." And then we'd say that `bagel` was a subclass within the `doughnut` class.

Object-oriented programming deals with some of these same issues: how can pieces of code be grouped together by their common characteristics, and then how can these commonalities be used to save a programmer from reinventing the wheel every time she or he has to write a program that will manipulate data in a way that's been done before?

The organization of objects within a particular OOP program is determined by the concept of **class**. As we saw above in the discussion of bagels, objects can be grouped according to common characteristics. The determination of what qualifies as a group of common characteristics is a subjective decision: the `bagel` class could have existed on its own, or been part of the `doughnut` class.

To distinguish the relative positions of certain elements within any class hierarchy, we can say that each subclass **is a** specialization of a given class, while a object is an **instance of** a class. A `seeded_bagel` **is a** `bagel`; a sesameseed bagel is an **instance of** the class of `seeded_bagel`. These objects can inherit ways of dealing with data (just as you might have inherited a good sense of smell or myopia) from their ancestors in the hierarchy. Thus the `seeded_bagel` has all the characteristics of the regular `bagel`, but with the addition of seeds.

This concept is not too difficult to understand if we remember that objects are basically abstract data types. In conventional C, a number can

```
class complex_number
  float real, imag;
methods:
  float get_real() { return (real); }
  float get_imag() { return (imag); }
  void set_value(float x, float y) {
    real = x;
    imag = y;
  }
  float add_to(complex_number y) {
    real += y.get_real();
    imag += y.get_imag();
  }

end class complex_number
```

Figure 1.2: A Class of Complex Numbers

be represented as an integer (`int`), a `float`, a `double`, or a `long`. These are all *instances* of abstract data types (objects) representing numbers. When a particular `int` called `john` is assigned the value of 7, while another `int` called `martha` is assigned 8, both `john` and `martha` are still `int`'s, even though they contain different values.

1.2.1 Defining A Class

Suppose we were to define a class of complex numbers. Each object of that class would need to remember its value, provide some access to the value, and have some procedure for setting the value. One possible implementation is for each complex number object to keep values for the real part and the imaginary part of the complex number, and allow those values to be set, accessed, and added to another complex number. Our design translates very naturally into MeldC (see Figure 1.2).

Each object of the class `complex_number` keeps its value in the **instance variables** `real` and `imag`. These are the private variables of each `complex_number` object. The instance variables are declared in conventional C syntax, and can be assigned a value upon declaration. If we wanted to initialize all new objects of class `complex_number` to $4+3i$ for some reason, we would say so in the class definition, just like C:

```
class ComplexNumber ::=
  float real = 4;
  float imag = 3;
methods:
  ...
```

Instance variables must be declared before the `method` keyword of the class definition, so that they might be available for use by those methods.

The `method` section contains all the **methods** for the class in the form of C-like functions containing conventional C and MeldC statements. Take a look at the method called `add_to`. It takes as a parameter another object of the `complex_number` class, referred to as `y`. To add two complex numbers together, we retrieve the *values* of the `real` and `imag` instance variables of `y` via `y`'s own `get_real` and `get_imag()` methods[1]. These C-like functions treat the instance variables as if they were global variables of a conventional C program, but they are the only functions that have any access to them. Methods declared in *other* objects cannot reference these variables.

This encapsulation of values (as we noted previously) is a key means by which objects can "compartmentalize" information. Just as conventional C can limit the scope of certain values by treating them as static variables, an object created in MeldC (and most other object-oriented programming languages) can hide the information it holds from other objects seeking to directly access, or even change, a given value.

1.3 Objects

The MeldC concept of objects is relatively simple to understand. One important thing to remember is that each class is like a cookie cutter, shaping every object it creates in its own image. Objects have types, just as variables do in conventional C. If an object `myzucchini` was of type `zucchini`, for example, it would be declared so in the area where variables are declared. Figure 1.3 shows the skeleton of a feature in which the object `myzucchini` is declared globally.

[1] Note the special syntax. It resembles the way in which we can access a part of a struct written in conventional C.

```
feature vegetables_i_have_known
interface:
implementation:
  zucchini myzucchini;
  ...
end feature vegetables_i_have_known
```

Figure 1.3: Global Variable Declaration

1.3.1 Object Creation and Destruction

There is a difference between conventional C variables and MELDC objects, however. When a conventional C variable is declared, space for the variable is automatically created. When a MELDC object is declared, only the name is locked in place; the object has yet to be created. That object can be created at any time, but possibly the most common time would be when it is declared. For example, if we wanted to declare and create an object `myzucchini` of class `zucchini`, we would write:

```
implementation:
  zucchini myzucchini = zucchini.Create();
  ...
```

Though the construction `zucchini.Create()` is peculiar to MELDC, note that the assignment is otherwise similar to C syntax: we are simply assigning to a variable a value in the same statement as its declaration.

An object is destroyed in an analogous manner. The statement

```
zucchini.Destroy(myzucchini);
```

will handle it. Notice that both `.Create()` and `.Destroy()` come after the *class* name.

1.3.2 The `init()` and `dest()` Special Methods

`init()` and `term()` are methods that are called on object creation and destruction, respectively. If a class definition includes an `init()` method, then whenever an object of that class is created the statements of the method will be executed. Likewise, when an object is destroyed the statements in it's `dest()` method, should one exist, will be executed.

The `init()` method is especially important to MELDC programs. Since there is no `main()` procedure as there would be in conventional C, there is no predetermined place for execution of the program to start. Instead, whenever the first object of the program is created (as well as whenever any other object is created), a method begins executing. Thus, not every class need have an `init()` method, but one in every program certainly should; if there were none, nothing at all would happen on program execution.

1.3.3 The MELDC Function Call

After objects are created, what good are they? How can the MELDC program access them? Well, what we have called **methods** bear a strong resemblance to conventional C functions, and **classes** resemble conventional C programs with those classes' instance variables corresponding to conventional C global variables. In fact, we manipulate objects in MELDC with the MELDC **function call**, much like the one in conventional C [2]. The syntax is a bit different but familiar nonetheless: since there are a number of objects in the system at any one time, a function call must have some way of distinguishing between the alike-named methods of different objects. MELDC does this by calling a method by its name and the name of its object[3]. The basic syntax is:

object.method(*parameters*)

Like C function calls, MELDC function calls return a value (unless the method called was declared type `void`), so the function call can be a single statement ended with a semi-colon, or it can be embedded in some larger statement in an expression. As we will see in Chapter 3, there is more to

[2] We can use conventional C function calls within methods as well, but they manipulate data within an object rather than interact with other objects.

[3] Once again, we can harken back to conventional C and note a similarity to the way a structure's members are addressed.

these function calls than meets the eye, for they are closely related to the idea of **message passing**.

As a final note, if an object is dynamically created—that is, if it is not created at the same time it is declared—a little extra syntax is needed for the MELDC function call. If the return value is needed then the type of the return value must be explicitly cast. For example, if an object `foo` had a method `bar()` that returned an integer value, the assignment of its return value to variable `gurble` would look like

```
gurble = (int) foo.bar();
```

1.3.4 Special Objects

There are a number of special objects (actually special object names) of differing importance to MELDC. Possibly the most important is `$self`, a self-reflexive object which, when used within a method, refers to the object executing the method. One important place we use `$self` is in the `init()` method to start up the program, if the object is to send itself a message when it is created. If, for example, we had a class `dreamer` with a method `pinch_me` which "awakens" him or her, we might want to have that method called as soon as a `dreamer` object is created. We could do that with the following `init()` method:

```
void init() { $self.pinch_me(); }
```

Less used than `$self`, but important nonetheless, is `$sender`. `$sender` refers to the object that called the current method with the MELDC function call. Without this special object, objects have no way of telling who calls their methods, a piece of information that is sometimes very useful.

1.4 Features

A MELDC program is built of **features**. If objects are black boxes, limited to access through their selectors, then features are the black casing around a set of black boxes. Inside of a feature are the global objects, variables, and classes that everything inside the feature can reference.

```
feature feature-name
interface:
    imports and exports
implementation:
    global declarations
    class definitions
end feature feature-name
```

Figure 1.4: The Skeleton of a Feature

The MELDC feature has a skeleton, shown in figure 1.4. It has two parts, the **interface** and the **implementation**. The interface contains information on how it interacts with other features in the program, while the implementation is the actual body of the feature, the global variable declarations and class definitions. In programs with only one feature the whole program will be in the implementation section and the interface will be empty.

Though the feature skeleton is fairly clear, there are a few things to remember about it:

- Each file of a program contains exactly one feature. The name of the feature and the name of the file that stores it are completely unrelated. They may be the same or different, whatever makes the programmer's life easier.
- Class don't have to be defined *before use* in the implementation section, but they must be defined somewhere in the feature.
- Occasionally, MELDC programs will require `#include` directives. These must come before the first line of the feature: "`feature` *feature-name*".
- `struct`s, `typedef`s and `union`s must be declared in the implementation section. They may be `#include`d before the feature, but then they must be inside a `.h` file.
- The *feature-name* in "`end feature` *feature-name*" is optional, but should be included in the interest of clear programming.

1.4.1 Multiple Features and Interfacing

Though each *file* in a MELDC program consists of only one feature, each *program* may be many features long. And just as objects have ways of interfacing with other objects—namely, the MELDC function call—features must be able to interact with other features when multiple features are used in a program. The way features interact is through **exporting** and **importing** classes and objects.

A feature can **import** a class or object that is defined in another feature. Once it does so, the class or object behaves (and is accessed) as if it were defined locally. That is, if feature `A` imports a class that is defined in feature `B`, that class can be used in feature `A` as if it were defined there. Likewise for objects.

There is a catch, though. Feature `A` cannot import anything from feature `B` unless feature `B` has **exported** it. In other words, a feature can't just *steal* something from another...it must be *offered* it. This is another data-abstraction aid: imagine a feature that dealt with keeping an object-oriented database. Though the feature might have all kinds of the classes and objects necessary for upkeep of the database, it would only offer certain of these for access to other features. The rest would operate invisibly from the standpoint of the larger MELDC program.

Each MELDC feature has a special place to list imports and exports, its **interface** to other features. This interface section lies at the start of the feature and begins, not surprisingly, with the keyword `interface` (which may be followed by a colon for clarity). Here the programmer lets MELDC know how the feature is to interface with other features, through the `imports` and `exports` he or she declares.

Importing an object or class is straightforward. To import something, we list it on a line after the keyword `imports`. We must know the name of the class or object to import, and the feature to import it from; the syntax for this is *featurename*[*classname*]. If we wanted to import the class `vegetable` and the object `broccoli1` that were exported from a feature called `health_foods`, for example, a line of the interface section would be:

```
imports health_foods[vegetable], health_foods[broccoli1]
```

but we needn't really go into that much detail. We can import everything exported from `health_foods`, objects and classes, with

```
imports health_foods
```

Putting the object- or class-name in square brackets specializes the import, while using only the feature-name imports everything exported from that feature.

Exports from a feature are even easier, since the feature doesn't have to be specified. To export something, use the line "`exports` *something*" in the `interface` section—that *something* can be the name of any class or globally defined object. We can even export more than one thing on a line by separating the items with commas: if we wanted to export the objects `broccoli1` and `cauliflower33`, and the class `vegetable`, we could write

```
exports broccoli1, cauliflower33, vegetable
```

The ordering of the interface section is important in one respect: All imports must be declared before anything is exported. Apart from that, anything goes. The class, objects and features listed on each line need not be ordered, and MELDC doesn't care whether all exported or imported items are on the same line, each one on its own line, or something in between. The only reason for ordering imports is that if a class or object of a particular name is imported, and another class or object with the same name is imported, only the first is actually imported; the second is ignored. For example, consider the **interface** line:

```
imports foofeature, barfeature
```

If both features `foofeature` and `barfeature` had the definition of a class `bazclass`, only `foofeature`'s `bazclass` would be imported.

There is one caveat about exporting and importing. When a feature imports an entire feature, or exports itself with only one export declaration, it only imports or exports the *classes* in the feature. Objects cannot be automatically imported or exported by putting an entire feature in the interface section, they must be imported or exported on an individual basis.

1.4.2 Declaration and Scope Rules

There are four scope levels at which a variable can be declared. The lowest scope is that of **local** variables. Local variables are just those that are

```
class lemming ::=
  static int number_of_lemmings;
methods:
  void init() {
    number_of_lemmings++;
    if (number_of_lemmings > 100)
       jump_off_cliff();
    else
       mill_around_aimlessly();
  }
end class lemming
```

Figure 1.5: A Class of Lemmings

declared in the body of a method, and are analogous to local variables in C functions. Local variables are only visible and accessible within the method in which they are defined.

One step higher are `instance` variables. As an instance of a particular class, an object has its own internal copies of variables from the original definition of the class. These instance variables have the same names as those that exist in their class's definition, but their scope is limited to the object. As twins can be born with identical characteristics (their instance variables) but diverge at birth from this ideal of equivalence and lead distinct lives, so, too, objects take on unique identities after they are created[4].

There are cases, though, where we would like to have a variable that can be accessed and modified by any object of a given class. Say we had a class whose objects behaved differently depending on the number of other objects of its class in existence. Imagine, for instance, that we had a class called `lemming` whose objects would do one thing when there were less than a hundred other lemmings on the system, and another if they were overpopulated. How could the objects of class `lemming` (or just "lemmings" for short) detect how many other lemmings there were? One simple way would be to make a global counter `number_of_lemmings` which is incremented every time a new object is created. This would solve the problem, since all lemmings could access the global variable. Everything else in the feature could access the variable as well, however—not good data encapsulation.

[4] As we will soon see when we look at inheritance, objects are not necessarily created through asexual reproduction. They may have more than one parent class.

MeldC's method for hiding a variable from the rest of the feature yet sharing it with all objects of some class is the **class** variable: If, in the class definition, an instance variable is declared `static`, a separate variable is not created for each object; one class variable is created and all objects of that class can reference it. A correct definition of our `lemming` class—using class variables—is shown in Figure 1.5.

At the highest level of scope are **global** variables. Declared at the start of the `implementation` section of the MeldC program, they are visible everywhere in the feature. They may be accessed and changed from every method of every object defined in the feature or imported into it.

Chapter 2

Inheritance

Our conception of object-oriented programming and how it is implemented in the MeldC language has so far covered issues of class, object creation, and message passing between those objects. We now come to an important variation on the idea of class in MeldC: the concept of **inheritance**.

In MeldC, classes can be grouped into larger categories called **superclasses**, as in the animal world species can be grouped into ever-larger categories, from genus up to kingdom. When a class is grouped into a more encompassing class, it is said to **inherit** from its superclass. To view from a different angle, subclasses can be said to "reuse" information in existing superclasses to create something more specialized.

There are as many ways to visualize inheritance as there are natural examples that deal with inheritance. One way to look at inheritance is the **Venn diagram** of Figure 2.1. In it, the class `bagel` inherits from classes `torus` and `breakfast_foods`, and is shown as being *inside* the intersection of its two superclasses. The class `seeded_bagel` is shown as inside the class `bagel`, which it inherits. Since `seeded_bagel` is also inside the intersection of `torus` and `breakfast_foods`, it is clear that it inherits from its "grand-superclasses" as well.

Venn diagrams tend to get complicated quickly, though, so in this manual, inheritance schemes will be illustrated by directed, acyclic graphs, like family trees (see Figure 2.2). It is important that the graph be *acyclic*, so as to avoid the case of an object inheriting from a class which inherits from

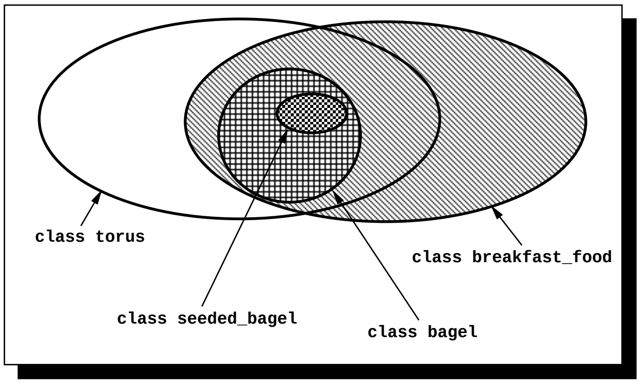

Figure 2.1: A Venn diagram of an inheritance scheme.

itself — as in real life, you can't be your own father... unless your name is Oedipus.

Up to this point we've learned how to write simple MeldC programs that let us make instances of classes and send messages between those instances. Unfortunately, we still lack the ability to create more specific classes, and that is truly a gaping hole in our use of the language. One of the most fundamental tools of object-oriented programming is inheritance. Without it, our picture of the MeldC world is rather childlike: it's as if the animals, minerals, and vegetables in our world are different from one another (as they should be) but all the animals are dogs, all the minerals are "dirt," and all the vegetables are... "yucky green stuff."

The world we'd like MeldC to reflect is the diverse world of a prominent zoo or natural history museum, one in which we can group subclasses of animals under the most general class animal. Perhaps mammals and insects fall under animal, and perhaps lemmings fall under mammals. In that case any instance of the class `lemming` would also be an instance of the classes `mammal` and `animal`. Yet we would also like the ability to start with a general class and construct more specific instances of it. These two views of inheritance are really two sides of the same coin, the "top-down" side and the "bottom-up" side, and both are valid in MeldC.

2.1 Creating Classes using Inheritance

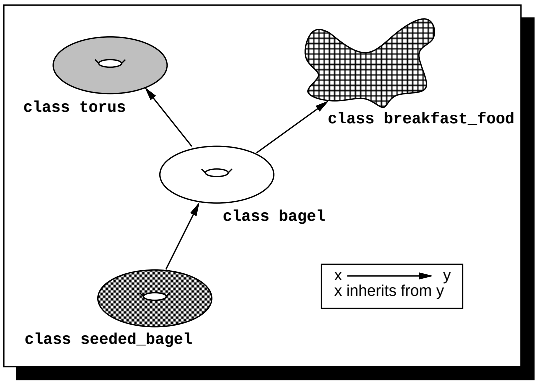

Figure 2.2: An inheritance scheme for class `seeded_bagel`.

In the inheritance scheme of Figure 2.2 the class `bagel` inherits every aspect of both `torus` and `breakfast_food`, and includes in itself some aspects of its own. It, in effect, **merges** the attributes of its parent classes with its own attributes to make a satisfying whole. It is as if we lived in a world where a child started off remembering everything his or her parents knew (which would make school quite a bit easier for the child of two university professors). Thus, to define a class that inherits from other classes (or to define a class that inherits from just one other class), we use MELDC's `merges` keyword:

`class` *class-name* `merges` $parent_1$, $parent_2 \ldots$ `::=`

where the *parents* are the superclasses we want *class-name* to inherit from. Our bagel, for example, was the product of two parents: the class of `breakfast_foods` and that of `torus`. So to define the class `bagel` we would start off with:

```
class bagel merges breakfast_foods, torus ::=
```

and continue on with `bagel`'s instance variables and methods.

Just as children can't be too picky about who their parents are, MELDC allows parent classes to come from almost anywhere, even other features, without forcing us to specifically import them. When we inherit from outside the current feature, we use similar syntax as when we import classes, but the "imported" class is only visible to its inherited children: nobody else can use it to `Create()` objects or for anything else. If (as is likely) the definitions of the classes `torus` and `breakfast_food` aren't in the same feature—say, `torus` is in the feature `solids`—we can still create class `bagel` (without any import statements in the interface section) starting with:

```
class bagel merges breakfast_foods, solids[torus] ::=
```

where we assume the class `breakfast_foods` is in the current feature, and the class `turus` is in the feature `solids`. Even if the `torus` class or the `solids` feature *is* imported in the interface section, this inheritance syntax must be followed.

2.2 How Objects with Inheritance Behave

We now understand how to write classes which inherit from other classes, but what is so special about them. In terms of MELDC, what does it mean to say that a child class inherits "aspects" of its parent or parents?

We already know that an object consists of its methods and its instance varibles, both of which are defined in the object's class definition. Well, if an object's class inherits from another class, then the object has the instance variables and methods of both its base class and of all inherited classes. Thus, in Figure 2.2, an object of class `bagel` will have all instance variables specified in the definition of the class `bagel`, as well as all instance variables defined in the classes `torus` and `breakfast_foods`. Likewise, it will have access to the methods defined in all three classes.

So, we use the keyword `merge` because inheritance *merges* the structure of all inherited classes into one great superstructure. This kind of thing, however, can have reprecussions if there are conflicts in names in the inheritance hierarchy.

2.3 Conflicts and Override Inheritance

It's not precicely true that MeldC's classes are merged by the merge statement of inheritance, but the only way to tell this is when there are name conflicts.

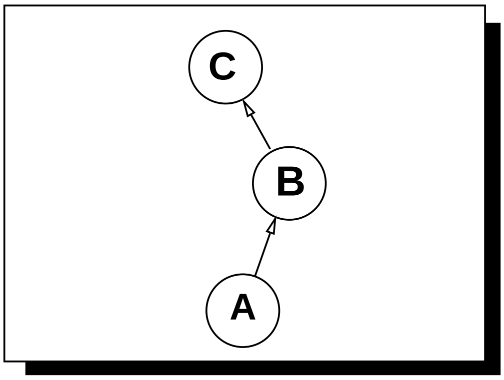

Figure 2.3: Basic Override Inheritance.

2.3.1 Inheritance of Methods

Consider the class hierarchy of Figure 2.3, where A is the child of B and B is the child of C. If all three classes have the method `foo()`, then A's `foo()` will be executed when an object of class A receives the `foo()` message. If only B and C have definitions for `foo()`, however, B's `foo()` will execute. In short, the method *closest* to the base class is executed, and nothing else.

Conflicts in this rule are caught at compile time. In Figure 2.4, for example, both B and C are equally "close" to A, as A inherits both classes. If B and C have definitions for method `bar()`—and if A has no such definition—the program will not compile.

2.3.2 Masking and the `::` Operator

There is a problem with this notion of inheritance: It appears as if certain methods can be permanently inaccessible in the inheritance hierarchy. If, in figure 2.3, classes B and C both have a method `foo()`, C's `foo()` will never run when `foo()` is called in an object of class A. No methods were "starved out" like this under merge inheritance.

There is indeed a way to access these "stranded" methods. If one method is overridden by another, it can still be accessed and called by the construction

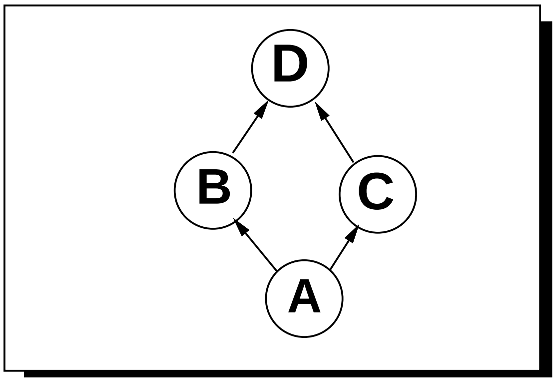

Figure 2.4: Multiple Override Inheritance.

class-name`::`*method-name*

For example, consider a version of Figure 2.3 where B and C have `foo()` methods. If `A_obj` is an object of class A, then the MELDC function call

```
A_obj.foo();
```

will trigger B's `foo()` method, but the call

```
A_obj.C::foo();
```

will trigger the method defined in class C.

There are a number of possible cases which might arise under override inheritance. All of these refer to the inheritance hierarchy of Figure 2.4:

- If both A and B have methods of the same name, the program will compile, and objects of class A will have direct access to A's method.
- If both B and C have methods of the same name, and A doesn't have such a method, the program will not compile due to the conflicts between A's parent classes.
- If A, B and C have methods of the same name, the program will compile and objects of class A will use A's method.

2.3.3 Inheritance of Instance Variables

Inheritance of instance variables with name conflicts is very similar to method inheritance. Instance variables of the same name in the inheritance hierarchy are not precicely merged. Rather, space is reserved for each instance variable of each class in the hierarchy. Normally, all methods of a class can access that class's instance variables and any of it's superclasses' instance variables, so long as there isn't a name conflict. If there is such a conflict, a class can only access one variable of each name without using the `::` operator. So, in Figure 2.3, if classes B and C each had an instance variable named `cost`, an object of class A (or B, for that matter) could only access B's `cost` variable. But this statement, using the `::` operator,

```
cost = cost + C::cost;
```

would add the value of C's `cost` variable to B's. So all instance variables are accessible, but those overridden by "closer" variables must use some extra syntax.

2.3.4 Type Conflicts

Type conflicts are not allowed under override inheritance. If any of the classes in an inheritance hierarchy have instance variables of the same name *and* same type, a "private" copy is made for each variable name in each subclass where they are declared, and each class's methods have access to that class's variable. If any of the classes have instance variables of the same name and different types, however, the program will not compile, as MeldC cannot be sure how to store values in conflicted variables. **Any** instance variables of the same name with different types inside any inheritance hierarchy will stop compilation and signal an error.

2.3.5 Virtual parent versus Non-virtual parent

Figure 2.4 describes the inheritance scheme correctly up to a point, but a little extra work needs to be done by the programmer to get an inheritance scheme to look like that. Examine Figure 2.5. Both pictures in this figure describe the same inheritance scheme, where classes B and C both inherit from D, and class A inherits from B and C. The left side of the figure is

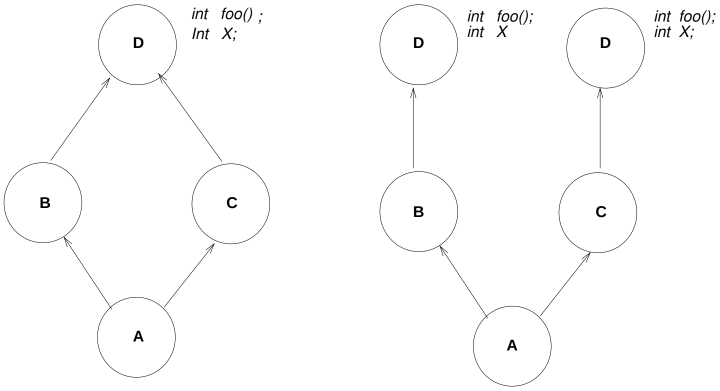

Figure 2.5: Virtual parent verses Non-virtual parent

the Virtual Parent scheme for inheritance. This scheme will create a parent once and only once. So in this figure, classes B and C share class D. There is also the scheme used in the right side of the figure, Non-virtual Parent. For this scheme, there is a separate class object for each inheritance. So, both classes B and C have a copy of class D.

When the Virtual parent scheme is used, less memory will be used since all duplicate objects will not be created. This is one of the reasons that this method is used by MeldC, as compared to using the Non-virtual parent scheme. This scheme is also used for the reason of traversing inheritance trees and for the reason that it makes the removal of objects easier.

2.3.6 $self versus $vself

Both $self and $vself are used by the programmer to specify the method to use when the method exists multiple times in an inheritance hierarchy. For an example of what we mean, see Figure 2.6. In this case, we have three different methods named foo. One in object B, another in object C and one more in object D. The question is how to access the one that you want to use. That is where the $self and $vself come in. The $vself is used to state that we want to use the foo that is closest to the base of the inheritance tree. So when object E makes a call **$vself.foo()**, we will start at the bottom of the inheritance tree and search upwards until a foo is found. So using the figure, we will check and see that object A does not have a foo. The next object to check is object B. This does have a foo, so this foo will

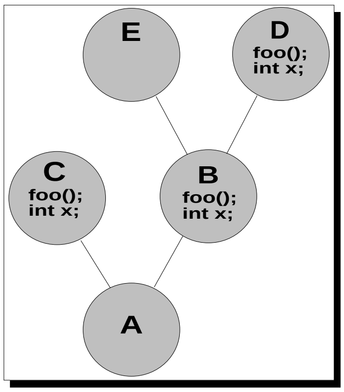

Figure 2.6: Virtual parent verses Non-virtual parent

be executed when the call is made in object E.

Now, let's say that object D wants to use the foo that it has. It can do this by simply making the call **$self.foo()**. The $self call states that we want to find a foo, but instead of starting at the bottom of the inheritance hierarchy, start at this object making the call. Another way of looking at this is, the $vself said to start at the base object, the lowest in the hierarchy. The $self says to start at the base, but make the object making the call the temporary base object. So when object D makes the above call, the foo it owns will be executed.

As another example of $self, when object A makes the call **$self.foo()**, we will check object A for a foo. Since there is none, go to object B. Here we will find a foo and execute it. Notice that the call **$vself.foo()** from object A will equal the **$self.foo()** call.

2.4 Conflicts and Merge Inheritance

There is another, experimental inheritance semantic on name conflict in MeldC, one that is more complicated, yet perhaps more powerful. This "merge" semantic (not to be confused with the MELDC keyword `merge`) behaves the same as MELDC's normal, override, semantic in the absense of name conflicts, but has intriguing ways of handling those conflicts.

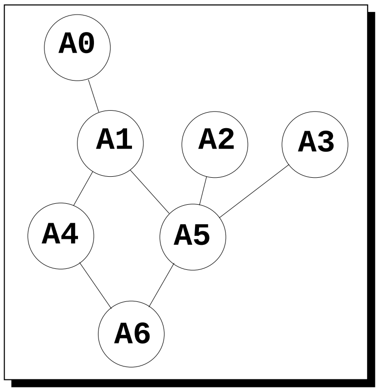

Figure 2.7: An Inheritance Tree

MeldC handles instance variable conflicts very simply. In figure 2.7, for example, if we had defined an integer variable `arthur` both in class `A2` and `A6`, any object of class `A6` would have an integer variable `arthur`. In fact, we could declare `arthur` in all the inheritance tree's classes and we'd get *one* variable—as long as the variable is always declared of the same type.

If one instance variable `arthur` is declared `int` and another `float`, however, the program won't compile. For different classes to share an instance variable name this way, all must know and agree on that variable's type. The same applies to making a variable `static` to turn it into a class variable: either all declarations of a variable in an inheritance tree must be static, or none can be.

2.4.1 Multiple Methods

The merge semantic doesn't consider multiple declarations of a method in an inheritance tree a conflict. That is, declaring a method `ford()` in both `A6` and `A2` doesn't force a compiler error. MeldC will simply execute *all* the methods named `ford()` whenever an object receives the proper message. If *all* of the classes `A0` to `A6` had definitions of `ford()`, and an object of class `A6` received the message `ford()`, a method from *all seven* classes would be executed. If an object of class `A5` received the message, however, only `A0, A1, A3, A4,` and `A5`'s `ford()` methods would be executed. Simple.

Not quite. The complications to this simple scheme (and yes, there always do have to be complications) are the *order* in which the methods are executed, and the way method parameters are handled.

Remember when we said that the order of the classes in the `merges` statement was important? Well, it is, but only if there are methods that are defined more than once in the resulting inheritance tree. The ordering of the `merges` statement defines the order in which a class's superclasses are searched for methods: methods are executed in the order of a depth-first traversal of the superclass tree, starting with the first parent class and continuing to the last.

Let's say the method `zaphod(x)` was defined in each of the classes of Figure 2.7. If an object of class `A6` were sent the message `zaphod(42)`, the methods would be executed in the order:

```
A0, A1, A4, A2, A3, A5, A6
```

not

```
A0, A1, A4, A2, A5, A3, A6
```

Note that the depth-first-traversal doesn't pass from `A5` to `A1`: Any class's methods can be executed only *once*, so the depth-first-traversal will not enter classes it has already passed through.

For methods that return values, the value returned to the calling object is the return value of the *last* method called. For Figure 2.7, that would be the method in `A6`.

There's another problem with calling more than one method: How can we be sure their parameters as well as their names agree? MELDC doesn't know how to handle two methods in the same inheritance tree with a different number or type of arguments, and so will generate a compiler error unless the program has some helpful extra information in it. Without the programmer doing something extra, the MELDC compiler will not accept, for example, having `zaphod(int x, float y)` defined in `A0` and `zaphod(char x)` defined in `A5`, and the program will not compile.

When we declare a method, we can not only declare *its* parameters, but also the parameters that method will take if defined in an immediate parent. To clarify this, let us suppose that in `A6` we have a method `zaphod(int x)` and in `A5` we have `zaphod(int x, int y)`. Also, suppose that we knew we wanted to execute `A5`'s `zaphod(x,y)` always with the parameter `x` equal to `y`, and both of them equal to the parameter of the message: That is, if an object received the message `zaphod(3)`, we would execute `zaphod(3,3)` in `A5`. We could accomplish this by defining `zaphod` in class `A6` as:

```
int zaphod(int x), A5(x,x) {
  ...
}
```

If we wanted some parameter change in `A2` we would declare it in `A5`: All parameter changes would then percolate up if we send a message to an object of class `A6`.

In general, defining a method in a class `C` that inherits a method of the same name is done by:

return-type method-name(*parameters*), *super*$_1$(*super*$_1$*-params*), ...

Where all of the *super-params* are either parameters of the method as defined in `C` or instance variables of class `C`.

2.5 The `-merge` Compiler Switch

The behavior of merge inheritance is selected with a compiler switch, `-merge`. When a MeldC program is compiled, the whole program is compiled with either merge or override inheritance. Thus they cannot be used together in the same program. Normally this shouldn't be a problem, but it is important to remember which files were compiled with override and which with merge inheritance when dealing with separate compilation; such files cannot be linked together.

A final note of caution: The syntax of inheritance (as well as multiple inheritance) is the same under both merge and override inheritance:

```
class A merges B ::=
```

Don't let the `merge` keyword mislead you: The inheritance behavior could be either merge or override, depending *only* on the compiler switch.

Chapter 3

Concurrency

3.1 Inter-Object Communication

Up to this point we have considered the definitions of objects and classes, the creation of objects and classes, and the structure of object hierarchies as systems of classes. Basically, we have been looking at a vertical scheme. Now it's time to stop looking up and down, and start looking around at eye-level.

We already know that classes are able to inherit characteristics *from* other classes, and to "pass on" new characteristics *to* their own descendants. Those characteristics may be instance variables or methods of dealing with variables.

Information, then, is being transmitted through families of classes much as humans can pass on family legends from generation to generation. The information passed by the classes, though, is of a limited sort because members of the same generation (instances of the same class) or their cousins (the instances of other classes) so far have no means of direct communication with each other. We have created these objects to act as if they were mini-programs within a common file, but now they might just as well be separate programs in separate files with no connection. Something crucial is missing from this scheme of things.

We briefly examined something called "message passing" in the previous

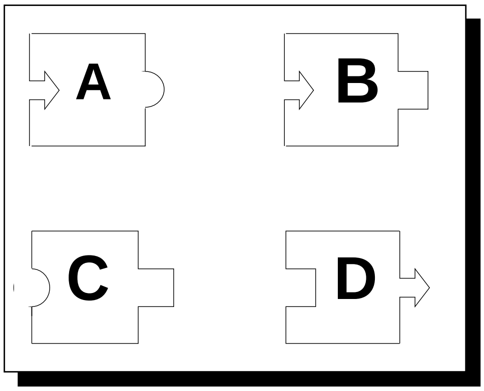

Figure 3.1: Selective Message Reception

section, but only in the simplest way. Like procedure calls, where one program can "pass the baton" to another, message passing enables objects to pass information between each other. These messages may be values needed for equations, instructions to start or stop an action, requests for more data, or any other transfer of information needed to commence or complete an action.

The catch, however, is that a given object will only react to incoming information if it receives what it is *allowed* to receive. For example, an object that only accepts a certain form of character string will be ill-prepared to accept a message consisting of a floating point number to be converted to scientific notation. Like DNA, where the amino acid thymine can only pair off with adenine, and guanine only accepts cytosine, Object A may "accept" messages of the type passed to it by Object B, but not those from Object C, because that object does not know the correct form of an acceptable message.

We can see in Figure 3.1 a simple representation of this sort of selectivity. Let Object (A) be an object that can only receive arrow-type messages, and transmit semi-circle-type messages. Object (B), on the other hand, happens to only transmit square-form messages; messages passed by it to (A) will be rejected because the interface for (A) only shows the outside world a receptor for arrow-forms. This is an example of the principle of uniform external interfaces we saw in the previous chapter. No matter how many times (B) tries to poke (A) with its message, (A) will turn a cold shoulder. If, however, a third object, one that *does* transmit messages with the arrow-format, like (D), tries to send a message, (A) will respond. It is all quite literally a matter of the impossibility of fitting a square peg in a round (in this case an arrow-shaped) hole.

Now consider (B). Once it receives a message it can process, it notes the source of its incoming message, invokes the method(s) associated with such a message, and transmits the result *back to the message's source.* A square-form reply from (B) will *only* go to (D), the object that sent the original message, and to *no other.* (D) clearly has a receptor for the type of messages (B) can transmit.

This specialization of response can be seen in the human body. If a lit match is waved under your bare foot, a message will be sent up through the nerves to the pain center of your brain, processed, and a response message (probably in the form of an instruction to jerk the foot away from the match) will be sent to that particular foot, not to the other foot nor to a hand. If, however, a feather is waved under your foot, a different message will be transmitted to your brain: this one will go to your pleasure center and result in a response message instructing that foot to curl its toes. In each case the FOOT object sends out a different message to the BRAIN object; the interface of the BRAIN that can respond to a pain message takes that instruction; the interface in charge of pleasure messages takes the second instruction. The BRAIN object keeps track of the origin of the message and sends a directed reply appropriate to the nature of the original message back to the sender of that message.

We've already discussed the MELDC **function call** (in Section 1.3.3), where one object sends out a message to another and waits for a reply before continuing. This technique of targeting messages to a particular object may be further enhanced by something called **asynchronous message passing**. Here an object is permitted to send out a message, continue its business, and receive a reply at some later time[1]. It is clear that such a system provides for more efficient use of objects: *all* of our objects can be working at the same time, rather than having half of them waiting for the other half to finish before they can continue.

[1] Or never get a reply, if one is not needed.

3.2 Message Passing in MELDC

Communication between objects in MELDC is implemented through **message passing**. Messages are passed by a statement of the form of C function calls

method_name(parameters)

and are sent from one object to another. In conventional C, if we wanted to call the function `factorial` on the argument 84, the syntax would be `factorial(84)`. In MELDC, we would instead send `factorial(84)` as a message to an object containing the method `factorial`.

Although there are two different ways of *sending* messages, objects behave uniformly upon *receiving* a message. It is important to realize that the distinction between synchronous and asynchronous messages only exists from the standpoint of the sender. The object that receives the message has no way of determining whether it was sent synchronously or asynchronously.[2] An object *can*, however, find out who sent the message by using the special object `$sender`.

3.2.1 Synchronous Message Passing

We already know of synchronous message passing through its alias, the MELDC function call, so named because of its similarities with conventional C's function call in both behavior and syntax. When MELDC reads the statement

```
foo.bar(42)
```

it sends the message `bar(42)` to the object `foo` and waits until the method `bar` finishes and returns a value (or no value if `bar` is of type `void`).

If a synchronous message accesses a method with a return value, the message can be embedded in some larger expression. To find the area of a rectangle `rect1`, where the class `rectangle` is defined in Figure 3.2, we might use the statement:

[2] In the case of an asynchronous send, `return` does not actually return to its point of origin (its sender), so its return "type" can be considered to be "void".

```
class rectangle ::=
  int width, height;

methods:
  int get_width() {
    return(width);
  }
  int get_height() {
    return(height);
  }

end class rectangle
```

Figure 3.2: Class `rectangle`

```
area=rect1.get_width()*rect1.get_height();
```

3.2.2 Asynchronous Message Passing

When an object sends an asynchronous message, it *does not wait* for the called method to finish, as in the case of synchronous sends. Instead, it continues on with the next statement after the message send.

To emphasize the differences between this behavior and that of synchronous messages, the syntax for the asynchronous message employs two MeldC keywords solely for this purpose: `send` and `to`. If we wanted to asynchronously send the message `bar(42)` to the object `foo`, for example we would simply write:

```
send bar(42) to foo;
```

instead of the

```
foo.bar(42);
```

we would otherwise use for a synchronous send.

Asynchronous message passing is the key to **concurrency**. Recall that objects are distinct from one another, and at any given moment several can

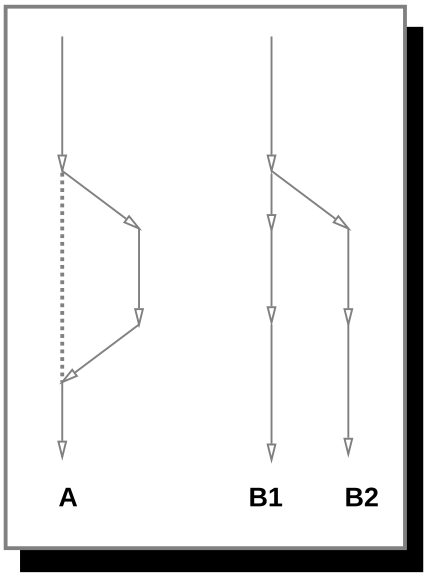

Figure 3.3: On the left, a synchronous thread. On the right, an *asynchronous* thread that has split in two.

be at different stages of operations performed on their input. A map of a given input's traversal of objects and program code is called its **thread of execution**. In a larger context, a program in execution is known as a **process**.

If there were only synchronous messages, there would be only one thread of control, jumping from routines to subroutines and back again. With asynchronous messages, however, a new thread is begun with each new asynchronous message. The thread of control, in effect, splits in two, with both the sender and the receiver having a thread to itself.

Threads are actually of two types: heavyweight and lightweight. Programs in execution are considered to be *heavyweight* operations in the computer, because the operating system itself must control them within the main memory of the computer. Each object within a particular program, however, may pursue its own *lightweight* thread within the entire program's address space.

An analogous situation would be that of two different college lectures (*definitely* heavyweight processes), each with a professor and a group of students (the lightweight threads), with blackboards and notebooks (their total surface area being the process's address spaces). In Professor Igor Tistical's classroom, the students sit in rapt attention as the lecturer expounds upon his specialty, and they intently copy into their notebooks exactly what he writes on the blackboard. In Professor I.M. Boring's classroom, however, students doodle in the margins of their own notebooks, scribble notes on those of their neighbors, and even run up to the blackboard to draw diagrams explaining their questions to him...and some of them even take lecture notes.

In the first address space, the threads are executing in parallel, each performing an identical task; in the second address space, the threads are involved in message passing and writing all over their common address space. On the one hand, the second space may be a messy whirlwind of activity as compared to the first, but we should remember that the doors to each classroom/process are locked, and the walls are soundproofed: no student/thread from one space may invade another space and write something there. Under normal conditions, the heavyweight threads are executing *oblivious* to each other[3].

3.3 Pseudo-Parallelism

While several lectures may be given within a college building simultaneously, a computer only *appears* to be accomplishing several tasks at the same time. User 1 might be editing a program, while User 2 is reading e-mail, User 3 is using the debugger, User 4 is cursing at a long list of compiler error messages being scrolled up the screen, and so on. Each of them, sitting in front of a different terminal connected to the same computer, seems to be able to use the computer's resources at exactly the same time as her or his neighbor, but this is not so. The simultaneity is an illusion, created by the computer running each user's process into and then blocking at scales of time so small as to make it appear that everything is happening at the same time. The **context switching** that the computer employs to start and stop processes shows a major bottleneck in the computer: the central processing unit, or CPU.

At any one time, the CPU can only handle one process, but because it can switch bits in and out in nanoseconds, a human observer can be fooled

[3] A different situation occurs in the case of remote objects, a subject we'll deal with in Chapter 4.

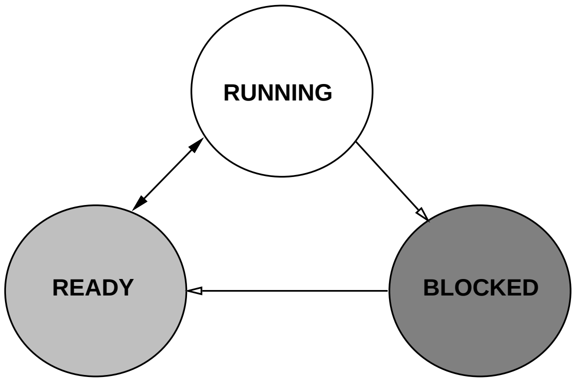

Figure 3.4: The three possible states, and transitions, for a process.

into thinking that all processes are being worked on at the same time. This is comparable to a “flip-book” of pictures which, if one flips through the pages fast enough, gives the illusion of an animated image.

Actually, the illusion of simultaneity within the computer is further complicated by the fact that the myriad of processes within the computer may be in any one of three different states: **ready**, **running**, or **blocked**. When a process is ready, it is waiting for a chance to continue running; when it is running, its thread of execution is allowed to continue working on the program; and, when it is blocked, the process is waiting for a resource (e.g., printer, tape drive, network connection, etc.) which is presently unavailable. Of course, while one process is blocked, another is usually running in the CPU. This **scheduling** of resources must balance the needs of users (who each want to get their own job done ASAP) with the capacities of the system (we can’t have everyone sending a file to the printer at the same time; the resulting papers would be hopelessly garbled).

3.4 Synchronization Issues in Concurrency

Object-oriented systems are inherently parallel: if we have a group of interactive objects within a system, they should all be able to operate at the same time, in some approximation of simultaneity. Anything less would waste much of the benefit of the data abstractions known as objects.

This **concurrency** is an easy concept both to understand and to model, *if* the concurrent threads never interact. Clearly there could be no problem, for example, for two threads to run simultaneously if one were searching a database and the other computing a factorial. Problems arise, though, when two or more threads try to access—and change—the same data[4], or when one thread needs the result of another to continue.

3.4.1 Race Conditions and Mutual Exclusion

Like two Indy 500 racers trying to push each other out of the way to get to the pole position on the track, each process (given the chance) will try to push another out of the way and finish first. Clearly, in their zeal to zoom through a narrow bit of track and come out the other end in first position, the racers can pull alongside, lock wheels together, and prevent each other from moving into proper position; both lose.

To prevent such a thing happening to threads within the computer, we make note that certain structures of the code, like the pole position on the racetrack, are so-called **critical sections** where we must be extra careful to avoid collisions of threads. Suppose that four threads are approaching a data structure: two (A and B) want to read from it, two (C and D) want to write to it. The data structure itself is a critical section of code because it must be completely intact when any of the threads tries to enter it. If C or D enters the data structure, their goals will be to alter the data structure, so that during their transit of its contents the data structure will *not* be intact; if A or B enters the data structure, they will only try to extract information, but they will not try to alter it, so the data structure *will* be intact during their transits.

Now here's the problem: if A or B is within the data structure while C or D is in it, too, then the readers can't be assured that they have an intact version of the data structure. Indeed, they may have some sort of hybrid form of the data structure, containing old data plus *some* new data, but not *all* of the changes C or D tried to enter into the data structure. Likewise, if *both* C and D are within the data structure at the same time, one could overwrite part of a change of the other, and the result would also be a bizarre hybrid version of the correct data structure.

The solution is to protect these critical regions through the principle of

[4] The same problems crop up when accessing peripherals, but we will concentrate on shared memory.

mutual exclusion: block all threads until a single thread can complete its business within the critical region. For the example above, commonly known as **the readers and writers problem**, we allow only one writer at a time, and lock out all other readers *and* writers attempting to enter; when the writer's done, someone else can come in. Readers, on the other hand, since they make no changes to the critical section, have no limits on the number of them that can come in at any one time. If, however, one writer knocks on the door to be let in, all readers have to leave and be locked out.

3.4.2 Atomic Blocks

To protect instance variables from being accessed or modified by more than one thread at a time, MELDC provides the **atomic block**. Atomic blocks look much like the normal MELDC blocks, but begin and end with angle brackets rather than curly braces. Atomic blocks also behave like MELDC blocks in most respects—including the ability to declare variables at the beginning of them. The important difference is that only one thread of control can enter an atomic block at a time. In fact, not only the atomic block is protected, but the whole object is. If thread A enters an atomic block, for example, all *other* threads attempting to access a method in the object containing the atomic block (through a message sent to one of the object's methods) will go to sleep until thread A leaves the atomic block. In addition, any thread already executing one of the object's methods will also go to sleep. This ensures that no unwanted reader or writer can sneak into the critical section at an inopportune moment and refer to the object's instance variables. If more than one thread reaches the start of the block at the same time, only one is allowed into it and the rest are put to sleep and queued up, the first one entering the atomic block only after the active thread has left it.

Note that the atomic block only puts all *other* threads to sleep if they attempt to access the object. The thread inside the atomic block can call any of the object's methods and continue to execute, which allows recursive calls under the atomic block's protection.

To use an atomic block, simply put angle brackets around the protected code, the critical section, as in Figure 3.5.

```
class bank_account ::=
  int balance;
methods:
  int deposit(int dep_amount) {
  <         /* atomic block begins */
    balance += dep_amount;
    return (balance);
  >         /* atomic block ends */
  }
  ...

end class bank_account
```

Figure 3.5: A Bank Account Object using an Atomic Block

3.4.3 `delayuntil` and `respond`

`delayuntil` and `respond` are MELDC's means of synchronizing threads of control. The basic syntax for both of them is the keyword followed by a string (a string constant or character array or pointer). This string acts as a label for multiple uses of `delayuntil` to match their respective `respond`s. Thus

```
delayuntil "hark, who goes there?";
```

matches

```
respond "hark, who goes there?";
```

`delayuntil` and `respond` statements can also be tailored to specific objects. If we wanted to have an object wait until it received a response from a particular object, we would use the pair `delayuntil` and `from`:

```
delayuntil "hark, who goes there" from rosencrantz;
```

If this statement were found in an object named `guildenstern`, `guildenstern` would sleep until an object named `rosencrantz` executed the statement:

```
respond "hark, who goes there?" to guildenstern;
```

When a thread executes a `delayuntil` statement, it suspends its execution until some other thread executes a matching `respond` statement. The thread, in effect, goes to sleep until it is told to reawaken by a personalized wake-up call.

Many different threads can execute the same `delayuntil` statement. In this case, *all* the threads go to sleep. They are then queued up to wait for a suitable response. When another thread executes a matching `respond` statement, the first delayed thread wakes up. Subsequent responses wake up the rest of the delayed threads.

It is not an error for a thread to execute a `respond` statement that doesn't match any previous `delayuntil`. If a thread executes a general `delayuntil` (that is, one not directed `to` any particular object) matching a previous respond, it is considered automatically responded to. Though this smells of time travel, it is actually a necessary part of the language. Since the user has no control over the speed of his or her threads, this behavior synchronizes them no matter which is faster.

`Delayuntil` and `respond` only relate to the concept of the MeldC thread. They have nothing to do with features; it is allowed (even necessary at times) for objects belonging to one feature to `delayuntil` and wait for a `respond` from an object in another feature.

3.4.4 A Warning

MeldC is a language specifically designed with concurrent programming in mind. Therefore, solutions for the problems of race conditions and synchronization have been implemented as parts of the language. Yet bugs stemming from concurrency are very difficult to find and fix. They reveal themselves irregularly, usually lurking in the background, ready to spring out at some especially inopportune moment. Taking some time to master `delayuntil`, `respond` and **atomic blocks** will prevent much weeping and gnashing of teeth.

Chapter 4

Distributed Programming

4.1 The General MeldC Model

There are two reasons for concurrency in MeldC:

- It fits well with the object-oriented paradigm.
- It can be more efficient for some computations to run in parallel.

If we are limited to pseudo-parallelism for our concurrency—that is, if our threads are all running on one processor—we aren't gaining any efficiency at all from concurrency. If, however, we had the ability to create threads of control on other processors or machines, to distribute work over some network, the possible efficiency gains are quite large. MeldC is designed for the distributed environment, and the way in which one MeldC program on one machine interacts with another is by **getting** and **putting** objects.

MeldC attempts to be **transparent** in its approach to distributed programming. Most of the time we don't have to be concerned about whether a particular object actually exists on this machine or the other. Apart from the "creation" of these external objects, they appear just like any other local object—though this transparency does have its limitations, as we will see at the end of this chapter. Accessing remote objects is easy; it is the getting and putting of them that is out of the ordinary.

4.1.1 The Protocol Object

Networks and connections between computers are complex by nature. For MELDC to communicate in this complex environment it needs a **protocol**, an agreed-upon means of communication to define how a MELDC program running on one machine can talk to the program running on another. MELDC, as an object-oriented language, views this protocol as an object like any other object: complicated, yet having the same basic overall structure as `complex_number` or `rectangle`. A **protocol object** provided by MELDC must be imported by any feature dealing with remote objects, yet the file containing the object need not be included on the command line when the MELDC program is compiled since protocol objects are `library objects`, which are explained more fully in Chapter 7.

4.2 The Nameserver

Normally, distributed programming is accomplished in MELDC through the use of a **nameserver**, a process which acts as an intermediary between different processes and machines. When an object is **put** from a feature, it's name, place and other information is regestered with the nameserver, and becomes available for other processes to **get** from the nameserver. To use the nameserver the `ns_protocol` object must be imported from feature `NS_Protocol_Obj`:

```
interface:
  imports NS_Protocol_Obj[ns_protocol]
```

4.2.1 Getting and Putting

For one MELDC process to access a remote object, there must be some remote machine **offering** that object. A MELDC program shares its objects with other machines and programs through the `PutObj()` MetaClass method.

Figure 4.1 shows a simple example of how an object is **put**. From it we see that the method `PutObj()` takes three arguments. The first is the object to put, the second a string indicating the name of the object as

```
feature put_feature
interface:
  imports NS_Protocol_Obj[ns_protocol]
  imports public_feature[public_class]
implementation:
  public_class public1 = public_class.Create();
  driver_class driver = driver_class.Create();

  class driver_class ::=
  methods:
    void init() {
      public_class.PutObj(public1,
                          "public1",
                         ns_protocol);
    }
  end class driver_class
  ...

end feature put_feature
```

Figure 4.1: An Example of `PutObj()`

it will be known on other machines, and the third is the protocol object, `ns_protocol`. `PutObj()` is a MetaClass method [1]: in the example the message `PutObj(...)` is sent to class `public_class`. It is important that the class the `PutObj()` message is sent to, the *class-name* in

classname.`PutObj(`*object*, *object-name*, `ns_protocol);`

be the same as the class of `object`, the object to be put.

This process of putting an object is actually one of *registering* the object with the nameserver. Objects are registered with the name *object-name*; if another object is subsequently registered with the nameserver with the same name, it supplants the old object.

Getting objects from the nameserver is just as easy, using the `GetObj()` MetaClass method. In this case, only the name of the object desired and the nameserver protocol object are arguments to `GetObj()`:

[1] The MetaClass is the class of the class; a slightly mind-bending concept, but one that is explained in chapter 4

object = *class-name*.`Getobj`*(object-name,* `ns_protocol`*)*;

If no object of *object-name* is regestered with the nameserver, *class-name*.`GetObj()` will return a null-object.

4.3 The Low-Level Approach

At times, more direct connections may be needed between processes, connections that shouldn't pass through the nameserver. MELDC's most primitive mode of distributed programming involves each process's **putting** and **getting** objects from other processes. To access this level, import the protocol object `remote_protocol` from the feature `RemoteObj`:

```
imports RemoteObj[remote_protocol]
```

4.3.1 Putting an Object

The process of putting objects, under the low-level approach to distributed, isn't much different from how we do it using a nameserver. The only difference involves the protocol object used. Instead of using the `ns_protocol` protocol object from feature `NS_Protocol_Obj`, we use the protocol object `remote_protocol` from the feature `RemoteObj`. So a feature using the low-level approach must include

```
interface:
  import RemoteObj[remote_protocol]
```

To actually put the objects, a normal `PutObj()` call is used:

classname.`PutObj(`*object, object-name,* `remote_protocol);`

4.3.2 Getting an Object

Without a nameserver, MELDC program not only needs to know the *name* of the object to get, but also the *place*. Thus this is a more complicated task than putting, and more information is needed.

```
#include <sys/socket.h>

feature get_feature
interface:
  imports RemoteObj[remote_protocol]
  imports public_feature[public_class]
implementation:
  public_class public2;
  driver_class driver = driver_class.Create();

  class driver_class
    struct sockaddr *sin;
  methods:
    void init() {
      remote_protocol.dest_addr(&sin,
                                "cunixb.cc.columbia.edu",
                                6001);
      public2 = public_class.GetObj("public1",
                                    remote_protocol,
                                    sin);
    }
  end class driver_class
  ...

end feature get_feature
```

Figure 4.2: An Example of `GetObj()`

Figure 4.2 shows a sample program employing the `GetObj()` method. We will go through it step by step.

We include `<sys/socket.h>` because we find the definition of `struct sockaddr` there, which is necessary for the `GetObj()`. A variable of the type `struct sockaddr` is where we store the origin of the remote object. In the example we have used the variable `sin`.

To assign valuable data to `sin` we use the protocol object `remote_protocol`'s method `dest_addr()`. This method accepts as parameters the address of a variable of type `struct sockaddr` and two other arguments—a `char` pointer representing a hostname, and an integer representing a port number—and encodes the hostname and port number into the `struct sockaddr` variable.

Host addresses should be fairly familiar, but port numbers seem slightly arcane. Understanding ports is thankfully not necessary to use remote objects, but it is necessary to choose a port number greater than 6000, as ports addressed lower than that are generally reserved for the operating system's use. Using port 9954 may also be a bad idea, as that port is used by the MeldC nameserver.

After `sin` has been set to the correct value, the last step in getting a remote object is using the `GetObj()` method itself. `GetObj()` takes three arguments: the name of the object to look for (a `char` pointer), the protocol object, and the `struct sockaddr` variable with the information on where to look. If successful, it returns the object requested, which must be cast to the class of the object. The general form of statements using `GetObj()` is:

object `=` *class*`.GetObj(`*object-name*,`remote_protocol`,*socket-addr*`)`

You may notice that `GetObj()` takes three arguments here, while it only took two when we used the nameserver. *socket-addr* is an optional parameter in the definition of `GetObj()` for low-level use.

4.4 Transparency

MeldC remote objects are designed to work transparently, by which we mean that to a user there's no apparent difference between the behavior of remote and local objects. However there are some situations in which this is impossible.

4.4.1 Passing Objects to Methods

When objects are passed as arguments to methods, synchronously or asynchronously, they are passed by reference, not by copy. In effect, they are passed like pointer values are passed to conventional C functions. This shouldn't matter in most cases, but it does make passing objects as parameters to remote objects difficult. If the remote process doesn't have the class definition for the passed object, it will arrive at the remote machine as so many uninterpretable bytes.

In short, objects may only be passed as parameters to remote methods if their class definition exists on the remote machine. And classes cannot be passed as parameters at all.

4.4.2 Network Failure

If the network dies between two machines, one of which has `put` an object and the other has **got** it, use of the remote object is impossible. If communication fails there are two noticeable effects:

- `GetObj()` returns null. This will occur whenever `GetObj()` is unable to fulfill a request, whether for the reason that the host given it by the *socket-addr* variable is unreachable, or because no object of the correct name has been **put** on that host.
- *remote_object.method*() returns zero. Any messages sent to a remote-object with a broken connection will get a zero for a return value, or a null pointer if the method would normally return a pointer.

It's easy to test if `GetObj()` returns a null: a few test statements in our code can catch errors resulting from network failures if we are trying to get an external object. If we already have an external object, however, gracefully handling a network failure is more difficult, for zero is often used as a return value. If the network is trustworthy, handling network failures at this level is probably more trouble than it is worth.

4.4.3 Synchronization and Atomic Blocks

Currently `delayuntil`, `respond`, and atomic blocks do not work in MeldC's distributed environment. This is an area that is under development, and active research is ongoing on the subject.

4.5 Garbage Collection

It should be noted that MeldC is not uniform in passing pointers for local and remote applications. Here are some of the cases:

1. An integer pointer cannot be passed to a remote source.
2. A structure cannot be passed to a remote source.
3. A char * is passed as NULL.

There is also a problem with objects. There will be two copies of an object. One on the remote side and one on the local side. When you free the memory of an object on your local side, the remote side object copy will remain. You are not allowed to free the object on the remote side. This problem will leave objects that no longer needed on the remote side, there by creating a garbage collection problem.

Chapter 5

Persistent Programming

5.1 The General MeldC Model

Often in large systems it is desirable to imbue objects with the quality of *persistence* for purposes of fault tolerance or system-related aspects. This persistence can have many forms, but the common thread among them is that the data for the objects with the property of persistence should be safe from deletion upon the object's (or system's) demise.

5.1.1 The Persistent Protocol Object

MeldC provides most primitive mode of persistent programming involves each process's `putting` and `getting object` from a external storage (i.e. the disk). To use the persistent object protocol, import the protocol object `persistent_obj_protocol` from the feature PersistentObj:

```
interface:
  imports PersistentObj[persistent_obj_protocol]
```

5.1.2 Getting and Putting

The process of putting objects to the external storage device and mark the persistent, isn't much different from distributed programming. The only difference involves the protocol object used. Instead of using the `remote_protocol` protocol object from feature `RemoteObj`, we use the protocol object `persistent_obj_protocol` from the feature `PersistentObj`. So a feature using that uses the persistent object must include

```
interface:
  import PersistentObj[persistent_obj_protocol]
```

To actually put the objects, a normal `PutObj()` call is used:

classname.`PutObj(`*object, object-name,* `persistent_obj_protocol);`

Under the our current `persistent_obj_protocol`, once the object acquire the persistent behavior, it will stay persistent until the object is destroyed. Issue a `PutObj()` to a persistent object will flush the current object's state to the external storage.

Getting the object from the external storage is just as easy, using the `GetObj()` MetaClass method. In this case, only the name of the objects and the persistent object protocol are the argument to the GetObj:

object `=` *classname*.`GetObj(object-name, persistent_obj_protocol);`

The Getobj() will return null-object, if the object does not exist in the external storage.

Figure 5.1 shows a simple example of how an object is putting to a external storage as well as restoring from a external storage.

```
feature Persistent_feature
interface:
  imports PersistentObj[persistent_obj_protocol]
  imports public_feature[public_class]
implementation:
  public_class public1 = public_class.Create();
  driver_class driver = driver_class.Create();

  class driver_class ::=
  methods:
    void init() {
      public_class public2;

      public_class.PutObj(public1,"public1",
                          persistent_obj_protocol);

      public2 = public_class.GetObj("public1",
      persistent_obj_protocol);
    }
  end class driver_class
  ...

end feature put_feature
```

Figure 5.1: An Example of `Putting and Restoring the Persistent Object`

Chapter 6

Dynamic Composition of Object's Behavior

Sometimes, a programmer needs to change the behavior of a MeldC program "on the fly." The power to dynamically change program behavior, traditionally only provided by the dangerous practice of self-modifying code, can be extremely useful in debugging large programs and auditing large object-bases. It has more advanced uses, as well: All of MeldC's distributed programming aspects are based on the concept of dynamically modifying program behavior to form a link to other MeldC processes. In MeldC however, this power does not come from self-modifying code. Rather, MeldC offers the dynamic extension of object behavior through the use of the reflective architecture. The extended behavior of an object is referred to as its *secondary behavior* to distinguish from the *primary behavior* defined in the class of the object. Extending object behavior in MeldC is characterized by two properties: (1) composability and (2) decomposability. Composability states that primary behavior, which implements the interface of objects, can be modified by composing with multiple secondary behaviors without changing the objects' interface. Decomposability describes the reverse property of composability. Primary behavior encompasses an object's functionality as defined in the object's class definition or provided through an inheritance mechanism. Secondary behavior encompasses dynamically added functionality which is in most cases orthogonal to primary behavior and to other secondary behaviors.

MELDC provides a mechanism call *shadowing* to implement secondary bevaviors. The idea "shadow" implies a dynamic, transient and orthogonal effects upon primary behavior. A shadow object is an object which intercepts messages addressed to some base object and processes them before (perhaps) sending them to the base object. The shadow object can also process the value returned by the base object. Thus, by controlling all access to the base object, it can redefine the behavior of that base object without actually modifying code.

6.1 Writing a Shadow Object

Shadow objects are created just as other objects are; through their *class-name*`.Create()`. Defining the class for a shadow object is quite normal as well, but in order to have it act properly as a shadow object, some special methods must be defined. Using these methods (and thus using shadow objects) requires that the file `"meldc_user.h"` be `#include`'d in the program file before the feature declaration.

The entry-point method A method (of any name) may be used as the "entry-point method" for the shadow object. This method will be executed whenever the base object is sent a method; in other words, this method intercepts any messages going to the base object. This method must have both a return value and a parameter of type `struct _frame *`. If we had named the method `intercept`, for example, it might be defined as:

```
struct _frame * intercept(struct _frame *fp)
   ...
```

Advanced uses of shadow objects such as those used for remote objects arise from the manipulation of the passed **frame** parameter, but they are beyond the scope of this manual. Those wishing information about this subject are welcome to look at the MELDC code for the implementation of remote objects.

The return value for this method determines whether or not the base object's method will execute. If the parameter frame is returned, the base object will execute as if it received the message originally passed. If this entry-point method returns a NULL, however, the base object will not execute.

The exit-point method Another method may be defined as the exit-point method for the shadow object. This method will execute after the base object finishes its execution — and thus will not execute at all if the shadow object's entry-point method returns a NULL. The exit-point method is of return-type `void` and has two parameters, one `struct _object *` and one `struct _threadid`. For example:

```
void clean-up (struct _object *obj, struct _threadid t)
   ...
```

Again, the parameters can be used for advanced applications, but such applications are beyond the scope of this manual.

The init-point and dest-point methods Two other special methods can be defined; the init-point and dest-point methods. These can also have any method name. The init-point method executes when the shadow object is attached to its base object (see Section 6.2), and takes as parameters those passed by the `AttachObject` method. The exit-point method executes when the shadow object is detached from a base object, and gets passed all parameters of the `DetachObject` method.

6.2 Attaching and Detaching

Attaching and detaching shadow objects are simple operations. To attach a shadow object to a base object we use:

> *base-class*.AttachObject(*base-object, shadow-object,*
> *entry-point-method, exit-point-method,*
> *init-point-method, dest-point-method*);

where *base-class* is the class of the base object, *shadow-object* is the shadow object to be attached, and the other parameters (of type `char *`) being the names of the corresponding methods in the shadow object. If a method is not defined in the shadow object it may be safely omitted from the parameter list, or a NULL pointer may be used as a placeholder. Only the entry-point method must exist in shadow objects.

To detach an object we use:

base-class.DetachObject(*base-object, shadow-object*);

If a NULL is passed as the *shadow-object* parameter, *all* shadow objects will be detached from the base object, otherwise only the specified shadow object will.

6.3 Multiple Shadow Objects

One base object may have more than one shadow object. In fact, there are two basic ways this might occur: when many shadow objects are attached directly to a base object, or when a "chain" of shadow objects are attached to a base object.

In the first case, that of many shadow objects directly attached to their base object, all shadow objects will always execute their entry-point methods whenever a method is directed at the base object. If *all* shadow objects' entry-point methods return their passed frame pointer rather than a NULL, the base object will then execute. The base object will not execute if any entry-point method does return a NULL.

If the setup is a chain of shadow objects, the behavior is slightly different. When a message is directed to the base object, first the outermost shadow object will intercept it. If its entry-point method does not return a NULL, the next shadow object down the line will intercept the message, and so on down to the base object. So if any shadow object does return a NULL, all execution stops there.

6.4 Shadow Object Examples

6.4.1 Object Composition

Shadow objects must be thought of as a regular object that simply extends the functionality of it's parent class. The idea of *object composition* consists of taking a shadow object and attaching it to another object, it's parent object. The result is effectively a cross-product of the two objects, where each individual object's identity is preserved. Dynamic composition allows the programmer to dynamically enhance, add or eliminate, a statically defined object.

6.4.2 An Example of Object Composition

Let us look at a simple example of dynamic composition. A class **Savings_Account** describes two methods, **deposit** and **withdrawal**. Depositing and withdrawal are the primary behaviors of every instance of **Savings_Account**. Yet a manager may decide to audit the activities of a particular savings account. To do so, he does not need to modify the definition of the class. All he needs to do is attach a shadow object with the "audit" behavior to the account object. This shadow object is simply a modifier to the **deposit** and **withdrawal** methods, so that the behaviors are now audit deposit and audit withdrawal. When the manager is ready, he can then remove this shadow object at anytime.

6.4.3 A Tracing Example

If you the user wanted to write a tracing program, it could be implemented rather easily using Shadow Objects. Instead of having to modify your code, a Shadow Object can be created that will implement a trace for us. Then the Shadow Object could be attached to the object(s) that you would like to trace.

When the program is run, the Shadow Object will intercept the message. The Shadow Object will then run the trace code, such as printing the information you want to the screen so that you know what object is being executed. The Shadow Object will then pass the received message to it's Parent Object. The Parent Object will execute the message and return some value.

6.4.4 A Persistent Object Example

If a Shadow Object was attached to a Persistent Object, then the Shadow Object will do the following. It will first intercept any messages for it's

Parent Object. The Shadow Object will then flush the Persistent Object to a disk to save the object's state, keeping the object as Persistent. Once the object has been flushed to disk, the Shadow Object will give the intercepted message to the Parent Object. The Parent Object will then execute the message. Once it is done executing, the Parent Object will return some value to the Shadow Object. The Shadow Object will then forward the message to the object that sent the message.

6.4.5 A Remote Object Example

When a message is received at a Shadow Object associated with some Remote Object, the Shadow Object will send the message across the network to the Remote Object. The Remote Object will receive the message through it's own Shadow Object and process this message. When it is done, the message will be returned by the Remote Shadow Object to the original Shadow Object that intercepted the message. This Shadow Object will then forward the return value to the object sending the original message. All of this will make it seem as if the Remote Object is on the local side.

Chapter 7

MeldC Library Objects

The MELDC distribution contains a number of predefined objects which should be used for various system and I/O-related tasks. These objects, ananogous to conventional C's library functions, are treated just as any other MELDC object is, but for the fact that the files containing them need not be included on the compiler's command line—the MELDC compiler knows where to find the files in the event the objects are imported.

This chapter will introduce the the **system object** and **memory objects**. **Protocol objects**, library objects used for more sophisticated I/O and inter-process communication, are covered in Section 4, Distributed Programming.

7.1 The System Object

MELDC's construct for communicating with the operating system is the system object. Like all library objects (and all objects not defined in the current feature), we must **import** it to use its methods:

```
imports Unix[sys_obj];
```

The system object is used in the manner of any other MELDC object, through synchronous or asynchronous messages. The messages and parameters the system object accepts depend on the operating system on which

MeldC is running (The UNIX system calls MeldC supports are listed in Appendix D). The messages accepted by the system object, though, are invariably in the same format as the corresponding C system call, with the same name and accepting the same arguments. Thus, if we wanted to use the `write()` system call under UNIX, the statement would be:

```
sys_obj.write(fd, buf, nbyte);
```

or

```
send write(fd, buf, nbyte) to sys_obj;
```

This brings up the question of why we can't just use a conventional C function call in an object method, rather than using these system objects. There are two reasons:

- Since system calls are actually communications with the operating system, it is natural to view the operating system as a coherent object that can have messages sent to and received from it.
- If a MeldC thread uses a conventional C function call, the operating system will put the MeldC program into a blocked state. If, on the other hand, a MeldC thread sends a message to the system object, *only that thread will be blocked.*

That last point deserves some clarification. In Chapter 3 we discussed the concept of the thread, and saw that threads can be in one of three states:

ready The thread is about to be executed, and is waiting to be scheduled.

running The thread has been scheduled, and will continue running until it is put back in a ready state by the scheduler to allow time for another thread to run, or until it is blocked.

blocked The thread is not running, and cannot run until some event happens. A thread is blocked, for example, when it is waiting for some I/O to finish. When the thread is unblocked it enters the ready state and waits to be scheduled.

Using C system calls not only blocks the MeldC thread that makes the system call, but all other threads in the current MeldC program as well by putting the entire program process in a blocked state. In other words, conventional C system calls put heavyweight threads in blocked states, while messages sent to system objects block only lightweight threads.

7.1.1 The `exit()` Method

Some care must be taken in exiting MeldC programs. While the C statement

```
exit();
```

will stop the program's run, it may also produce unintended side effects. The graceful way to cause a MeldC program to finish is to call the **`exit()`** method of the system object:

```
sys_obj.exit();
```

While this takes care of MeldC's requirements for a clean exit, it may still cause strange behaviour if more than one thread is active when the program exits. It is the programmer's responsibility to make sure all other threads have finished executing (if all threads in that program do need to finish) before any thread reaches the exit method-call.

7.2 The Shell Object

The shell object is used to get information from the shell from where the MeldC program is run: In particular, it is used to get the values of **`argc`**, **`argv`**, and **`envp`**, the number of the arguments used when the program is run, a list of the arguments, and a pointer to the shell's environment string. It's use is straightforward: Any feature that needs access to this shell information must first import the shell object:

```
imports Unix[shell]
```

`shell` has three methods, `argc()`, `argv(num)`, and `envp()`—`argc()` and `envp()` return the corresponding values, and `argv(num)` returns the *num*-th argument in the argument list. So, in order to get the third argument, we would use

```
arg = shell.argv(3);
```

after importing the `shell` object.

7.3 Memory Objects

The conventional C functions `malloc()` and `free()` should not be used in a MeldC program. To access free memory, operations of this type must pass through a **memory object**.

There is no single memory object as there is a system object. Instead, any class whose methods need to use `malloc()` and `free()` should *inherit* the memory class. If class `foo` had a method that was to malloc a block of memory, the class's declaration should begin with

```
class foo merges MemoryAllocation[Memory] ::=
```

thus merging the class `foo` with the class `Memory` of the `MemoryAllocation` feature. As is normal for inheritance, it isn't necessary to import the `Memory` class in the interface section; inheriting it will automatically import it. Nor is it necessary to specify a filename for the file containing featuer `MemoryAllocation` on the `mcc` command line: Since it is a library class, MeldC knows where to find `Memory` when it is called for.

Once a class is merged with the `Memory` class, its methods can malloc and free memory by calling its own `malloc()` and `free()` methods inherited from the `Memory` class. As explained in Section 1.3.4, the special object `$self` is used when objects are to send messages, or call, their own methods. One of the methods in `foo`, for example, could malloc 128 bytes of memory with the statement

```
ptr = $self.malloc(128);
```

The class can use a corresponding free statement (in that method or another) to deallocate the memory:

```
$self.free(ptr);
```

There is a caveat to allocation and deallocation with the memory object, however: An object cannot free memory allocated by a different object. So if one object mallocs a block of memory, only that object can free it. Normally this shouldn't be much of a problem, as the pointers to malloc'd memory will usually be instance variables, only visible to the object that created it. If, however, class variables (static instance variables) or global variables have access to malloc'd memory, there may be problems which could lead to coredumps or other inexplicable behaviour.

Chapter 8

Software Tools

8.1 The MeldC Debugger

The MeldC debugger, `mcgdb` is built upon `gdb`, the Gnu Project's `C` debugger. This allows `mcgdb` to provide a means to debug MeldC code with a standard interface, yet also to use `gdb`'s ability to step through and examine the `C` statements which may compose the majority of a MeldC program's methods. As with `gdb`, in order for a MeldC program to be debugged with `mcgdb`, it must have been compiled with the `-g` compiler switch set.

This section is not a primer on general debugging; a basic knowledge of the use of `gdb` is assumed. Only MeldC specific commands of `mcgdb` are explained, as well as various points necessary for the successful debugging of a MeldC program.

8.1.1 Scope and the $ Separator

All variables are printable inside of an `mcgdb` debugging session, and breakpoints can be set at any method, but it'll take more than a few keystrokes to do so. The full address of any variable or method is the featurename and object or classname followed by variable or methodname. For example, to display the variable `area` of the object `rectangle` in the feature `geometry`, we would use:

```
print $geometry$rectangle$area
```

Or to set a breakpoint in that same object's `lengthen` method:

```
break $geometry$rectangle$lengthen
```

Which brings up the subject of scope, since it is not actually necessary to use the full "address" for each operation, thus saving valuable keystrokes (and perhaps averting carpal tunnel syndrome). In the middle of a trace or after a debug, any variables local to the current method can be accepted just by using their names. Any local to the current class can be accessed through *classname$variablename*. And, of course, anything can still be addressed through its full address.

8.1.2 `mcgdb` Commands

print This command displays the contents of a variable. Variables local to the current method may be addressed just by name, those local to the class by *classname$variablename*, and any variable may be addressed by *$featurename$classname$variablename*.

break This command sets a breakpoint at which execution will stop. The breakpoint specified must be a method. The first breakpoint specified *must* be specified by its full address (feature, class and method), though all subsequent breakpoints set can be addressed in any allowable manner.

where This command displays the MeldC stack; that is, the stack of MeldC function calls for the current executing thread. For most debugging purposes this should suffice.

c-where This displays the stack of C function calls, rather than MeldC function calls. Though the debugging resolution can be finer with this command, the information given can be confusing.

up, down These commands allow movement up or down in the MeldC runtime stack.

c-up, c-down These allow movement along the C stack.

list This displays the MeldC statement which will be executed next.

meldcq When used without an argument, this displays the first method name in each of the queues used for concurrency. When a number is given as an argument, the names of all the methods in that particular queue are shown.

mccprintfc This displays all feature and class names in the MeldC program.

mccprintfcm And this command displays all feature, class and method names.

Chapter 9

Advanced Topics

9.1 MeldC Optional Parameters

The MeldC language supports the use of optional parameters of methods. Programmers can declare a MeldC method with optional parameters and reference the optional parameters using macros. This is a very useful feature in a programming language which allows programmer more flexibility by making modules more compatible and maintainable; little changes is required if suddenly a method needs an extra parameter.

9.1.1 Syntax

9.1.2 Declaring a method with optional parameter

```
return_type method_name(formal_parameter_declarations, optional)
{
  ...
}
```

The keyword `optional` appearing after (always) the required formal_parameter_declarations says that this method have optional parameters.

```
        int foo(int x, optional)         /* x is required
```

```
{                                  * & other optional parameters
  ...                              */
}
int bar(optional)                 /* no required parameters
{                                  * just optional parameters
  ...                              */
}
```

9.1.3 Referencing a particular optional parameter

When the programmer references a a particular optional parameter passed into the method, it is assumed that the programmer knows the type of the optional parameter he is expecting, and also the position of the optional parameter, whether it's the 1st or 2nd or the nth optional parameter.

A pointer variable of that type has to be declared first, and the macro `ARG_ADDR(i)`, where i is the position (range from 0-the number of optional parameters passed in), can be used to reference the address of the desired optional parameter in the following fashion :

```
parameter_type *ptr;
ptr = ARG_ADDR(i);
```

And then by dereferencing the pointer variable, the programmer can access the value of the optional parameter. For example, assume that the function foo() has 1 optional parameter and it's of type int.

```
int foo(optional)
{
   int *opt_ptr;
   int x;

   opt_ptr = ARG_ADDR(0);
   x = *opt_ptr;
}
```

9.1.4 Referencing All Optional Parameters Passed in as a Whole

Sometimes it may be desirable to reference all the optional parameters passed in as a whole and pass into another method as an argument. This kind of reference can be done by $option anywhere in the method.

This illustrates the flexibility that optional parameters allow. The method foo() does not have any knowledge about what parameters it gets, but just directly pass them down to the method bar() which will eventually decode them. It the parameter declaration of method bar() need to change, the parameter declaration of method foo() will not be affected.

```
e.g.
obj.method($option);
        send method($option) to obj;
        obj.method(1, "hello", $option);

        int foo(optional)
        {
           $self.bar(1, "hello", $option);
        }
```

9.1.5 Mapping Optional Parameters

```
-------------------------------------------------------------------------------
actual arguments        formal parameters        mapping
-------------------------------------------------------------------------------

(i, "hello")            (int x, char *s)        i -> x, "hello" -> s

(i, "hello")            (int x, optional)       i -> x, MSG has 1 arg: "hello"

(i, "hello")            (optional)              MSG has 2 args: i, "hello"

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

(i, $option)            (int x, char *s)        i -> x,
                                                1st arg in $option -> s
                                                rest of arg in $option discarded
```

```
(i, $option)            (int x, optional)       i -> x, all of $option -> MSG

(i, $option)            (optional)              i, all of $option -> MSG

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

($option)               (int x, char *s)        1st arg of $option -> x
                                                2nd arg of $option -> s
                                                rest of $option discarded

($option)               (int x, optional)       1st of $option -> x
                                                rest of $option -> MSG

($option)               (optional)              all of $option -> MSG

-------------------------------------------------------------------------------
```

9.1.6 Referencing individual optional parameters with Macros

Description of the Macros

Macros have been created to allow you, the user, access to the optional parameters. With the macros, you will be able to find out the argument size, the argument type, the address of the argument and the postfix type of the argument. The argument macros take an integer, passed as a parameter, that represents the arguments slot number in the list of arguments. The slot number of the argument is a well known number, since the order of the arguments is known. Below we will explain each macro separately and show you how it is carried out by showing you the actual code.

To find the size of some given argument, you can use the ARG_SIZE macro. Given the slot number, i, it will return the size of the argument in that slot.

```
#define ARG_SIZE(i)                                                         \
  ( (MSG != NULL) && ((int) MSG->_num_arg) < i ?                            \
    0:                                                                      \
    MSG->_para[i].type->size )
```

To find the postfix type of an argument in slot number i, you can use the ARG_POSTFIX_TYPE macro.

```
#define ARG_POSTFIX_TYPE(i)                                        \
  ( (MSG != NULL) && ((int) MSG->_num_arg) < i ?                   \
    (char  *) NULL :                                               \
    MSG->_para[i].type->postfix_type )
```

If you need to find out the argument type, (int, char, ...), then you can use the ARG_POSTFIX_TYPE macro. When you call this macro by giving the slot number, i, the macro will return the argument type.

```
#define ARG_TYPE(i)                                                \
  ( (MSG != NULL) && ((int) MSG->_num_arg) < i ?                   \
    (char  *) NULL :                                               \
    MSG->_para[i].type->c_type_desc )
```

To find the address of an argument, you can use the ARG_ADDR macro. Calling this macro by providing the slot number will give you the address of the argument. This address is found by adding the address of the message, the beginning, to the offset of the argument.

```
#define ARG_ADDR(i)                                                \
 ( MSG == NULL || MSG->_num_arg <= i ?                             \
    (printf("MeldC Error : Insuffiicent optional parameters \n"),\
     kill(getpid(),SIGQUIT),                                       \
    (int) NULL)  :              \
    (int *)((int)(MSG->_para[i].offset) + ((int) MSG)) )          \
```

9.2 Delegation

When MeldC receives a message that it does not understand from some remote location, we check to see if a function meldc_default exists or not. This function will then process the message and forward the message to the proper recipient when done. This concept is known as *Delegation*.

It is assumed that the meldc_default function takes at least one argument, the name of the method. The other arguments are optional and are provided by the user. These other arguments can be:

- The caller_name, which represents the name of the object that sent this message.
- The resp_message, which is a pointer to the return message.
- The message, which is a pointer to the message that was received that the meldc_default function must process.

9.3 Active Values

Traditional computer languages process instructions in a linear fashion, computing a result from some initial data. Many mathematical concepts favor a more open-ended approach, however. There are many simple equations in the form of the basic Fahrenheit to Celsius conversion:

$$C = 5/9(F - 32)$$

The equal sign here implies not only that $C = 5/9(F - 32)$ but also that $F = 9/5C + 32$.

MELDC's construct for handling this type of calculation is the **active value**. Active values are variables that, when assigned a value or changed, cause some side effect (typically assigning a value to another variable) to occur immediately after the assignment. Using active values we can assure that no matter whether F or C is changed in a Fahrenheit-Celsius relation, both values will be correct.

9.3.1 Active Value Syntax

MELDC active values are local to classes, and as such are declared in the class definition—in the methods section. The methods section of the class definition, in fact, consists entirely of a sequence of active value declarations and class definitions in any order, but the code is clearer if all active values are declared before any methods are introduced.

To explain the syntax of active values we have defined class `weather_station` in Figure 9.1. Our `weather_stations` store the temperature in both Fahrenheit and Celsius, but read the current temperature only in Celsius. To keep the value of `F` current, the instance variable `C` of this class is an active value, declared by the line:

```
class weather_station ::=
  float F, C;
methods:
  (C) --> { F=(9/5)*C+32; }

  float get_fahrenheit() { return(F); }

  float get_celsius() { return(C); }

  float set_temperature(float temp) {
    C=temp;
    return(F);
  }

end class weather_station
```

Figure 9.1: A Weather Station

```
(C) --> { F=(9/5)*C+32; }
```

This means that whenever `C` is assigned a new temperature, the variable `F` is adjusted to reflect the new temperature in Fahrenheit units. Specifically, whenever `C` is assigned a value, the statement

```
F=(9/5)*C+32;
```

is executed, setting `F` to the correct temperature.

Only instance variables (and class variables—`static` instance variables) can be active values: neither local nor global variables can have this property. It is also useful to note that active values are never inherited. An object will only have active instance variables if the variables are declared active in that object's class, *not* in an ancestor class. If a variable is declared active in some ancestor class, it is still an available instance variable—it simply isn't active.

In general, active values are declared in the form:

(*list-of-active-values*) `-->` { *code* }

Here, *list-of-active-values* is a list of instance variables and *code* is MeldC code to be executed whenever one of the active values are triggered. The *code* cannot have any message-passing statements *at all*, whether synchronous or asynchronous.

The behavior of an active value is simple. Whenever an assignment to an active value occurs, that active value's code is executed. To trigger the active value code, though, the assignment operator must be applied to the actual variable with the active value. The code will not be triggered through pointer indirection. So, for example, if a MeldC object had an active value defined as

```
(profs_salary)-->{admins_salary=profs_salary+10000;};
```

where `profs_salary` and `admins_salary` are initialized to 30000 and 40000, respectively, then evaluating

```
profs_salary += 500;
```

would trigger the active value, and `admins_salary` would be set to 40500. However, if there was a variable declared as

```
int *salary_pointer = &profs_salary;
```

then

```
*salary_pointer += 500;
```

while raising the base salary of professors to $30,500, will *not* trigger the active value; the administrators will stay at the $40,000 level.

9.3.2 Assignment vs. Change

MeldC distinguishes between assignment of a variable and change of a variable in its handling of active values; they can be set to trigger when the variable is *assigned to*, or alternatively when it is actually *changed*.

Triggering the code when the variable is assigned a value is the default behavior for MeldC. This means if we declared the active value

```
(profs_salary) --> { admins_salary += 1000; }
```

we would clearly have a world where the base pay for administrators always rises, regardless of how much faculty are paid. But this wouldn't be the end of the injustice. If `profs_salary` were 30000 and MELDC executed the statement

```
profs_salary = 30000;
```

`admins_salary` would still go up by $1,000. Administration would get a pay raise for just reminding the faculty of its current salary.

If the active value had been

```
(profs_salary@) --> { admins_salary += 1000; }
```

(with an at-sign following `profs_salary`) the code would only have been triggered when `profs_salary` actually *changed*, not when it was simply assigned a value. Following any active value with an at-sign forces it to behave this way.

9.3.3 Active Values and Complex Data-Types

When dealing with variables of simple types the meaning of a variable being active is fairly clear. When dealing with more complex user-defined types like arrays and structures, however, questions arise about what part of a compound variable can be made active. MELDC offers as much flexibility as possible with active values and complex data.

- *Arrays as a whole can be made active.* For example, if we had an instance variable `char a[25][3]` and declared the active value

  ```
  (a) --> { printf("Lox\n"); }
  ```

  we would get the word Lox printed every time there was an assignment to *any* element of the array `a`.

- *Elements of one-dimentional arrays can be made active.* If we declare `int b[13]` we can declare the active values `(b@)`, `(b[10])`, `(b[0]@,b[1])`, et cetera. The active value `(b@)` will fire whenever any element of the array is changed, and the active value `(b[0]@,b[1])` will fire whenever the zeroth element of `b` is changed or the first element is assigned to. Remember that this individual addressing of active values works only for *single*-dimensional arrays—if we use arrays of two or more dimensions we can have *only* the whole array active.

- *Structures and unions can be made active.* In general, any portion of a `struct` or `union` may be made active, from the entire conglomerate to the smallest sub-parts of it. Thus, if we had a variable `b.a.a`, we could make `b, b.a`, or `b.a.a` active, with well-defined behavior for each one. If any element of the structure `b` is assigned to, the active value `(b)` will fire. If `b.a` were assigned a value both `(b)` and `(b.a)` would fire, and if `b.a.a` were assigned to all three active values would fire.
- *No kind of pointer can be made active.* This includes variables containing '`-->`' as well as '`*`', so neither `*a` nor `a->b` can be active values.

9.3.4 Active Values and Inheritance

Active values and inheritance do not mix well in the current implementation of MeldC. Thus, inheriting classes which employ active variables is not such a good idea, and will result in unpredictable and undefined behaviour.

9.4 The MetaClass

What does

```
seeded_bagel mybagel = seeded_bagel.Create();
```

have to do with

```
int i = mybagel.get_seeds();
```

where `seeded_bagel` is a class and `mybagel` is an object of class `seeded_bagel`?

There certainly is an astounding similarity in syntax here. And we know exactly what syntax the second statement means: we are assigning to an integer the result of the method `get_seeds()` of object `mybagel` which was defined in class `seeded_bagel`. The first statement, however, we have (until now) simply accepted as syntactic convention: we tack "`.Create()`" to the end of a class-name and we get something that returns an object.

Until now, we have been extremely silent about an important aspect of MeldC. Well, the secret is out, if the syntax didn't already give it away. `Create()` is actually a **method**, just as `get_seeds()` is. And, just as `get_seeds()` is defined in `mybagel`'s class, `Create()` is defined in `seeded_bagel`'s class.

`seeded_bagel`'s class? But `seeded_bagel` *is* a class. True, and the class of all classes, and thus the class of `seeded_bagel`, is what MeldC calls the **MetaClass**. Just as all `seeded_bagel` objects are **instances** of the class of seeded bagels, all class objects (what we have called classes) are instances of the MetaClass.

What does this mean? Well, for one thing, it explains the strange "syntactic" methods we have seen, `Create()`, `Destroy()`, `GetObj()` and `PutObj()`, among others. They are all MetaClass methods—methods that are used by classes and are defined in the MetaClass, much like `get_seeds()` is used by `mybagel` and defined in the class `seeded_bagel`. The MetaClass shows that MeldC is a thoroughly object-oriented language, extending to almost all facets of its use. There are very few exceptions to the object-oriented paradigm, which makes the language consistent and easy to use.

It also explains one more syntactic element about the language that may have been noticable. `Create()`, `Destroy()`, `GetObj()` and `PutObj()`—all MetaClass methods—all start with a capital letter. This is a stylistic convention for MeldC and isn't a necessary part of the language. But it does help keep MetaClass methods and regular class methods from being confused, and is a helpful hint to keep in mind when writing and debugging MeldC programs.

One final mind-bending note on the MetaClass: If the MetaClass is the class of classes, the MetaClass is itself a class. Which means that the MetaClass is the class of the MetaClass—the MetaClass is its own class. Ignoring the looming infinite regress this kind of construct creates, this makes programming MeldC much easier than it could have been: there are only three levels of the class hierarchy to keep track of.

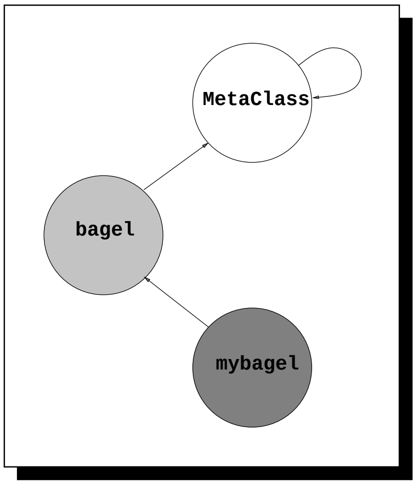

Figure 9.2: The MetaClass Hierarchy

9.4.1 Destroy()

The Destroy feature is used for the sole purpose of removing either an object or a class. This would want to be done to clean up an object. This can be done as follows:

metaClass.Destroy(*object-name*);

This will destroy an object if *object-name* refers to an object. If the *object-name* refers to a class instead, then the entire class will be destroyed. It should be pointed out that the destroy function will work recursively. That is, say we have the hierarchy of Figure 9.3. In this case, if we make the Destroy call from A, we must then also destroy B and C due to the fact that B and C are inherited from A.

Notice what this implies. When we make the call **metaClass.Destroy();**, we are saying that we want to destroy the MetaClass. This will cause a series of steps that will destroy all classes that exist, including the MetaClass. Notice that this call can only be done from the MetaClass.

We must also point at the case of when a Destroy is called on a object

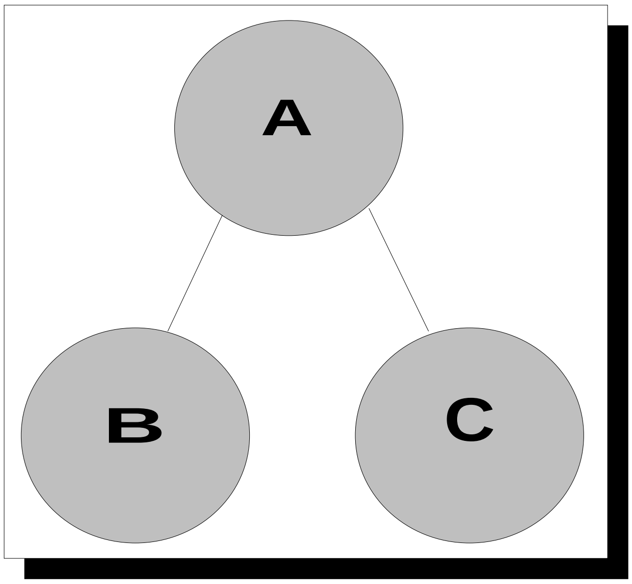

Figure 9.3: Destroy call in a hierarchy.

that has a shadow object attahced to it. When this happens, the Destroy routine will issue a Detach call to remove the shadow object from the object to be destroyed.

9.5 String Selectors

When a MELDC object receives a message, its default behavior is to match the message with one of its selectors and then to fire off the method corresponding to the matched selector. With this in mind, a simple improvement might be for the object to match the incoming message with some other predicate other than that of equality; that is, for the object to fire a method if a message matches the method's selector in some programmer-defined way *other* than being exactly alike.

For reasons of efficiency, most MELDC messages only fire off a method if the message-name exactly matches one of the object's selectors. For example, the message `square(5)` sent to the object `arithmetic` will only work properly if there is some method in `arithmetic` with selector `square`;

if there is some method defined by, say:

```
implementation
    ...
  int square(x) { return x*x; }
```

MELDC can be more flexible in its message-triggering using a special kind of selector: a **string selector**. Instead of defining a method-name with an identifier, the programmer can use a string-constant, which may be a regular expression using metacharacters. When a string message is sent to the object, the object will trigger the first method whose string selector matches the incoming string message-name (it will fire the *first*, since the possibility exists that more than one selector could match the message). Thus, MELDC objects can be simple pattern-matching entities.

There is a price to pay for this flexibility, however. Methods which have string selectors cannot have any arguments passed to them.

9.5.1 Syntax for Selectors and Messages

String selectors are defined much like regular selectors are, but a string constant is used rather than an identifier. For example, if we had a method `foo` defined by:

```
implementation
    ...
  int foo()  ...
```

and we instead wanted to define a method which would trigger not when the object receives the message `foo`, but when it receives the string messages `"foo", "fo", "fooo", "foooo"`, and the like. We would then define the new method with:

```
int "foo*"() { ...  }
```

as `"foo*"` is the regular expression for the set of strings of the form "fo" followed by zero or more "o" 's. It is perfectly allowable to use a string selector without metacharacters (i.e., `"foo"`); though the only string message matching it will be its exact copy.

The difference is important, though, between `foo`, the selector and identifier, and `"foo"`, the string selector and string constant. The former can only be triggered by the message `foo`, while the latter only by the *string* message `"foo"`. There is a specific syntax for sending a *string* message to an object, though not a complicated one: simply replace the method-name in the call with a string.

If we were to send the normal message `foo` to the object `spam`, for example, we would use the statements `spam.foo()` or `send foo() to spam`, depending on whether we were sending synchronously or asychronously. If `spam` had a string selector `"foo"` or `"foo*"`, however, we would use `spam."foo"` or `send "foo" to spam`.

In fact, we aren't limited to string *constants* in sending string messages. If `bar` were a variable of type `char *`, the calls `spam.bar` and `send bar to spam` are also legal. The actual string message sent would be whatever string is referenced by the character pointer `bar`. Thus, if `bar` pointed to the string `"fooo"`, the call `spam.bar` would trigger the method with string selector `"foo*"`.

9.5.2 The $selector Keyword

It is often useful to have some record of the actual string message sent to a string selector. For example, if our `"foo*"` selector is triggered, it may be necessary to know whether it was triggered by `"foo"` or by `"foo1"`. The $selector keyword may be used inside a string-selected method to find this information. $selector will return a character pointer representing the string which triggered the method. Remember, though, that the $selector keyword only has meaning inside a string-selected method.

9.5.3 Limitations

The price of the ability to use regular expressions for selectors is the argument list: Methods with string selectors can never have arguments, and must always be declared with an empty argument list. This is not to say that *objects* with string-selected methods cannot have arguments for *any* of its methods. Objects may freely mix methods with string selectors and those with regular selectors. The order of their listing isn't important, apart from the fact that a string message will only trigger the *first* method it matches. Methods with differing types of selectors are treated completely separately. Thus, it is important to note that string messages will not match regular selectors, or vice versa. The regular message `foo` can never cause a method selected by `"foo*"` to fire.

9.6 The MELDC Scheduler

As hinted in Chapter 4, normal MELDC concurrency can be described as *pseudo*-parallelism, not true parallelism: although there may be more than one MELDC thread running "concurrently," all computation is actually executed on a single processor[1], with each MELDC thread given a certain amount of processor time, or a certain length "time slice," before the processor moves onto another MELDC thread.

There are a number of possible policies which might govern the lengths of the time slices for each thread — policies to govern when one thread is put on hold to devote processor time to another. Though MELDC supports a number of these, one common aspect is that all threads are kept in a queue. When the MELDC process moves from one thread to another, its current one is put onto the end of the queue and must wait until all other threads are attended to before receiving any more processor time.

9.7 Dynamic Linking of Classes

In its support for dynamic programming, MELDC provides facilities for externally constructed entities, both classes and objects, to be incorporated on-the-fly into a running MELDC program. These facilities can be cleanly divided into the dynamic linking of classes and the loading and saving of persistent objects. Persistent objects are not supported in the current version of MELDC, but the dynamic linking of classes is well supported — though only under the Sun 4 architecture.

It is sometimes the case that not all class definitions will be available when a MELDC program is executed. At others, it is simply not desirable to reserve space in a MELDC executable for a host of class definitions, many of which may never be used. In such cases it is often wise to only include a class definition into the program if and when it is actually needed — to

[1] Note that this scenario describes the basic concurrency constructs built into the MELDC kernel. MELDC support for remote objects allows true parallelism

dynamically link that class definition into the already-running program. Using such dynamically-linked classes imposes some overhead, and care must be used in creating objects with them, but they can be extremely useful.

9.7.1 Syntax

Dynamic linking is a fairly straightforward process, but it involves something we haven't seen before: A MetaClass method sent to the MetaClass itself, as well as an "object" of class MetaClass.

First of all, in order to make any use at all of the dynamic linking abilities of MeldC, the protocol object `persistent_class_protocol` must be imported from feature `PersistentClass`:

```
interface:
  PersistentClass[persistent_class_protocol]
```

There must also be some existing variable which will be the placeholder for the dynamically linked class: the new "name" for the class. This variable will have the type `MetaClass`, for just as object variables have classes as types, so do class variables have `MetaClass` as a type. For example, if we were going to dynamically link some class and use the variable `foo` to hold the class definition, we must have the declaration:

```
MetaClass foo;
```

somewhere where the variable `foo` is visible for the next step:

For the class to actually be dynamically linked into the MeldC program, four pieces of information are necessary: The class name, the name of the protocol object, `persistent_class_protocol`, the name of the feature which contains the class, and a search path where the already compiled feature can be found. All this information is used for a call to the MetaClass method `GetObj`, already seen in Chapter 4:

```
class-variable = (MetaClass)
              MetaClass.GetObj(class-name,
                               persistent_class_protocol,
```

```
                                    feature-name,
                                    search-path);
```

So if we wanted our `foo` class variable to contain the class definition for class `rectangle` of feature `geometry` which was compiled in the default directory, we would use:

```
foo = (MetaClass)
      MetaClass.GetObj("rectangle",
                       persistent_class_protocol,
                       "geometry",
                       (char *)NULL);
```

9.7.2 Using the Dynamically Linked Class

The most common use for a newly linked class is to create an object. This is also relatively straightforward, but there are a few catches. Firstly, the object variable for the newly created object must have been declared to be of type (or class) `object`. That is, if we want to set the variable `bar` to be a newly created object of class `rectangle`, just after linking that class into the program as `foo`, we would declare `bar`:

```
object bar;
```

And to actually create the object, we would use:

```
bar = (object) foo.Create();
```

The presence of the (object) typecast is mandatory: One price of dynamic power is a lack of typechecking, so all return values of these dynamic entities must be cast.

9.7.3 Preemptive vs. Non-Preemptive Scheduling

For some architectures (Sun 4), the MeldC distribution includes a choice of schedulers, a preemptive and a non-preemptive one. The choice of schedulers is not under programmer control; the local maintainers of MeldC make the decision as to which scheduler to offer when MeldC is compiled on site. Though it is usually unwise to write programs which differ in behavior depending on the scheduler, information about how and why threads are scheduled can be useful in very advanced applications.

The non-preemptive scheduler is the default for MeldCUnder this scheduling policy, a MeldC thread runs until a MeldC function call is made, either synchronously or asynchronously. Whenever a statement such as *object.method* or `send` *method* `to` *object* is reached, the current thread is placed on the end of the queue and all other threads are given processor time before the MeldC function call is executed. This is an efficient policy, as it removes much of the task-switching overhead of a more active policy, but sometimes it may not be a good enough simulation of parallelism. For example, a thread with a "tight loop" of C statements without any MeldC function calls might starve all other threads of processor time, for the thread would receive processor time until the (perhaps quite long) loop was finished. The preemptive scheduler may be a closer approximation to true parallelism: With it, threads can be switched in the middle of a method execution; in the middle of a list of C statements. C *functions*, however, will not be preempted, nor will C code inside the MeldC kernel.

9.8 Preventative Debugging

Even with the help of the MeldC debugger, attempting to make programs bug-free can be an extremely time-consuming business. We highly recommend that programmers working in MeldC make use of a form of "defensive programming." By this we mean that extra care must be taken to assure that code is written correctly, according to both conventional C and MeldC syntax and grammar.

The single greatest source of errors in conventional C code has to be the confusion of the assignment operator (a lone equals sign, "=") with the notation for equality (a pair of equals signs, "=="). More hair has been torn out of the heads of beginning C programmers over this simple oversight than any other pitfall of the language. Since MeldC has adopted this syntax, programmers should pay careful attention to its correct usage.

Actually, the single vs. double problem shows up elsewhere in both conventional C and MeldC. A lone ampersand (&) will signify a bit-wise AND operation to be performed on a pair of variables, while a pair of ampersands (&&) signal a *logical* AND operation performed on a pair of variables. Likewise, bit-wise and logical OR operations follow the same pattern but use a vertical line (|) or lines (||) instead.

More troublesome to beginning MeldC programmers will be certain operators or operations in MeldC that closely resemble completely different features of conventional C. Due to MeldC's adoption of C grammar as the basis for its own, these similarities will undoubtedly cause much confusion. A few of the more prominent examples are listed in Figure 5.1.

Once all syntactical errors are corrected, if the MeldC file is still producing error messages, it's important to check for programming errors. Here are a few ways to avoid some of the more mystical errors that can occur within MeldC:

1. *Confirm that* `delayuntil/respond` *pairs match not only what they're passing but also their departure and arrival points.* Theoretically, an object could wait forever if its `respond` was misdirected or didn't match the `delayuntil` message which was waiting for it. Imagine an object `Krazy_Kat` that declares `delayuntil ''Iloveyou'' from Ignatz`, but `Ignatz` never actually sends a `respond ''Iloveyou'' to Krazy_Kat`. Or else `responds` with `brick`. In either case, `Krazy_Kat` would wait in vain for a proper response.

Conventional C		MeldC	
Explanation	**Example**	**Example**	**Explanation**
Member "b" of struct pointed to by "a"	a->b	a-->b()	Active value
Assignment of 42 to member "bar" of struct "foo"	foo.bar = 42	foo.bar(42)	Synchronous send of message "bar" with value 42 to object "foo"
Function named "create"	create()	Create()	Global object declaration at start time
Type integer	int	init()	Entry point for most threads

Figure 9.4: Some common sources of confusion.

2. *Avoid cyclical delayuntil/respond groupings.* This is the opposite of number 1, for in this case the delayuntil/respond pairs are doing their jobs *too* well by causing a deadlock in the system. If A waits for a response from B that will only come if A responds to B, then the objects will never get any work done.

3. *Avoid cyclical merges.* We might call this "the Oedipus Syndrome", in which an object will inadvertently (through careless merges) attempt to inherit features from itself. In other words, the object tries to become its own parent. Such bootstrap techniques may work in other areas of computer science, but object-oriented programming forbids them.

4. *Methods returning an* `int` *must specify so.*

5. *Methods cannot contain empty statements.* If you have a method named, for instance, `wait_for_Godot()`, it must contain meaningful statements. The MeldC compiler does not permit an "open" curly brace and a "close" curly brace with absolutely no code between them.

6. *Carefully examine the context of a pair of colons to determine whether something is missing or something is extra.* Depending on the loca-

tion, a pair of colons may signal the need for an equals sign, or may suggest that one of the colons should be removed.

7. *Be aware of the limits of encapsulation.* Values an object wishes to access may be "hidden from view" if they are within an object of another class than that of the object seeking access.

8. *Be aware of the proper syntax for braces, brackets, and parentheses in MeldC.* Braces are used to denote the scope of C statements. Angle brackets indicate the location of atomic blocks. Parentheses enclose expressions.

9. *Make certain to include an* `init()` *function in at least one class within a given feature.* Don't be overly concerned, however, if the MeldC compiler complains about a particular class lacking an `init()` function. If class A has an `init()` function, while class B does not, the compiler will send a warning message about B. The output file is still runnable, however.

10. *Make certain to initialize an object using* `Create()`*.* If you don't `Create()` an object, MeldC's system function will generate a "core dump" when you try to call that object. This program failure will undoubtedly be a source of befuddlement when you try to debug your MeldC code and find everything else syntactically-correct.

11. *Typedefs, unions, and structs must be defined in the* `implementation` *section or in a header file to be* `#include`*d in a MeldC program.* If this is not done, the compiler will either ignore the definitions or generate a "syntax error" message.

12. *Warning about the Create call.* When we make the Create call, we will call the init function asychronousily. So, when we call Create all of the parameters will also be passed to the init function. Because of this, you should make sure that any variables you pass to Create exist until the init fucntion has finished. For this reason, you should not pass addresses of a local variables to Create.

Chapter 10

Examples

10.1 Inheritance: Clock Radios

A real-life example of the inheritance from two distinct classes can be found in the case of a common clock radio. If we implement in MeldC a merged class of `clock_radio`, the class could take elements of the `clock` class hierarchy *or* the `radio` class hierarchy, *or* both. A side effect of this merger of elements from two different classes can be seen in the case of an incoming message seeking the appropriate interface to enter an object. If, for some reason, a particular item in the `clock_radio` class does not have the correct interface to respond to a message, then the message is sent up the line to a superclass which does have a receptive selector (either something in the `clock` lineage or in the `radio` lineage.

This sample program details the code necessary to create objects of the class `clock_radio`. Important aspects of multiple inheritance, as well as object creation and testing, are explained.

```
feature inheritance               /* the mandatory feature name */

interface:                        /* a one-feature program, so  */
                                  /* we have an empty interface */
implementation:
  Clock_Radio a_clock_radio =     /* create a Clock_Radio       */
    Clock_Radio.Create();         /* object globally            */
  test test_obj = test.Create();
```

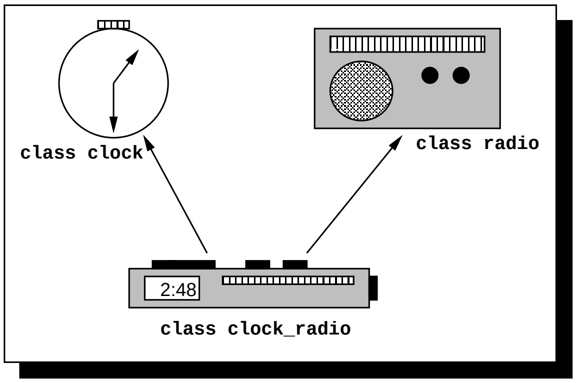

Figure 10.1: The Clock Radio Inheritance Tree

```
class test ::=
methods:
  void init() {                   /* all testing done inside    */
    int i, time;                  /* the init() special method  */

    printf("\n\tWake me at eight\n};
    a_clock_radio.radio_on();
    a_clock_radio.set_station(89.9); /* set to WKCR, of course   */
    a_clock_radio.set_alarm(8);
    a_clock_radio.reset_time();

    for (i = 0; i< 12; i++) {
      a_clock_radio.tick();
      if (a_clock_radio.get_time() ==
          a_clock_radio.get_alarm_time()) {
        a_clock_radio.sound_alarm(); /*sound_alarm is a method   */
      }                           /* both superclasses          */
    }

    printf("\n\tDON'T wake me at eight\n");
    a_clock_radio.alarm_off();
    a_clock_radio.reset_time();
    for (i = 0; i< 12; i++) {
      a_clock_radio.tick();
      if (a_clock_radio.get_time() ==
          a_clock_radio.get_alarm_time()) {
        a_clock_radio.sound_alarm();
      }
    }

    printf("\n\t7 o'clock.  NO RADIO.\n");
    a_clock_radio.radio_off();
    a_clock_radio.set_alarm(7);
    a_clock_radio.reset_time();
```

```
    for (i = 0; i< 12; i++) {
      a_clock_radio.tick();
      if (a_clock_radio.get_time() ==
          a_clock_radio.get_alarm_time()) {
        a_clock_radio.sound_alarm();
      }
    }
  }
end class test

class Clock ::=                /* The first of our superclasses */
  int time;
  int alarm_time;
  int alarm_on;
methods:
  void init() {
    time = 1;
    alarm_on = 0;
  }
  void set_alarm(int set_time) {
    alarm_on = 1;
    alarm_time = set_time;
  }
  void alarm_off() {
    alarm_on = 0;
    printf("Alarm turned off\n");
  }
  void reset_time() {
    time = 0;
  }
  void tick() {                        /* each tick is one hour */
    time += 1;
    printf("time is %d o'clock\n",
           time);
  }
  void sound_alarm() {
  {
    if (alarm_on)
      printf("Alarm: %d o'clock WAKE UP!!! \n",
     time);
  }
  int get_time() {
    return time;
  }
  int get_alarm_time() {
    return alarm_time;
  }
end class Clock

class Radio ::=                /* the second of our superclasses */
  float station;
  int on;
  int vol;
methods:
  void init() {
    on = 0;
```

```
    station = 92.7; /* Now let's tune in K-Rock */
    vol = 4;
  }
  void radio_on() {
    on = 1;
  }
  void radio_off() {
    on = 0;
  }
  void set_station(float new) {
    station = new;
  }
  void sound_alarm() {
  {
    if (on)
      printf("Alarm: station %f.\n",
             station);
  }
  void alarm_off() {
    on = 0;
  }
end class Radio

class Clock_Radio merges Clock, Radio  ::=
methods:                            /* looks boring by itself...  */
  void init() {                     /* all its variables and      */
  {                                 /* methods are in its parents */
    printf("\n\tA new clock radio\n");
  }
end class Clock_Radio

end feature inheritance
```

10.2 Concurrency: The Producer-Consumer Problem

Our second large example program discusses MeldC's treatment of concurrency issues is based on a familiar programming problem from the study of operating systems. Imagine a pair of programs, where one *produces* data of some sort that is *consumed* by its partner. The consumer has a buffer of a fixed size into which the producer can send the data and from which the consumer can read ("consume"). The problem is: how do we prevent the producer from writing data into a full buffer, and how do we prevent the consumer from attempting to read from an empty buffer?

The key is to synchronize the two actions in such a way that the producer will only write into the buffer when there's a space waiting for a deposit, and the consumer will only attempt to read from the buffer when there's something in the buffer. In MeldC, as can be seen from the example below, this synchronization is accomplished through several of the constructs we've discussed in Chapter 3: `delayuntil` and `respond`; atomic blocks, and message passing.

```
extern int printf();
extern int random();

feature Producer_and_Consumer

interface:

implementation:

  ProducerConsumer PCobj = ProducerConsumer.Create();

/*
 * Producer-Consumer program using synchronous delayuntil
 * 1-to-1 synchronization between message-sending and delayuntil
 */


class ProducerConsumer ::=
  int buffer[10];
  int total_space = 10;
  int P = 0;
  int C = 0;

methods:

void init()                 /* create 10 "buffers", really respond */
    {                       /* statements ready for delayuntils    */
    $self.spacing(total_space);
    send world() to $self;
    }

void spacing(int x)         /* the actual "buffer" creator         */
    {
    printf("space(%d)\n", x);
    if(x > 0)
        {
        respond "SpaceAvailable";
        send spacing(x-1) to $self;
        }
    }


int decide()                /* a glorified coin-flipper            */
    {
    if( (long) random() >= (long) (1024*1024*1024) )
        return(1);
    else
        return(0);
    }


int world()
    {
```

```
    int i;
    int c = 0;
    int p = 0;

    printf("Start ..............\n");

    for(i = 0; i < 40; i++)        /* run the simulation for 40 */
        {                          /* productions/consumptions  */
        if( $self.decide() )
            {
            send consumer(c++) to $self;
            }
        else
            {
            send producer(p++) to $self;
            }
        }
    }

int consumer(int i)
    {
    printf("Consumer %d consumes %d\n", i,$self.Consume());
    }

int producer(int i)
    {
    printf("Producer %d produces %d\n", i, $self.Produce(i));
    }


int Produce(int x)
    {
    delayuntil "SpaceAvailable";/* wait until there is no space   */
    <                           /* and begin the critical section */

    buffer[P] = x;
    printf("...Producer(%d) at buffer[%d] = %d\n",x, P, buffer[P]);
    P = (P + 1) % total_space;
    respond "EntryAvailable";
    return(x);

    >

  }

int Consume()
    {
    delayuntil "EntryAvailable"; /*wait until nothing to consume  */
    <                            /*and enter the critical section */
    int i;

    i = C;
    C = (C + 1) % total_space;
    printf(".....Consume() at buffer[%d] = %d \n", i, buffer[i]);
    respond "SpaceAvailable";
    return(buffer[i]);
```

```
    >
    }

end class ProducerConsumer

end feature Producer_and_Consumer
```

10.3 Shadow Objects: Another Producer-Consumer

This is simply our old producer-consumer program with a number of shadow objects attached which display messages as they are entered and exited. This makes clear the behavioral differences between "linear" and "circular" schemes of attachment with the use of an #ifdef preprocessor directive.

```
#include "meldc_user.h"

#define Linear

feature ShadowObjectTest

interface:
    imports Unix[sys_obj]
    imports MemoryAllocation[Memory]

implementation:
  testing test_obj = testing.Create();

class ShadowObject1 merges MemoryAllocation[Memory] ::=

methods :


struct _frame * shadow_object_entry_point(struct _frame *fp)
    {
    /*
     * reconstruct the frame and return the frame.
     */

    printf("enter Shadow object entry point 1 \n");

    return(fp);

    }

void shadow_object_exit_point(object obj, struct _threadid t)
    {
    /*
     * reconstruct the frame and return the frame.
```

```
     */

    printf("enter Shadow object exit point 1 %s\n",
    ((struct _object *)obj)->$.name);

    return;

    }


end class  ShadowObject1

class ShadowObject2 merges MemoryAllocation[Memory] ::=


methods :

void shadow_object_entry_point(struct _frame *fp)
    {
    /*
     * reconstruct the frame and return the frame.
     */

    printf("enter Shadow object entry point 2 \n");

    return(fp);

    }

void shadow_object_exit_point(object obj, struct _threadid t)
    {
    /*
     * reconstruct the frame and return the frame.
     */

    printf("enter Shadow object exit point 2 %s\n",
   ((struct _object *)obj)->$.name);

    return;

    }

end class  ShadowObject2

class testing ::=
ProducerConsumer pcobj;
ShadowObject1 shadow_obj1;
ShadowObject2 shadow_obj2;

methods:

void init()
    {
    shadow_obj1 = ShadowObject1.Create();
    shadow_obj2 = ShadowObject2.Create();
```

```
    pcobj = ProducerConsumer.Create();

#ifdef Linear

    /*
     * Real_object <- shadow_obj1 <- shadow_obj2
     *
     * When a message is sent to the base object, the
     * the message is first sent to shadow object 2.
     *
     * If any shadow object should fail, the subsequent shadow
     * object will not be executed.
     * and the base object will not be executed.
     */

    ProducerConsumer.AttachObject((object)pcobj,shadow_obj1,
  "shadow_object_entry_point",
  "shadow_object_exit_point",
  "",
  "");

    ShadowObject1.AttachObject(shadow_obj1,shadow_obj2,
       "shadow_object_entry_point",
       "shadow_object_exit_point",
       "",
       "");
#endif

#ifdef CIRCULAR
    /*
     *  Real_object  <- shadow_obj1
     *                              <- shadow_obj2
     *                              <- shadow_obj2
     *                              <- shadow_obj3
     *                              <- shadow_obj4
     *                              <- shadow_obj5
     *
     * When you send a message to the base object, the
     * the message will be sent to the shadow object 5 first.
     *
     * Should any shadow object fail, the subsequent shadow object
     * will be executed but the real object will not be.
     */

    ProducerConsumer.AttachObject((object)pcobj,shadow_obj1,
  "shadow_object_entry_point",
  "shadow_object_exit_point",
  "",
  "");
    ProducerConsumer.AttachObject((object)pcobj,shadow_obj2,
  "shadow_object_entry_point",
  "shadow_object_exit_point",
  "",
  "");
#endif
```

```
    }
end class testing

/*
 * Producer-Consumer program using synchronous delayuntil
 * 1-to-1 synchronization between message-sending and delayuntil
 */
class ProducerConsumer ::=
  int buffer[10];
  int total_space = 10;
  int P = 0;
  int C = 0;

methods:

void init()
    {
    /*
     * create 10 buffers.
     */
    $self.spacing(total_space);
    send world() to $self;
    }

/*
 * Creating the buffer spaces.
 */

void spacing(int x)
    {
    /*
     * create n spaces
     */
    printf("space(%d)\n", x);
    if(x > 0)
        {
        respond "SpaceAvailable";
        send spacing(x-1) to $self;
        }
    }


int decide()
    {
    if( (long) random() >= (long) (1024*1024*1024) )
       return(1);
    else
       return(0);
    }


int world()
    {
    int i;
    int c = 0;
    int p = 0;
```

```
    printf("Start ..............\n");

    for(i = 0; i < 40; i++)
        {
        if( $self.decide() )
            {
            send consumer(c++) to $self;
            }
        else
            {
            send producer(p++) to $self;
            }
        }
    }

int consumer(int i)
    {
    printf("Consumer %d consumes %d\n", i,$self.Consume());
    }

int producer(int i)
    {
    printf("Producer %d produces %d\n", i, $self.Produce(i));
    }


int Produce(int x)
    {
    /*
     * Wait if there is no space.
     */
    delayuntil "SpaceAvailable";

    /*
     * Critical Section.
     */
    <

    buffer[P] = x;
    printf(".....Producer(%d) at buffer[%d] = %d\n",x, P, buffer[P]);
    P = (P + 1) % total_space;
    respond "EntryAvailable";
    return(x);

    >

  }

int Consume()
    {
    /*
     * Wait if there is nothing to consume.
     */
    delayuntil "EntryAvailable";
```

```
    /*
     * Critical Section.
     */

    <
    int i;

    i = C;
    C = (C + 1) % total_space;
    printf(".....Consume() at buffer[%d] = %d \n", i, buffer[i]);
    respond "SpaceAvailable";
    return(buffer[i]);

    >
    }

end class ProducerConsumer


end feature ShadowObjectTest
```

10.4 Dynamic Linking: Yet Another Producer-Consumer

This example program simply loads the class `ProducerConsumer` from the feature `Producer_and_Consumer` and creates an object of that class. Note, however, that if it is run from the directory where the program in Section 10.2 was compiled, this program will execute the exact same code.

```
feature Conference

interface:
  imports PersistentClass[persistent_class_protocol]

implementation:
  LocalUser local_object = LocalUser.Create();

class LocalUser ::=
methods:

void init()
    {
    MetaClass pc_class;
    object pc_object;

    /*
     * 1st class name.
     * 2nd the the protocol object.
```

```
     * 3rd feature name
     * 4th Search path for the object code.
     *     (i.e -E../../examples -E../../../examples)   Exactly the
     *     same function as the compiler -E switch.
     */

    pc_class = (MetaClass)
MetaClass.GetObj("ProducerConsumer",persistent_class_protocol,
"Producer_and_Consumer",(char *)NULL);

    pc_object = (object)pc_class.Create();
    }

end class LocalUser

end feature Conference
```

Appendix A

The MeldC Program File

A.1 Feature Name

Each MeldC program file contains exactly one feature, which must have a name. The name of the feature and the name of the file where it is stored are completely independent: they can be the same or different. The name of the feature is declared at the beginning of the program file by

`feature` *feature-name*

where *feature-name* is any valid identifier.

A matching

`end feature` *feature-name*

line marks the end of the feature, and must always be placed at the end of the program file. Since there can be no confusion about which feature is being ended, however, the *feature-name* label in this statement is optional. All `#include` statements must appear at the top of the file, before the feature is declared.

A MeldC feature is divided into two parts, the **interface** and the **implementation**.

```
feature feature-name
interface:
  exports class, object . . .
  imports feature[class], feature[object] . . .

implementation:
   global-variable-declarations
   class-definitions

end feature feature-name
```

.

Figure A.1: The MeldC Feature

A.2 Interface

The **interface** section of a MeldC feature declares how it interacts with other features. The section begins with the keyword `interface` (optionally followed by a colon for clarity). The `interface` label is necessary to begin the section even though it may be empty if the feature neither imports nor exports anything.

A feature can **export** its classes and global objects for other features to use, and **import** classes and global objects which other features have exported. Consequently, each line of this section consists of either of the keywords `exports` or `imports`, followed by a list of things to export or import, respectively. All import lines must occur before all export lines.

A.2.1 Exports

The syntax for exporting objects and classes is relatively simple. Each export line consists of the keyword `exports` followed by a list of things to export. If, for example, we wanted to export the classes `shape` and `animal` and the objects `circle1` and `account39`, we would use the statement:

```
exports shape, animal, circle1, account39
```

There are three things to note about this line:

- The order of the objects and classes listed is unimportant. Here we listed exported classes before exported objects, but only for clarity.

- All four exports here are on one line, but we could have used two, three or four `export` statements to accomplish the same thing.
- There is no semi-colon ending the statement.

A.2.2 Imports

To import something we must provide both the name of the object or class we wish to import as well as the MELDC feature from which to import, but other than that the syntax is the same as exporting. Each item on an import line is of the form *feature*[*thing*], so if the class `shape` and the global object `circle1` were exported in feature `geometry`, `animal` was exported in feature `zoo`, and `account39` was exported in `Lincoln_Savings`, we might use the following statements:

```
import geometry[shape], geometry[circle1]
import zoo[animal], Lincoln_Savings[account39]
```

Alternately, we can just import a whole feature, getting everything exported from that feature in one fell swoop. If `Lincoln_Savings` exported all of its accounts, we could import them with the statement

```
import Lincoln_Savings
```

A.3 Implementation

The bulk of the MELDC program is usually in this second section of the feature. It begins with the keyword `implementation` (with an optional colon for clarity).

A.3.1 Global Variable Declarations

Variables that are global to the feature are declared (and possibly initialized) with C syntax at the beginning of the implementation section before any classes are defined:

type-cast variable-name `[ =` *assignment* `] ;`

Each declaration has

```
class class-name  merges list-of-superclasses
  instance-variable-declarations
 methods:
  active-value-definitions
  method-definitions
 end class  class-name
```

Figure A.2: The MELDC Class

- A type-cast, declaring the variable to be of a certain type. The type can be one of conventional C's primitive types, one user-defined through `typedef`, or a class name (which would be the type of an object of that class),
- Any valid identifier as the variable's name, and optionally
- An initial assignment to the variable.

If we wanted to create the object `account28` of the class `bank_account` when the MELDC program starts to execute, for example, we would use:

```
bank_account account28 = bank_account.Create();
```

Which not only creates the variable `account28` of type `bank_account`, but also assigns to it the result of `bank_account.Create()`. `Create()` is actually a MetaClass method, and as such must be capitalized (see Section 9.4).

A.3.2 Class Definitions

The rest of the implementation section is devoted to one or more class definitions. Each class begins with the line

`class` *class-name* `::=`

where *class-name* is any valid identifier. If we want to have the class inherit from one or more other classes, we define it with:

`class` *class-name* `merges` *list-of-superclasses* `::=`

where *list-of-superclasses* is a list of class names (or *feature-name[class-name]*'s) separated by whitespace or a feature-name with square brackets. For example, if we were to define the class `bagel` as a subclass of `torus` and `breakfast_food`, the first line of the class definition would be:

```
class bagel merges breakfast_food, torus ::=
```

The order of the list is important and defines some of the behavior of the inherited class (see Chapter 2).

The end of each class must be explicitly defined by

`end class` *class-name*

but, like *feature-name*, *class-name* is an optional addition for clarity.

Each class definition consists of declarations of **instance variables**, the keyword `methods` optionally followed by a colon, and definitions of the class' **active values** and **methods**. Class definitions may not be nested: Classes can only be defined in the `implementation` section of the MeldC feature, nowhere else.

Instance Variables

We declare variables in this section as we did for global variables. The difference is the scope of the variables created. Where global variables are visible to any object in the feature, each object's instance variables are only visible to that object. If instance variables are declared `static` they are **class variables** and can be accessed by any object of the class where the variable is defined.

Active Values

The basic syntax to declare an active value is:

(*list-of-active-values*) `-->` { *code* }

Where *list-of-active-values* is a comma-delineated list of instance variables and *code* is a series of MeldC statements. If an active value is declared, whenever one of the instance variables in *list-of-active-values* is assigned to or changed, the *code* statements are executed.

Active values are constructs with complex behavior and syntax, and are described in Section 9.3. There are a few simple things to remember about them, however:

- Declaring a global variable to be active is not allowed, nor can active variables be local to a method; active values can only be instance variables.
- If an active value is followed by an at-sign ("@") the code is triggered only when the value is actually changed. Without the at-sign, the code is triggered whenever an assignment is made to the active value.
- The arrow above ("`-->`") is formed by two hyphens and a greater-than symbol. Using one hyphen will unnecessarily confuse the compiler.
- There is no semi-colon after an active value definition.

Methods

The syntax of a method definition is the same as a corresponding C function definition (although the body of the method can use MELDC constructs as well as conventional C constructs):

```
type-cast method-name(argument-list) {
  method-body
}
```

Where *method-body* is a sequence of C and MELDC statements. Unlike conventional C, MELDC local variables may be declared anywhere in a method, not just at the beginning. They must be declared before they are used, however.

Note that the MELDC is not as forgiving as the C compiler, and will *not* assume all un-typed methods are of type `int`. Any method definition without a type-cast is an error. If we want a method to be of type `int` we must declare it so.

Appendix B

Backus-Naur Form

B.1 A Short Introduction to BNF Concepts

We've spent much of this manual talking about hierarchies of information. Mostly, these hierarchies have been classes within MELDC. Now it's time to consider the *language* of MELDC as a hierarchy in its own right.

Computer scientists often describe new computer languages in terms of that language's BNF, or Backus-Naur Form. In a BNF, all possible *legal* combinations of language primitives (keywords) are described in a concise set of choices.

To describe a variety of terms based on combinations of simple primitives, we must be able to define alternate syntaxes and multiple instances of a particular keyword or term. In a BNF, these needs are fulfilled by employing the OR-symbol | (just like the logical OR found in conventional C) to indicate alternatives, the Kleene star (an asterisk after a keyword or term) to indicate zero or more instances of that word, the subscript $_{opt}$ to indicate zero or one instance of that word, and, the plus-sign + to indicate *at least* one instance of that word.

A simple example of a BNF is one to describe colors. Here, as in the BNF for MELDC we'll see shortly, we start from the top and work our way down to the basic keywords:

```
color ::= primary + color
          |
          primary
          |
```

```
        ;

primary ::= red
            |
            blue
            |
            yellow
            ;
```

In the first line we named the item to be described ("color") and then used a "::=" to indicate that it will be assigned the attributes of what follows immediately afterward: a list of the ways to combine simpler elements to create the item. Notice that each way is separated by an or-symbol, to show that each is an alternative to consider. Within each alternative, we can include a "+" to indicate combinations. In this case, both "color" and "primary" are considered to be *nonterminals* since they can be further broken down into simpler parts: red, blue, or yellow. The latter three are called *terminals*, since we can make them no simpler.

By this BNF, blue is a color. Green is also a color, because it can be expressed as "blue + yellow." The third term of the definition (the "+" is considered the second term) is "color", which can itself be a primary color. On the other hand, "color" could be a combination of a primary color and an existing color. In this way we could describe "aqua", as a combination of blue and green. Thus, aqua can also be described as "blue + green", which is really "blue + blue + yellow". Obviously, we can join terminals and nonterminals in endless combinations, and create virtually any color from this simple definition.

Notice, also, that we can describe the color black by this BNF, too. Since black is the absence of color, we just use the null string, which is indicated by the lone semi-colon after the second or-symbol. Normally, a definition will conclude with a semi-colon, as in the case of the definition of "primary". Here, in "color", the semi-colon without anything preceding it is given as an optional attribute (note the or-symbol just before the semi-colon in the "color" definition), which is meant to show that "color" can be defined as null.

B.2 The MeldC BNF

In the BNF that follows, constructs for methods, class definitions, and other components of objects are shown, as well as extensions of the C language for special use by MeldC.

Readers who are not familiar with standard C syntax (either Kernighan and Ritchie C or ANSI C) are referred to Appendix A13 of "The C Programming Language"(Second Edition) by Kernighan and Ritchie, where a complete BNF for conventional C can be found. Since MELDC has adopted many of the elements of conventional C, only those grammatical elements that differ between the two languages are listed here. For example, the following non-terminals, although used in MELDC, are not defined here; their definitions can be found in K&R:

compound-statement
declaration-list
declarator
expression
identifier-list
init-declarator-list
pointer
statement-list
storage-class-specifier

likewise,

declaration-specifiers	is modified (see the definition below)
type-specifier	is extended (see the definition below)

The complete MELDC BNF follows:

program ::=
    `feature` *feature-name*
    `interface` *externals**
    `implementation`
        body
    `end feature` *feature-name*$_{opt}$

externals ::=
    `exports` *port-list*
  | `imports` *port-list*

port-list ::=
    port
  | *port-list* , *port*

port ::=
    class-name
  | *object-name*
  | *feature-name* `[` *class-name* `]`
  | *feature-name* `[` *object-name* `]`

body ::=
    *meldc-decl**
meldc-decl ::=
    *data-declaration-list**
  | *class-decl**

data-declaration-list ::=
    data-declaration+

data-declaration ::=
    declaration-specifiers init-declarator-list$_{opt}$ *;*

class-decl ::=
    `class` *class-name ::=*
        class-body
    `end class` *class-name*$_{opt}$
  | `class` *class-name* `merges` *super-class-list ::=*
        class-body
    `end class` *class-name*$_{opt}$

super-class-list ::=
    super-class
  | *super-class-list* `,` *super-class*

super-class ::=
    class-name
  | *featurename* `[` **class-name** `]`

class-body ::=
    inst-variables `methods` *method-part-list*

inst-variables ::=
    *data-declaration-list**

method-part-list ::=
    *method-part**
method-part ::=
    method-decl
  | *active-value-rules*

active-value-rules ::=
    (active-value-list) `-->` *compound-statement*

active-value-list ::=
    active-value
  | *active-value-list , active-value*

active-value ::=
    identifier

| *identifier* @

method-decl ::=
*type-specifier func-dcltr para-declarations merge-parameter-list** *method-body*

func-dcltr ::=
func-dcltr-1
| *pointer func-dcltr*

func-dcltr-1 ::=
func-dcltr-2
| *func-decl* ( )
| *func-decl* [ ]

func-dcltr-2 ::=
(*func-dcltr*)
| *identifier* (*func-formals-list*$_{opt}$)
| **string** ( )

func-formals-list ::=
func-formal-declaration
| *func-formals-list* , *func-formal-declaration*

func-formal-declaration ::=
type-specifier declarator
| *identifier*

para-declarations ::=
para-decl+

para-decl ::=
type-specifier declarator

merge-parameter-list ::=
merge-parameter
| *merge-parameter-list merge-parameter*

merge-parameter ::=
, *class-name* (*identifier-list*$_{opt}$)

method-body ::=
decl-stat-list

decl-stat-list ::=
decl-stat+
decl-stat ::=
decl-stat-list declaration-list
| *decl-stat-list stmt-list*

stmt-list ::=
    statement-list ;
  | *send-stmt* ;
  | *delayuntil-stmt* ;
  | *respond-stmt* ;

send-stmt ::=
    obj-expression . *message*
  | `send` *message* `to` *obj-expression*

message ::=
    identifier (*argument-expression-list*$_{opt}$)
  | **string** ( )

delayuntil-stmt ::=
    `delayuntil` *message*
  | `delayuntil` *message* `from` *obj-expression*

respond-stmt ::=
    `respond` *message*
  | `respond` *message* `to` *obj-expression*

obj-expression ::=
    expression

declaration-specifiers ::=
    storage-class-specifier declaration-specifiers$_{opt}$
  | *type-specifier declaration-specifiers*$_{opt}$

primary-expression ::=
    identifier
  | *constant*
  | *string*
  | *(expression)*
  | *$sender*
  | *$selector*
  | *$self*
  | *$thread*

type-specifier ::=
    `void`
  | `char`
  | `short`
  | `int`
  | `long`
  | `float`
  | `double`
  | `signed`
  | `unsigned`

| *enum-specifier*
| *struct-or-union-specifier*
| *typedef-name*
| **class-name**
| `object`

feature-name ::= identifier

class-name ::= identifier

object-name ::= identifier

Appendix C

MeldC **Keywords**

MeldC has fifty-two reserved keywords that cannot be used for variables or changed by the user. In addition to these keywords, identifiers created by the user may not include the dollar-sign symbol ("$"), the at-sign ("@"), or begin with two adjacent underlines ("__").

Keywords marked below with a dagger ("†") are C keywords as well as MeldC keywords.

$selector	delayuntil	if †	short †
$self	do †	implementation	signed †
$sender	double †	imports	sizeof †
$thread	else †	int †	static †
auto †	end	interface	struct †
begin	enum †	long †	switch †
break †	exports	merges	to
case †	extern †	methods	typedef †
char †	feature	object	union †
class	float †	register †	unsigned †
const †	for †	respond	void †
continue †	from	return †	volatile †
default †	goto †	send	while †

Appendix D

The UNIX System Object

Features using UNIX system calls should do so through importing the object `sys_obj` from the feature UNIX:

```
imports Unix[sys_obj]
```

The uses of the system object are described in Section 7.1.

The following calls `are` available through the UNIX system object:

accept	dup	read	sendto
access	dup2	readlink	setsockopt
bind	listen	recv	sleep
close	lseek	recvfrom	socket
connect	lstat	select	stat
exit	open	send	write

Bibliography

[1] Carlo Ghezzi and Mehdi Jazayeri. *Programming Language Concepts*. John Wiley & Sons, Inc., New York, New York, second edition, 1987.

[2] Gail E. Kaiser and David Garlan. Composing software systems from reusable building blocks. In *Twentieth Hawaii International Conference on System Sciences*, volume II, pages 536–545, January 1987.

[3] Gail E. Kaiser and David Garlan. Meld: A declarative language for writing methods. In *Sixth International Phoenix Conference on Computers and Communications*, pages 280–285, February 1987.

[4] Gail E. Kaiser and David Garlan. Melding data flow and object-oriented programming. In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 254–267, Orlando FL, October 1987. Special issue of *SIGPLAN Notices*, 22(12), December 1987.

[5] Gail E. Kaiser and David Garlan. Melding software systems from reusable building blocks. *IEEE Software*, 4(4):17–24, July 1987.

[6] Gail E. Kaiser and David Garlan. Synthesizing programming environments from reusable features. In Ted J. Biggerstaff and Alan J. Perlis, editors, *Software Reusability*, volume II, pages 35–55, Reading MA, 1989. Addison-Wesley.

[7] Gail E. Kaiser, Wenwey Hseush, Steven S. Popovich, and Shyhtsun F. Wu. Multiple concurrency control policies in an object-oriented programming system. In *2nd IEEE Symposium on Parallel and Distributed Processing*, pages 623–626, Dallas TX, December 1990.

[8] Gail E. Kaiser, Steven S. Popovich, Wenwey Hseush, and Shyhtsun Felix Wu. Melding multiple granularities of parallelism. In Stephen Cook, editor, *3rd European Conference on Object-Oriented Programming*, British Computer Society Workshop Series, pages 147–166, Nottingham, UK, July 1989. Cambridge University Press.

[9] Al Kelley and Ira Pohl. *A Book on C*. The Benjamin/Cummings Publishing Company, Inc., Redwood City, California, second edition, 1990.

[10] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, second edition, 1988.

[11] Andrew Koenig. *C Traps and Pitfalls*. Addison-Wesley Publishing Company, Reading, Massachusetts, 1989.

[12] Steven S. Popovich, Shyhtsun F. Wu, and Gail E. Kaiser. An object-based approach to implementing distributed concurrency control. In *11th International Conference on Distributed Computing Systems*, Arlington TX, May 1991. To appear.

[13] Andrew S. Tanenbaum. *Operating Systems: Design and Implementation*. Prentice-Hall, Inc., Englewood Cliffs, New Jersey, 1987.

[14] Shyhtsun F. Wu and Gail E. Kaiser. Network management with consistently managed objects. In *IEEE Global Telecommunications Conference, Communications: Connecting the Future*, volume 1, pages 304.7.1–304.7.6, San Diego CA, December 1990.