THE MELD PROGRAMMING LANGUAGE

USER MANUAL

Bill N. Schilit
Wen-Wey Hseush
Shyhtsun Felix Wu
Steven S. Popovich

Technical Report
CUCS-461-89

Columbia University
Department of Computer Science
New York, NY 10027

28 September 1989

Research in Programming Systems is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, Citicorp, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

Schilit is supported in part by the Center for Advanced Technology. Hseush, Wu, and Popovich are supported in part by the Center for Telecommunications Research.

Notes to the Reader

The Structure of this Manual

This manual is divided into two parts, a tutorial introduction to the MELD programming language, and a language reference manual. The index spans both parts. A bibliography of MELD publications and technical reports is available in Appendix II. If you are interested in the decisions leading to the current design you should refer to this literature; the intent of this manual is to provide a working description of the current language and its implementation, and not an exposition of the language issues. Chapter 9 describes the language features which are not yet implemented.

A Note on Object Oriented Programming

The primary programming paradigm used in MELD is object oriented. We assume the reader is familiar with the concept of object oriented programming, and if not, urge them to refer to the Smalltalk-80 [Goldberg 85] and C++ books [Stroustrup 86], or the articles on Flavors [Keene 85; Moon 86] which are common examples of the paradigm.

A brief review of the terminology used in OOPLS (Object Oriented Programming Languages) is in order. Object oriented programming is an approach to software development involving the use of *objects*, a notion similar to abstract data types. Both are ways of encapsulating actions, however objects usually refer to the notions of messages, class hierarchy, and inheritance as well. An object has private procedures, called *methods*, private storage, called *instance variables*, and a public interface to the procedures, sometimes called the *protocol* and in MELD this is called the *selector* and *parameters*. Objects tie together procedures and data to form a functionality that is accessible only through an interface.

When programming in an OOPL, you must create a description (called a *class*) for the object, usually by *specializing*, i.e, adding "special case" code to a more general class. For example, you would define class "Pinto" to be a specialization of "Car" with the difference that the gas tank of a "Pinto" explodes on a rear end collisions of more than 45 M.P.H. You then declare instantiations (*instances*) of the object, as you would declare variables in C. Of course, you can also create instances during the execution of a method. Methods are invoked by sending *messages* to objects instead of the normal notion of a subroutine call. Messages are employed because the binding, that is, the decision of what code gets executed, occurs at runtime for many OOPLS[1]. It is this runtime matching of message to method that gives OOPLS their power, because if a message does not *match* any method selector (also called *protocol*), then the message is forwarded to the parent class (*superclass*). In our example the message "turn right" is understood by "Pinto" even though we never wrote the "turn" method because "turn" would be defined by the superclass of "Pinto", namely "car." In this way, an object *inherits* the actions of the more general superclass. Because of *inheritance* the programmer only needs to specify how

[1] MELD determines what code to run at compile time

some desired action differs from the superclass action, thereby making software reuse a natural part of the paradigm.

A Note on Dataflow Programming

Data flow programming is a paradigm useful for achieving a high degree of parallel execution. The concept behind dataflow is simple: statements (instructions, subroutines, etc.) may be executed as soon as their inputs become available. Dataflow can take advantage of architectural parallelism since at any given time a number of statements may be available for execution. Dataflow programming, however, is somewhat different from normal programming because the approach is declarative, that is, you specify the statements to be executed, but not the order of their execution.

MELD employs the dataflow paradigm, and MELD programmers may take full advantage of dataflow programming at the statement level (or not, if they choose). MELD's dataflow does not presume a particular underlying machine architecture.

For further information on dataflow and dataflow programming languages see [Wadge 85; Broy 85; Sharp 85].

A Note on the Examples

The examples in this manual show all MELD reserved words in `CAPITALS` and all other program components in `Mixed` case. This is merely a convention, case is not significant for MELD reserved words. Case is, however, significant for variable and selector names. The output examples in this manual show user typein in *italics* and the program output in **bold**.

The example programs are tested MELD programs, available in the `/proj/meld/man/examples` directory. Those examples marked ‘‘Program Fragment’’ are not complete programs.

I Tutorial

1. A Tutorial Introduction

This chapter will give you a quick introduction to the MELD programming language. Our aim is to show enough of the basic elements of MELD to write a few small programs.

1.1. Getting Started

How hard is it to write a "Hello, World!" program in MELD? Not very. Here is an example:

```
FEATURE World

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    world : Main := Main.CREATE;

  CLASS Main ::=

    METHODS:

      printf("Hello, World!!\n");

  END CLASS Main
END FEATURE World
```

Figure 1-1: Hello, World! Program

First put this program into the file `hello.m` in your working directory or copy the file `hello.m` from the examples directory[2]. The compiler is run from a shell script, `meld`, that is stored in the `/proj/meld/bin` directory. You will either need to use the full `/proj/meld/bin/meld` name when invoking the compiler, or you may prefer to add the MELD bin directory to your shell `PATH` variable.

To compile (and then run) with input `hello.m` and executable output specified as `hello`, you would type:[3]

$ *meld -o hello hello.m*
$ *hello*

The program would then output:

Hello, World!!

[2]Examples are in `/proj/meld/man/examples`

[3]We use *italics* to denote user typein and **bold** to denote program output.

From this example you can see that a MELD program begins with a `FEATURE` name, and consists of an `INTERFACE:` and `IMPLEMENTATION:` part. A MELD `FEATURE` is a unit of reusability, similar to an Ada package, that is, a collection of private storage and class types with a well defined interface. The `INTERFACE:` portion of a MELD program is also a mechanism for reusability. It is employed when a number of files (actually `FEATURE`) are being used together, and, as you might expect, it defines the interface of a feature. See section 4.5.

The `IMPLEMENTATION:` part of a MELD program contains the storage and code declarations. You define and initialize global object variables in the `OBJECT` section, and specify the behavior for each class of objects in the `CLASS` section.

The basic syntax of MELD is illustrated in Figure 1-2 (the full grammar is presented in 8)[4]. Note that there are two types of object declarations. Declarations directly after the keyword `OBJECT` denote global variables, and declarations inside the `CLASS` part define variables local to the object, i.e., instance variables.

In this example, the global variable `Main` is initialized to a new instance of the `World` class using `CREATE`. This is a very useful sort of initialization, and most often is the one you want to use. If no initialization is done on a variable, its value will start out as the null object, **nil**, and messages sent to it will not do what you expect; they will not do anything. This common bug can be avoided by making sure that every variable is initialized in its declaration.

The executable code in a MELD program is contained in the `METHODS` part of a class definition. In this case we have a special type of method called a *constraint* that can be identified by the fact that it has no selector. Selectors are like templates for matching incoming messages.

A constraint statement has no selector because it executes when an object is created or when an object's variables change, and not directly upon receipt of a message. Constraints are further discussed later on in this chapter, and in chapter 6. It suffices to say that the constraint statement in our example is executed when the object `World` is created.

Let's continue with another program that does some input as well as output, see 1.1.

In this example we have a method with a *string selector*, `"Hello"`. The basic syntax for a method is:

selector --> *statement*

The *statement*, as you'll see later, can be a block of statements, and the *selector* can be a string or symbolic form. A string selector method has simple semantics: when a string message sent to an object matches a string selector, the statement will be executed. The string selector may contain a regular expression or scanf() like arguments, so matching and extracting arguments is very

[4]The grammar here is simplified for readability, you can actually have objects and classes appear in any order. See the full grammar for details.

```
program        ::= FEATURE identifier-list
                     INTERFACE: externals*
                     IMPLEMENTATION:
                       OBJECT
                        object-def*
                       CLASS LABEL :=
                        object-def*
                        METHODS method*
                       END CLASS identifier-list_opt
                   END FEATURE identifier-list_opt
```

Figure 1-2: Basic Syntax in MELD

```
FEATURE Animal World

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    Polly : Parrot := Parrot.CREATE;

  CLASS Parrot ::=

  METHODS:

    "Hello"--> printf("How are you?\n");

  END CLASS Parrot
END FEATURE Animal World
```

Figure 1-3: Parrot Program

flexible. For example:

```
 "Deposit %d"(cash : INTEGER) -->
printf("Deposited %d\n",cash);
```

will extract the integer 10 from the string message "Deposit 10." For further information on string selectors see section 4.4.2.

To run this program, you should type:

$ *meld -o hello-2 hello-2.m*
$ *hello-2*
Hello
How are you?

This program acts a little like a parrot. Whenever it hears "Hello," it responds with "How are you?"

The program operates as follows. The global object `Polly` is created during initialization. Since no other action is possible the runtime system waits for user input. When the user types a string followed by a return, the runtime system matches the input line against all string selectors in all objects. The runtime finds a match with a selector in object `Polly` and sends it the line of input. If other string selectors matched, then they too would get messages.

1.2. Variables and Types

You can define variables with basic types, constructor types or user defined types (that is, classes). Basic types include integer, boolean, char, string, real, and double. Types are described in Chapter 2. Here, we only give a simple example of how to define instance variables, using the basic type `INTEGER`.

```
FEATURE Money World

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    Ruckhouser : Person := Person.CREATE;

  CLASS Person ::=
    Salary : INTEGER := 40000;

  METHODS:

    "you get a raise"-->
       Salary := Salary + 4000;

    "what is your salary"-->
       printf("My current salary is %d\n",Salary);

  END CLASS Person
END FEATURE World
```

Figure 1-4: Money World Program

The "what is your salary" method in this example, like the earlier "Hello" method, is a *string selector*, and is executed whenever the string "what is your salary" is typed at the terminal.

1.3. Action Equations and Data Dependency

One significant difference between MELD and other object-oriented languages is the notion of *action equations*. An action equation is a statement that is executed by data dependency rather than sequentially. Normally, statements within a begin end block are executed one after another. The execution of action equations, however, is determined by when their inputs (variables on the right hand sides) become available.

A parallel block in MELD is enclosed by curly braces:

action-block ::= { *statements* }

```
METHODS:

M1() --> {  A := B + C;
            B := 5;
            C := 7; }
```

```
                        +---------+
                 /---> | B := 5  | --\
      +----------+/      +---------+    \-> +-------------+
----> |  Method  |                          | A := B + C  |
      +----------+\      +---------+    /-> +-------------+
                 \---> | C := 7  | --/
                        +---------+
```

Figure 1-5: Data Dependency Relations Among Action Equations in MELD

Consider the method in figure 1-5. In this example, there are three action equations, "`A := B + C`", "`B := 5`", and "`C := 7`". It is clear that the first statement relies on variables `B` and `C`. The second two statements are assignments to the constants `5` and `7` so they have no reliance on values of variables.

The data flow dependencies force "`A := B + C`" to be executed after both "`B := 5`" and "`C := 7`", but do not specify which of those two must be executed first. On a multiprocessor system, they might conceivably be executed in parallel, although is this particular example there is little merit. If, on the other hand, B and C were each being set to the return value of some complicated function (with no side effects), executing them in parallel might be very worthwhile.

Let's consider the following example program, which implements the part of a simple "savings account" that handles deposits. We will add to this example at various points in this manual.

In this example, if you were to change the order of the two action equations in the "Deposit" method, you would find that, due to data dependency:

```
balance := balance + cash;
printf("the balance is %d",balance);
```

is equivalent to

```
printf("the balance is %d",balance);
balance := balance + cash;
```

By examining Fig. 1-5, you can also find that:

```
A := B + C;
B := 5;
C := 7;
```

is equivalent to

```
B := 5;
A := B + C;
C := 7;
```

```
FEATURE Bank

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    CitiSaver : Savings_Account := Savings_Account.CREATE;

  CLASS Savings_Account ::=
    balance : INTEGER := 0;

    METHODS:

      "Deposit %d"(cash : INTEGER) --> {
         balance := balance + cash;
         printf("the balance is %d\n",balance);
      }

    END CLASS Saving_Account
END FEATURE Bank
```

Figure 1-6: Bank Program

and, in fact, all orderings of these three action equations are equivalent. Regardless of ordering, A will be equal to 12 after these three action equations are executed.

1.4. Messages to Objects

Our examples so far have used string messages and string selectors in the methods. The runtime system, as we saw, sends lines of user input to objects with matching string selectors.

Another form of selector is called the *symbolic selector*, which matches symbolic messages sent to an object. The symbolic message and symbolic selector look very similar to a procedure call and a procedure header in Pascal. Whereas string selectors are used for messages that interact with the user, symbolic selectors are used for messages internal to the program.

Here is an example of a method using symbolic selectors. Note that selectors look like procedure headers, and you must have a type for each selector argument:

```
CLASS PositiveInteger ::=

  METHODS:
    Add(i,j : INTEGER) -->
                      return(i+j);

    Sub(i,j : INTEGER) -->
                      if ((i-j) > 0) then
                        return(i-j);
                      else
                        return(0);

END CLASS PositiveInteger
```

You can invoke methods using a synchronous or asynchronous message. The synchronous message looks like a normal C function call except it is preceded by a variable name and a dot. The message is sent to the object represented by the variable.

For example, if you have an object `balance` of type `PositiveInteger` as defined in the previous example, you can send synchronous messages using the syntax:

```
balance.Add(6,10);
```

The synchronous message is also called the MELD function call, since, along with `SEND` (see below), it is the principal mechanism for calling other methods. Note also that MELD allows calling external C functions using the normal C function call syntax.

MELD is a concurrent programming language. Two (or more) methods of any object may be active at the same time. One way to achieve concurrency is to use the asynchronous form of the `SEND` statement. With `SEND` execution continues immediately after sending the message, and in effect, you create a new thread of control. You can use `SEND` with symbolic or string messages.

For example, you can use `SEND` to output on your terminal by sending a string message to the system defined object `stdout`:

```
SEND "Hello, World!\n" to stdout;
```

Everything sent to `stdout` will be displayed on the standard output (usually your terminal).

Figure 1.4 is an example of a method that interacts with the user for input and then uses MELD functions.

`CitiSaver.Withdraw(cash)` is a MELD function call that sends the message `Withdraw(cash)` to object `CitiSaver` and then waits for the result sent back from `CitiSaver`. Since the return value will be assigned to `balance` and the following statement (2) depends on the value of `balance` (by the rules of data dependency), statement (2) will not be executed until the function call returns and `balance` is assigned its new value.

Next, we show an example of using asynchronous sending to initiate another thread of control (see figure 1.4).

```
FEATURE Bank

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    teller : Teller := Teller.CREATE;
    CitiSaver : Savings_Account := Savings_Account.CREATE;

  CLASS Teller ::=

    METHODS:

    "withdraw %d"(cash : INTEGER) --> {
       balance : INTEGER;

       balance := CitiSaver.Withdraw(cash);      {*  (1)  *}
       printf("the balance is %d\n",balance);    {*  (2)  *}
    }
  END CLASS Teller

  CLASS Savings_Account ::=
    balance : INTEGER := 1000;

  METHODS:

    Withdraw(cash: INTEGER) --> {
      balance := balance - cash;
      RETURN(balance);
    }

  END CLASS Savings_Account
END FEATURE Bank
```

Figure 1-7: Interactive Bank Program

In figure 1.4, statement (1) sends the message `Withdraw(cash)` to object `CitiSaver`. Statement (2) may then run concurrently with statements (3) and (4) and the output could be either ordering of statements (2) and (4). Note that the `teller` does not know the new balance, and in fact cannot tell when the withdrawal has been completed! This situation could be avoided by having `withdraw` explicitly `SEND` a notification message back to its caller, but in most cases where information must be returned to the caller, synchronous message passing (function call) should be used, as in the previous example.

```
FEATURE Bank

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    teller : Teller := Teller.CREATE;
    CitiSaver : Savings_Account := Savings_Account.CREATE;

  CLASS Teller ::=

  METHODS:

    "withdraw %d"(cash : INTEGER)--> {
      SEND Withdraw(cash) TO CitiSaver;                {*(1)*}
      printf("your transaction is being processed\n"); {*(2)*}
    }
  END CLASS TELLER

  CLASS Savings_Account ::=
    balance : INTEGER := 0;

  METHODS:

    Withdraw(cash : INTEGER) --> {
      balance := balance - cash;              {*(3)*}
      printf("the balance is %d\n",balance);  {*(4)*}
    }
  END CLASS Saving_Account
END FEATURE Bank
```

Figure 1-8: Asynchronous Bank Program

1.5. Constraints

We have already seen a simple example of a constraint statement (in figure 1.1). A constraint statement differs from a method in that it has no selector. Once a constraint statement is defined for an object it will be executed whenever the right hand side variables in the statement change, and when the object is created.

Figure 1.5 shows an example of using a constraint statement in a class. We have a constraint specifying the equivalent temperature in degrees Fahrenheit (F) for one given in degrees Celsius (C). This constraint causes `F` to be set to a new value immediately following any change to `C`, keeping the Fahrenheit temperature consistent with the Celsius temperature. It does not, however, cause `C` to respond to changes in `F`.

In general, a constraint defining a relation between instance variables in a class will be executed whenever a variable on the right-hand side changes, but *not* when a variable on the left-hand side

```
FEATURE Weather World

INTERFACE:
IMPLEMENTATION:

  OBJECT:
    thermometer : FConverter := FConverter.CREATE;

  CLASS FConverter ::=
    F,C : REAL := 0;

    METHODS:
      F := 32 + C * 9.0/5.0;   {* constraint *}

     "set %f"(x : REAL) --> C := x;

     "F"--> printf("F = %f\n",F);
  END CLASS FConverter
END FEATURE Weather World
```

Figure 1-9: Weather Program

changes. A two way constraint may easily be specified by giving two constraints, one for each direction. For example, the two constraints

```
F := 32 + C * 9.0/5.0;      {* constraint *}
C := (F - 32) * 5.0/9.0;    {* inverse constraint *}
```

together specify that whenever either the Fahrenheit or Celsius temperature changes, the other one should be updated in a consistent manner.

Our next example (see figure 1.5) shows a useful debugging feature — how a constraint can be used to display all changes to a variable as they happen. It shows a modification of our banking example to display the `balance` every time it changes.

Constraints are declarative. In order to display the balance whenever and wherever changed, the programmer need only declare one constraint. In the absence of any effective sort of debugger for MELD, constraints are the only way available to trace instance variables when debugging. Eventually, MELD will have a debugger (MD), but until then, remember this example.

1.6. Method Interleaving

MELD is a concurrent programming language; calls to two (or more) methods of an object may be active simultaneously. Here, we show an example of what can happen when two methods are activated concurrently.

```
FEATURE Bank

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    teller : Teller := Teller.CREATE;
    SuperSaver : Savings_Account := Savings_Account.CREATE;

  CLASS Teller ::=

  METHODS:

    "withdraw %d"(cash : INTEGER)--> {
      SEND Withdraw(cash) TO SuperSaver;               {*(1)*}
      printf("your transaction is being processed\n"); {*(2)*}
    }
  END CLASS TELLER

  CLASS Savings_Account ::=
    balance : INTEGER := 0;

  METHODS:

    {*  constraint  *}
    printf("the balance is %d\n",balance);

    Withdraw(cash : INTEGER) --> {
      balance := balance - cash;   {*(3)*}
      printf("the balance is %d\n",balance);  {*(4)*}
    }
  END CLASS Saving_Account
END FEATURE Bank
```

Figure 1-10: Bank Program with Debug Constraint

```
printf("a = %d",A);  {*  (1)   *}
printf("b = %d",B);  {*  (2)   *}
printf("c = %d",C);  {*  (3)   *}
printf("d = %d",D);  {*  (4)   *}

FOO (X : INTEGER)-->
{
     A := X + 1;  {*   (5)   *}
     C := B + 1;  {*   (6)   *}
}
BAR (X : INTEGER)-->
{
     B := X + A;  {*   (7)   *}
     D := C + 1;  {*   (8)   *}
}
```

If method `FOO` and method `BAR` are called at the same time, the statements in the methods will interleave according to the rules of data dependency. In this example, there is only one valid execution ordering, (5)->(1)->(7)->(2)->(6)->(3)->(8)->(4). This ordering occurs because data dependencies force (5) to be executed before (7), (7) before (6), and (6) before (8), giving us a partial execution path, before constraints are considered, of (5)->(7)->(6)->(8). Then, since constraints are executed as soon as possible after their right-hand-side variables change, the complete execution path will be as given above.

Race condition may occur because of the timing of message passing. For example, if the message for `FOO` arrives earlier than the message for `BAR` then `FOO` might start executing and the interleaving of the statements may cause adverse effects.

2. Declarations, Variables, Literals and Types

MELD is a strongly typed language -- you must declare all of your variables and the compiler will enforce certain rules in their use.

Some basic variable types are offered by MELD, and of course, using classes, MELD programmers can create new types for themselves. In this chapter we talk about what types are offered by MELD, how to declare them, and how to use them.

2.1. Declarations

A declaration in MELD consists of one or more variable names, a colon, a type specifier, and an optional initializer:

object-def	::=	*var-list* `:` *type-specifier* `;`
	|	*var-list* `:` *type-specifier* `:=` *signed-constant* `;`
	|	*var-list* `:` *type-specifier* `:=` *LABEL* `.` *CREATE* `;`

where `LABEL` is a declared class name.

The initializer assigns the value to each element in the *var-list*, so for example:

```
Bob,Carol,Ted,Alice : People := People.CREATE;
```

will create four objects and

```
BobAge,CarolAge : INTEGER := 24;
```

will assign `24` to both `BobAge` and `CarolAge`. You would use two declaration statements in order to initialize these variables to different values.

Depending on its placement in the MELD program, the declaration statement can be used to define global variables, instance variables, or local variables. Global variables are accessible by all classes in the feature (and, if exported, they are accessible to other features as well). Instance variables are private to each object. When an object is created, it gets a private copy of the instance storage. And local variables are private to each method; like local variables in a subroutine, their value is not preserved across calls.

The three types of variables are shown in the example of figure 2.1. This example shows the use of a global variable `FDIC_Limit` that defines the limit of insurance per savings account offered by the Federal Deposit Insurance Corporation. Clearly we want this constant to be the same for all `Account` objects so it is defined in the `OBJECT` section and made global.

The MELD message `CREATE` is useful for initializing variables, see section 4.1 for more information.

2.2. Types

There are six basic types in MELD:

Integer	an integer, typically reflecting the natural size of integers on the host machine.

```
FEATURE VarExample

INTERFACE:
IMPLEMENTATION:

  OBJECT:
    Main : V_Test := V_Test.CREATE;
    global : INTEGER := 1;

  CLASS V_Test ::=
    instance : INTEGER := 2;

    METHODS:

      "show" --> {
        local : INTEGER := 3;

        printf("%d %d %d\n",global,instance,local);
      }

  END CLASS V_Test

END FEATURE VarExample
```

Figure 2-1: Variables in a Program

Real	a single-precision floating point number.
Double	a double-precision floating point number.
Boolean	True(1) or False(0).
Character	one byte, machine-dependent but usually ASCII.
String	a one dimensional array of characters.

The declarations `Real` and `Double` will reflect their types ‘‘natural’’ size on the host machine.

2.2.1. Compound Types

In addition you can define a type using the following constructors:

`ARRAY [` *INTEGER* `..` *INTEGER* `] OF` *basic-type*
`ARRAY [` *INTEGER* `..` *INTEGER* `] OF` *LABEL*

Here, *basic-type* is one of the types defined above.

Note: initialization does not work for arrays, and there is no range checking on the indices.

2.3. Constants (Literals)

String, character, integer, and floating point constants may be used in your MELD program. The format is similar to the C language format, and is described in section 7.3.4. In addition the

```
FEATURE SavingsAndLoan

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    FDIC_Limit : INTEGER := 200000;
    Savings : Account := Account.CREATE;

  CLASS Account ::=
    balance : INTEGER := 0;

    METHODS:

      printf("Your balance is %d\n",balance);

      "deposit %d"(cash : INTEGER)--> {
         balance := balance + cash;
         if (balance > FDIC_Limit) then
           printf("careful, $%d is uninsured\n",
                (balance-FDIC_Limit));
      }
    END CLASS Account
END FEATURE SavingsAndLoan
```

Figure 2-2: Global Variables

MELD language defines a number of system constants for your use[5]:

```
TRUE     1
FALSE    0
NIL      0
NULL     0
```

2.4. Variables

A variable name starts with a character of the alphabet and continues with any number of alphanumeric or underscore characters. Case is significant in variable names.

You may not use a variable name that conflicts with one of the MELD reserved words, see 7.3.3, or one of the C language reserved words (refer to your local C compiler reference manual).

[5]These constants may not be redefined.

2.4.1. System Variables

System variables are distinguished by a dollar sign as the first character in their names. These variables are generally read only, unless specified otherwise, and include:

`$SELF` Refers to the object that is executing the current statement.

`$SENDER` Refers to the object that sent the current message.

`$SELECTOR` The entire string message that invoked this method. For symbolic selectors `$SELECTOR` holds the selector name as a string. When used with the `*` selector you may inspect `$SELECTOR` to see what the message received was.

3. Operators, Expressions and Statements

This chapter covers the expression and statement syntax of MELD. This syntax is conveniently similar to C except that the assignment operator is := instead of =.

3.1. Operator Summary

MELD supports arithmetic, relational, and boolean operators.

3.1.1. Arithmetic Operators

In MELD there are three classes of arithmetic operators:

binary arithmetic operators	+, -, *, and /.
modulus operator	%, which produce the remainder when the first operand is divided by the second operand.
unary operators	the + and - signs, for example, -3.

The precedence of these operators in MELD is the same as in the C language.

3.1.2. Relational Operators

The relational operators are:

>	Greater than.
>=	Greater than or equal to.
<	Less than.
<=	Less than or equal to.
= (or) ==	Equal to.
<> (or) !=	Not equal to.

3.1.3. Logical Operators

OR (or) |	OR operator.
AND (or) &	AND operator.
XOR (or) ^	Exclusive OR operator.
NOT (or) !	NOT operator.

3.2. Expressions

Expressions in MELD are identical to expressions in C.

3.3. Statement Summary

The statement in MELD is the unit of execution and concurrency. Chapter 5 describes how to combine statements into blocks that execute their statements sequentially or in parallel. Section 7.4.7 describes statements used as constraints within a class, and chapter 4 describes those statements used for communicating among objects.

This section will provide you with an description of the basic statements in MELD: assignment, condition, return, and the interface to external C procedures.

3.3.1. Assignment Statement

The syntax of the assignment statement is:

assignment ::= *variable asgn-operator expression*

where *variable* can be a global variable, an instance variable, or a local variable declared in a method. The expression is the same as a C language expression, and the *asgn-operator* is defined as:

```
asgn-operator  ::=  := | +:= | -:= | *:=
               |   /:= | %:= | >>:= | <<:=
               |   &:= | ^:= | |:=
```

The assignment operators are identical to their C language counterparts, except they use `:=` instead of `=`. Expressions such as

```
i := i + 1;
```

in which the left hand side is repeated, can be written as

```
i +:= 1;
```

where `+:=` is an `asgn-operator`. The possible assignment operators are:

:=	Simple assignment.
+:=	Addition.
-:=	Subtraction.
*:=	Multiplication.
/:=	Division.
%:=	Remainder.
>>:=	Right shift.
<<:=	Left shift.
&:=	Bitwise AND operator.
^:=	Bitwise XOR operator.
|:=	Bitwise OR operator.

3.3.2. Conditional Statement

The syntax of the `IF-THEN-ELSE` statement is

if-statement ::= `IF` (*expression*) `THEN` $statement_1$
| `IF` (*expression*) `THEN` $statement_1$ `ELSE` $statement_2$

Where *expression* is a boolean expression, and *statement* is a statement described in this chapter, or a block statement described in chapter 5. As is normally the case, any ambiguity is resolved by associating the `ELSE` with the closest else-less `IF`.

3.3.3. Return Statement

The return statement is used for a method to return a value. The format is:

return-statement ::= `RETURN` *expression*

3.3.4. C Interface

MELD provides an interface to the C language. MELD programs can simply use C function calls or procedures by using the ordinary C function call syntax. A C function call is legal wherever a MELD function call (synchronous send) is legal.

The parameter types in MELD programs will be converted as follows:

INTEGER	int
REAL	float
DOUBLE	double
STRING	char *
BOOLEAN	int
CHARACTER	char

The current implementation of MELD does not do any checking on the types or number of arguments to C functions. Our programs using `printf()` throughout this manual are examples of an external C function call.

3.4. Comments

The format for a comment in MELD is:

`{*` *comment* `*}`

3.5. A Small Example

The following example, in figure 3.5, uses some of the language features covered in this chapter. The example also uses a construct that will be explained fully in chapter 5, namely, a group of statements enclosed in square brackets:

`[` *statements* `]`

These statements will be executed sequentially, one right after another, the same as using `BEGIN` and `END` in Pascal, or `{}` in C.

```
FEATURE HiLo

{* HiLo.   Play the game of High-Low!  Try to guess the computer's
 * secret number.
 *}

INTERFACE:
IMPLEMENTATION:

  OBJECT:
    Main : UserInterface := UserInterface.CREATE;

  CLASS UserInterface ::=
    val : INTEGER := 0;

    METHODS:

      {* Initialization constraint... *}
      send "Help" to $SELF;

      {* Help method.  Show user valid commands *}
      "Help" --> [
        printf("Welcome to the game of Hi-Lo.\n");
        printf("The object of the game is for you to guess");
        printf("a number.  Commands are:\n");
        printf("  help  exit  play <number>\n\n");
      ]

      {* Internal method, called when user wins *}
      Winner() --> [
        printf("Congratulations, you are a winner!\n");
        printf("Don't you want to play again? (say play)\n");
      ]

      {* Integer guess method, see if it is high or low *}
      "%d"(guess : INTEGER) -->
        if (guess > val) then
          printf("High\n");
        else if (guess < val) then
          printf("Low\n");
        else $SELF.Winner();

      {* play method.  Start a new game *}
      "give up" --> printf("The val was %d\n",val);

      "play" --> val := rand()*10;

  END CLASS UserInterface
END FEATURE HiLo
```

Figure 3-1: Language Features Example

4. Classes, Objects, Methods and Messages

4.1. Overview

All executable statements in MELD exist within classes. A class is a *definition* of an *abstract data type* consisting of private storage and operations on that storage. The storage is called *instance variables* and the operations are called *methods*. Since the class is only a description of the storage and operations, you *instantiate* the class, that is, create an instance, as you would create a new record structure in Pascal given its template.

The method we have seen for instantiating a class is by using `CREATE` in a variable declaration[6]:

```
OBJECT:
  mazda : Car := Car.CREATE;
```

This creates a new `Car` *object*, called `mazda`. We say that `mazda` is an *instance* of the class `Car`. An object has its own private storage known as *instance variables*.

Once an object has been created you can invoke the *methods* defined by the class. Using the `SEND` statement or the MELD function call you send messages to an object and invoke one of its methods. The methods themselves may invoke other methods using `SEND` and the MELD function call, or they may use external C procedures, or the simple MELD statements described in chapter 3.

4.2. Classes and Instances

A MELD class definition is:

class-decl	::=	`PERSIST`$_{opt}$ `CLASS` *IDENTIFIER* `:=`
		*object-def**
		methods-part$_{opt}$
		`END CLASS` *identifiers*$_{opt}$

The data of class objects may persist across program executions by using the `PERSIST` keyword (see the next section for details). The *object-def* section is used to declare instance variables, and the *methods-part* is used to declare functions and constraints. A single *IDENTIFIER* is used to name the class, and an optional list of identifiers is allowed for the `END CLASS`, these identifiers do not need to match.

For information on how to declare objects see section 2.1.

4.2.1. Persistent Classes

Objects of a class declared with the `PERSIST` keyword remain in existence even when the MELD program terminates. The next time the program (or another) is invoked all data from persistent objects may be restored. Objects are stored and retrieved from the file `MELD.db` in the currently connected directory. This file should not be removed unless you want to ‘‘forget’’

[6] `class.CREATE()` may also be used as a function call to create objects.

the persistent objects.

To create and access persistent objects you first declare the class with the `PERSIST` attribute and then use `class.CREATE("`*name*`")` to create and identify the name of the object. MELD will look for *name* in the database, and restore all class variables to the new object; otherwise MELD creates a new object and initializes the class variables as determined by the class declaration.

Figure 4.2.1 is an example of using `PERSIST` in this way.

Any instance variables changed during the execution of a method of a persistent object will be written back to the database at the time the method finishes executing.

Note: The current MELD compiler will not allow arguments to `CREATE` if `CREATE` is part of the object declaration. Instead you need to use `class.CREATE()` in an assignment statement as part of a method or constraint.

Note: There is no way to remove persistent objects aside from deleting the `MELD.db` file. `DESTROY` does not exits in the current implementation.

Note: The name can be any ascii string, however, don't use ''$'' in the name.

4.2.2. Remote Objects

Remote objects are named objects that are registered through a network wide Naming Service. When you send a message to a remote object the message is routed to the machine containing the object, then the object executes the message, and if a (synchronous) MELD function call with a return value was used, the result is sent back.

When you use `class.CREATE("`*name*`")`, the following steps take place:

1. The local database is searched for a persistent object of the same name. If one is found it is returned.
2. The Naming Service is asked to locate remote objects of the same class and name on the network. The Naming Service will not locate the object unless it (or a member of the same class) is being used in a running MELD program. If an object is found it is returned.
3. If steps (1) and (2) are not able to provide an object, a new object is created and given the specified name. The name is also registered with the Naming Service for future accesses.

If you require that the object already exist you can use `class.GET("`*name*`")`. This is similar to `CREATE` described above, except for the last step; `GET` will never create a new object. Instead, when `GET` is unable to locate a named object it returns a place-holder object that will re-lookup and try to locate the named object at the time of use. If lookup is still unsuccessful at time of use, the message is discarded. A similar result occurs when the server system containing

```
FEATURE Linger

{* Linger.  Example of how class storage can persist. *}

INTERFACE:
IMPLEMENTATION:

  OBJECT:
    Main : UserInterface := UserInterface.CREATE;

  PERSIST CLASS LingerClass ::=
    val : INTEGER := -1;

    METHODS:

    set(n : INTEGER) --> val := n;

    show() --> SEND "The value is %d\n"(val) TO stdout;

  END CLASS LingerClass

  CLASS UserInterface ::=

    LingerOne : LingerClass;

    METHODS:

      send "init" to $SELF;

      "init" --> LingerOne := LingerClass.create("One");

      "show"--> LingerOne.show();

      "set %d"(n : INTEGER) --> LingerOne.set(n);

  END CLASS UserInterface

END FEATURE Linger
```

Figure 4-1: Persistent Class

a named object crashes, or the MELD program containing a named object exits.

One technique for discerning whether a named object was actually located by `GET` is to call a simple method (created just for this purpose) that returns a non-nil value, something like `you_there()`.

Note: The name can be any ascii string, however, don’t use ‘‘$’’ in the name.

4.3. Methods

Methods are the basic program units defined in classes. The format for a method is:

selector `:` *result-type* `-->` *statement*

or

selector `-->` *statement*

depending on whether the method returns a result or not. Here, *selector* is a description of, or template for, a message sent to the object. The *result-type* is a declaration of the datatype of the method and is only necessary if this method uses the `RETURN` statement to return a value. The possibilities for *statement* have been covered in chapter 3 and the class specific statements are described further along in this chapter. The symbol "-->" is used to associate *selector* with *statement*. When *selector* is received the *statement* is executed.

4.3.1. Send Statement

Meld provides an easy way to create another thread of control: asynchronous sending. Unlike the "procedure call" or synchronous message, an asynchronous send does not delay the caller. The message is sent, and execution continues normally. Any return value from the method called asynchronously is lost.

The syntax of asynchronous sending is:

`SEND` *symbolic-message* `TO` *object*

or

`SEND` *string-message* `TO` *object*

where *symbolic-message* is an identifier with a sequence of typed parameters; *object* is an object variable; *string-message* is a string with a sequence of typed parameters.

A method will be activated when receiving a matched message. The receiving method and the sending method will run in parallel (as far as is possible, due to data flow and other execution ordering constraints). For example:

The statement `X := 1;` and `Y := 2` will be executed in parallel, or in either order on a uniprocessor.

4.3.2. Synchronous Send

The synchronous send is also known as the MELD function call. Synchronous sending causes the sender process to stop and wait for the result from the receiver process. The syntax of synchronous sending is:

object`.`*message*

where *object* is an object identifier; *message* can be only be a *symbolic-message*.

For example, figure 4.3.2 shows how to use the return statement and synchronous sends using `$SELF`.

Other examples are:

```
Foo_Obj : Foo := Foo.CREATE;
Bar_Obj : Bar := Bar.CREATE;

CLASS Foo ::=

  y : INTEGER;

  METHODS:

     Foo()--> { SEND Bar() TO Bar_Obj; y := 2; }

END CLASS Foo

CLASS Bar ::=

  x : INTEGER;

  METHODS:

    Bar()--> x := 1;

END CLASS Bar
```

Figure 4-2: Send Program Fragment

```
FEATURE Bank

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    CitiSaver : Savings_Account := Savings_Account.CREATE;

  CLASS Savings_Account ::=
    balance : INTEGER := 0;

  METHODS:
    "Deposit %d"(m : INTEGER) -->
        printf("Your balance is now %d\n",$SELF.Deposit(m));

    Deposit(a : INTEGER) : INTEGER -->
        { balance := balance+a;
          RETURN(balance);
        }

    END CLASS Saving_Account
END FEATURE Bank
```

Figure 4-3: Return Statement

```
Bar_Obj.Bar();

Foo_Obj.Foo(x, y, z);
```

The arguments of a function call can be any arbitrary expression.

4.3.3. DelayUntil Statement

The `DELAYUNTIL` statement is used to wait for a message to be received by the current method. The syntax is:

`DELAYUNTIL` *selector-name*

where *selector-name* is a symbolic selector that does not need to be defined in the current method (see below example). The statement causes the current thread of execution to wait for a message that matches the *selector-name*, and then continues executing. That is, when a message is received by an object, any methods in the current object that were using `DELAYUNTIL` on that method will continue their execution.

The `DELAYUNTIL` statement may be used for synchronization of two methods by, for example the program fragment in figure 4.3.3 shows how to do this. The selector `COMPLETION` does not need to be defined as a method; in this example a method would be extraneous.

```
FOO() --> [
        :
        :
        SEND COMPLETION() TO $SENDER;
        ]

BAR() --> {
        SEND FOO()
        :
        :
        DELAYUNTIL COMPLETION()
        }
```

Figure 4-4: DELAYUNTIL Example Fragment

4.4. Selectors

There are two kinds of selectors that can be employed. The symbolic selector is designed for internal use for messages between program components; a symbolic selector looks much like a Pascal procedure header. As we will see, methods with symbolic selectors are used similarly to procedures in other languages.

4.4.1. Symbolic Selectors

The symbolic selector receives symbolic messages. The format is:

sym-selector ::= *IDENTIFIER* ($\textit{param-list}_{\text{opt}}$)

Here, *IDENTIFIER* is a name for the method, and *param-list* consists of the typed parameters, for example:

```
Withdraw(cash : INTEGER): INTEGER -->

Sort(a : ARRAY[1..20] OF INTEGER)-->
```

4.4.2. String Selectors

The string selector, is intended for external uses of programs (i.e, input). When a string message is sent to an object, it is matched against all of the object's string selectors. Any that match are invoked in parallel.

The syntax of the string selector is:

str-selector	::=	*regular-exp*
	|	*regular-parm-exp* (*param-list*)

where *regular-exp* is a regular expression that defines a set of strings; *regular-parm-exp* is a regular expression with some notations to specify parameter occurrences. For a description of regular expressions refer to grep(1) in the Unix^{TM} 4.3BSD programmer's and user's manuals.

The *param-list* is a list of typed parameters:

param-list	::=	*identifier-list* : *type-specifier*
	|	*param-list* ; *param-list*

Some examples of string selectors are:

```
"hello"--> {* When "hello" is received, then ... *}

"hello [a-zA-Z]+"-->
{* when "hello " concatenated with a name is received, .. *}

"[0-9]+"--> {* when an integer is received, then ..... *}
```

In *regular-parm-exp*, the notations for specifying parameter occurrences is similar to the scanf() function. The current Meld implementation supports "%d", "%f" and "%s", which indicate the occurrences of variables of types `INTEGER`, `REAL` and `STRING`, respectively. Examples are:

```
"X is %d"(x : INTEGER)-->

"Withdraw %f" (cash : REAL)-->

"My name is %s Doe" (name : STRING)-->
```

When multiple parameters are specified, they must occur in the same order in the input string as in the parameter list. The first example below is incorrect; the second shows the correct parameter list for the same input string.

```
"Withdraw %f from account #%d" (account : INTEGER; amount : REAL) -->

"Withdraw %f from account #%d" (amount : REAL; account : INTEGER) -->
```

4.4.3. Line-Oriented Matching

After an object receives a message, it checks whether the message matches any string selectors. The matching is line-by-line, where a line ends with newline ("\n"). So, each line of the message will be compared with the string selectors. For example:

```
incoming message: "my name is Ishmael"

string selector: "my name is %s"(name : STRING)
```

The incoming message will activate the method and the variable will be bound to "Ishmael". Another example shows a non-matching case:

```
coming message: "They call me Ishmael"

string selector: "my name is %s"(name : STRING)
```

The input doesn’t match the string selector even though part of the input does match the string selector. In order for input to match a string selector, it must match from the beginning of the line.

4.4.4. More Than One Matching

Since two string selectors can describe two overlapping sets of strings, incoming messages may match more than one selector. That is, more than one method might be activated by one message. The activated methods are not ordered in any way, and may be executed in parallel. For example:

```
coming message: "Open the cargo doors HAL"

string selector: "Open the %s"(name : STRING)-->statement1;

string selector: "Open*"-->statement2;
```

Both methods will be activated when the message arrives, and both *statement*$_1$ and *statement*$_2$ will be executed in parallel.

4.4.5. Special Selectors

A special form of selector is available that will match when no other symbolic method selector matches:

any-selector ::= `*`

Figure 4.4.5 shows how this is done.

Note: There is no string form that matches *only* when no other selector matches. The string `"*"` will always match even if their are other matching selectors.

4.5. External Methods and Variables

As discussed in section 1.1 the `INTERFACE:` clause is used when a number of features are separately developed and combined into one program. The syntax for the features clause is

```
FEATURE Match_Any

INTERFACE:
IMPLEMENTATION:

  OBJECT:
    Intf : UserInterface := UserInterface.CREATE;

  CLASS UserInterface ::=

    METHODS:

      * --> printf("Received an unknown message (%s)!\n",$SELECTOR);

      "do" --> $SELF.FUBar();

  END CLASS UserInterface

END FEATURE Match_Any
```

Figure 4-5: Any Example

program	::=	`FEATURE` *identifiers*
		`INTERFACE:` *externals**
		`IMPLEMENTATION:` *body*$_{opt}$
		`END FEATURE` *identifiers*$_{opt}$
externals	::=	`EXPORTS` *port-list*$_{opt}$
	|	`IMPORTS` *port-list*$_{opt}$
port-list	::=	*IDENTIFIER*
	|	*IDENTIFIER* `[` *identifier-list* `]`
	|	*port-list port-list*

The programmer specifies the external features to be used by means of the `IMPORTS` clause, and those classes and globals that are visible to other features with the `EXPORTS` clause.

The `IMPORTS` clause:

```
IMPORTS Feature_Random[Random_Int_Class,Random_Count];
```

declares that `Feature_Random` is an external feature, and what follows inside the brackets is a list of methods and global variables to be used by the importing feature. If no list is supplied, then all methods and global variables declared in the `EXPORTS` list for `Feature_Random` are available.

The `EXPORTS` clause:

```
EXPORTS Random_Int_Class[Get_Random_Int],Random_Real_Class,
         Random_Count;
```

declares that `Random_Int_Class` is a class available to external features, and if followed by a bracketed list of method names, only those methods are visible. If no methods are specified after the class name, then all methods are accessible to external features. The `EXPORTS` list may also contain global variables accessible to external features.

When you import a class you are able to declare variables of the external types, to create instances of the classes, and to send messages to instances of the class (perhaps through global objects). However, you cannot use the class name in a `MERGES` statement (described below), that is you cannot create subclasses of the external class.

Figures 4.5 and 4.5 are programs that show how to use external features[7].

```
FEATURE Random_Feature

INTERFACE:

  EXPORTS Random_Int_Class[Get_Random],
          Random_Count,Random_Real_Class,RandomInt,RandomReal;

IMPLEMENTATION:

  OBJECT:
    Random_Count : INTEGER := 0;
    RandomInt : Random_Int_Class := Random_Int_Class.CREATE;
    RandomReal : Random_Real_Class := Random_Real_Class.CREATE;

  CLASS Random_Int_Class ::=

    METHODS:
      Get_Random() --> {
        Random_Count := Random_Count+1;
        return(rand(0));
       }
  END CLASS Random_Int_Class

  CLASS Random_Real_Class ::=

    METHODS:
      Get_Random() --> {
        Random_Count := Random_Count+1;
        return(rand(0)*1.0);
       }
  END CLASS Random_Real_Class

END FEATURE Random_Feature
```

Figure 4-6: Random External Example

[7]Note: external features do not work with the current compiler as of 9/28/89.

```
FEATURE Do_Random_Things

INTERFACE:
  IMPORTS Random_Feature[Random_Count,RandomInt]

IMPLEMENTATION:

  OBJECT:
    Main : Be_Random := Be_Random.CREATE;

  CLASS Be_Random ::=

    METHODS:
      "random" --> {
        send "The %d random is %d\n"(
            RandomInt.Get_Random(),Random_Count) to stdout;
       }
  END CLASS Be_Random

END FEATURE Do_Random_Things
```

Figure 4-7: Use Random External Example

4.6. Merges Statement

The `MERGES` statement is used to create subclass of existing classes. The syntax is:

merge-decl ::= `MERGES` *identifier-list* `AS` *IDENTIFIER*

Where *identifier-list* is a list of existing classes, and *IDENTIFIER* is a new class name. Normally, `MERGES` merges together the methods of all classes mentioned in the statement and produces a new class. If two methods contain the same name, the statements are merged into one method and executed in dataflow order. It is possible however to specify a combination of three attributes in a class definition to customize the `MERGES` behavior. These attributes are:

`OVERRIDE` On a method of the new class, means MELD uses only the methods described by the new class and not any of the other methods of the same name.

`DEFAULT` On a method of the merged classes, means this method is the default action to be taken if no other (non default) method of the same name occurs in any of the other merged classes. If two default methods occur they get merged. Note that `DEFAULT` on the new class only applies to classes that later `MERGES` new class.

`INSIST` On a method of the merged classes, means the subclass cannot override it.

5. Block Structure

There are three types of blocks in MELD: parallel, sequential, and atomic. The block type defines the runtime behavior of the statements within the block. Statements within a parallel block will be (potentially) executed in parallel according to the partial ordering of data dependency. A sequential block will execute statements sequentially according to their textual ordering. Statements within an atomic block are executed without interleaving with statements outside the block; by default, they are executed "in parallel", or in data-flow order, but atomic and sequential blocks may be combined to give sequential execution without interleaving with other statements.

The MELD grammar defines a block as:

block	::=	`BEGIN` *statements* `END`
	|	`{` *statements* `}`
	|	`[` *statements* `]`
	|	`(` *statements* `)`

A *block* consists of zero or more statements enclosed by delimiters. We have covered the different statement types in chapter 3 and 4. These are:

statement	::=	*block* `;`$_{opt}$
	|	*assignment* `;`
	|	*if-statement* `;`
	|	*object-def* `;`
	|	*function-call* `;`
	|	*send-statement* `;`
	|	*delay-statement* `;`
	|	`RETURN` *expression* `;`
statements	::=	*statement*
	|	*statements statement*

The runtime execution of the statements is determined by the delimiters surrounding the block, as follows:

`BEGIN` *parallel* `END`	Equivalent to `{` *statements* `}`.
`{` *parallel* `}`	The statements are (potentially) executed in parallel according to the partial ordering of data dependencies.
`[` *sequential* `]`	The statements enclosed in the block are executed sequentially.
`(` *atomic* `)`	The statements enclosed in the block are executed without interleaving with any statements outside the block. By default the statements are executed in data-flow order, but a sequential block may be nested inside an atomic block to provide sequential execution without interleaving with other statements.

An atomic block executes without interruption from statements outside the block. Normally, a method may have more than one thread of execution running at one time; the atomic block allows entry of only one thread of execution at a time.

5.1. Parallel Block

A parallel block is a set of statements enclosed by `{}`. The syntax is:

{ *statements* }

where *statements* can be any number of statements including parallel, sequential or atomic blocks.

5.1.1. Partial Ordering of Data Dependency

For every parallel block there is a partial ordering of statement inputs and outputs (i.e, right hand and left hand side variables), according to the rules of data dependency. The execution order is determined by this partial ordering.

In figure 5.1.1 we have an example of a parallel block associated with the selector `ADD` of class `Bank`. Because no data dependency exists between statements (1) and (2), they will be executed in parallel.

```
CLASS Bank ::=

Add(x : INTEGER) --> {
  counter := counter + 1;      {* (1) *}
  all := all + x;              {* (2) *}
}
END CLASS Bank
```

Figure 5-1: Parallel Block Program Fragment

In the cases that data dependency does exists between statements, the partial ordering is determined among statements. For example:

```
CLASS Bank ::=

METHODS:

Add(amount : REAL) --> {
  tax : REAL;

  counter := counter + 1;    {* (1) *}
  total := total + tax;      {* (2) *}
  tax := 0.0825 * amount;    {* (3) *}
}
END CLASS Bank
```

A data dependency relation exists between statement (2) and statement (3) through local variable `tax`. In other words, a value for `tax` must be computed and assigned before statement (2) can be executed. Therefore only the ordering requirement that statement (3) runs before statement (2) exists. The execution ordering between statement (1) and statement (2), and between statement (1) and statement (3) is not restricted, and hence nondeterministic from the programmers point of view. That is, (1) and (2) are concurrent and so are (1) and (3). The partial ordering below indicates how two threads of control might be used to execute the three statements:

```
--> (3) --> (2) -->
------> (1) ------>
```

5.2. Sequential Block

A sequential block is a set of statements enclosed by `[]`. The syntax is:

`[` *statements* `]`

where *statements* can be any number of statements including parallel, sequential or atomic blocks.

Sequential blocks are appropriate when ordering of the statements must be one after another:

```
ReadFile() --> [
    open();
    x := read_x();
    close();
]
```

Here, the programmer requires the file to be opened first, then read, then closed. By enclosing these statements in `[]`, the block will be executed sequentially.

Note: `ReadFile()`, `open()`, `close()`, and `read_x()` are not parts of the MELD language or runtime, they are made up just for this example.

5.3. Atomic Block

An atomic block is a set of statements enclosed by `()`. The syntax is:

`(` *statements* `)`

where *statements* can be any number of statements including parallel, sequential or atomic blocks.

The statements are run in dataflow order when the block is activated and the whole block is treated as a unit. The statements enclosed by the atomic block are executed without interleaving with the statements outside the block. For example:

The execution order is (3)->(1)->(2)->(4). Since statement (1) and statement (2) are in a atomic block, no other statements can be executed between them.

In the next chapter, we will show the use of constraints in MELD programming. It is sometimes very important to use atomic blocks to avoid premature activation of constraints. For example, a constraint is used to automatically move the point `(X, Y)` on the screen when the internal data `X` or `Y` are changed.

```
CLASS Foo ::=

 a, b, c, d : INTEGER;

METHODS:

Foo() --> {
    (
     a := b + 1;                      {* (1) *}
     c := d + 1;                      {* (2) *}
    )
    d := 1;                           {* (3) *}
    SEND "A is %d\n"(a) TO stdout;    {* (4) *}
}

END CLASS Foo
```

Figure 5-2: Atomic Block Program Fragment

```
CLASS Move ::=
  x, y : INTEGER;

  METHODS:

    {* constraint to move the point whenever x or y changes *}
    Move_Point(x,y);

    MOVE(dx,xy : INTEGER) --> (
      x := x + dx;   {*   (1)   *}
      y := y + dy;   {*   (2)   *}
    )

END CLASS Move
```

Without using a atomic block, the `move_point()` constraint would be activated twice. The first time is when `x` is changed in statement (1) and the second is when `y` is changed in statement (2). The point on the screen will be moved horizontally and then be moved vertically. Using a atomic block, the constraint is activated only once.

5.4. The Rules of Data Dependency

5.4.1. First Rule of Data Dependency - Assignment Rule

For two different assignment statements, if the left-hand side variable of one assignment also appears in the right-hand side of another assignment, then there is a data dependence relation between these two assignments.

We say the left-hand side ‘‘affects’’ the right-hand side, and the right-hand side ‘‘depends’’ on the left-hand-side.

A single assignment statement cannot depend on itself. Even if a left-hand side variable also appears on the right-hand side of the assignment, there is no data dependency.

5.4.2. Second Rule of Data Dependency -- If-Then-Else Rule

There is no data dependency between the statements in the if-branch and the statements in the else-branch. The reason is that only one branch is active at any time in an If-Then-Else statement.

5.4.3. Third Rule of Data Dependency -- Change-Type Rule

All Statements have a depend on relation to the variables they use (that is, read).

5.4.4. Fourth Rule of Data Dependency - sequential block rule

There is no data dependency between the statements in a sequential block. But data dependency can, however, exist between a statement in the sequential block and a statement outside the sequential block.

```
CLASS Foo ::=
 x, y : INTEGER;

METHODS:

Exchange() --> [
    temp : INTEGER;

    temp := x;    {* (1) *}
    x := y;       {* (2) *}
    y := temp;    {* (3) *}
]
END CLASS Foo
```

Figure 5-3: Fourth Rule Example Fragment

Even though statement (1) has `x` as an input variable (right-hand side variable) and statement (2) has `x` as an output variable (left-hand side variable), there is no data dependency between (1) and (2), because both statements are in the same sequential block. The statements will be executed sequentially (i.e, (1)->(2)->(3)).

Another example shows the data dependency between the statements in a sequential block and the statements outside the block.

```
CLASS Foo ::=
 a, b, c, d : INTEGER;

METHODS:

Foo() --> {
    [
      a := 1;       {* (1) *}
      c := b;       {* (2) *}
    ]
    [
      b := a + 1;   {* (3) *}
      d := c + 1;   {* (4) *}
    ]
}
END CLASS Foo
```

The execution order is (1)->(3)->(2)->(4).

5.4.5. Fifth Rule of Data Dependency -- Constraint Rule

A constraint has no effect on the dependency of any statements, either statements in a method or another constraint. For example:

```
CLASS Foo ::=
 a, b, c : INTEGER;

METHODS:

    a := b+1;    {*   (1)   *}

   Foo() --> {
    c := a+1;    {*   (2)   *}
    b := c+1;    {*   (3)   *}
}
END CLASS Foo
```

The execution order is (2)->(3)->(1). There is no circularity of data dependency, since the constraint has no "affect" relation with statement (2) and statement (3).

5.4.6. Sixth Rule of Data Dependency - Dynamic Rule

When two methods in one object are executing concurrently and the first five rules of data dependency have been applied to each method, the execution order of statements among the two methods is further limited by the Dynamic Rule. The Dynamic Rule says that when two methods in the same object are running concurrently, the total effect of execution order is determined by interleaving the two partial orderings according to the rules of data dependency.

5.4.7. Data Dependency Deadlock

Dynamic data dependency sometimes causes a deadlock situation due to the circularity of the effect of statements. That is, a circular chain of depends and effects exists, and no statement is ready to run. This situation is considered a fatal error and the MELD runtime will detect and print an error message when a deadlock occurs.

Figure 5.4.7 is an example of a deadlock situation. If the messages `FOO()` and `BAR()` arrive at the object at the same time, the execution order of statements cannot be determined because of the circularity (1)->(2)->(3)->(4)->(1).

```
CLASS Foo ::=
  a,b,c,d : INTEGER;

METHODS:

FOO()--> {
        a := b + 1;    {*   (1)   *}
        c := a * 2;    {*   (2)   *}
        }

BAR()--> {
        d := c + 2;    {*   (3)   *}
        b := d + 4;    {*   (4)   *}
        }
END CLASS FOO
```

Figure 5-4: Deadlock Example Fragment

5.4.8. Seventh Rule of Data Dependency - Recursion Rule

There is no data dependency between two activations of the same method because of the possibility of deadlock; the methods are treated separately.

5.4.9. Race Condition

A race condition is the possibility for some unpredictable outcome to occur as the result of two or more interacting concurrent processes. The example in figure 5.4.7 would have caused a deadlock if both messages arrived at exactly the same time. However, if message `FOO` arrives slightly earlier than message `BAR`, then statement (1) might have gotten executed before dynamic data dependency is applied; and if `BAR` arrives slightly earlier, statement (3) will be executed first. These cases lead to three different results solely because of the timing of messages, therefore a race condition exists.

Race conditions are sometimes difficult to avoid, you may need to resort to atomic or sequential blocks to control the interaction of concurrent methods.

6. Constraints

As we have seen already, statements defined in a class can be associated with a selector as part of a method, or they can be *constraints*, without a selector. In this chapter, we will further explore the constraint statements and their uses. A constraint defines a relation between instance variables in the class. The relation will hold true through the life-time of the object. Technically speaking, constraints are declarative with the highest priority of execution, so the constraints will be evaluated whenever the associated input instance variables are changed. There are three types of constraints: *equation* constraints, *change-type* constraints and *conditional* constraints.

6.1. Equation Constraints

An equation constraint defines an equation relation between a set of input instance variables and one output instance variable. During the lifetime of an object, whenever the values of input variables change, the value of the output variable will be updated so that the relation is maintained. The syntax is:

output := expression-inputs

where *output* is an instance variable declared in the class; *expression-inputs* is an expression with input instance variables. The expression may be any valid MELD expression. See figure 6.1 for example.

```
CLASS Temp::=
  c,f : REAL;

METHODS:

  c := (f - 32.0) * 5.0 / 9.0;

INC(x : REAL) --> f := f + x;

END CLASS Temp
```

Figure 6-1: Constraint Program Fragment

6.2. Change-Type Constraints

Instead of defining a relation between the output instance variable and the input instance variables, a change-type constraint defines a relation between a fact or action and the input instance variable. Change-type constraints do not have an output instance variable defined in the statement. A change-type constraint defines an action that will be evaluated at the time of the object creation and whenever the input instance variables are changed. The action can be a regular C procedure call, an asynchronous send statement or a synchronous send statement.

A special case of a change-type constraint is a constraint with no inputs. This constraint is evaluated only at object creation, and is useful for initializations.

6.2.1. C procedure Constraints

A C procedure constraint is a C procedure call that has one or more than one input parameters. The input parameters should be the instance variables declared in the class. Some examples are described below.

- Change-type constraints might be used to define the relation between the internal data and the screen display: Assume a point on the screen that is represented as X and Y in a class. Whenever X or Y is changed, the point will be moved to the proper location. A C procedure constraint can be defined to simplify the program.

```
CLASS TerminalGraphics ::=
  x, y: INTEGER;

  METHODS:

  Move(x, y);

  Move_X(d : INTEGER) --> x := x + d;

  Move_Y(d : INTEGER) --> y := y + d;

END CLASS TerminalGraphics
```

  where `Move(X, Y)` is a C procedure constraint; `Move_X()` and `Move_Y()` are methods defined in class `Coor`. `Move(x, y)` will move the point in the screen to the location `(x, y)`.
- C procedure constraints might be used to display messages associated with some instance variables. For example:

```
CLASS Bank ::=
  money : INTEGER;

  METHODS:

  printf("the  money is %d\n",money);

  Deposit(x : INTEGER) --> money := money + x;

END CLASS Bank
```

  In sequential programming, in order to trace the history of `money`, printf statements would need to be inserted wherever the instance variable `money` is changed. Here, only one printf statement is needed.

6.2.2. Asynchronous-Send Constraints

An asynchronous-send constraint is a send statement that will send a message to an object and activate a method. The syntax is the same as that defined in a method.

`SEND` *message* `TO` *object;*

Here, *message* can be either a symbolic message or a string message and *object* is an object identifier. One example is to use a send constraint to self initialize an object.

```
CLASS sort ::=

  METHODS:

  SEND Randomize() to $SELF;

  Randomize()--> { srand(0); }

END CLASS Sort
```

As we can see, the message `Radomize()` in the constraint has no input parameter, so that the constraint will be evaluated exactly once when the object is created.

In some cases, an object wants to notify other objects when some instance variables are changed.

```
CLASS Bank ::=
  money : INTEGER;

  METHODS:

  SEND Display(money) TO Display_Object;

  Deposit(x : INTEGER) --> money := money + x;

END CLASS Bank

CLASS Display ::=

  Display(x : INTEGER) --> printf("%d\n",x);

END CLASS Display
```

6.3. Conditional Constraints

The syntax is the same as the conditional statement used in methods (see section 3.3.2). There are two sets of constraints associated with a conditional constraint: if-set and else-set. Only one set of statements is considered active at any time. For example:

```
CLASS Foo
  balance : INTEGER;
  debug : BOOLEAN;

  METHODS:

    IF (debug) THEN
      printf("Balance is now %d\n",balance);
    ELSE
      printf("debug disabled\n");
```

The change-type constraint `printf("Balance is now %d\n",balance)` is active when `debug` is `TRUE`. The constraint `printf("debug disabled\n");` is active when `debug` is `FALSE`, and is executed only once, when `debug` changes to `FALSE`. Only one of these two constraints is active at any time. The value for `balance` will be printed out when `debug` is set to `TRUE` and from then one when `balance` is changed, or the message "debug disabled" will be printed out when `debug` changes from `TRUE` to `FALSE`.

6.4. Limitations

Some limitations for using constraints:

1. There is no block structure. Except for the blocks associated with if-then-else constraints, no block (parallel block, sequential block, etc.) is allowed in constraints.
2. Only parallel blocks can be used to enclose if-set statements and else-set statements in conditional constraints.
3. The implementation disallows synchronous-sends in constraints.

6.5. A Small Example

Figure 6.5 shows an example program using the three types of constraints discussed in this chapter.

```
FEATURE Bank

INTERFACE:

IMPLEMENTATION:

  OBJECT:
    CitiSaver : Savings_Account := Savings_Account.CREATE;

  CLASS Savings_Account ::=
    cash_flow : INTEGER := 0;
    balance : INTEGER := 0;
    debug : BOOLEAN := FALSE;

    METHODS:

      if (debug) then
        printf("balance is now %d\n",balance);
      else
        printf("debugging is disabled\n");

      "[Dd]ebug %s"(s : STRING) -->
        debug := strcmp(s,"on");

      "Deposit %d"(cash : INTEGER) --> {
         balance := balance + cash;
      }

    END CLASS Saving_Account
END FEATURE Bank
```

Figure 6-2: Constraint Example

II Reference Manual

7. MELD Reference Manual

7.1. Introduction

The MELD language is an active research project at Columbia. This manual tries to provide a reference point for questions and issues arising from the use of the language, however, the final arbiter is the source code and your own test programs. In particular you can look at the compiler and preprocessor source code in the project directories `/proj/meld/baseline/compiler` and `/proj/meld/baseline/preproc`.

The implementation of MELD generates C code, which is then compiled into an executable program. Restrictions of the C language and compiler may therefore affect your program. For example, using the C reserved word `void` as a MELD identifier will produce a C compilation error, but not a MELD compilation error. This manual does not itemize all such restrictions, so the MELD programmer must remain aware of the inherent limitations of the compilation process.

This manual describes the MELD programming language as of June 1989.

7.2. Syntax Notation

The syntactic notation used in this manual is a modified BNF form. Syntactic categories (nonterminals) are presented in italics, for example *expression*. The subscript *opt* means the symbol is optional. For example,

$$\textit{remote-list} \quad ::= \quad \textit{remote-list}_{\text{opt}} \ \textit{remote-object-decl}$$

denotes a repetition of one or more *remote-object-decl*. Zero or more repetitions of a symbol is represented with a star superscript, as in `externals`*.

Literal words (terminals) are written in a typewriter style and capital letters, e.g., `BEGIN`, `END`.

Lexical categories, such as *integer*, *string*, etc., are presented in uppercase italics, for example *INTEGER*. All lexical categories are defined using regular expressions based on their lex counterpart definitions.

7.3. Lexical Conventions

There are six lexical categories: identifiers, keywords, constants, strings, operators, and punctuation. Each of these lexical categories is described in detail below.

White space, consisting of space, tab, and newline characters, as well as comments, is ignored by the lexical analyzer and serves only to delimit input tokens as necessary.

7.3.1. Comments

The characters `{*` start a comment that must be ended with `*}`.

7.3.2. Identifiers

A identifier is any number of letters, digits, and underscore characters (_) starting with a letter or underscore. Uppercase and lowercase letters are distinct.

IDENTIFIER ::= [a-zA-Z_][a-zA-Z_0-9]*

7.3.3. Keywords

The following identifiers are reserved for use as keywords:

```
$SELECTOR          $SELF          $SENDER
ALL                AND            ARRAY
AS                 AT             BEGIN
BOOLEAN            BY             CHAR
CLASS              DEFAULT        DELAYUNTIL
DOUBLE             ELSE           END
END                EXPORTS        FALSE
FEATURE            HIGH           IF
IMPLEMENTATION     IMPORTS        IN
INSIST             INTEGER        INTERFACE
IS-PROD            KEY            LOW
MERGES             METHODS        NAME
NIL                NOT            OBJECT
OF                 OR             ORDERED
ORDERED-SET        OVERRIDE       PERSISTENT
REAL               RECORD         REMOTES
RETURN             SEND           SET OF
STRING             THEN           TO
TRUE               UNION          USES
VIEWS              XOR
```

Note: because MELD generates C, all C reserved words are also illegal as identifier names, and will cause C compiler errors.

7.3.4. Constants

7.3.4.1. System Constants

There are several constants predefined by the MELD compiler:

```
TRUE      1
FALSE     0
NIL       0
NULL      0
```

7.3.4.2. Integer Constants

An integer constant consists of a sequence of digits 0 through 9 and is always interpreted in base 10.

INTEGER ::= [1-9][0-9]*
| [0]

Note: this format is different from C, which allows hexadecimal and octal integer constants.

7.3.4.3. Floating Point Constants

Floating point constants consist of an integer (the integer part), a decimal point, and an integer (the fraction part). Both the integer part and the fraction part must be present.

FLOATP ::= [0-9]+"."[0-9]+

Note: this format differs from C, which allows an exponent, and either the integer or the fraction part (but not both) to be missing.

7.3.4.4. Character Constants

A character constant is a printable character surrounded by single quotation marks, as in `'g'`.

CHARACTER ::= [’][\40-\176][’]

This format only allows a single *printable* character, not a backslash followed by a number; the backslash form above is only used to describe the valid range of character inputs.

Note: There is currently no facility for entering characters outside the printable range of 40 (space) to 176 (tilde) ascii. This deficiency exists in the tokenizer `meld.l` that only accepts `['][\40-\176][']` (i.e. *one* character in quote marks) as a character constant. However, it should be possible to allow the fuller C syntax for character constants since MELD generates C code.

7.3.5. Strings

A *STRING* is a sequence of characters starting and ending with quotation marks, and possibly including backslash quoted quote marks as in C.

7.3.6. Embedded C Code

The characters `%{` start a block that must be ended with `%}`. All text between `%{` and `}%` is sent directly to the generated C file, and therefore you may include arbitrary C code between the delimiters.

7.4. Meld Programs

A MELD program is a named *feature* (i.e., modular unit or package), consisting of name, interface, and implementation. A *feature* usually consists of related classes bundled together to provide a coherent functionality. The feature name may be any arbitrary string of *IDENTIFIERS*, for example: `Double Pane Window Manager`.

The feature may also end with some arbitrary string of *IDENTIFIERS*, but not necessarily the same string.

Note: You cannot use a MELD reserved word, such as `AND`, in the *IDENTIFIERS* string. This is a bug.

The `INTERFACE:` part is used to declare any imported or exported classes and variables; the

`IMPLEMENTATION:` part consists of the storage and class declarations.

```
program         ::= FEATURE identifiers
                      INTERFACE: externals_opt
                      IMPLEMENTATION: body_opt
                    END FEATURE identifiers_opt

identifiers     ::= IDENTIFIER
                |   identifiers IDENTIFIER
```

7.4.1. Interface

The interface clause consists of an optional declaration specifying the imported features, exported classes and objects:

```
externals       ::= EXPORTS port-list_opt
                |   IMPORTS port-list_opt
                |   externals externals
```

The `EXPORTS` clause lists those classes defined in the implementation that may be used externally.

The `IMPORTS` clause lists those features whose exported classes may be used internally. Import supports reuse through data abstraction and information hiding in the same fashion as Ada packages. The imported classes may be used in the implementation body as (1) the types of instance variables, (2) the types of global objects, and (3) within a merges clause.

In the `EXPORTS` clause, the *port-list IDENTIFIER* denotes an exported class or global object. When a class is exported, the names within braces denote instance variables and methods that will be visible outside the defining feature. In the `IMPORTS` clause the *IDENTIFIER* denotes a feature, and names within braces denote classes that will be used inside the current feature.

```
port-list       ::= IDENTIFIER
                |   IDENTIFIER [ identifier-list_opt ]
                |   port-list  port-list

identifier-list ::= IDENTIFIER
                |   identifier-list  IDENTIFIER
```

7.4.2. Implementation

The body of the `IMPLEMENTATION` part consists of global object definitions, class declarations and merge clauses. These components may be used in any order and repeated as many times as desired.

```
body            ::= class-decl
                |   merge-decl
                |   object-decl
                |   body body
```

7.4.3. Object Declaration

The *object-decl* is used to define and initialize global object variables. These objects are accessible to all methods in the feature, but in order to be used by other features they must be included in the `EXPORTS` statement.

Objects may be initialized to a constant value or via a method such as `CREATE`. If no initializer is specified then the value will be set to the nil object.

The object declaration portion begins with the keyword `OBJECT` and then any number of object definitions, of the form

object : type := initializer

A list of objects of the same type may be declared in one statement:

```
object-decl    ::=  OBJECT: object-def*

object-def     ::=  identifier-list : type-specifier ;
               |    identifier-list : type-specifier := signed-constant ;
               |    identifier-list : type-specifier := identifier . CREATE ;
```

A *signed-constant* is defined in section 7.4.10. The *identifier* in *identifer.CREATE* must be a class name.

7.4.4. Merges

The *merges-decl* clause provides the inheritance mechanism for MELD. Merging permits objects to inherit instance variables and methods defined in the imported features. The syntax for the merges clause is:

```
merge-decl     ::=  MERGES identifier-list AS IDENTIFIER
```

The merges clause combines the classes and instance variables of any number of features into a single new class (the class named after the AS).

The `MERGES` clause is separate from any class declarations so that it is simple to define a class that is solely the combination of two or more other classes.

If an instance variable or a method with the same name is inherited from more than one MELD superclass, MELD automatically merges the code in each method using dataflow dependencies.

7.4.5. Class Declarations

The class declaration is used to specify the methods (procedures) and instance variables (private object storage) of a class. The declaration includes the optional keyword `PERSIST`, described below, the class name, which may be any valid *IDENTIFIER*, the object definitions for instance variables, and the methods themselves:

class-decl ::= `PERSIST`$_{opt}$ `CLASS` *IDENTIFIER* `:=`
object-def*
methods-part$_{opt}$
`END CLASS` *identifiers*$_{opt}$

methods-part ::= `METHODS` *method**

The `PERSIST` keyword is used to preserve any class objects across executions. This is implemented with the aid of a file stored in the users connected directory (see section 4.2.1).

The *object-def* defines types and (optional) initial values for instance variables. Instance variables are accessible only by the methods of the class they are defined in. Each object (an *instance* of the class) manages a private copy of these variables.

The *methods-part* starts with the keyword `METHODS` and is followed by any number of method declarations and constraints.

7.4.6. Methods

Methods are the basic program unit in MELD, similar to a C subroutine but invoked with a message instead of a procedure call. A method consists of a selector and statements to be executed when a message matching the selector is received:

selector --> *statement*

selector : result-type --> *statement*

A method without a selector is called a constraint and is defined in section 7.4.7.

method ::= *method-attr*$_{opt}$ *constraint*
| *method-attr*$_{opt}$ *sym-selector selector-type*$_{opt}$ `-->` *statement*
| *method-attr*$_{opt}$ *str-selector* `-->` *statement*
| *method-attr*$_{opt}$ *any-selector* `-->` *statement*

method-attr ::= `OVERRIDE`
| `DEFAULT`
| `INSIST`
| `OVERRIDE DEFAULT`
| `OVERRIDE INSIST`

selector-type ::= `:` *type-specifier*

The *method-attr* field defines the behavior of the method when used in a `MERGES` statement:

`OVERRIDE` On a method of the new class, means you use only the methods described by the new class and not any of the other methods of the same name.

`DEFAULT` On a method of the merged classes, means this method is the default action to be taken if no other (non default) method of the same name occurs in any of the other merged classes. If two default methods occur they get merged.

	Not that `DEFAULT` on the new class only applies to classes that later `MERGES` new class.
`INSIST`	On a method of the merged classes, means the subclass cannot override it.

Because MELD is strongly typed, you must specify a *selector-type*, that is a result datatype, if the method returns a value.

The selector may be a symbolic, string, or a special form. These are described below.

Note: A string selector can only be called asynchronously (with the `SEND` statement), and a string selector method cannot return a value.

7.4.6.1. Selectors

When a message is received by an object, the run-time system searches for a matching selector and invokes the appropriate method. The symbolic selector is designed for internal use by programs and resembles a procedure header:

sym-selector ::= *IDENTIFIER* (*param-list*$_{opt}$)

Here *IDENTIFIER* is the selector identifier, and the *param-list* specifies the names and types of the formal parameters:

param-list ::= *identifier-list* : *type-specifier*
| *param-list* ; *param-list*

The string selector is intended for external use (i.e., input from users of MELD programs). When a string message is sent to an object, it is matched against all of the object's string selectors and all matching selectors are invoked. The string selector format is:

str-selector ::= *regular-exp*
| *regular-parm-exp* (*param-list*)

A *regular-exp* and *regular-parm-exp* are quoted strings containing a regular expression as described in the manual entry for ed(1).

regular-exp ::= *STRING*

regular-parm-exp ::= *STRING*

A *regular-parm-exp* allows parameters through the use of "%x" placed within the regular expression indicating the occurrence of parameters. The implementation supports:

```
%d          Integer
%f          Real
%s          String
```

For example:

```
    "x is %d" (x : Integer) -->
```

The *any-selector* is a special form that matches only when no other symbolic selector matches:

any-selector ::= `*`

7.4.7. Constraints

Constraints are statements in the `METHODS` part of a class definition that have no associated selector; they are not invoked directly by a message. Constraints define a relation between variables and actions. When the value of a variable changes some predefined action is triggered. Constraints are declarative. They are executed whenever the associated input variables are changed, and not by some explicit command. There are three types of constraint equations:

constraint ::= *equation-constraint* `;`
| *change-constraint* `;`
| *condition-constraint* `;`

7.4.7.1. Equation Constraints

An *equation-constraint* defines a relation between a set of input instance variables and one output instance variable. The syntax is that of an assignment statement:

equation-constraint
::= *variable* `:=` *expression*

where *IDENTIFIER* is the output instance variable defined by the class, and *expression* is a C *expression*. When any input (i.e., right hand side) variable changes value, the expression is reevaluated and assigned to the output variable.

7.4.7.2. Change Type Constraints

A change type constraint does not have an output variable. This constraint defines an action that will be evaluated when the class object is created and whenever the input instance variables are modified. The action can be a C procedure call, a synchronous send or an asynchronous send. The parameter list of variables are the input instance variables in a change type constraint.

change-constraint
::= *send-statement*
| *function-call*

7.4.7.3. Conditional Constraints

Constraints in the form of an if statement are called conditional constraints. A conditional if is recomputed in response to changes in the arguments to the conditional expression:

condition-constraint
::= *if-statement*

See section 7.4.9.3 for the *if-statement* syntax.

7.4.8. Type Specifiers

There are six basic types in MELD:

basic-type ::= `INTEGER`
| `BOOLEAN`
| `CHAR`
| `STRING`
| `REAL`
| `DOUBLE`

The *type-specifier* is

type-specifier ::= *basic-type*
| `ARRAY [` *INTEGER* `..` *INTEGER* `] OF` *basic-type*
| `ARRAY [` *INTEGER* `..` *INTEGER* `] OF` *IDENTIFIER*
| *IDENTIFIER*

7.4.9. Statements

statement ::= *block* `;`$_{opt}$
| *assignment* `;`
| *if-statement* `;`
| *object-def* `;`
| *function-call* `;`
| *send-statement* `;`
| *delay-statement* `;`
| `RETURN` *expression* `;`

statements ::= *statement*
| *statements statement*

7.4.9.1. Compound Statement, or Block

Several statements can be used anywhere one is expected by surrounding the sequence of statements by the delimiters described below. The delimiters define the type of concurrency (parallel or sequential) and whether the block is executed atomically.

block ::= `BEGIN` *statements* `END`
| `{` *statements* `}`
| `[` *statements* `]`
| `(` *statements* `)`

The statement *block* consists of zero or more statements enclosed by delimiters. The runtime execution of the statements is determined by the delimiters as follows:

`BEGIN` *parallel* `END`
Equivalent to `{` *statements* `}`.

`{` *parallel* `}` The statements are (potentially) executed in parallel according to the partial ordering of data dependencies.

`[` *sequential* `]` The statements enclosed in the block are executed sequentially.

`(` *atomic* `)` The statements enclosed in the block are executed without interleaving with any statements outside the block. By default the statements are executed in data-flow order, but a sequential block may be nested inside an atomic block to provide sequential execution without interleaving with other statements.

An atomic block executes without interruption from statements outside the block. Normally, a method may have more than one thread of execution running at one time; the atomic block allows entry of only one thread of execution at a time.

7.4.9.2. Assignment Statement

The assignment statement sets *variable* to be the value of *expression*. The *variable* may be a global, instance or local variable:

assignment ::= *variable asgn-operator expression*

asgn-operator ::= `:=` | `+:=` | `-:=` | `*:=`
| `/:=` | `%:=` | `>>:=` | `<<:=`
| `&:=` | `^:=` | `|:=`

The operators have the same meaning as in C (the C form is the same, but without the colon).

7.4.9.3. Conditional Statement

if-statement ::= `IF` (*expression*) `THEN` *statement*$_1$
| `IF` (*expression*) `THEN` *statement*$_1$ `ELSE` *statement*$_2$

The expression is evaluated and if it is non-zero then statement$_1$ is executed. If `ELSE` is used then statement$_2$ is executed only when the expression evaluates to `FALSE` (0). As is normally the case, any ambiguity is resolved by associating the `ELSE` with the closest else-less `IF`.

7.4.9.4. Procedure Call

The procedure call is a synchronous message to a MELD object, when the dotted form is used, or a subroutine call to an external C subroutine.

In the dotted form *object* is an instance, global, or local variable (or system defined object name, such as `$SENDER`) containing the object that will receive the message, and *symbolic-message* consists of the method name and parameters.

procedure ::= *object* `.` *symbolic-message*
| *symbolic-message*

Note: a string message is not allowed in this form, use the `SEND` statement.

When calling C procedures, the parameter types are converted as follows:

```
MELD         C
-----       -----
INTEGER      int
REAL         float
STRING       char *
BOOLEAN      int
CHAR         char
DOUBLE       double
```

No type checking is performed on the arguments for external procedure calls.

7.4.9.5. Send Statement

The `SEND` statement sends a message to an object and immediately continues execution (i.e., the message is sent asynchronously).

send-statement ::= `SEND` *symbolic-message* `TO` *variable*
| `SEND` *string-selector* `TO` *variable*

The *symbolic-message* is a selector identifier with a sequence of parameters.

symbolic-message ::= *IDENTIFIER* `(` *exp-list*$_{\text{opt}}$ `)`

exp-list ::= *expression*
| *exp-list exp-list*

The *string-message* is a string with a sequence of optional parameters.

string-message ::= *STRING*
| *STRING* `(` *exp-list* `)`

7.4.9.6. System Variables $Sender, $Self and $Selector

Two system defined variables are provided for referencing the object itself (`$SELF`) and the message sender (`$SENDER`). These variables cannot be used on the left hand side of assignment statements.

`$SELECTOR`
The entire string message that invoked this method. For symbolic selectors `$SELECTOR` holds the selector name as a string. When used with the `*` selector you may inspect `$SELECTOR` to see what the message received was.

7.4.9.7. Delay Statement

The `DELAYUNTIL` statement is used to wait for a message to be received by the current method. The syntax is:

`DELAYUNTIL` *selector-name*

where *selector-name* is a symbolic selector that does not need to be defined in the current method. The statement causes the current thread of execution to wait for a message that matches the *selector-name*, and then continues executing. That is, when a message is received by an object, any methods in the current object that were using `DELAYUNTIL` on that method will continue their execution.

7.4.9.8. Return Statement

A method returns a result by means of the *return-statement*.

return-statement ::= `RETURN` *expression*

7.4.10. Constant

A constant in MELD consists of a user specified number, string or character, or one of the system constants:

constant	::=	`TRUE`
	|	`FALSE`
	|	`NIL`
	|	`NULL`
	|	*FLOATP*
	|	*INTEGER*
	|	*STRING*
	|	*CHARACTER*

See section 7.3 for the lexical definitions of these constant types. *FLOATP*, *INTEGER*, *STRING* and *CHARACTER* are lexical classes of constants.

A *signed-constant* is a constant with an optional unary sign:

signed-constant	::=	*constant*
	|	`+` *constant*
	|	`-` *constant*

8. Complete Grammar

program	::=	FEATURE *identifiers*
		INTERFACE: *externals**
		IMPLEMENTATION: *body*$_{opt}$
		END FEATURE *identifiers*$_{opt}$
externals	::=	EXPORTS *port-list*$_{opt}$
	|	IMPORTS *port-list*$_{opt}$
port-list	::=	*IDENTIFIER*
	|	*IDENTIFIER* [*identifier-list*]
	|	*port-list port-list*
body	::=	*class-decl*
	|	*merge-decl*
	|	*global-object-decl*
merge-decl	::=	MERGES *identifier-list* AS *IDENTIFIER*
global-object-decl	::=	OBJECT*: object-def**
class-decl	::=	PERSIST$_{opt}$ CLASS *IDENTIFIER* :=
		object-def; *
		methods-part$_{opt}$
		END CLASS *identifiers*$_{opt}$
methods-part	::=	METHODS*: method**
object-def	::=	*identifier-list* : *type-specifier*
	|	*identifier-list* : *type-specifier* := *signed-constant*
	|	*identifier-list* : *type-specifier* := *variable* . *CREATE*
method	::=	*method-attribute*$_{opt}$ *constraint*
	|	*method-attr*$_{opt}$ *sym-selector selector-type*$_{opt}$ --> *statement*
	|	*method-attr*$_{opt}$ *str-selector* --> *statement*
	|	*method-attr*$_{opt}$ *any-selector* --> *statement*
method-attr	::=	OVERRIDE
	|	DEFAULT
	|	INSIST
	|	OVERRIDE DEFAULT
	|	OVERRIDE INSIST
selector-type	::=	: *type-specifier*

```
sym-selector      ::= IDENTIFIER ( param-list_opt )

param-list        ::= identifier-list : type-specifier
                  |   param-list ; param-list

str-selector      ::= regular-exp
                  |   regular-parm-exp ( param-list )

regular-exp       ::= STRING

regular-parm-exp  ::= STRING

any-selector      ::= *

constraint        ::= equation-constraint ;
                  |   change-constraint ;
                  |   condition-constraint ;

equation-constraint
                  ::= IDENTIFIER := expression

change-constraint
                  ::= send-statement
                  |   function-call

condition-constraint
                  ::= if-statement

signed-constant   ::= constant
                  |   + constant
                  |   - constant


basic-type        ::= INTEGER
                  |   BOOLEAN
                  |   CHAR
                  |   STRING
                  |   REAL
                  |   DOUBLE

type-specifier    ::= basic-type
                  |   ARRAY [ INTEGER .. INTEGER ] OF basic-type
                  |   ARRAY [ INTEGER .. INTEGER ] OF IDENTIFIER
                  |   IDENTIFIER

block             ::= BEGIN statements END
                  |   { statements }
                  |   [ statements ]
                  |   ( statements )
```

```
statement         ::= block ;_opt
                  |   assignment ;
                  |   if-statement ;
                  |   object-def ;
                  |   function-call ;
                  |   send-statement ;
                  |   delay-statement ;
                  |   return-statement ;

statements        ::= statement
                  |   statements statement

if-statement      ::= IF ( expression ) THEN statement_1
                  |   IF ( expression ) THEN statement_1 ELSE statement_2

assignment        ::= variable asgn-operator expression

asgn-operator     ::=  := | +:= | -:= | *:=
                  |   /:= | %:= | >>:= | <<:=
                  |   &:= | ^:= | |:=

function-call     ::= variable . symbolic-message
                  |   symbolic-message

delay-statement   ::= DELAYUNTIL IDENTIFIER

return-statement  ::= RETURN( expression )

send-statement    ::= SEND symbolic-message TO variable
                  |   SEND string-message TO variable

symbolic-message  ::= IDENTIFIER ( exp-list_opt )

string-message    ::= STRING
                  |   STRING ( exp-list )
```

```
variable         ::=  IDENTIFIER
                 |    IDENTIFIER [ expression ]
                 |    variable . variable

object           ::=  IDENTIFIER
                 |    IDENTIFIER [ expression ]

primary          ::=  variable
                 |    function-call
                 |    constant
                 |    ( expression )

expression       ::=
                 |    primary
                 |    + expression
                 |    - expression
                 |    ( basic-type ) expression
                 |    expression operator expression

operator         ::=   -  |  +  |  *
                 |    <>  | !=  | ||  | &&  | >=  | <=
                 |    <<  | >>  | <   |  >  |  %  | !
                 |     /  |  |  |  ^  | ==  |  =

constant         ::=  TRUE
                 |    FALSE
                 |    nil
                 |    NULL
                 |    FLOATP
                 |    INTEGER
                 |    STRING
                 |    CHARACTER

exp-list         ::=  expression
                 |    exp-list  exp-list

identifier-list  ::=  IDENTIFIER
                 |    identifier-list  IDENTIFIER

identifiers      ::=  IDENTIFIER
                 |    identifiers IDENTIFIER
```

9. Unimplemented Features and Bugs

The MELD implementation is incomplete. A number of features have been discussed in the literature but are not present in the current implementation. This section outlines the missing features.

9.1. C Loops

All of the C loop statements should be added to MELD.

9.2. Type Specifier

A number of the sophisticated types are missing from *type-specifier*, including:

type-specifier ::= *basic-type*
| `RECORD BEGIN` *objdecl-part* `END`
| `SET-OF` *type-specifier* `KEY` *IDENTIFIER*
| `ORDERED-SET-OF` *type-specifier* `KEY` *IDENTIFIER*
`ORDERED` *direction* `BY` *IDENTIFIER*

9.3. Views

Views are not present:

assignment ::=
| *var* `:= VIEWS: IS-PROD (` *IDENTIFIER* `)`
| *var* `:= VIEWS: IS-PROD (` *IDENTIFIER* `)`
BOOLOP express

9.4. Union

The notion of union [Kaiser 87a; Kaiser 89a] is not present:

body-unit ::= *union*

union *GETS IDENTIFER := IDENTIFER * IDENTIFER*

9.5. Data Dependency Bug

```
BALANCE := BALANCE + CASH;
SEND "the balance is %d"(BALANCE) TO STDOUT;
```

There is a bug in the data dependency analysis. This example will output the old balance. The proper behavior may be obtained by using **printf** rather than SEND to do the output, *i.e.* `printf("the balance is %d", BALANCE);`.

9.6. FEATURE name

The `FEATURE` name may be a list of words, but if you include a reserved word in that list (such as `AND`) you get a very obtuse error message. Try naming a feature "Savings and Loan" and see what happens!

9.7. Comments

The comment characters are `{* comment *}`. `/* comment */` should also be allowed.

9.8. Object Class

There is no system defined type hierarchy, and no Object class.

9.9. Feature Names

Feature names consist of an arbitrary string of identifiers, but you cannot include a MELD reserved word in that list!

9.10. Type checking

Since there is no type checking on procedures, lint should be called as part of the MELD script.

9.11. Bug in string selectors

The selector:

```
     "Open the %s doors HAL"(what : STRING) -->
           printf("match 1 ’%s’\n",what);
```

does not match the string "Open the cargo doors HAL". This seems like a bug.

Appendix I
Running Meld

The procedure for compiling and linking a Meld program is simple. The command

```
/proj/meld/bin/meld foo.m
```

will compile the single file `foo.m` and link it with the runtime library, producing an `a.out` file. The command

```
/proj/meld/bin/meld -o foo foo.m
```

will work the same way, but will produce an executable file named `foo` rather than one named `a.out`. The command

```
/proj/meld/bin/meld foo.m bar.m
```

will compile both files and link them into a single `a.out` file. A switch may be added, as above, to change the name of the executable file. For true separate compilation, use the command

```
/proj/meld/bin/meld -c foo.m
```

This compiles `foo.m`, but without producing an `a.out` file. The switch is similar to the one in the C compiler that performs the analogous function. Compilation produces three files: `foo.o`, `foo.mo` (used by ml to build its tables), and `FooFeat.fo` (if the feature contained in foo.m is named FooFeat -- this file contains a listing of the feature's exports). If you have previously compiled `foo.m` and `bar.m` in this way, you should then use the command

```
/proj/meld/bin/ml foo.o bar.o
```

which will generate some tables necessary for linking separately compiled Meld files and link everything together, producing an `a.out` file. The name of the executable file may be changed, as in the compiler examples above, by using the `-o` switch.

There are other switches. The switch `-v` prints out messages as each compiler phase runs, and `-t` times each phase.

Appendix II
Bibliography

[Broy 85] Manfred Broy (editor).
Control flow and data flow: concepts of distributed programming.
Springer-Verlag, New York, 1985.

[Goldberg 85] Adele Goldberg.
Smalltalk-80.
Addison-Wesley, Reading, Mass., 1985.

[Hseush 88a] Wenwey Hseush and Gail E. Kaiser.
Concurrent Breakpointing.
Technical Report CUCS-402-88, Columbia University Department of Computer Science, October, 1988.

[Hseush 88b] Wenwey Hseush and Gail E. Kaiser.
Data Path Debugging: Data-Oriented Debugging for a Concurrent Programming Language.
In *ACM SIGPlan/SIGOps Workshop on Parallel and Distributed Debugging*, pages 236-246. Madison WI, May, 1988.
Special issue of *SIGPlan Notices*, 24(1), January 1989.

[Kaiser 87a] 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.

[Kaiser 87b] Gail E. Kaiser and David Garlan.
MELD: A Declarative Notation for Writing Methods.
In *6th Annual International Phoenix Conference on Computers and Communications*, pages 280-285. Scottsdale AZ, February, 1987.

[Kaiser 87c] Gail E. Kaiser and David Garlan.
Composing Software Systems from Reusable Building Blocks.
In *20th Annual Hawaii International Conference on System Sciences*, pages 536-545. Kona HI, January, 1987.

[Kaiser 87d] Gail E. Kaiser and David Garlan.
Melding Software Systems from Reusable Building Blocks.
IEEE Software :17-24, July, 1987.

[Kaiser 88] Gail E. Kaiser.
Concurrent MELD.
September, 1988
Presented at the Workshop on Object-Based Concurrent Programming. Available from the author.

[Kaiser 89a] Gail E. Kaiser and David Garlan.
Synthesizing Programming Environments from Reusable Features.
Software Reusability.
Addison-Wesley, Reading MA, 1989, pages 35-55, Chapter 2.

[Kaiser 89b] Gail E. Kaiser.
Object-Oriented Programming Language Facilities for Concurrency Control.
Technical Report CUCS-439-89, Columbia University Department of Computer Science, April, 1989.
Submitted for publication.

[Kaiser 89c] 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*, pages 147-166. Cambridge University Press, Nottingham, UK, July, 1989.

[Keene 85] Sonya E. Keene and David A. Moon.
Flavors: Object-oriented Programming on Symbolics Computers.
In *Common Lisp Conference*. December, 1985.

[Moon 86] David A. Moon.
Object-Oriented Programming with Flavors.
In *Object-Oriented Systems, Languages, and Applications Conference*, pages 1-8. Portland, OR, September, 1986.
Special issue of *SIGPlan Notices*, 21(11), November 1986.

[Popovich 88] Steven S. Popovich and Gail E. Kaiser.
MELDing Parallel and Distributed Programming.
Technical Report CUCS-402-88, Columbia University Department of Computer Science, October, 1988.

[Sharp 85] J. A. Sharp.
Data flow computing.
Halsted Press, New York, 1985.

[Stroustrup 86] Bjarne Stroustrup.
The C++ programming Language.
Addison-Wesley, Reading, Mass., 1986.

[Wadge 85] William W. Wadge and Edward A. Ashcroft.
Lucid, the dataflow programming language.
Academic Press, New York, 1985.

Index

Table of Contents