MELDing Multiple Granularities of Parallelism

GAIL E. KAISER, STEVEN S. POPOVICH, WENWEY HSEUSH AND SHYHTSUN FELIX WU

Columbia University

1. INTRODUCTION

Programming language design for parallel and/or distributed systems typically follows one of three approaches. The first is to parallelize and/or distribute conventional sequential languages such as Fortran, C and Lisp *via* a parallelizing compiler and remote procedure call. Although this may be the best approach for parallelizing existing software for a designated concurrent architecture, it has serious problems when the target architecture involves multiple grains of concurrency (*e.g.*, a network of cubes) and/or when writing new systems because it requires a hide-and-seek approach to parallelism — the programmer may be cognizant of the concurrency issues of the architecture, but writes in a sequential style effectively hiding this knowledge from compiler, which must then find it again. This is clearly unlikely to lead to high productivity, performance and reliability.

The second approach is to add concurrent constructs to what is fundamentally a sequential language, as was done for Ada, Argus [Liskov 87], Avalon [Herlihy and Wing 86] and Modula-2. Ada extended a Pascal-like sequential language with tasks and rendezvous; Argus added guardians and actions to CLU, and Avalon similarly extended C++ with servers and transactions; and Modula-2 added processes with shared variables and signals to a modularized Pascal. In all of these cases, the concurrency features are a relatively small addition to an otherwise sequential language. The third is to design a fundamentally concurrent language, as has been done for ConcurrentSmalltalk [Yokote 86], Emerald [Black 86], Mentat [Grimshaw 87] and SISAL [McGraw 85]. Both of these approaches in effect add a nice set of facilities for one style of concurrent programming while ignoring other styles. For example, Ada's rendezvous and Argus' guardian neatly match the large grain client/server model of distributed systems, but are not terribly useful for expressing fine grain parallelism. We are developing a concurrent programming language, MELD, that supports a range of concurrent styles by supporting multiple programming paradigms at multiple levels of granularity.

MELD integrates four paradigms:

- Object-oriented, with encapsulated classes, multiple inheritance and active objects. Object-oriented is the dominant paradigm.
- Macro dataflow, where the "macro" implies the dataflow is at the source code level — among statements — rather than at the machine level.
- *Module interconnection, with modular units called features that bundle*

together related classes and export classes for use by other features.

- Transaction processing, for fault tolerance and concurrency control within applications and among users.

These four paradigms are combined to support sound software engineering principles — encapsulation and reusability — for more robust software as well as several styles of concurrent programming.

In this paper, we present MELD's facilities for concurrent programming and emphasize how the object-oriented paradigm provides the appropriate framework for multiple granularities of parallelism. Our previous papers [Kaiser 87a; Kaiser 87b; Kaiser 87c; Kaiser 87d] focused on encapsulation and reusability issues, although a paper describing the MELD Debugger (MD) [Hseush 88] briefly introduced concurrent MELD.

First we give a brief overview of the language, ignoring concurrency issues. Then we describe how MELD integrates several granularities of parallelism, in particular, concurrency among users, among objects, among methods and among statements. We compare MELD to a number of other concurrent object-oriented programming systems, and briefly describe the implementation. We conclude by summarizing our contributions. The appendix gives a small demonstration program.

2. OVERVIEW OF MELD

MELD's four paradigms support three granularities of encapsulation and reusability: statements, classes and features.

Classes provide medium grain encapsulation. Each MELD class defines a number of instance variables, methods and *constraints* (explained later). Instance variables are strongly typed, where the type is a built-in class (`integer`, `string`, *etc.*), a built-in constructor (`array`, `sequence`, `set`, `table`), or a user-defined class. An initial value may be defined, optionally, for each instance variable as part of its declaration. New instances of classes are created by sending the `Create` message to the class (conceptually — MELD classes are not represented as objects); other methods are invoked by sending the corresponding message to the receiver object using the common notation "receiver.selector(arguments)". Methods take an arbitrary number of arguments, which are passed by reference (unique object ID), or for built-in types by value-result, and may optionally return a value; methods may define local variables using the same notation as for instance variables. Each user-defined class may specify zero or more superclasses. Rather than give the superclasses as part of the class definition, they are given separately in a *merge clause*; this is convenient for classes that are solely compositions of existing superclasses and add no new facilities of their own. As shown in Figure 1, the merge clause merges the definitions of one or more superclasses (`S`, `T`) into a subclass (`D`).

Statements provide fine grain encapsulation, smaller than methods. (MELD provides the standard kinds of statements: assignment, conditional, and so on; a statement can also be an arbitrary C subroutine call.) Statement-level encapsulation is one of the most unusual aspects of MELD. It is manifest in *constraints* and in the distinction between a *sequential block* and a *dataflow block*.

A constraint is an individual statement associated with a class but not part of any

```
/* class */
CLASS C ::=
    /* instance variables */
    a, b: integer;
    d: integer := 0;
    f: sequence of D;
    g: S := nil;

METHODS:
    constraint
    method

END CLASS

/* merge clauses */
MERGE S, T AS D;
MERGE U AS C;
```

Figure 1: Class

method. A constraint typically defines a relationship among instance variables that must always hold. Each constraint has zero or more input instance variables and zero or more output instance variables. When one or more of the inputs changes in value, the constraint statement is re-executed to restore consistency with the outputs. Thus MELD's constraints are unidirectional rather than bidirectional, unlike most other "constraint" systems [Leler 88], and permit objects to behave as active values [Stefik 86]. Those constraints with no inputs are executed only when a new object is created, as initialization; a constraint with no outputs may print a message or perform some other action. An example is illustrated in Figure 2. Whenever o executes, it recomputes the value of b. Whenever the new value is different from the old value, this triggers the execution of the constraint to recompute the value of a.

```
CLASS C ::= a, b, d: t;

METHODS:
    /* constraint */
    a := b.F();          /* 2nd */

    /* method */
    o (c, e: t) -->
    [ b := c.G(); ]      /* 1st */

END CLASS
```

Figure 2: Constraint

Constraints may take other forms than the simple assignment statement; another useful form is the *conditional constraint*, which is a conditional (**if**) statement that is not part of any method. In this case, whenever an variable in the conditional expression changes, the condition is re-evaluated and the appropriate branch of the conditional statement is executed. In the commonest usage of conditional constraints, only one branch of the

statement is specified; this form of constraint allows detection of, and response to, exceptional conditions in the data.

A sequential block is a compound statement: a list of statements, which is executed in exactly the sequence given. Figure 3 gives an example.

```
CLASS C ::= a, b, d: t;

METHODS:

    O (c, e: t) -->
    [ a := b.F();       /* 1st */
      b := c.G();       /* 2nd */
      d := e.H(); ]     /* 3rd */

END CLASS
```

Figure 3: Sequential Block

A dataflow block is a list of statements, which is executed in *data-dependency order* rather than in the sequence given. A dataflow block is enclosed within curly braces "{}" and a sequential block within square brackets "[]". Each statement in a dataflow block has zero or more inputs and zero or more outputs. The statements are treated as simultaneous equations, except that each particular statement is executed only after all its inputs (if any) have reached their final values. Figure 4 gives an example. Method O consists of three statements executed in dataflow order. In particular, "b := c.G()" must be executed before "a := b.F()" since b is the output of the former and the input of the latter. "d := e.H()" can be executed before, in between, or after these two statements since there is no data dependency. Circularities among statements in a dataflow block are detected by the MELD compiler. The only exception is that circularities within a single statement, such as "a := a + 1", are permitted; a is treated as an output but not an input for the purposes of dataflow.

```
CLASS C ::= a, b, d: t;

METHODS:

    O (c, e: t) -->
    { a := b.F();       /* 2nd */
      b := c.G();       /* 1st */
      d := e.H(); }     /* any time */

END CLASS
```

Figure 4: Dataflow Block

We developed the dataflow block as a solution to the "multiple inheritance" problem. When a class inherits from multiple superclasses, it may inherit methods with the same name from more than one ancestor. There are three standard approaches to solving this problem that appear in other object-oriented languages. One is to require the class to

explicitly choose one of the inherited methods. Another is to flatten the ancestor graph into a precedence list (for example, left to right depth first up to joins) and choose the first or last occurrence. The third is to execute all occurrences of the method, in the order given by a precedence list. In MELD, all the inherited methods are executed, but interleaved in the order implied by the dataflow dependencies among the statements in the methods; this is why we claim statements as a level of encapsulation. This makes most sense when there are semantic assumptions among the methods of a superclass, that is, when those methods depend upon one another's side effects, which is almost always the case. It is not correct to override some but not all of these interdependent methods due to the (often incidental) precedence ordering; the resulting violation of assumptions can cause other methods, not overridden, to behave improperly. (The programmer can, however, change this behavior if necessary; MELD achieves many of the goals of the other schemes with a complicated but very flexible system of attaching `default`, `insist` and `override` keywords to inherited and overriding methods.)

```
CLASS S1 ::= a, b: t;
METHODS:
    O () -->
    { a := b.F(); }      /* 2nd */
END CLASS

CLASS S2 ::= d: t;
METHODS:
    O (w: t) -->
    { d := w.J(); }      /* any time */
END CLASS

MERGE S1, S2 AS C;      /* C inherits from S1 and S2 */

CLASS C ::= a, b, d: t;
METHODS:
    O (c, e: t) -->
    { b := c.G();        /* 1st */
      d := e.H(); }      /* illegal */
END CLASS
```

Figure 5: Multiple Inheritance

Method interleaving is illustrated in Figure 5. Class C inherits from S1 and S2, since they are specified as its superclasses by the merge clause. The O methods from each of these three are interleaved in dataflow order. Combining these methods, each with a different number of parameters, is legal because the subclass C simply specializes its superclass S2's method by adding an additional parameter; the parameters w in S2 and c in C are two equivalent names for the first parameter passed to the O method. S2's method, declared with only one parameter, ignores the second parameter, and the code inherited from S1, whose O method takes no parameters, ignores both parameters passed to the combined method.

Ignoring the second statement of the O method from C, b is computed from argument c and then a from the new value of b. d would be computed from w if it were not for the illegal conflict with the separate assignment of d in C's method for O. This would be

detected as a fatal error at compile-time.

This solution to the multiple inheritance problem permits every class to be written independently in most cases, since only the set of instance variables updated by each method must be made public. Other languages either require subclasses to be cognizant of the details of their ancestors or potential ancestors of the same subclass to be aware of each other's implementation details. Since many possible interactions are invariably overlooked, the resulting software is likely to be unreliable and difficult to debug.

Dataflow blocks may be nested inside sequential blocks and *vice versa*, and both dataflow and sequential blocks may trigger constraints. The intricacies of interactions between constraints, dataflow blocks and sequential blocks are discussed in our previous papers.

Features provide large grain encapsulation. Classes are the standard unit of reusability in object-oriented languages, but we believe classes are too small — the cost of retrieving and adapting the reusable unit is likely to be larger than the cost of developing it from scratch. A practical reusable unit consists of related classes bundled together to provide a coherent functionality. MELD's feature construct is a modular unit with a public interface consisting of an *export clause*, an *import clause*, a *remotes clause* (our rather simplistic means for naming remote objects, not discussed in this paper), and a private implementation. The export clause lists those classes defined in the implementation that may be used externally for types of instance variables or as superclasses. The import clause lists those features whose exported classes may be used internally. The implementation part consists of global object definitions, classes and merge clauses.

MELD provides additional facilities to export a particular *view* [Garlan 86] of each class, which defines which instance variables and methods of each exported class are visible to external subclasses and clients. Some other object-oriented languages provide similar facilities [Schaffert 86]. MELD also supports *generic features* analogous to Ada's generic packages. This support for reusability is explained in our previous papers. An example feature is shown in Figure 6.

3. INTEGRATING PARALLEL AND DISTRIBUTED PROGRAMMING

MELD's multiple paradigms also lead to three granularities of concurrency:

- Macro dataflow for fine grain concurrency. Two statements in the same dataflow block are ordered when an output of one statement is among the inputs of another; otherwise, they are unordered and may execute in parallel. When multiple methods execute simultaneously in the same object, concurrency is maximized at the cost of nondeterminism. *Atomic blocks* solve this problem by enforcing critical sections with respect to the object; atomic blocks may be either dataflow or sequential.
- Objects for medium grain concurrency. MELD supports both synchronous and asynchronous message passing among local or remote objects.
- Transactions for large grain concurrency. Transactions appear to execute atomically and in serial order with respect to other transactions; their data cannot be corrupted by other MELD code executing concurrently. They are much more powerful than atomic blocks, since they can cut across methods

```
FEATURE F

INTERFACE:

    EXPORTS C;

    IMPORTS G, H;

    remotes

IMPLEMENTATION:

/* global variables */
OBJECT:
    a: array[lb..ub] of X;
    b: string := "xyzzy";

/* classes and merges */
CLASS C ...

MERGE S, T AS D;

END FEATURE
```

Figure 6: Feature

and objects.

A MELD method may be invoked synchronously or asynchronously. In the synchronous case, the caller waits for return with respect to its own thread of control; this uses the standard notation "receiver.selector(arguments)". In the asynchronous case, the caller continues and the invocation creates a new thread of control; this uses the distinct notation "SEND selector(arguments) TO receiver". MELD programs may involve an arbitrary number of threads, which may be created dynamically during program execution. Several threads may operate within the same address space and one thread may operate across multiple address spaces, in the sense that the local thread suspends while until the new remote thread returns control. In either case, the invoked method runs concurrently with any other methods currently active on the same object. These other methods may be reading and writing the same instance variables. The default synchronization among such methods is done by dataflow, as within a single method as described in the previous section.

There is a serious problem with this approach. The program illustrated in Figure 7 operates as indicated in the comments if M and N happen to begin execution at exactly the same time, due to "simultaneous" arrival of messages M and N from other objects. d is computed from the value of c given as argument to N, b is computed from the new value of d, and a from the new value of b. However, if message N arrives a bit later than M, then b is computed from the old value of d, and then the new value of d is computed from the argument c and a is computed from the new value of b. In this case, both statements in N could be executed concurrently since there is no dataflow between them. On the other hand, if message M arrives a bit later than N, then b may be computed from either the

```
CLASS D ::= a, b, d: t;

METHODS:
    M () -->
    { b := d.G(); }     /* 2nd */

    N (c: t) -->
    { a := b.F();       /* 3rd */
      d := c.H(); }     /* 1st */

END CLASS
```

Figure 7: Concurrent Methods

old or new value of d and a may be computed from either the old or new value of b, depending on race conditions. This may be rather bewildering for the programmer, since it is necessary that the resulting computation be deemed 'correct' in all of these cases.

```
CLASS D ::= a, b, d: t;

METHODS:
    M () -->
    [ b := d.G(); ]     /* 1st */

    N (c: t) -->
    [ a := b.F();       /* 2nd */
      d := c.H(); ]     /* 3rd */

END CLASS
```

Figure 8: Concurrent Methods with Sequential Blocks

Sequential blocks remove concurrency within methods, making them easier to write without the need for the single-assignment mindset, but do not affect concurrency among methods. The comments in Figure 8 indicate the ordering if M and N happen to start at the same time. b is computed from the old value of d, a from the new value of b and then the new value of d from the argument c. But if there is a race condition — which is more likely than not — a may see the new value of b or b may see the new value of d, but not both.

Figure 9 illustrates an atomic block, where method O executes atomically with respect to the receiver object. Atomic blocks are indicated with parentheses "()". Both b and d are updated, and only after O terminates is the constraint triggered. Actually, the two statements in O may themselves be executed in either sequential or dataflow order, since it is necessary to include them within an inner sequential or dataflow block, not shown, if the atomic block is used. In this case, of course, it does not matter.

Methods M and N in Figure 10 are both atomic blocks, so they execute in the serial order determined by which happens to begin first. MELD implements atomic blocks by locking the entire object at the computational grain size of individual blocks. One problem with this is that the critical section cannot begin in one method and end in another, which is

```
CLASS C ::= a, b, d: t;

METHODS:
    a := b.F();          /* 2nd */

    O (c, e: t) -->
    ( b := c.G();        /* 1st */
      d := e.H(); )

END CLASS
```

Figure 9: Atomic Block

```
CLASS D ::= a, b, d: t;

METHODS:
    M () -->
    ( b := d.G(); )    /* 1st or 2nd */

    N (c: t) -->
    ( a := b.F();      /* 1st or 2nd */
      d := c.H(); )

END CLASS
```

Figure 10: Concurrent Methods with Atomic Blocks

often bad programming practice but sometimes is necessary.

```
CLASS E ::= c: integer;

METHODS:
    P (a, b: integer) -->
    { c := a + b;
      return c; }

    Q (x: integer) -->
    { c := 2 * x; }

END CLASS

CLASS F ::= d: E;
            e: integer;

METHODS:
    R () -->
    ( e := d.P(1, 2); )

END CLASS
```

Figure 11: Problem with Atomic Blocks

Atomic blocks have one serious flaw. Consider the example in Figure 11, where R is an

atomic block but P is not, and R invokes P synchronously. The programmer presumably expects that e will be assigned to 3, but this need not be the case. Consider the case where Q is invoked immediately after "c := a + b" executes, but before "return c" executes. Just because R is atomic does not imply anything about the other methods it invokes!

```
CLASS E ::= c: integer;

METHODS:
    P (a, b: integer) -->
    { c := a + b;
      return c; }

    Q (x: integer) -->
    { c := 2 * x; }

END CLASS

CLASS F ::= d: E;
            e: integer;

METHODS:
    R () -->
    < e := d.P(1, 2); >

END CLASS
```

Figure 12: Transaction Block

The solution is serializable transactions. Transaction blocks are indicated with angle brackets "<>", but transactions may cut across methods using `begin-transaction`, `commit-transaction` and `abort-transaction` statements. Transactions can be arbitrarily nested, and can enclose arbitrarily nested dataflow, sequential and atomic blocks. The problematic example is repeated in Figure 12, where now it is guaranteed that e will be set to 3. Since MELD supports distributed transactions on arbitrary sets of objects, this behavior is guaranteed even if the objects of classes E and F are located on separate processors.

4. RELATED WORK

Many systems provide a subset of the facilities provided by MELD, but we know of only one whose range of concurrent programming styles is comparable to that of MELD. Like MELD, SR [Andrews 88] is based on a small number of underlying concepts, but supports a wide range of facilities including local and remote procedure call, rendezvous, dynamic process creation, asynchronous message passing, multicast and semaphores. MELD supports all these facilities and more, for example, multiple threads within the same object.

MELD's dataflow blocks support finer parallelism than other macro dataflow languages. For example, Mentat [Grimshaw 87] supports dataflow among the inputs and outputs of actors, which correspond to MELD's methods. SISAL [McGraw 85] applies macro

dataflow at the statement level, but differs substantially from MELD in being applicative and not object-oriented.

Concurrent object-oriented languages typically provide synchronous or asynchronous message passing, but not both. Eden [Almes 85] and ConcurrentSmalltalk [Yokote 86] do provide both options.

MELD's constraints permit active objects, but in an essentially passive manner. Most other active objects are like those of Emerald [Black 86], where there is a process continually running within each object. This process is written as sequential code and can carry out arbitrary activities — but in practice does little except wait for messages to arrive. Another difference between MELD and Emerald is the latter's type system is built on the notion of type conformance — only interfaces, not implementations, can be shared between classes. In contrast, MELD provides the standard form of implementation hierarchy — the only novel aspect of inheritance is the interleaving of multiply inherited methods.

MELD is like Argus [Liskov 87] in that it executes each incoming message to an object in a separate thread. Meld objects, however, are much smaller than typical Argus guardians. Guardians are servers, with one multi-threaded guardian per heavyweight process; smaller objects within the guardian are implemented as CLU clusters. In contrast, MELD would implement a server as one or more features, each consisting of many objects, within a common address space. MELD uses the same object model to represent the interface object (not discussed in this paper) corresponding to the Argus guardian as it does to represent the smaller objects corresponding to the clusters within the guardian, and so is likely to execute many more parallel threads than the equivalent guardian. MELD also differs substantially from Argus in concurrency control: where Argus has atomic and non-atomic *objects*, binding concurrency control to one level of implementation, MELD allows "free-form" concurrency control at any level and gives the programmer the flexibility to combine atomic and non-atomic actions on the same object.

Although Clouds [Dasgupta 88] is an operating system rather than a programming language, there are many similarities between Clouds and MELD. Clouds also uses a single object model for all granularities of objects. Its concurrency control is not bound to the object level, and atomic and non-atomic operations on an object may be mixed. Clouds objects, however, are completely passive; there is no facility corresponding to MELD's constraints. The Clouds object model is also compromised by processes, which are not associated with or "inside" any object. In MELD, all executable code is declared either as a method of some object class or as a constraint.

Mach [Jones 86], another operating system, also provides many facilities useful in the construction of distributed programming systems: ports to represent objects, messages for communication between objects, remote procedure calls, and multiple threads of execution within a task. A distributed transaction facility, Camelot [Spector et al. 86], has been implemented under Mach; Avalon [Detlefs 88] provides language support for Camelot. The Avalon model of concurrency control was heavily influenced by Argus, and *also binds atomicity to the object: each class may be a subclass of resilient, atomic or dynamic, which are themselves in a linear hierarchy. Executable code is written in C++,*

and parallelism is available only if the C++ code explicitly starts up multiple threads. In MELD, parallelism is always available, reduced only when necessary by the particular synchronization facilities employed.

Hybrid [Nierstrasz 87] uses a single granularity of encapsulation, the object (type) level. Its concurrency facilities are less flexible than MELD's. Hybrid allows only a single thread of control within a domain, as opposed to MELD's multiple threads within each object. Hybrid binds thread creation to particular operations designated as "reflexes"; other operation invocations cannot start a new thread of control. Hybrid has an atomic block construct which provides atomicity across multiple objects, but blocks other code from executing within any of those objects until the atomic block commits or aborts. MELD's atomic blocks work similarly, but on single objects only; MELD's transactions provide atomicity across multiple objects, but permit much more concurrency.

Most other object-based systems provide only a single form of concurrency control; for example, Coral3 [Merrow 87] uses only two-phase locking, while GemStone [Maier 86] uses only an optimistic approach.

5. IMPLEMENTATION STATUS

The first implementation of MELD was completed in March 1988. This implementation was missing several vital features, notably inheritance, which has been added since *via* a preprocessor, and a reasonable distributed name service, which is in progress (we're also working on a design for a general query language). This mainline version supports only atomic methods, not serializable transactions; a separate version of MELD with transactions diverged in early 1988 and is now being integrated. Currently, both atomic methods and transactions are implemented purely for concurrency control, and crash recovery is not supported.

MELD is translated into C. Both the translator and run-time support are written in C and run on Berkeley Unix, using sockets for interprocess communication. The current implementations simulate the multi-threaded dataflow parallelism within a heavyweight process in order to share the address space, but objects may be distributed across a network of Sun, MicroVAX, and IBM RT workstations[1]. Eventually, we plan to reimplement MELD on Mach, taking advantage of Mach's abilities to have multiple threads of control sharing the same address space rather than simulating this in the runtime system, and to allow true parallelism by running the separate threads of a task on separate processors of a multiprocessor machine.

6. CONCLUSION

In our initial work on MELD, we took one simple idea — encapsulation — and applied it at several granularities within an object-oriented programming framework. The result addressed two important problems: multiple inheritance and scale of reusable components. More recently, we took another simple idea — a block of statements as the unit of synchronization policy — and again applied it at several granularities. The result was

[1] *The RT version has not been adequately tested.*

support for a wide range of concurrent programming styles.

ACKNOWLEDGMENTS

David Garlan and Gail Kaiser jointly developed the original, non-concurrent design of MELD. Nicholas Christopher, Jeffrey Gononsky, Nanda S. Kirpekar, Marcelo Nobrega, David Staub, Seth Strump, Kok-Yong Tan, and Jun-Shik Whang also contributed to the implementation effort. An earlier version of this paper was handed out at the *Workshop on Object-Based Concurrent Programming*, San Diego CA, September 1988. Kaiser's Programming Systems group is supported by National Science Foundation grants CCR-8858029 and CCR-8802741, by grants from AT&T, DEC, IBM, Siemens, Sun and Xerox, by the Center for Advanced Technology and by the Center for Telecommunications Research.

REFERENCES

[Almes 85] Guy T. Almes, Andrew P. Black, Edward D. Lazowska, and Jerre D. Noe.
The Eden System: A Technical Review.
IEEE Transactions on Software Engineering SE-11(1):43-59, January, 1985.

[Andrews 88] Gregory R. Andrews, Ronal A. Olsson, Michael Coffin, Irving Elshoff, Kelvin Nilsen, Titus Purdin and Gregg Townsend.
An Overview of the SR Language and Implementation.
ACM Transactions on Programming Languages and Systems 10(1):51-86, January, 1988.

[Black 86] Andrew Black, Norman Hutchinson, Eril Jul and Henry Levy.
Object Structure in the Emerald System.
In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 78-86. Portland, OR, September, 1986.
Special issue of *SIGPlan Notices*, 21(11), November 1986.

[Dasgupta 88] Partha Dasgupta, Richard J. Leblanc Jr. and William F. Appelbe.
The Clouds Distributed Operating System: Functional Description, Implementation Details and Related Work.
In *8th International Conference on Distributed Computing Systems*, pages 2-9. San Jose CA, June, 1988.

[Detlefs 88] David Detlefs, Maurice Herlihy and Jeannette Wing.
Inheritance of Synchronization and Recovery Properties in Avalon/C++.
Computer :57-69, December, 1988.

[Garlan 86] David Garlan.
Views for Tools in Integrated Environments.
In Reidar Conradi, Tor M. Didriksen and Dag H. Wanvik (editors), *Lecture Notes in Computer Science*. Volume 244: *Advanced Programming Environments*, pages 314-343. Springer-Verlag, *Berlin, 1986.*

[Grimshaw 87] Andrew S. Grimshaw and Jane W.S. Liu.
Mentat: An Object-Oriented Macro Data Flow System.
In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 35-47. Orlando FL, October, 1987.
Special issue of *SIGPlan Notices*, 22(12), December 1987.

[Herlihy and Wing 86] M. P. Herlihy, J. M. Wing.
Avalon: Language Support for Reliable Distributed Systems.
Technical Report CMU-CS-86-167, Carnegie Mellon University, November, 1986.

[Hseush 88] 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.

[Jones 86] Michael B. Jones and Richard F. Rashid.
Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems.
In *Object-Oriented Programming Systems, Languages and Applications Conference*, pages 67-77. Portland, OR, September, 1986.
Special issue of *SIGPlan Notices*, 21(11), November 1986.

[Kaiser 87a] 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 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.
Melding Software Systems from Reusable Building Blocks.
IEEE Software :17-24, July, 1987.

[Kaiser 87d] 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.

[Leler 88] Wm Leler.
Constraint Programming Languages Their Specification and Generation.
Addison-Wesley Pub. Co., Reading MA, 1988.

[Liskov 87] Barbara Liskov, Dorothy Curtis, Paul Johnson, and Robert Scheifler.
Implementation of Argus.
In 11th ACM Symposium on Operating Systems Principles, pages 111-122. Austin TX, November, 1987.
Special issue of Operating Systems Review, 21(5), 1987.

[Maier 86] David Maier, Jacob Stein, Allen Otis, and Alan Purdy.
Development of an Object-Oriented DBMS.
In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 472-482. October, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

[McGraw 85] James McGraw, S. K. Skedzielewski, Stephen Allan, Rod Oldehoeft, John Glauert, Chris Kirkham, Bill Noyce, and Robert Thomas.
SISAL: Streams and Iteration in a Single Assignment Language: Reference Manual Version 1.2
Manual M-146, Rev. 1 edition, Lawrence Livermore National Laboratory, Livermore, CA, 1985.

[Merrow 87] Thomas Merrow and Jane Laursen.
A Pragmatic System for Shared Persistent Objects.
In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 103-110. Orlando FL, October, 1987.
Special issue of *SIGPlan Notices*, 22(12), December 1987.

[Nierstrasz 87] O. M. Nierstrasz.
Active Objects in Hybrid.
In *Object-Oriented Programming Systems, Languages and Applications Conference Proceedings*, pages 243-253. Orlando FL, October, 1987.
Special issue of *SIGPlan Notices*, 22(12), December 1987.

[Schaffert 86] Craig Schaffert, Topher Cooper, Bruce Bullis, Mike Kilian and Carrie Wilpolt.
An Introduction to Trellis/Owl.
In *Object-Oriented Systems, Languages, and Applications Conference*, pages 9-16. Portland, OR, September, 1986.
Special issue of *SIGPlan Notices*, 21(11), November 1986.

[Spector et al. 86] Alfred Z. Spector, Joshua J. Bloch, Dean S. Daniels, Richard P. Draves, Dan Duchamp, Jeffrey L. Eppinger, Sherri G. Menees, Dean S. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.
Also available as Technical Report CMU-CS-86-166, Carnegie-Mellon University, November 1986.

[Stefik 86] Mark Stefik and Daniel G. Bobrow.
Object-Oriented Programming: Themes and Variations.
AI Magazine 6(4):40-62, Winter, 1986.

[Yokote 86] Yasuhiko Yokote and Mario Tokoro.
The Design and Implementation of ConcurrentSmalltalk.
In *Object-Oriented Programming Systems, Languages, and Applications Conference*, pages 331-340. October, 1986.
Special issue of *SIGPLAN Notices*, 21(11), November 1986.

I. EXAMPLE

This program is intended for two Sun 3 workstations sitting side by side, each displaying a single large window. There is a "hand" in each window and they play catch by tossing a "box" back and forth, apparently from screen to screen. A square box rather than a round ball is used for simplicity in graphics hacking. The windows, hands and box are objects and all interactions are done by message passing. Another copy of the toss feature currently must be compiled separately with the roles of box1, hand1, window1 and box2, hand2 and window2 reversed, for the two different machines. $WORKSTATION is a place holder which, in an actual program, would be replaced by the name of the "other" machine; this location information is temporarily required due to the lack of a distributed name service.

```
FEATURE toss

INTERFACE:
        EXPORTS box, window
        REMOTES box1 : box at $WORKSTATION
                hand1 : hand at $WORKSTATION
                window1 : window at $WORKSTATION

IMPLEMENTATION:

OBJECT:
        box2: box;
        hand2: hand;
        window2: window;
        gravity : double := 0.12;
        get_number() : double;


CLASS box ::=
        active : integer := 0;
        center_x : double := 2028.0;
        center_y : double := 300.0;
        speed_x : double := 0.0;
        speed_y : double := 0.0;
        status : integer := 0;

METHODS:

{* set up the path between standard input and this object. *)

path(0, $self);

if(active = 1) then status := window2.box(center_x, center_y, 1);.

"help"--> (
  printf("\nBOX::\n  (1) x speed = d (between 1.0 and 100.0)\n
         (2) y speed = d (between -20.0 and 20.0)\n\n");
  printf("Box position : (x = %lf, y = %lf)\nBox speed  :
         (x = %lf, y = %lf)\n\n", center_x, center_y, speed_x, speed_y);
)

impel(x, y: double)-->(
 if(speed_x = 0.0 && speed_y = 0.0 && (x != 0.0 || y !=0.0)) then
```

```
    send move() to $self;
  speed_x := x;
  speed_y := y;
)

set_x_y(x, y :double)-->(center_x := x; center_y := y;)

move()-->
(
  hand2.light(center_x, center_y);
  if(speed_x != 0.0 || speed_y != 0.0) then [
    speed_y +:= gravity;
    ( center_x +:= speed_x; center_y +:= speed_y; )
    if(status = 1) then send move() to $self;
    else (
      if(status = 0) then (
        speed_y := 0.0; speed_x := 0.0;
      )
      else (
        if(box1.activate(center_x, center_y, speed_x, speed_y) = 1)
        then (
          active := 0;
          window2.box(center_x, center_y, 0);
{*          send box(center_x, center_y, 0) to window2;*)
        )
      )
    )
  ]
)

activate(cx, cy, x, y: double)--> (
  if(active = 1) then return(0);
  else (
    if(window2.box(cx, cy, 1) = 1) then (
      center_x := cx;
      center_y := cy;
      active := 1;
      speed_x := x;
      speed_y := y;
      if(speed_x != 0.0 || speed_y != 0.0) then send move() to $self;
      return(1);
    )
    else (
      printf("activate: box is not inside the window\n");
      return(0);
    )
  )
)
END CLASS box

CLASS window ::=
        left : integer := 1050;
        right : integer := 2100;
        up : integer := 50;
        down : integer := 850 ;
        open_side : integer := -1;

METHODS:
  fb_open();
  path(0, $self);
```

```
"help"-->
    printf("window = (left = %d, right = %d, up = %d, down = %d)\n",
            left, right, up, down);

box(x, y : double; flag : integer)-->{
  box(x - left, y, flag);
  return($self.inside(x, y));
}

hand(x, y : double)-->{
  if(open_side = 1) then right_hand(x - left, y);
  else left_hand(x - left, y);
}

inside(x, y : double)-->{
  if(x >= left && x <= right && y >= up && y <= down) then return(1);
  else {
    if(open_side = 1 && x > right) then return(-1);
    else {
      if(open_side = -1 && x < left) then return(-1);
      else return(0);
    }
  }
}

END CLASS window

CLASS hand ::=
        center_x : double := 2029.0;
        center_y : double := 300.0;
        speed_x : double := -35.0;
        speed_y : double := -2.0;
        dx : double;
        dy : double;
        y1 : double;
        t : double;
        range : double := 20;
        status : integer := 0; {* 1 = grasp; 2 = toss; 0 = not both *}
METHODS:

  path(0, $self);
  send hand(center_x, center_y) to window2;

"help"-->{
  printf("Hand position : (x = %lf, y = %lf)\n\n  (1) hand x = d\n
          (2) hand y = d\n  (3) toss\n", center_x, center_y);
}

"x*speed*=*\n"--> {
  speed_x := get_number($selector);
  printf("x speed = %lf\n", speed_x);
}

"y*speed*=*\n"--> {
  speed_y := get_number($selector);
  printf("y speed = %lf\n", speed_y);
}

"hand*x*=*\n"-->{
```

```
  center_x := get_number($selector);
  printf("hand x = %lf\n", center_x);
}

"hand*y*=*\n"-->{
  center_y := get_number($selector);
  printf("hand y = %lf\n", center_y);
}

"init"-->{
    send set_x_y(center_x - 1.0, center_y) to box2;
    send impel((double)0.0, (double)0.0) to box2;
    status := 1;
}

"toss"-->{
  if(status = 1) then {
    send set_x_y(center_x - 1.0, center_y) to box2;
    send impel(speed_x, speed_y) to box2;
    status := 2; {* toss *}
    send signal(center_x - 1.0, center_y, speed_x, speed_y) to hand1;
  }
}

light(x, y: double)-->{
  if(x < center_x + range && x > center_x - range &&
     y < center_y + range && y > center_y - range) then {
    if(status = 2) then return(0);
    else {
      status := 1; {* grasp *}
      box2.impel((double)0.0, (double)0.0);
      printf("I catch the box\n");
      return(1);
    }
  }
  else {
    status := 0;
    return(0);
  }
}

signal(x, y, sx, sy : double)-->{
  printf("signal (%lf, %lf, %lf, %lf)\n", x, y, sx, sy);
  t := (center_x - x)/sx;
  if(t > 0) then {
    y1 := y + sy * t + gravity * t * t / 2.0;
    if(window2.inside(center_x, y1) = 1) then {
      send move_to(center_x, y1) to $self;
    }
    else printf("I am not going to catch the box\n");
  }
}

move_to(x, y : double)-->[
  if(center_x + 10 < x) then dx := 10;
  else dx := x - center_x;
  if(center_x - 10 > x) then dx := -10;
  else dx := x - center_x;
 if(center_y + 6 < y) then dy := 10;
 else dy := y - center_y;
```

```
  if(center_y - 6 > y) then dy := -10;
  else dy := y - center_y;
  if(dx != 0 || dy != 0) then [
    center_x +:= dx;
    center_y +:= dy;
    send move_to(x, y) to $self;
  ]
]

END CLASS hand

END FEATURE toss
```