MELD: A Multi-Paradigm Language with Objects, Dataflow and Modules

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027

David Garlan
Tektronix, Inc.
Computer Research Laboratory
Beaverton, OR 97077

December 1987

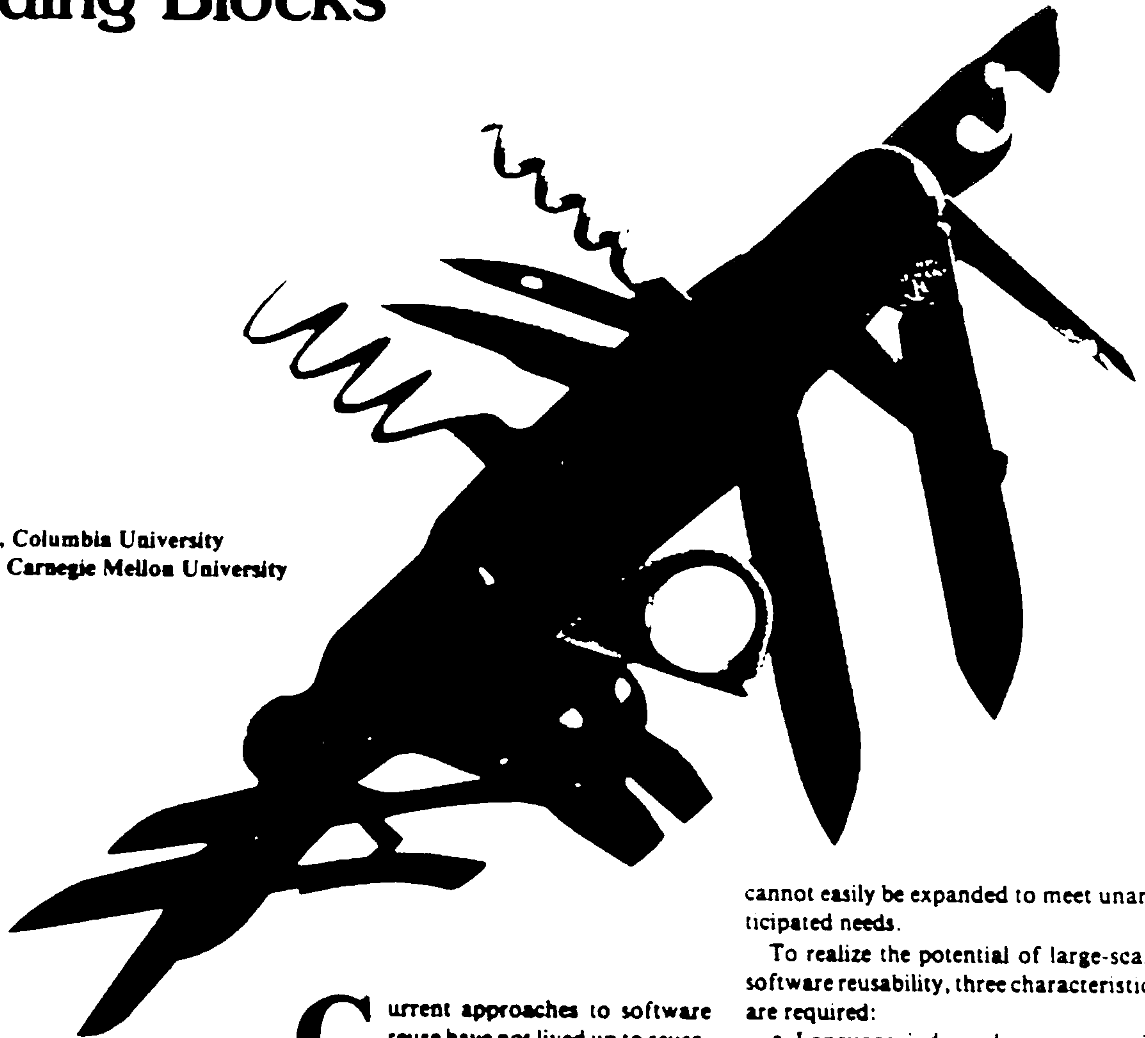
CUCS-281-87

Abstract

This technical report consists of the two most recent papers from the MELD project. *Melding Software Systems from Reusable Building Blocks* describes MELD as a declarative language that combines facilities from the package library, software generation and object-oriented language approaches to reusability. *MELDing Data Flow and Object-Oriented Programming* emphasizes the multi-paradigm nature of MELD and introduces overriding of inherited facilities and generic features, and discusses compile-time error detection.

Prof. Kaiser is supported in part by grants from AT&T Foundation, IBM, Siemens Research and Technology Laboratories, and the New York State Center of Advanced Technology — Computer & Information Systems, and in part by a Digital Equipment Corporation Faculty Award. When this research was conducted, Dr. Garlan was supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and in part by ZTI-SOF of Siemens Corporation, Munich, Germany.

Melding Software Systems from Reusable Building Blocks



Gail E. Kaiser, Columbia University
David Garlan, Carnegie Mellon University

This declarative language takes the best features from the three most popular reusability approaches, but overcomes their flaws. It supports language independence, component composition, and tailoring.

Current approaches to software reuse have not lived up to reusability's potential to dramatically improve software productivity and maintainability.

The shortcomings of the three most popular approaches are discussed in the box on p. 15. The primary deficiency of these approaches is that they tightly couple reusable components to their original context. Either the reusable components are written in one language and cannot easily be reused in another language or they implement a specific abstract data type and/or function and cannot easily be converted into the subtly different requirements of another application — or even the original application when it is repaired or enhanced. Furthermore, the options for tailoring a reusable component are determined when the component is written and cannot easily be expanded to meet unanticipated needs.

To realize the potential of large-scale software reusability, three characteristics are required:

- Language independence, to avoid early implementation decisions. Of course, every notation is a "language," but it is best to avoid concrete representations and control structures until actual reuse, when the most efficient solutions can be chosen.
- Composition of components, to build large software systems to meet complex requirements. Procedural invocation of separately developed components works to some extent, but it does not promote reuse of large-scale components.
- Flexibility, to expand reusability beyond the capabilities and applications anticipated by the original implementor. Most approaches require that the complete set of tailoring options be defined when the reusable component is created. These options are not always sufficient.

0740-7459/87/0700/0017/$01.00 ©1987 IEEE

We have developed a new approach to reusable software that has these three characteristics. The basis of this approach is Meld,* a declarative language that can be translated to an efficient implementation in a conventional programming language.

An entire software system can be written in Meld and then translated to the desired programming language. System maintenance is done with the Meld representation rather than the implementation language. Or an individual component can be written and maintained in Meld, and its translation integrated into an application written in a conventional language. Thus, old code can be combined with components written in Meld.

Overview of Meld

Meld is an object-oriented language in that it supports the encapsulation of data and operations as objects defined by classes and the inheritance of separately defined data and operations from ancestor classes. In the context of object-oriented languages, such data are called instance variables and such operations are methods.

But Meld has two essential aspects that set it apart from other object-oriented languages, *features* and *action equations*.

Features. Features are reusable building blocks. Like Ada packages, their interfaces are separate from the implementation. A feature bundles a collection of interrelated classes in its implementation. The interface exports a subset of these classes and a subset of their methods.

In effect, a feature is a reusable unit larger than a subroutine, on approximately the same scale as an Ada package, that permits the reuse of the glue among subroutines (methods) and, in fact, among abstract data types (classes). A generic feature is similar to a generic package because the interface can be instantiated according to the requirements of the desired application.

Moreover, unlike Ada packages, features have the flexibility of classes. A feature's interface imports a group of features, making the classes exported by these features available within its implementation. There, an imported class can be merged, either with a locally defined class or with another imported class.

Merging has the effect of inheritance: One class inherits (reuses) all the facilities already defined for another class and can augment and replace any subset of these facilities.

Action equations. Unlike classes, however, features can combine, as well as augment and replace, inherited methods. This is accomplished by writing methods as action equations. Action equations should not be confused with mathematical equations. They were developed to extend attribute grammars for semantics processing of programming environments.[1]

Action equations define (1) the relationships that must hold among objects and among parts of objects and (2) the dynamic interaction among objects and between objects and external agents (such as users, the operating system, and utilities).

When classes merge, separately inherited methods can be combined by assigning them the same selector (the same method name). Because methods define relationships and dynamic interactions in the abstract — as action equations — rather than by a sequence of statements, the behavior of the methods is combined implicitly. The resulting composite behavior does not depend on any sequential ordering of the methods.

Implementation. Meld is implemented by translating each feature into a conventional programming language and by a special runtime environment that supports execution of systems built from features. We derived the implementation algorithms from previous work on software generation, specifically language-based editors, where performance as good as handcoded editors has been achieved.

This implementation technique ensures language independence and portability because all existing features can be produced in a new implementation language or executed on a new architecture simply by reimplementing the translator and runtime support.

Thus, Meld combines the advantages of libraries, software generation, and object-oriented programming — without assuming the limitations of these approaches.

However, there are certain important issues in software reusability we did not address. Meld is no better than the widely used approaches at helping programmers determine if a particular component is suitable for reuse in a particular application. Meld also does not address the separate problem of categorizing and retrieving reusable components, but we are working on this, using a conceptual clustering approach derived from research in artificial intelligence.

An extended example

The important aspects of Meld can be illustrated with an extended example.

Memory manager. Suppose a programmer wants to implement a facility for loading and storing arbitrary entities to disk. Traditionally, he might modify the entities themselves to support memory management. Or he might add the facility directly to the runtime support of the programming language, perhaps as a generic package.

Neither approach promotes reusability. In the first case, memory management is specific to the entities. In the second case, the memory manager is reusable only in the sense that a text editor or a compiler is: I use it today on one file, you use it tomorrow on another file. This memory manager cannot be tailored to the particular needs of the application.

However, with Meld the programmer constructs a reusable building block that can be incorporated into any system that requires memory management.

The Memory Manager describes the world as seen from a simple memory manager's point of view. The world consists of a collection of memory-managed entities grouped under a memory-managed root. Each memory-managed entity has a unique identifier, a disk location, a designation of whether it is loaded in core or not, and a time stamp representing the most recent access to it.

*The dictionary definition of "meld" is "melt plus weld," or "to merge." Meld also stands for Multiple Elucidations of Language Descriptions, which was suggested by David Barstow.

The information about each memory-managed entity is always maintained in core; the actual content of the entity is what the memory manager loads and stores. The memory manager loads the content of an entity when it is first accessed. When primary memory is nearly full, entities are stored according to a least-recently-used policy.

Figure 1 shows the classes for a generic memory manager. This description is encapsulated into a feature, which has a name, an interface, and an implementation.

Interface. The interface lists the classes exported by the feature in its exports clause and lists the other features that are imported in its imports clause.

In this case, the ROOT and ENTITY classes are exported. Any exported instance variables are listed in the brackets following the class name; only the listed components are accessible outside the feature. The instance variables of ENTITY are entirely hidden, but the maxentities variable of ROOT is available to other features that import the memory-manager feature. This is more powerful than information hiding in Ada, where either all or none of the fields of a record are exported.

Implementation. The implementation part defines two classes, ROOT and ENTITY, followed by the instance variables and their types. The action equations that describe the behavior of the root and memory-managed entities are given in Figures 2 and 3, respectively.

The ENTITY class represents the entities managed by the memory manager. It defines four instance variables — unique-id, incore, lastuse, and diskid — that contain obvious information. Each instance variable is typed; strong typing is enforced. enforced.

The ENTITY class represents only the stub for an entity — there are no instance variables representing the content of the memory-managed entity. Instance variables that do represent the content are added when the Memory Manager feature is merged with one or more other features that provide entities that require memory management.

The ROOT class defines the memory-

What's wrong with the popular approaches

Three approaches to software reusability have achieved widespread use: subroutine libraries, software generation, and object-oriented programming.

None has achieved marked gains in productivity outside of a small set of application areas, but all treat reusability as part of the design rather than an afterthought of the implementation. It is rarely feasible to decompose an existing software system into reusable components that can be then used to construct other systems. Reusability must be engineered from the start.

Subroutine libraries. Libraries have had a significant effect on the production of mathematical software systems, as well as string manipulation and I/O, but they are not sufficient to achieve a large-scale improvement in software productivity and maintenance.

An individual subroutine is too small and the glue necessary to make many subroutines work together is too large. Small size makes subroutines more amenable to reuse than larger units, since they tend to be relatively simple and context-free, but only a small amount of code is actually reused by each subroutine call.

Libraries are written in a particular programming language, so decisions about primitive data types, constructors for structured data types, and subroutine linkages have already been made.

Subroutines are written with all the details filled in, so it is not possible to change the number or types of the parameters or to pick out part of the algorithm encapsulated in the subroutine without a proliferation of library versions or hand-copying of reused code.

Ada packages extend subroutine libraries, expanding the size of the reusable unit and encapsulating some of the glue among subroutines, so the glue can also be reused. Generic packages permit the types of selected parameters to be specified by the application, but do not support changing the number of parameters and do not help the programmer in specializing algorithms. Ada packages are better than subroutines when it comes to maintenance, since the Ada Programming Support Environment supports version control.

Code skeletons also extend subroutine libraries. They consist of many subroutines, or even many packages, making even more of the glue reusable. Code skeletons are barren, however: Instead of supplying the glue, the programmer must supply the contents. If the data structures and algorithms are not included in the skeleton, they must be provided by the programmer; if either is included, opportunities for reuse are restricted to those planned by the original implementor.

Software generators. Applied successfully to report generators, compiler-compilers and language-based editors, software generation meets the criterion of language independence. The generator can be changed to produce software in a different implementation language without significantly affecting existing input specifications.

Software generation produces code several orders of magnitude larger than the specification, but a new generator can be developed only after the application area is relatively well-understood and standardized — and standardization cannot keep pace with new applications.

It is difficult to combine the output produced by different generators, and large-scale patches to the generated code abandon the advantages of generation for later maintenance.

Object-oriented languages. This approach supports great flexibility in defining and composing reusable components. New classes inherit the facilities of specified existing classes, and a class may augment or replace any subset of the inherited data structures and operations.

In addition to augmenting and replacing, however, it is often necessary to combine distinct operations provided by different reusable components. Some object-oriented languages permit a collection of operations to be combined by sequencing — calling each one individually in some prespecified order — but with no support for conditional choice or interleaving.

Object-oriented programming languages imply particular decisions regarding the implementations of data types and operations, drastically limiting, a priori, the contexts in which a class can be reused.

```
Feature Module Interconnection Language

Interface:

  Exports all
  Imports Programming Language

Implementation:

  Class PROJECT :: = name: identifier
                     modules: seq of MODULE

  Class MODULE :: = name: identifier
                    imports: seq of identifier
                    exports: seq of SIGNATURE
                    components:
                      IMPLEMENTATION

End Feature Module Interconnection Language
```

Figure 4. The Module Interconnection Language feature.

ULE class can be tailored to the desired language by importing a Programming Language feature that defines the SIGNATURE and IMPLEMENTATION classes. In the case of Ada, which defines its own module construct, the MODULE class might be merged with an imported PACKAGE class.

Merging. Continuing with the example, the Memory Manager and MIL features are combined in a small system. In this system, the memory manager will manage the modules and their implementations.

Figure 5 illustrates how we use Meld to do this. We establish a connection between the ROOT and PROJECT classes on one hand, and between the ENTITY, MODULE, and IMPLEMENTATION classes on the other.

A feature may combine a group of imported features. In this case, the Programming-in-the-Large feature imports both Memory Manager and MIL, and certain classes from these two features are merged in the implementation.

For example, when the ROOT class is merged with the PROJECT class, each instance of the *renamed* PROJECT class has all the instance variables from both the PROJECT class of the MIL feature and the ROOT class of the Memory Manager feature.

However, the only instance variable from the ROOT class that can actually be accessed here is maxentities, since it is the only instance variable exported by Memory Manager. A new action equation overrides its default value of 100 and changes the constant value of maxentities to 200, so the memory manager now maintains at most 200 entities in core. Overriding default methods and constraints makes it easy to tailor features to a wide variety of applications.

The programming-in-the-large feature operates as follows. When a human user tries to read the text of a module, the primitive operation Access is received by the module, which then activates any action equations that are part of the Access method for the MODULE class.

In this case, the only equations are those that were inherited from the ENTITY class in Figure 3. These equations update lastuse to the current time and check if the incore instance variable has the value true. If not, the Load message is sent to self, meaning the module. This has the effect of loading the content of the accessed entity. If there are now too many entities in core, the least recently used entity is stored on disk.

A real system would probably merge many such features, as listed in Figure 6.

```
Feature Programming in the Large

Interface:

  Exports: . . .
  Imports: Module Interconnection Language,
           Memory Manager

Implementation:

  Merges ROOT, PROJECT as PROJECT
         ENTITY, MODULE as MODULE
         ENTITY, IMPLEMENTATION as IMPLEMENTATION

  Class PROJECT :: = maxentities: integer

    Methods:
    maxentities : = 200

End Feature Programming in the Large
```

Figure 5. The Programming-in-the-Large feature.

Implementation

Three early versions of Meld have been implemented.

The first, the Representation Description Language, barely resembles Meld except that it uses the same notation for instance variables. RDL development began in 1981 as part of the Display-Oriented Structure Editor[3] system and has been used for rapid prototyping of several small systems at Carnegie Mellon University and Siemens Research and Technology Laboratories.

The second, Genie, is much closer to Meld. It has the notion of features, then called views, but applied them only to support display mechanisms. Genie was used in 1985 to implement the MacGnome[4] educational programming environment for Pascal, which is expected to be released

```
Feature A Larger System

Interface:

   Exports: . . .
   Imports: Module Interconnection Language, Memory Manager,
            Compilation Unit, Documentation Facility,
            My Error Handler, . . .

Implementation:

   . . .

End Feature A Larger System
```

Figure 6. A larger system.

by Apple Computer as a commercial product.

The most recent implementation, Mercury, was completed in February 1987. Mercury supports constraints but not messages and methods, using parallel algorithms in a distributed environment.[5,6] Mercury has been used to implement demonstration-quality, multiple-user programming environments for small subsets of Modula-2 and Ada.

RDL was originally implemented in extended Pascal on the Perq operating system, was ported to Accent in 1982 and to C on Sun 3.0 (essentially Berkeley Unix) in 1985. Genie was implemented in a different extension of Pascal for the Macintosh. Mercury, written in C, was implemented by modifying the Cornell Synthesizer Generator[7] and runs on Digital Equipment Corp. VAX 750s under Unix 4.3 BSD on a 10M-baud Ethernet.

We are now designing a fourth implementation that will support all of Meld as presented here. The implementation has four parts: an environment for developing and maintaining Meld features, a translator for the structural components of classes and features, a translator for action equations, and runtime support.

The environment is itself described in Meld and will be implemented through a bootstrapping procedure. This environment provides mechanisms for choosing particular concrete representations (for example, for collections) and control structures (for example, sequential versus parallel equation evaluation) appropriate to the implementation language and the reusing application. It also supports evolution of features using techniques previously demonstrated in TransformGen,[8] which automates updates to existing objects when their definitions are modified.

The translator for the structural components is responsible for generating object definitions in a conventional programming language. An object in a Meld system is a synthesis of one or more classes, perhaps defined in different features. The classes themselves are not actually combined in the translation; instead, the physical representation of each object consists of several facets, where each facet corresponds to one of the classes. This is necessary because only some facets of an object may be active at any time; this was seen in the memory-manager example, where the stub for an object could be loaded and manipulated independently of its content.

It is easy to translate individual classes into the corresponding data types in conventional programming languages. For example, in Pascal each facet would be represented by a record, where each field in the record is a component or a pointer to a component (depending on the type of the component). The difficulty arises in maintaining the connections and consistency among the various facets of the same object. This is handled by generating new constraints that update the instance variables of one facet in response to changes in another.

To the popular reuse approaches Meld adds the unique concept of combining both data and operations.

Unlike the sets of instance variables, the methods for a merged class are combined in the translator for action equations. A local dependency graph is constructed to represent the action equations attached to the same selector. The nodes in the graph represent equations, and the edges represent the dependencies among the inputs and outputs of the equations. The local dependency graph is also constructed for constraints — those action equations (for the same synthesized type) that are not attached to a selector. These dependency graphs are used by the runtime support to determine the evaluation order for active equations.

The translation of action equations also involves translating each individual action equation into a procedure that performs the activities in the equation. The procedures take advantage of the facilities provided by the implementation language, as well as the primitives provided by the runtime support.

The runtime support provides all the necessary primitives for creating, deleting, and accessing objects, as well as for interacting with users, the file system, and the operating system. It sends messages corresponding to these primitives as necessary; for example, the Access message is sent to an object whenever the object is accessed. It also provides primitives to send the new messages defined in the Meld description and manages a queue of pending messages.

The most important job of the runtime support is to order the evaluation of active action equations. This is done using an adaptation of Reps' incremental attribute evaluation algorithm,[9] which was developed to generate language-based editors from attribute grammars. The basic idea is that the local dependency graphs are combined into a composite dependency graph at runtime to reflect the actual connections among objects. Only the graphs for the current message, plus the graphs for constraints, are considered in the composition. A topological sort of the composite graph determines the order in which equations are evaluated. This algorithm is linear in the number of affected objects, and is thus optimal.

Meld meets the three criteria essential to large-scale reuse and is superior to the widely accepted approaches to software reusability. The notation abstracts away the details of any particular programming language, although almost any language is suitable for implementation. Meld supports composition of components through imports and merging and tailoring through renaming and overriding equations.

Meld, then, blends the package library, software generation, and object-oriented programming approaches to reusability and solves most of the problems of these three approaches.

From the package library approach came relatively large-scale modular units — features — that enforce information hiding. From software generation came the idea of a declarative notation that can be translated into an efficient implementation in a conventional programming language. From object-oriented programming came the concepts of inheritance and of encapsulating behavior with data structures.

To these approaches Meld adds the unique concept of combining *both* data and operations. Other object-oriented languages merge data structures, in the sense of creating objects with private memory for the instance variables defined by each ancestor class. But no other notation supports combination of algorithms on the basis of dependencies.

Software reuse will lead to large productivity gains only when it becomes practical to combine separately developed algorithms more flexibly than procedure invocation. Meld represents the first step in this direction.

Acknowledgments

Yael Cycowicz, Dannie Durand, Charlie Krueger, Josephine Micallef, David Miller, Benjamin Pierce, Calton Pu, and anonymous referees made useful criticisms and suggestions about earlier versions of this article. Bob Schwanke at Siemens RTL, members of the Gandalf project at Carnegie Mellon University and students in the Programming Languages and Translators I course at Columbia University have written several features in Meld and discovered certain problems that have since been solved.

We also thank Nico Habermann and Mark Tucker for motivating our interest in reusable software. Meld is an outgrowth of both authors' work on Gandalf.[10]

This article is an expansion of "Composing Software Systems from Reusable Building Blocks," which appears in pp. 536-545 of *Proc. 20th Hawaii Int'l Conf. System Sciences*, (Computer Society Press, Los Alamitos, Calif., 1987).

Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and New York State Center for Advanced Technology, and in part by a DEC faculty award. Garlan is supported in part by the U.S. Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, N.J. and in part by ZTI-SOF of Siemens Corp., Munich, West Germany.

References

1. Gail E. Kaiser, "Generation of Run-Time Environments," *SIGPlan 86 Symp. Compiler Construction*, ACM, New York, 1986, pp. 51-57.
2. David Garlan, "Views for Tools in Integrated Environments," *Proc. IFIP WG 2.4 Int'l Workshop Advanced Programming Environments*, Springer Verlag, Berlin, 1986, pp. 341-343.
3. Peter H. Feiler and Gail E. Kaiser, "Display-Oriented Structure Manipulation in a Multi-Purpose System," *Proc. Seventh Int'l Computer Software and Applications Conf.*, CS Press, Los Alamitos, Calif., 1983, pp. 40-48.
4. Ravinder Chandhok et al., "Structure Editing-Based Programming Environments: The GNOME Approach," *Nat'l Computer Conference 85*, AFIPS, Reston, Va., 1985, pp. 359-370.
5. Simon M. Kaplan and Gail E. Kaiser, "Incremental Attribute Evaluation in Distributed Language-Based Environments," *Proc. Fifth SIGACT-SIGOPS Symp. Principles Distributed Computing*, ACM, New York, 1986, pp. 121-130.
6. Gail E. Kaiser and Simon M. Kaplan, "Reliability in Distributed Programming Environments," *Proc. Sixth Symp. Reliability Distributed Software and Database Systems*, CS Press, Los Alamitos, Calif., pp. 45-55.
7. Thomas Reps and Tim Teitelbaum, "The Synthesizer Generator," *Proc. SIGSoft/SIGPlan Software Engineering Symp. Practical Software Development Environments*, ACM, New York, 1984, pp. 41-48.
8. David Garlan, Charles W. Krueger, and Barbara J. Staudt, "A Structural Approach to the Maintenance of Structure-Oriented Environments," *Proc. Second SIGSoft/SIGPlan Software Engineering Symp. Practical Software Development Environments*, ACM, New York, 1986, pp. 160-170.
9. Thomas Reps, Tim Teitelbaum, and Alan Demers, "Incremental Context-Dependent Analysis for Language-Based Editors," *ACM Trans. Programming Languages and Systems*, July 1983, pp. 449-477.
10. A.N. Habermann and D. Notkin, "Gandalf: Software Development Environments," *IEEE Trans. Software Engineering*, Dec. 1986, pp. 1117-1127.

Gail Kaiser is an assistant professor of computer science at Columbia University, where she received a DEC Faculty Award. Her research interests are software reusability, object-oriented languages, programming environments, evolution of large software systems, application of artificial intelligence technology to software development and maintenance, and distributed systems.

Kaiser received the PhD in computer science from Carnegie Mellon University, where she was a Hertz Fellow.

David Garlan recently received the PhD in computer science from Carnegie Mellon University, where he has been an active member of the Gandalf and Gnome projects. His thesis, *Views for Tools in Integrated Environments*, addresses the problem of building tools for highly integrated environments.

His research interests include tool integration, software development environments, reusable software, functional programming languages, educational programming environments, and object-oriented systems and languages.

Questions about this article can be addressed to Kaiser at Columbia University, Computer Science Depart., New York, NY 10027; CSnet kaiser@cs.columbia.edu.

IEEE Software

July 1987

THEME ARTICLES

6 **Reusability Comes of Age**
Will Tracz
Reusability has long held out the promise of issuing in software's industrial revolution, of transforming a cottage industry into a mass-production system. The tools to do this are now appearing.

9 **Frame-Based Software Engineering**
Paul G. Bassett
One of reusability's main problems is how to easily modify available components. This frame-based approach handles the problem.

17 **Melding Software Systems from Reusable Building Blocks**
Gail E. Kaiser and *David Garlan*
This declarative language takes the best features from the three most popular reusability approaches, but overcomes their flaws. It supports language independence, component composition, and tailoring.

25 **The Reusable Software Library**
Bruce A. Burton, Rhonda Wienk Aragon, Stephen A. Bailey, Kenneth D. Koehler, and *Lauren A. Mayes*
The RSL couples a passive database with interactive design tools to make reuse an integral part of the software development process.

34 **Software Reuse through Building Blocks**
Manfred Lenz, Hans Albrecht Schmid, and *Peter F. Wolf*
Specification, design, and code can all be reused easily if handled as a building block. An IBM group recently developed this concept and applied it to systems programming — with success.

43 **Reusability Issues and Ada**
Anthony Gargaro and *T.L. (Frank) Pappas*
How do you write reusable code when your methodology doesn't address reusability? These guidelines developed by a major defense contractor may help.

52 **Can Programmers Reuse Software?**
Scott N. Woodfield, David W. Embley, and *Del T. Scott*
An experiment asked programmers untrained in reuse to evaluate component reusability. They did poorly. Are reusability's promises hollow? Or are there some answers?

60 **Cognitive View of Reuse and Redesign**
Gerhard Fischer
Reusable components are not enough. Program designers need tools that help them understand the components and how to use them. Fortunately, some support tools do exist.

SPECIAL FEATURE

74 **Implementing and Optimizing Lisp for the Cray**
J. Wayne Anderson, William F. Galway, Robert R. Kessler, Herbert Melenk, and *Winfried Neun*
This Portable Common Lisp version for the Cray couples the language's strengths with the machine's power. The researchers who developed the implementation tell how they did it.

MELDing Data Flow and Object-Oriented Programming

Gail E. Kaiser
Columbia University
Department of Computer Science
New York, NY 10027
(212) 280-3856

David Garlan
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213
(412) 268-7698

Abstract

MELD combines concepts from data flow and object-oriented programming languages in a unique approach to tool reusability. MELD provides three units of abstraction — equations, classes and features — that together allow sufficient options for granularity and encapsulation to support the implementation of reusable tools and the composition of existing tools in parallel (*i.e.*, interleaved) as well as in series.

1. Introduction

MELD started out as a data flow language oriented not towards dataflow computers but towards the implementation of incremental tools for programming environments. Our two primary goals were to support reusability of tools across a spectrum of programming languages and software development projects, and to be able to compose tools in parallel as well as in series. By "in parallel" we do not mean the tools necessarily operated concurrently, but that the incrementality of the tools was enhanced by interleaving the operation of the various tools that manipulate the same software entities. This was made possible by MELD's data flow facilities. For example, symbol resolution and type checking tools could be defined separately to permit the same symbol resolution facilities to be reused for different programming languages. But the tools could still be interleaved, permitting the type correctness of an individual identifier use to be determined as soon as the identifier was bound to its definition.

MELD evolved into an object-oriented language as it became clear that software entities such as modules, procedures, statements, *etc.*, could best be viewed as objects, with their components represented as instance variables and the object-specific tool fragments as methods. Multiple inheritance became our mechanism for composing tools in parallel, since we could indicate that a fragment of one tool manipulated the same object as a fragment of another tool by making that object an instance of a class that inherited both tools. In the above example, an identifier class would inherit from both the symbol class

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Association for Computing Machinery. To copy otherwise, or to republish, requires a fee and/or specific permission.

© 1987 ACM 0-89791-247-0/87/0010-0254 $1.50

manipulated by the symbol resolution tool and the variable class of the type checking tool.

As we compared MELD with other languages, we found that most object-oriented programming languages have some serious deficiencies when applied to the implementation of realistic systems of tools. The primary form of abstraction in these languages is the class. However, tools are, in general, much more complex than classes. Thus we found that previous object-oriented languages are inadequate for defining reusable tools for five reasons:

1. Granularity of abstraction is too small (a single class) to represent realistic tools. Most tools, such as window managers and graphics packages, involve a collection of several interrelated classes.
2. Granularity of abstraction is too large (a single procedure) to flexibly combine the functionality of interrelated tools. Methods may be composed by serial invocation, but not by parallel composition in a sense similar to coroutines, where one tool depends on the partial results of another and *vice versa*.
3. The notation does not support the parallel combination of types that define tools. This is necessary to support interleaving, as described in the previous point.
4. The boundaries of abstraction between tools are not enforced. Although typically only an object's methods may manipulate its instance variables, every class generally knows about every other class and any object can send messages to any other object. This does not permit information hiding within tools that are larger than a single class.
5. Classes cannot be parameterized to enhance tool reusability among widely divergent contexts.

The data flow and object-oriented aspects of MELD fit together nicely to correct these defects and thus support reusable tools. One way to view MELD is:

```
Meld = Reusable Modules +
       Data Flow +
       Multiple Inheritance.
```

Each of the components in isolation provides an important aspect of tool definition and integration, but all three are needed to make the whole thing work.

In this paper we present MELD using a bottom-up approach, building up to a small example programming environment consisting of several tools. In Section 2 we briefly explain our use of classes and multiple inheritance. In Section 3 we present MELD's data flow notation for writing methods and describe in Section 4 how this notation supports parallel as well as serial combination of separately defined tool fragments. Finally, we address the construction and composition of reusable tools in Section 5.

Certain aspects of MELD have been described in three previous papers [7, 11, 12]. This paper describes our past work on MELD in the context of other research on object-oriented programming languages, and presents a more detailed rationale for our design decisions. There is also some material presented here for the first time, notably overriding, generic modules, and compile-time consistency checking. The algorithms underlying MELD's (running but only partially completed) implementation are described elsewhere [14, 13, 10, 8].

2. Basics

2.1. Classes and Unions

```
Class DEPUNIT ::=
            inputs: seq of ENTITY
            outputs: seq of ENTITY
            command: string
    ...

Class ENTITY ::=
            time: TIMESTAMP
            origin: ORIGIN
    ...

Union ORIGIN ::=
            DEPUNIT | PRIMITIVE

Class PRIMITIVE ::=
    ...

(* MELD keywords are underlined.
"::=", ":", and "|" are special sym-
bols defined by MELD. Built-in classes
and type constructors are given in bold
italics. *)
```

Figure 2-1: Classes and Unions for Make

Figure 2-1 illustrates the *classes* and *unions* defined for a tool similar to the Unix™ Make utility [6]. Each instance of the DEPUNIT class is a dependency unit. Its functionality is to define the outputs as dependent on the inputs; whenever one or more of the inputs change in value, the outputs are recalculated by applying the given command to the inputs. Each input or output is an instance of the ENTITY class, maintains a timestamp signifying when it was last changed, and indicates its origin. As defined by the ORIGIN union, the origin may be an instance of a dependency unit (if the object is derived) or an instance of the PRIMITIVE class (otherwise). The PRIMITIVE class has no instance variables, since it is an atomic value.

MELD classes are similar to those found in other object-oriented programming languages. They consist of a name, a list of instance variables, a collection of methods, and inheritance information; we ignore the methods and inheritance information for now. As in Simula [5], Trellis/Owl [20] and C++ [23], instance variables are typed. MELD unions provide a means for designating a set of alternative types. They are used to specify the type of an instance variable as any one of several classes.

The type of an instance variable is either a class, a union, or a *collection*. Collections fill the role of Smalltalk's indexed instance variables [9] and Trellis/Owl's type generators. MELD provides four kinds of collections: array, sequence, set and table. Arrays are of fixed length and sequences are dynamic length, both accessed by index. Sets and tables are of dynamic length and are accessed by keys. A designated instance variable serves as the key. The difference between sets and tables is that sets are unordered and tables are ordered by a particular relation among the keys.

2.2. Merging

```
Class MODULE ::=
            name: identifier
            imports: seq of identifier
            exports: seq of SIGNATURE
            body: IMPLEMENTATION
    ...

Merges DEPUNIT, MODULE As COMPUNIT
Class COMPUNIT ::=
            objcode: OBJECTCODE
            symtab: SYMBOLTABLE
    ...
```

Figure 2-2: Merging DEPUNIT, MODULE

Figure 2-2 shows how we *merge* the previously defined DEPUNIT class and the illustrated MODULE class into the new COMPUNIT class. A module consists of a name, a

list of imported modules, a list of exported procedures, types, variables, *etc.*, and a body written in the source code of some programming language. SIGNATURE and IMPLEMENTATION, not shown here, would be defined specifically for the desired programming language. Each instance of COMPUNIT represents a compilation unit, and adds the object code and symbol table results of compilation to the instance variables defined by DEPUNIT and MODULE.

Merging determines *inheritance*, in the sense of defining a class as a specialization or *subclass* of another class, its *superclass*. Like Flavors [16] and CommonLoops [3] and some other languages, MELD permits multiple inheritance, meaning that a class may be a specialization of more than one other class. Thus the inheritance structure is a lattice rather than a strict hierarchy.

The "Merge" clause lists the superclasses whose components are inherited by the subclass, given after the "As" keyword. Note that merging is written separately, rather than in the body of either the subclass or the superclasses. This supports an easy shorthand for defining a new class that simply merges together the components provided by all its superclasses, without adding any new components of its own.

MELD permits both instance variables and methods to be inherited. As with all systems that support multiple inheritance a problem arises if an instance variable or a method with the same name is inherited from more than one superclass; we describe in the next section how we deal with this for methods. MELD solves this problem for instance variables by asking the programmer at compile-time whether or not his intention is to *share* the instance variable among the methods for the separately defined superclasses. If not, MELD helps the programmer to rename one of the conflicting instance variables (in the context of this application).

3. Methods

3.1. Data Flow Equations

```
Class DEPUNIT ::=
            inputs: seq of ENTITY
            outputs: seq of ENTITY
            command: string

    Methods:

    MAKE() -->

        Send MAKE To inputs[all]

        If Min(outputs[all].time)
              < Max(inputs[all].time)
        Then Send APPLY To self

    APPLY() -->

        outputs[all].origin := self

        outputs :=
            Apply(command, inputs)

Class ENTITY ::=
            time: TIMESTAMP
            origin: ORIGIN

    Methods:

    MAKE() -->

        If TypeOf(origin) <> PRIMITIVE
        Then Send MAKE To origin

    CHANGE()
      On (any except time, origin) -->

        time := Now()

(* "-->" separates the method selector
and formal parameters from its local
variables (not shown in this example)
and body.  "all" refers to each ele-
ment of a collection. *)
```

Figure 3-1: Methods for Make

Figure 3-1 shows the *methods* for the DEPUNIT and ENTITY classes. DEPUNIT

has two methods, MAKE and APPLY. MAKE forwards the MAKE message to all inputs of the dependency unit, to ensure they are up-to-date. It checks whether any input has changed more recently than any output and, if so, sends the APPLY message to the dependency unit to actually rederive the outputs. In MELD the statements of a method are not purely sequential commands, but are treated as a set of *equations* that must be evaluated. When there are no dependencies between the equations, as is the case here, they may be evaluated in either order, or concurrently. The evaluation of equations is described in detail later in this section.

The APPLY method calls the Apply function (not shown) on the command and input instance variables. Apply uses a system call — illustrated in the next section — to ask the operating system to invoke the given command with the given inputs, and return the indicated outputs. Once computed, the origin of each output is set to the dependency unit. Since the second equation computes the outputs used in the address expression on the left hand side of the first equation, the second must be scheduled before the first; we explain shortly how we use data flow techniques to accomplish this scheduling.

The ENTITY class also has two methods. If the object is not primitive, the MAKE message is forwarded to rederive the object (if necessary). The CHANGE method responds to MELD's built-in CHANGE message, automatically sent by the run-time support to each object that is modified by any another method or by an external agent (*e.g.*, the human user of the Make tool). Most classes do not provide a CHANGE method; since those classes that do are known at compile-time, such automatic propagation of built-in messages by the run-time support requires minimal overhead. Since the ENTITY class does define a CHANGE method, it is automatically invoked whenever any instance variable of the object is modified in any way. For obvious reasons, the On clause restricts the application of the CHANGE method to changes to instance variables other than time or origin; since the ENTITY class does not define any other instance variables, the CHANGE method is meaningless unless ENTITY is merged with some other class whose content changes in value. Thus ENTITY (and the other Make facilities) are intended to be used in the same manner as Flavors mixins.

Like other object-oriented languages, MELD methods consist of a name or *selector*, formal parameters, local variables, and a body. The formal parameters and local variables are typed in the same manner as instance variables. Argument transmission is the usual call-by-reference, where each object is itself passed rather than a copy. MELD enforces the abstraction common to most object-oriented programming languages: only the methods for an object's class can directly modify the instance variables of the object.

Although they look similar, as noted above, MELD methods are quite different from the methods of other languages. Almost all object-oriented languages define each method in a manner akin to a procedure, as a sequence of statements that are executed in the order written. Message passing normally works in the same way as a procedure call: by invoking a method, passing it the actual parameters, and waiting for the return results. Exceptions include certain concurrent languages, notably the various Actor languages [2], where methods may execute concurrently and a sending object does not necessarily block until receiving a response.

MELD is another exception, but different from other concurrent object-oriented languages in that the granularity of concurrency is much finer — at the level of a statement. The goal here is not to maximize concurrency, since methods are typically so short that performance improvements by this mechanism are unlikely, but to support data flow interleaving of separately defined methods in order to connect these tool fragments in parallel; this is explained in the next section.

```
a := F(b)        (* 2nd *)
b := G(c)        (* 1st *)
d := H(e)        (* any time *)
```

Figure 3-2: Dependencies among Equations

In the simplest case, the equations of a method can be evaluated in any order. However, in many cases one MELD equation performs a calculation using a value computed by another equation; we say the first equation *depends on* the second. It is then necessary to execute the second equation before the first one, in the sense of data flow languages [1]. Therefore, equation evaluations are scheduled in the order implied by the dependencies among equations, using an adaptation of algorithms originally developed for incremental evaluation of attribute grammars [19]. Figure 3-2 shows an example in which the computation F(b) in the first equation depends on the value b assigned by the second equation. Thus the second is automatically evaluated before the first. The third equation is independent of the other two, and can be evaluated at any time.

3.2. Constraints

```
Class DEPUNIT ::=
            inputs: seq of ENTITY
            outputs: seq of ENTITY
            command: string

Class MODULE ::=
            name: identifier
            imports: seq of identifier
            exports: seq of SIGNATURE
            body: IMPLEMENTATION

Merges DEPUNIT, MODULE As COMPUNIT
Class COMPUNIT ::=
            objcode: OBJECTCODE
            symtab: SYMBOLTABLE

    Methods:

    inputs[1] := body

    inputs[2,..] := imports[all]

    command := "Compile"

    objcode := outputs[1]

    symtab := outputs[2]

(* The "[i]" notation accesses the ith
component of an ordered collection
(array, sequence or table) while ",.."
refers to the rest of the collection.
*)
```

Figure 3-3: Constraints for COMPUNIT

When the DEPUNIT and MODULE classes are merged into the COMPUNIT class, it becomes necessary to specify how the inputs and outputs of the dependency units relate to the components of the module and the derived information of the compilation unit, respectively. Figure 3-3 shows five assignments that act as *constraints* for the COMPUNIT class. The first and second together ensure that the inputs of the dependency unit are always maintained as the current values of the body and imported resources of the module. The third behaves as a constant definition to set the dependency unit's command to "Compile", as appropriate for the context of module recompilation. The last two con-

straints define the objcode and symtab instance variables of the compilation unit to be the first two outputs of the dependency unit, to make sure the most recent object code and symbol table are always accessible.

MELD constraints are equations that are independent of any particular message, that is, they appear separately rather than as part of a method body. Constraints are automatically re-evaluated as necessary to maintain consistency. An assignment (":=") constraint is recalculated if any argument to the right hand side expression or the left hand side address expression changes in value. A conditional ("If") is recomputed in response to changes in the arguments to the conditional expression.

Constraints fill the same role as the active values of Loops [22] and some other object-oriented programming languages, to automatically maintain specified relationships among objects. The primary difference is that active values are normally implemented using lazy evaluation — that is, instance variables on the left hand sides are recomputed only when they are accessed; lazy evaluation is possible but not assumed for MELD constraints. MELD constraints are also related to Borning's constraints [4], which are bidirectional, while MELD's are unidirectional.

4. Method Interleaving for Multiple Inheritance

4.1. Data Flow among Methods

Figure 4-1 illustrates a case where distinct APPLY methods are defined by both the COMPUNIT class and its DEPUNIT superclass. The COMPUNIT method sends the (built-in) SystemCall message to itself to actually run its object code in the external environment, while the DEPUNIT method derives this object code. Thus the second DEPUNIT equation must execute first, followed by the COMPUNIT equation and the first DEPUNIT equation in either order (or concurrently), since neither depends on the other.

```
Class DEPUNIT ::= ...

    Methods:

    APPLY() -->

        outputs[all].origin := self

        outputs :=
            Apply(command, inputs)

Merges DEPUNIT, MODULE As COMPUNIT
Class COMPUNIT ::= ...

    Methods:

    objcode := outputs[1]

    APPLY() -->

    Send SystemCall("execute",objcode)
                  To self

(* "SystemCall" is one means whereby
MELD supports interactions with the
operating system and other facilities
in the external environment. *)
```

Figure 4-1: Distinct Method Definitions

A problem arises when a method with the same selector (*i.e.*, name) is inherited from multiple ancestors, as in this example. This is solved by data flow interpretation of methods. Specifically, MELD handles multiple inheritance of methods as follows: Consider all the methods collectively defined by a particular class, its superclasses, their superclasses, *etc.*, all the way up to the root of the lattice. Sometimes this collection contains more than one method with the same selector. When a message with this selector is received by an instance of the class, all of these methods are invoked, with the scheduling of

individual equations determined using the data flow techniques previously described. Thus, MELD may *interleave* separately defined methods inherited from distinct ancestors. (MELD also supports overriding inherited methods, as in standard multiple inheritance; we will get to this soon.)

4.2. Ancestor Class Encapsulation

In contrast to MELD, most object-oriented programming languages require that such multiple inheritance of the same method be disambiguated to choose exactly one of these methods to be triggered by the corresponding message. This is done by explicit selection of one method in the class itself, as in Trellis/Owl, or by picking the first method in some strict precedence ordering of the superclasses — such as Loops' rule of depth-first search with respect to the left-to-right ordering given in each superclass list.

We claim such solutions destroy the encapsulation of functionality defined by a superclass. The methods of a particular superclass make certain assumptions about each other, such as that one produces instance variable values to be consumed by another. These assumptions are violated when method selection is determined by incidental orderings and name conflicts. One might argue that the orderings and name conflicts are not incidental, but carefully understood by the programmer. This seems achievable only when all the ancestor classes are written by the same programmer or by a very small, tightly knit group of programmers; large scale reusability is impossible if such detailed understanding is required.

A few languages attempt to solve the problem of superclass encapsulation and the consequent inter-method assumptions by permitting more than one of the candidate methods to be triggered, but always in some explicit order. For example, Flavors uses a mechanism similar to Loops to chose a main method, but then orders *before* and *after* methods nested according to the class precedence list. We argue that such explicit ordering implies most of the disadvantages discussed above, and similarly restricts reusability to a small scale.

4.3. Overriding

```
Class DEPUNIT ::= ...

    Methods:

    APPLY() -->

        outputs(all).origin := self

        outputs :=
            Apply(command, inputs)

Merges DEPUNIT, MODULE As COMPUNIT
Class COMPUNIT ::= ...

    Methods:

    objcode := outputs(1)

    APPLY() -->

        Override outputs :=
            Smart(command, inputs)
```

Figure 4-2: Specialization

Consider the case where the programmer specializes the compilation unit to support 'smart recompilation' [25]. Smart recompilation refines the granularity of dependency: in the original combination of the MAKE and APPLY methods, all the outputs of dependency units depend on all the inputs; we now consider the language-specific dependencies among the source code symbols (procedure, type and variable names) defined within the input modules. Consider the case where a

module M imports a module N. Using the Make facilities as originally defined, M is a dependency unit that depends on N; this means that whenever N's interface changes, M's object code must be rederived. A smart recompilation strategy would keep track of the particular symbols exported by N that are actually used by M. Suppose N exports procedures p and q, and M uses only p. Then if changes to N only modify q, it is not necessary to recompile M. This refinement is accomplished by *overriding* the APPLY method (and taking the MAKE method as is) to instead call the Smart function (not shown). Smart performs various checks on the compilation unit's symbol table before actually passing the command in a system call.

Note that there is an important tradeoff between functional encapsulation of ancestors and functional specialization of descendants. Most other languages have decided to make specialization easy and such encapsulation nearly impossible, while we make encapsulation easy and specialization slightly more tedious for the programmer, but certainly not impossible.

5. Reusability

5.1. Features

Figure 5-1 shows the Module Interconnection Lang *feature*, which defines and exports the MODULE class discussed previously. This feature exports everything: the module class with all its instance variables and (omitted) methods. It imports the Programming Language feature, not shown, which provides SIGNATURE and IMPLEMENTATION classes appropriate to the specific programming language. In practice, there would be many such features, one per relevant programming language, that could be separately merged with the Module Interconnection Lang feature in order to extend each programming language with module interface facilities.

```
Feature Module Interconnection Lang

Interface:

    Exports all

    Imports Programming Language

Implementation:

    Class MODULE ::=
                name: identifier
                imports: seq of identifier
                exports: seq of SIGNATURE
                body: IMPLEMENTATION

End Feature Module Interconnection ...
```

Figure 5-1: Module Interconnection Lang

We have already explained that the primary motivation for MELD's unique method interleaving is to increase the potential for large scale reusability. Unfortunately, interleaving and other mechanisms for ancestor encapsulation (presented shortly) are not sufficient to achieve this goal. The most important difficulty is that a single class, even augmented by all the instance variables and methods inherited from its ancestor lattice, is rarely sufficient to implement an interesting tool. The Make utility consists of several classes (and a union); similarly, the Module Interconnection Language used in the example is not complete without the IMPLEMENTATION and SIGNATURE classes. Some mechanism is needed to bundle together those classes that together define a distinct tool or tool fragment that can be reused in many different applications.

MELD solves this problem by providing the

feature as a unit of modularity larger than the individual class. Each feature defines a tool fragment, such as a parser, that can be reused as part of many different tools, or a tool, such as a memory manager, that can be reused across many systems. MELD features can be mixed and matched to obtain the facilities desired for a system.

Like an Ada™ package, each feature consists of an *interface* (specification) and an *implementation* (body). The interface lists the facilities — classes and unions — exported by the feature and the other features whose facilities are imported. Unlike Flavors' packages, the interface is strict; it is not possible for a client of a feature to refer to an internal facility through a name qualification scheme. The implementation defines sufficient facilities that, perhaps together with imported facilities, provide the required exports and implement the functionality of the feature.

```
Feature System Modeller

Interface:

    Exports ...

    Imports
        Module Interconnection Lang,
        Make, Compiler, ...

Implementation:

    Merges DEPUNIT, MODULE As COMPUNIT
    Class COMPUNIT ::=
            objcode: OBJECTCODE
            symtab: SYMBOLTABLE

    Merges ENTITY, IMPLEMENTATION
                As IMPLEMENTATION
    Merges ENTITY, OBJECTCODE
                As OBJECTCODE

    ...

End Feature System Modeller
```

Figure 5-2: System Modeller

The System Modeller feature, shown in Figure 5-2, pulls together several tools into a complete system. The System Modeller could in turn be reused as a tool in many larger systems, such as the Cedar environment [24], from which we borrowed the "System Modeller" name and its basic functionality [15]. Note that the same name appears on both the left and right hand sides of two of the merges clauses. The use of "IMPLEMENTATION" on the left hand side refers to the facility imported from Module Interconnection Lang (via its own import of Programming Language), while the use on the right is a new subclass that merges the components of the imported ENTITY and IMPLEMENTATION classes. Throughout the rest of the implementation part of this feature, "IMPLEMENTATION" refers to this new subclass. MELD provides this simple scoping mechanism to make it easier for feature implementors to use mnemonic naming schemes.

Thus imported classes and unions may be used in two different ways within a feature's body: (1) as the types of instance variables (or the alternatives of unions) and (2) on the left hand side of a merges clause. This means that one or more superclasses defined by different features can be merged into a subclass defined by another feature. In the first case, where an imported class is used as the type of an instance variable, the messages for that class can be sent to the instance variable. This capability is typical, of course, for object-oriented languages, whether or not strongly typed: messages associated with a class can always be sent to instances of that class wherever they appear.

Figure 5-3 illustrates the Make feature. It exports the DEPUNIT and ENTITY classes,

and hides ORIGIN and PRIMITIVE, since these are useful only within Make. In addition to the usual advantages of such information hiding [17], this prevents proliferation of class and union names into the global name space, a problem with most other object-oriented languages.

```
Feature Make

Interface:

    Exports
        DEPUNIT[inputs, outputs,
                command, MAKE],
        ENTITY[]

    Imports Time

Implementation:

    Class DEPUNIT ::=
                inputs: seq of ENTITY
                outputs: seq of ENTITY
                command: string

    Class ENTITY ::=
                time: TIMESTAMP
                origin: ORIGIN

    Union ORIGIN ::=
                DEPUNIT | PRIMITIVE

    Class PRIMITIVE ::=

End Feature Make

(* The instance variables and methods
available for merging are listed be-
tween square brackets following the
class name in the exports list. *)
```

Figure 5-3: Make

DEPUNIT exports all its instance variables plus its MAKE method (from figure 3-1). These instance variables are accessible to subclasses of DEPUNIT, such as the previously described COMPUNIT, while the MAKE message can be sent by clients as well as by a compilation unit. The APPLY method remains internal, as do Smalltalk's private methods, but here the methods are private only with respect to clients and subclasses defined in other features. For example, if a method defined in the COMPUNIT class sends the APPLY message to itself, this does not have the effect of invoking DEPUNIT's APPLY method. Notice that the ENTITY class does not export anything at all; none of its components are available to subclasses. It is still necessary, however, to export the ENTITY class itself so it can be merged with other classes such as MODULE that are inputs or outputs of dependency units.

Trellis/Owl and CommonObjects [21] also provide stronger forms of encapsulation than most other object-oriented languages. Trellis/Owl's classes can define methods as subtype-visible, meaning these methods are accessible to subclasses but not to clients. CommonObjects' instance variables are inaccessible to subclasses as well as to clients. Neither provides any form of module larger than the class, however, nor any means for restricting an arbitrary subset of a class' instance variables and methods from inheritance.

5.2. Generic Features

```
Generic Feature System Modeller
    [Programming Language
        (SIGNATURE, IMPLEMENTATION)]

Interface:

    Exports ...

    Imports
        Module Interconnection Lang,
        Make, Compiler, ...
        ...

Implementation:
    ...

End Feature System Modeller
```

Figure 5-4: Generic System Modeller

MELD allows an implementor to parameterize tools by writing *generic features*. Figure 5-4 shows how the System Modeller Feature can be written as a generic feature, parameterized by the Programming Language Feature. The latter consists of the SIGNATURE and IMPLEMENTATION classes, which define the boundary between the System Modeller and a particular programming language. To use such a generic feature the implementor must *instantiate* it with a feature, such as C, that defines the facilities appropriate to a particular programming language. Instantiation binds the types and methods exported by the actual feature parameter to those anticipated by the formal parameter. Figure 5-5 illustrates how this is done. The resulting System Modeller has now been tailored to provide Make-like behavior and module interface checking for C.

```
Feature Programming Environment

Interface:

  Exports ...

  Imports ...
      SM Is New System Modeller
          (Programming Language = C
              (SIGNATURE = UNIT,
               IMPLEMENTATION = text))
      ...

End Feature Programming Environment
```

Figure 5-5: System Modeller Instantiated

MELD features are thus similar, but not identical to Ada packages [18]. Unlike Ada, generic packages features may be parameterized by other features. However, like Ada, parameter(s) to a generic feature are bound at compile time.

6. Conclusions

MELD's unique combination of reusable modules, data flow and multiple inheritance makes it possible to define reusable tools that can be composed in parallel as well as in series. It solves the problems posed in the introduction as follows.

1. Granularity of abstraction in other languages is too small — a single class — to represent realistic tools. MELD's *features*, however, allow the programmer to bundle up collections of classes as a tool unit.
2. Granularity of abstraction in other languages is too large — a single procedure — to flexibly combine the functionality of interrelated tools. In contrast, MELD's methods are made up of *equations* (rather than statements), which are intertwined by the run-time support using data flow techniques.
3. Other languages do not provide notation for the parallel combination of types that define tools. MELD's *merging* allows the programmer to compose tools in parallel, by combining the objects manipulated by the tools using a special form of multiple inheritance.
4. Information hiding is not enforced in other languages, except within a single class, whereas feature *interfaces* are strictly enforced at compile-time.
5. Classes in other languages are not parameterized, but *generic features* permit parameterization of both types and methods, enhancing tool reusability among widely divergent contexts.

Acknowledgements

We would like to thank Josephine Micallef and Calton Pu for reviewing a previous draft of this paper and making many useful criticisms and suggestions.

Dr. Kaiser is supported in part by grants from AT&T Foundation, Siemens Research and Technology Laboratories, and New York State Center of Advanced Technology — Computer & Information Systems, and in part by a DEC Faculty Award. Dr. Garlan is supported in part by the United States Army, Software Technology Development Division of CECOM COMM/ADP, Fort Monmouth, NJ and in part by ZTI-SOF of Siemens Corporation, Munich, Germany.

References

[1] William B. Ackerman.
Data Flow Languages.
Computer 15(2):15-25, February, 1982.

[2] G. Agha.
Actors: A Model of Concurrent Computation in Distributed Systems.
MIT Press, Cambridge, MA, 1986.

[3] Daniel G. Bobrow, *et al.*.
CommonLoops: Merging Common Lisp and Object-Oriented Programming.
In *ACM Conference on Object-Oriented Systems, Languages, and Applications*, pages 17-29. Portland, OR, September, 1986.

[4] A. H. Borning.
Constraints and Functional Programming.
In *Sixth Annual International Phoenix Conference on Computers and Communications*, pages 300-306. Scottsdale, AZ, February, 1987.

[5] Ole-Johan Dahl and Kristen Nygaard.
SIMULA— an ALGOL-Based Simulation Language.
Communications of the ACM 9(9):671-678, September, 1966.

[6] S.I. Feldman.
Make — A Program for Maintaining Computer Programs.
Software — Practice & Experience 9(4):255-265, April, 1979.

[7] David Garlan.
Views for Tools in Integrated Environments.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments*. Trondheim, Norway, June, 1986.
Lecture Notes in Computer Science 244, Springer-Verlag, 1986.

[8] David Garlan.
Views for Tools in Integrated Environments.
PhD thesis, Carnegie Mellon University, May, 1987.

[9] Adele Goldberg and David Robson.
Smalltalk-80 The Language and its Implementation.
Addison-Wesley Pub. Co., Reading, MA, 1983.

[10] Gail E. Kaiser.
Semantics of Structure Editing Environments.
PhD thesis, Carnegie Mellon University, May, 1985.
Technical Report CMU-CS-85-131.

[11] Gail E. Kaiser and David Garlan.
MELDing Software Systems from Reusable Building Blocks.
IEEE Software 4(7), July, 1987.

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