MELD/Features: An Object-Oriented Approach to Reusable Software

Gail E. Kaiser
David Garlan*

October 1986

CUCS-226-86

Abstract

This technical report consists of three related papers in the area of reusable software. *Synthesis of Programming Environments from Reusable Building Blocks* presents the notion of 'features' as an approach to reusable descriptions for the generation of programming environments. *Composing Software Systems from Reusable Building Blocks* presents MELD, a declarative language based on features, and generalizes features to the description of reusable software for general applications. *MELD: A Declarative Language for Writing Methods* focuses on MELD's capabilities for describing the behavior of software systems.

Part of this research was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA. *Mr. 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. Mr. Garlan's address is Department of Computer Science, Carnegie-Mellon University, Schenley Park, Pittsburgh, PA 15213.

Synthesis of Programming Environments from Reusable Building Blocks

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

David Garlan
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213

29 August 1986

Abstract

Generation of programming environments has proven to be a practical application of software reusability. The generator combines the language-independent kernel with a formal description of the desired programming language to produce a language-specific programming environment. Unfortunately, such reusability has been limited to the facilities provided by the kernel and it has not been practical to reuse the behavior specified in a language description. This has led to pushing more and more functionality into the kernel. We argue that this is unnecessary. We describe a unit of modularity, called a *feature*, that can be added to notations for language description. Features provide a means for dividing language descriptions into reusable pieces of functionality that can be mixed and matched to obtain the facilities desired for a particular programming environment. Further, features contribute to synthesis of reusable software for other applications.

This research 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. *Part of this research was conducted while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

1. Introduction

The generation of programming environments is one of the more promising areas of software reusability research. The dominant theme is to combine a language-independent kernel with a formal language description to produce a programming environment for the desired programming language. The kernel provides the common facilities, including the user interface, invocation of programming tools, the file system interface, and manipulation of the internal representation of programs and auxiliary structures. Representative examples are [33, 30, 6, 31, 20].

Unfortunately, such reusability is severely limited since only the language-independent kernel can be reused. The language-specific descriptions are invariably bound to linguistic and functional context — it is rare that parts of the descriptions can be reused for different tools or for environments for different programming languages. In response to this problem. more and more functionality has been pushed into the kernel itself, adding support for symbol tables, memory management, debugging, *etc.* But this doesn't really solve the problem: it is unrealistic to expect any set of basic facilities to satisfy all needs for reusability; further, the kernel cannot be tailored to specific needs, except through the language description.

We extend the generation of programming environments to support much wider reusability. We compose an environment from a <u>collection</u> of formal descriptions that each implement a basic unit of functionality, called a *feature*. An implementor can encapsulate a set of facilities as a feature, reuse features across multiple environments, integrate features with other features, and tailor features to specific environments. The benefits of this approach are:

- an environment can be built from manageable, interacting, functional units;
- these language-independent units support libraries of reusable building blocks;
- these units provide appropriate boundaries of abstraction without overly limiting data sharing.

Section 2 describes features as an extension of current notations for language description. An example of a small environment for programming-in-the-large illustrates the use of features in the synthesis of programming environments. Section 3 describes the implementation and Section 4 discusses related work. Although motivated by programming environments, our notation and algorithms extend well beyond these bounds. Indeed, we believe that features are an appropriate paradigm for reusable building blocks for any software system. The concluding section supports this claim.

2. Features: An Example

We start with the automatic generation of programming environments from formal descriptions. A generation system typically consists of two parts, the kernel and a translator from the formal notation into an internal representation understood by the kernel. The description also consists of two parts, describing the syntax and semantics, respectively, of the desired programming language. The first, an extended form of BNF [2], specifies the objects to be manipulated by the environment (procedures, statements, expressions, *etc.*). The second defines the tools provided by the environment (symbol resolution, type checking, code generation, and so on). The tools are usually written as attribute equations [32, 21] or action routines [26, 1]; in both cases, the tools manipulate auxiliary data structures called *attributes* that represent symbol tables, object code, *etc.*

We diverge from previous approaches in that we specify a programming environment not by one, but by a collection of formal descriptions, called *features*. Each feature defines both syntax and semantics of a unit of functionality that can be incorporated into a variety of environments. Some other systems, such as ALOE [8] and Mentor [7], also allow multiple descriptions, but these additional descriptions are strictly auxiliary — for describing the structure of attributes — and are independent of the 'primary' language description. In contrast, our primary description is obtained by automatically synthesizing the features that collectively define the facilities of the programming environment. Features extend our previous work on *views* [14] — a paradigm for generating environments from multiple descriptions — by adding interfaces, inheritance of facilities, and notation for specifying behavior; together, these extensions support reusability of functional units. We illustrate the approach by integrating a feature implementing system generation facilities into an environment for programming-in-the-large.

2.1. A Simple Environment for Programming-in-the-Large

```
Feature Module Interconnection Language
Interface:
    Exports all
    Imports Programming Language
Implementation:
    Use Programming Language
    MODULE          ::= mod-name: identifier
                        imports: seq of identifier
                        exports: seq of SIGNATURE
                        components: seq of COMPONENT
    COMPONENT       ::= MODULE | IMPLEMENTATION
End Feature Module Interconnection Language
```

This feature exports all the named types, including MODULE, COMPONENT, SIGNATURE and IMPLEMENTATION. The latter two are imported from the Programming Language feature, whose scope is opened by the Uses clause.

Figure 2-1: Specification of a Module Interconnection Language

Figure 2-1 shows a feature that defines a simple module interconnection language. It consists of a collection of *productions* and *unions* that define the data structures to be manipulated by that feature. A production has a list of named and typed components. The type is a primitive ("identifier", "integer", "text", and so on), another production, a union, or a collection such as "sequence" (an ordered list). A union gives a list of alternative productions and subunions. In this case, the syntax description consists of one production (MODULE) and one union (COMPONENT).

The MODULE production defines four components. The first, 'mod-name', gives the name of the module. The other three components are sequences. The 'imports' component lists the names of imported modules and 'exports' lists the signatures of exported facilities. SIGNATURE is imported from the Programming Language feature, and the Module Interconnection Language is not concerned whether it is a production or a union; a union is most likely, to indicate alternatives such as procedure, variable, type, *etc.* The 'components' component lists the subparts of the module, either other modules or IMPLEMENTATIONs (also imported from Programming Language). In addition to this syntax description, the full description of the environment would contain an associated operational component (for example, to check interfaces between modules), not shown here.

The most significant differences between the notation of features and classical syntax descriptions are (a) we group productions and unions together into *modular units*, *i.e.*, features, and (b) an *interface* defines the abstraction implemented by those productions and unions. In this case the interface exports all of the data structures defined within the feature. It imports another feature that defines the desired programming language, including the language-specific SIGNATURE and IMPLEMENTATION. As we will see, the Module Interconnection Language feature can be combined easily with additional features that augment its functionality.

2.2. Make

Suppose we would like to integrate our simple environment for programming-in-the-large with a facility, based on the Unix™ Make utility [11], for automatically regenerating an executable system after source code changes. Traditionally, we might embed this capability directly in the language description, but then we could not reuse it. Alternatively, we might add Make to the common kernel, allowing it to be reused across all environments, but without means to tailor or extend the facility for specific environments. Using features, however, we take a modular approach: we define an independent formal description that can be incorporated into and tailored for any environment requiring Make-like facilities.

Figure 2-2 describes the world from Make's point of view. That world consists of a collection of dependency units (DUs), each indicating a command that defines the relationship between input objects and output objects. Each object indicates the originating dependency unit and the time when it was produced. Certain objects are primitive, meaning they are generated outside Make. When the Make command is issued by some external agent, such as the human user of the programming environment, Make backtracks through the chain of outputs and inputs to determine whether each output object is up to date with respect to its input objects. If not, Make applies the corresponding command to rederive the object. We present the equations that implement this behavior in Section 2.4, but first we explain how to integrate the Make feature with the programming-in-the-large environment described earlier.

```
Feature Make
Interface:
    Exports all
    Imports Time
Implementation:
    Uses Time
    DU          ::= inputs: seq of OBJECT
                    outputs: seq of OBJECT
                    command: string
    OBJECT      ::= time: TIMESTAMP
                    origin: ORIGIN
                    content: any
    ORIGIN      ::= DU | PRIMITIVE
    PRIMITIVE   ::=
End Feature Make
TIMESTAMP is imported from the Time feature.  string and any are
built-in types.  PRIMITIVE is an atomic value.
```

Figure 2-2: Feature Description for Make

2.3. Merging Make and the Module Interconnection Language

Programming environments are synthesized by *merging* features with other features. The implementation part of a feature may include a Merges clause, which combines external object definitions imported from other features into a synthesized definition. Further, it may give each synthesized object an internal name and it can extend and/or export the synthesized object.

In Figure 2-3, the System Modeller feature merges (1) the DU and MODULE productions and (2) the OBJECT production and the IMPLEMENTATION production, retaining the latter name in each case. This means that the System Modeller includes a MODULE production and an IMPLEMENTATION production that inherit from the merged features. The local MODULE production has all the functionality of the DU production defined in the Make feature, as well as the facilities defined in Module Interconnection Language; additional facilities, in this case the 'objcode' and 'symtab' components, can be added by the merging feature. Similarly, IMPLEMENTATION inherits all the capabilities of OBJECT. This merge clause further defines OBJECTCODE as a new instantiation of OBJECT; it inherits all the capabilities of OBJECT as defined by the Make feature. Thus a module becomes a dependency unit, an implementation becomes an object, and object code is defined as an object.

A more realistic system would merge a large number of such features resembling, for example, Figure 2-4. The larger system includes all the data structures and incorporates all the functionality defined by all the synthesized features.

```
Feature System Modeller

Interface:

    Exports: ...

    Imports: Module Interconnection Language,
             Make

Implementation:

    Merges:

        Feature Module Interconnection Language
        Feature Make
            DU and MODULE as MODULE
            OBJECT and IMPLEMENTATION as IMPLEMENTATION
            OBJECT as OBJECTCODE

    MODULE      ::= objcode: OBJECTCODE
                    symtab: SYMBOLTABLE

    SYMBOLTABLE ::= ...

End Feature System Modeller
```

Figure 2-3: Merging Make and Module Interconnection Language

```
Feature A Larger System

Interface:

    Exports: ...
    Imports: ...

Implementation:

    Merges:

        Feature System Modeller
            ...
        Feature Compilation Unit
            ...
        Feature Documentation Facility
            ...
        Feature My Error Handler
            ...

End Feature A Larger System
```

Figure 2-4: Description of a More Realistic, Larger System

2.4. Equations for Make

Figures 2-5 and 2-6 contain the semantic equations needed to implement Make-like facilities. The details of these equations are not important to our argument. It suffices to understand that these equations are automatically evaluated as needed, and other features synthesized with the Make feature do not have to be concerned with these equations or their operation. This is because the semantics are written as equations rather than as routines. If routines were used, it would be necessary for the implementor to understand the details of the semantic routines provided by all the merged features in order to correctly order their invocations. In contrast, equations permit automatic ordering of evaluation according to the dependencies among the inputs and outputs of equations. In particular, each equation whose output appears as an input to another equation is automatically evaluated before the other equation. The algorithms for implementing this are briefly explained in Section 3.

The rest of this section explains how our equations implement system regeneration after source code changes, while the following section shows how behavior can be tailored to the particular environment by augmenting the set of equations and overriding default equations. The reader can skip to Section 3 without loss of continuity.

Our notation for equations is called *action equations* [23]. Action equations are an extension of attribute grammars [24], which have been applied previously to compiler-compilers [9, 12] and generation of programming environments [33, 20]. While attribute grammars support type checking, code generation, and other programming tools that inspect the source code, action equations can also define dynamic, interactive tools such as interpreters, debuggers, run-time support, *etc.* See [22] for a full treatment of action equations.

```
DU        ::= ...

     Equations:

     MAKE -->

          Propagate MAKE To inputs[all]

          Assert Min(outputs[all].time) > Max(inputs[all].time)
          Exception Propagate APPLY To self

     APPLY -->

          outputs[all].origin := self

          default outputs := Apply(command, inputs)
```

Figure 2-5: Equations for Dependency Units

Figure 2-5 associates four equations with the DU production. All four are attached to *events*, two each to MAKE and APPLY. An event is a named signal that can be sent to objects by the kernel or by other objects. Events are essentially parameterless messages. The kernel automatically sends the corresponding signal whenever a primitive operation (such as CREATE, DELETE, or ACCESS) is performed on an object. Further, one object can send an event to another using the *propagation equation*. The attribute equations attached to a particular event are recalculated when the object receives the matching signal.

The first equation is a propagation equation; a propagation equation sends a named event to one or more destination objects. This equation sends the MAKE event to every member of the sequence of inputs. This implements backtracking by propagating the signal to every input that transitively contributes to the desired output. The second equation is an *assertion*. An assertion causes the kernel to check that a certain condition is true; if the condition is false, an equation is activated to display an error, correct the situation, *etc*. This assertion detects when an input object is more recent than an output object, and propagates the APPLY event to rederive the output objects.

The next two equations are *constraints*; a constraint is essentially an assignment. The left hand side of a constraint addresses an object or a component of an object, while the right hand side is an arbitrary expression. The first equation sets the 'origin' component of each of the resulting outputs to the corresponding DU object. The second sets the value of the 'outputs' component to the result of calling the Apply function with the command string and its arguments. Equations attached to events are re-evaluated exactly once when the event is received. However, the inputs to each equation must be calculated before the equation can be re-evaluated. In this case, the second equation depends on the first, because the 'origin' component of an output object cannot be set until the output itself is available; thus, the kernel automatically evaluates the second equation before the first when the APPLY event is received by a DU object. Notice that the first constraint is qualified with the keyword "default". This means that the equation can be overridden by another equation with the same left hand side; we will explain this in the next section.

```
OBJECT     ::= ...

     Equations:

     MAKE -->

          Assert TypeOf(origin) = "PRIMITIVE"
          Exception Propagate MAKE To origin

     time := content Return Now()
```

Figure 2-6: Equation for Objects

The OBJECT production has two equations, shown in Figure 2-6. The assertion is activated by the MAKE event. It checks whether or not the OBJECT is primitive. If not, it propagates MAKE to the originating dependency unit to rederive the object. The second equation, a constraint, is not attached to an event. Therefore, it constrains its left hand side to always denote the value represented by its right hand side; in particular, the kernel automatically re-evaluates the equation whenever an argument to the right hand side changes in order to update the left hand side. In this case, the 'time' component is updated to the current time whenever the 'content' component changes in value.

A programming environment synthesized from the Make feature and the Module Interconnection Language feature would work as follows when the user gives the Make command. The kernel sends the MAKE event to the module, activating any attached equations. In this example, the only relevant equations come from the DU production, which is merged with the MODULE production in the System Modeller feature. These equations propagate MAKE through the chain of outputs and inputs until arriving at primitive source objects, that is, implementations. In each case where an implementation is more recent than its module's object code, the object code is rederived.

Note that the Make feature is completely generic; in particular, it does not know anything about files. In this example, the input objects were merged with modules and implementations and the output objects with object code, all of which are equivalent to files. However, the Make feature could be merged with other features that define completely different input and output objects. For example, we could merge input objects with some internal representation of arbitrary data structures and the output objects with the windows that display these data structures. Then the same behavior provided by the Make feature would update the windows whenever the data structures changed.

2.5. Tailoring Behavior

In Figure 2-3, the System Modeller feature adds the 'objcode' and 'symtab' attributes to the MODULE production, as places to save the corresponding object code and symbol table, respectively. It is necessary to also add equations that implement this behavior. The System Modeller feature associates three new equations with the MODULE production, as shown in Figure 2.5. All three are constraints. The first equation maintains the new 'objcode' component of the module to be the same as the first element of its 'outputs' component: that is, the object code for the module is the first of the output objects produced by applying the Make command to the module. Similarly, the second equation constrains the new 'symtab' component to be the second output.

```
MODULE  ::= ...
     Equations:
     objcode := outputs[1]
     symtab := outputs[2]
     APPLY -->
          outputs := Smart(command, inputs)
```

Figure 2-7: Augmenting and Overriding Behavior

Suppose we would like to tailor, rather than augment, the behavior defined by the Make feature to the context of the System Modeller. Our goal is to modify Make's processing to implement 'smart recompilation' [37]. Smart recompilation refines the granularity of dependency from the dependencies among inputs defined by dependency units to the dependencies among the source code symbols defined within the input modules. This change applies only to the System Modeller feature, and thus to any programming environments that incorporate this feature, not to all potential applications of Make facilities.

The System Modeller feature implements this change by overriding the default equation defined by the Make feature. The third constraint shown in Figure 2.5 is attached to the APPLY event. It has the same left hand side, the 'outputs' component, as the default equation defined for the DU production by the Make feature (Figure 2-5). Since MODULE has been merged with DU, it inherits all the equations defined for DU in the Make feature, including the default constraint. However, this new equation overrides the default equation, removing it from consideration during the kernel's ordering of equation evaluation. The difference is that the new equation calls the Smart function, whereas the default equation calls Apply. Smart is defined by a set of equations, not shown, that apply the compilation command only if the symbol tables for the input objects actually reference those source code symbols that have changed since the previous compilation. This is why we needed to add the 'symtab' component to save the module's symbol table.

3. Implementation

An earlier version of features was implemented for a Macintosh™ Pascal environment to be marketed commercially within the next year [5]. Routines rather than equations describe behavior and merging of data structures is not supported. This implementation demonstrates the practicality of merging alternative display descriptions [13]. A full implementation is being developed in CommonLoops [3].

This implementation requires translation of structural descriptions and action equations, plus run-time support. Each synthesized data structure combines productions from different features that have been merged together as *facets*, each facet corresponding to one production. Only some facets of an object need be active. Structural descriptions translate easily into corresponding data types. For example, each object could be represented by a record, where each field is a component or a pointer to a component (depending on the type of the component). The difficulty arises in maintaining connections and consistency among the various facets of an object: Auxiliary equations are generated to update certain facets in response to changes in other facets.

The equations for a synthesized data structure are combined and a *local dependency graph* represents all the equations attached to the same event. The vertices represent equations and the edges represent dependencies among the inputs and outputs of equations. Another graph represents all equations (for the same synthesized type) that are not attached to any event. The kernel orders the evaluation of active equations according to these graphs. Each individual equation is translated into an evaluation procedure that takes advantage of the implementation language facilities as well as the kernel primitives.

The kernel provides primitives for creating, destroying and moving among objects. It sends standard events as necessary. It also sends the new events defined in the features by selecting the corresponding local dependency graphs. The kernel's most important job is ordering evaluation of active equations. It uses an adaptation of Reps' incremental attribute evaluation algorithm [32], which generates language-based editors from attribute grammars [33]. The local dependency graphs are combined into a *composite dependency graph* at run-time to reflect the actual connections among objects and facets of objects. The composition considers only the graphs for the current event(s) and those not specific to any event. The graph is sorted topologically to order the evaluation of equations. This algorithm is asymptotically optimal, *i.e.*, linear in the number of affected objects. See [15] and [22] for algorithm details and further complexity results.

4. Related Work

Our approach both extends and unifies work from five major areas: structure-oriented environments, interface description languages, object-oriented programming, abstract data types, and specification languages.

Structure-Oriented Environments:[1] Our work extends current research by synthesizing the language description used to generate an environment from reusable building blocks. In our examples, features extend the notation used for the Display Oriented Structure Editor (DOSE) system [10]. Generation of structure-oriented environments in turns builds on compiler-compilers [19, 9]. Our results apply directly to these areas, since features are not specific to any particular formal notation for syntax description. However, in order to correctly merge semantics processing, the behavior must be described using equations or some similar declarative notation rather than routines; otherwise, the implementor of the structure-oriented environment is forced to combine the semantic routines by hand.

Interface Description Languages: IDL [28] is a formal notation for defining the data structures passed among tools; it grew out of research in compiler-compilers [25]. IDL has been extended to support tight integration among tools while still supporting reuse of tools [35]. Tool behavior is implemented by separately defined routines, permitting only sequential processing of data by tools. In contrast, features support interleaved operation by defining tool processing by equations, where equations for different tools are automatically interleaved, for example, if an equation for one tool depends on an equation for another tool which in turn depends on another equation for the first tool.

Object-Oriented Programming: Other than structure-oriented environments and IDL, our results are closest to object-oriented programming. Our merge clause, the glue that binds features together, is similar to the multiple inheritance of some object-oriented programming languages [36, 3, 27]. There are two important differences between merging and multiple inheritance. First, components with the same name are shared between separately inherited facilities, provided the types are 'compatible' [15]. Second, the behavior is merged without requiring the implementor to deal with the interactions among separately defined behavior; this is possible because the 'methods' are described by equations that are evaluated in the order implied by their dependencies, rather than by procedures that must be invoked in some explicit order.

[1]We use the term 'structure-oriented environment' synonymously with 'language-based editor', 'structure editor-based environment', 'syntax-directed editor', *etc.*

Abstract Data Types: Features resemble the encapsulated abstract data types of modern programming languages [34]. Features are strongly typed, with an interface and an implementation; they are similarly motivated: decomposability, abstraction, information hiding, protection, *etc*. However, abstract data types do not by themselves lead to a high degree of reusability since (a) they are language-dependent, and (b) they can be tailored to a particular context in limited ways — specifically, the subtypes of generic modules can be instantiated by each client. Features, on the other hand, are language-independent[2] and permit more specialization by their clients, with respect to both the data structure and the operations.

Specification Languages: Some specification languages [4, 16, 17] support composition of distinct functionalities in the style of multiple inheritance. However, these languages are oriented towards verification and cannot yet support completely automated translation to an efficient executable form. More significantly, they specify data implicitly, and thus cannot describe sharing. Furthermore, behavior is described axiomatically or algebraically rather than operationally, making it difficult to specify interactive software.

5. Reusability Revisited

Current approaches to software reuse have had relatively little effect on software engineering practice. Subroutine libraries have had the most success. However, subroutine libraries and most other existing approaches are highly tied to linguistic and/or functional context. A software building block can be reused only as the original programmer envisioned. A generic stack module in Ada™ manipulates only Ada stacks. A window manager manages only windows.

There are three important prerequisites to achieving an order of magnitude improvement in software production: (a) language-independence, (b) component reuse through composition, and (c) reuse of components in unanticipated ways. For example, we would like to reuse a window manager as a file system written in a different programming language. We believe this is realistic. A window manager creates and destroys windows, moves windows, defines subwindows, and reads/writes windows; a file system creates and destroys files, renames files, includes files in directories, and reads/writes files. The structures of the two programs are very likely similar, although the devices and implementation languages are quite different.

Our goal is to support this degree of reusability, without sacrificing previously written software. Our approach is broadly based on a framework where software building blocks can be transformed between two forms — programs and language-independent descriptions [18, 38]. 'Old code' is in a particular programming language, but new software could often be written as a language-independent description.

However, there is as yet no acceptable language-independent notation; we cannot transform automatically from programming languages to such a notation; and we cannot transform automatically back to the desired implementation language. All three problems must be solved.

[2]While our notation is itself a tool description language, descriptions of features written in that notation are translated during the environment generation process into some specific executable language (Pascal, C, *etc*.). Features are 'language-independent' in the sense the target language can be virtually any programming language.

We believe that features are a significant contribution to the first and third problems. This derives in part from Notkin's results [29]: He applied environment generation to a wide variety of integrated systems, including mail systems, document editors, and even ordinary programming — even restricting attention to environment generation, we potentially impact general software reusability. More concretely, our confidence comes from our design of several components for a programming-in-the-large environment, including a configuration manager and a module interconnection language with intermodule consistency checking ([14]), and incremental recompilation, interpretation and language-oriented debugging ([22], [23]).

Our research contributes directly to synthesis of programming environments from reusable building blocks. We can now describe abstract units of functionality as features, define features in terms of other features, and combine features with other features that specify both separate and shared components and distinct behaviors for the same objects. We can further compose synthesized features to generate arbitrarily complex programming environments incorporating retailored programming tools.

References

[1] Vincenzo Ambriola, Gail E. Kaiser and Robert J. Ellison.
An Action Routine Model for ALOE.
Technical Report CMU-CS-84-156, Carnegie-Mellon University, Department of Computer Science, August, 1984.

[2] John W. Backus.
The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.
In *International Conference on Information Processing.* 1959.

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

[4] R.M. Burstall and J.A. Goguen.
Putting Theories Together To Make Specifications.
In *Fifth International Joint Conference on Artificial Intelligence*, pages 1045-1058. Cambridge, MA, 1977.

[5] Ravinder Chandhok, David B. Garlan, Dennis Goldenson, Philip L. Miller and Mark Tucker.
Structure Editing-Based Programming Environments: The GNOME Approach.
In *National Computer Conference '85.* July, 1985.

[6] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang.
Programming Environments Based on Structured Editors: The Mentor Experience.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[7] Veronique Donzeau-Gouge, Gilles Kahn. Bernard Lang and B. Melese.
Documents Structure and Modularity in Mentor.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 141-148. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[8] Robert J. Ellison and Barbara J. Staudt.
The Evolution of the GANDALF System.
The Journal of Systems and Software 5(2):107-119, May, 1985.

[9] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4), October, 1984.

[10] Peter H. Feiler and Gail E. Kaiser.
Display-Oriented Structure Manipulation in a Multi-Purpose System.
In *IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 40-48. Chicago, IL, November, 1983.

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

[12] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Pub. Co., New York, NY, 1977.

[13] David Garlan.
Flexible Unparsing in a Structure Editing Environment.
Technical Report CMU-CS-85-129, Carnegie-Mellon University, Department of Computer Science, April, 1985.

[14] David Garlan.
Views for Tools in Integrated Environments.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments*. June, 1986.
Proceedings to appear as a book published by Springer-Verlag.

[15] David Garlan.
Views for Tools in Integrated Environments.
PhD thesis, Carnegie-Mellon University, 198x.
In progress.

[16] Joseph Goguen.
Parameterized Programming.
In *Workshop on Reusability in Programming*, pages 138-150. Newport, RI, September, 1983.

[17] John V. Guttag, James J. Horning and Jeannette M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

[18] Nico Habermann.
Private communication.
May, 1985
Regarding framework for reusable software.

[19] S.C. Johnson and M.E. Lesk.
Language Development Tools.
The Bell System Technical Journal 57(6), July-August, 1978.

[20] Gregory F. Johnson and Charles N. Fischer.
Non-syntactic Attribute Flow in Language Based Editors.
In *Ninth Annual ACM Symposium on Principles of Programming Languages*. January, 1982.

[21] Gregory F. Johnson and C.N. Fischer.
A Meta-Language and System for Nonlocal Incremental Attribute Evaluation in Language-Based Editors.
In *Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 141-151. January, 1985.

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

[23] Gail E. Kaiser.
Generation of Run-Time Environments.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 51-57. Palo Alto, CA, June, 1986.
Proceedings published as *SIGPLAN Notices*, 21(7), July, 1986.

[24] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[25] Bruce W. Leverett, Roderic G.G. Cattell, Steven O. Hobbs, Joseph M. Newcomer, Andrew H. Reiner, Bruce R. Schatz and William A. Wulf.
An Overview of the Production Quality Compiler-Compiler Project.
Technical Report CMU-CS-79-105, Carnegie-Mellon University, Department of Computer Science, 1979.

[26] Raul Medina-Mora.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March, 1982.

[27] David A. Moon.
Object-Oriented Programming with Flavors.
In *ACM Conference on Object-Oriented Systems, Languages, and Applications.* Portland, OR, September, 1986.

[28] John R. Nestor, William A. Wulf and David A. Lamb.
IDL — Interface Description Language: Formal Description.
Technical Report, Software Engineering Institute, Pittsburgh, PA, February, 1986.
Reprint of CMU Technical Report CMU-CS-81-139.

[29] David S. Notkin.
Interactive Structure-Oriented Computing.
PhD thesis, Carnegie-Mellon University, February, 1984.

[30] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[31] Steven P. Reiss.
Graphical Program Development with PECAN Program Development Systems.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 30-41. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[32] Thomas Reps, Tim Teitelbaum and Alan Demers.
Incremental Context-Dependent Analysis for Language-Based Editors.
ACM Transactions on Programming Languages and Systems 5(3):449-477, July, 1983.

[33] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 41-48. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[34] Mary Shaw.
Abstraction Techniques in Modern Programming Languages.
IEEE Software (4):10-26, October, 1984.

[35] Richard Snodgrass and Karen Shannon.
Supporting Flexible and Efficient Tool Integration.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments.* Trondheim, Norway, June, 1986.
Proceedings to appear as a book published by Springer-Verlag.

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

[37] Walter F. Tichy.
Smart Recompilation.
ACM Transactions on Programming Languages and Systems 8(3):273-291, July, 1986.

[38] Mark Tucker.
Private communication.
June, 1985
Regarding engineering of reusable software.

COMPOSING SOFTWARE SYSTEMS FROM REUSABLE BUILDING BLOCKS

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

David Garlan
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213

October 1986

This research 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. Part of this paper was written while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

To appear in Twentieth Hawaii International Conference on System Sciences, Kona, HA, January, 1987.

ABSTRACT

We argue that to achieve an order of magnitude improvement in software production, we need to support software reusability that has three important characteristics: (1) language-independence, (2) support for construction of systems from existing components, and (3) the ability to reuse a component in a way not anticipated by the original programmer. We describe a framework for achieving these three goals. The important components of the framework are *features*, a unit of modularity that can be composed in a manner similar to the multiple inheritance of object-oriented languages and *action equations*, a declarative notation for specifying the behavior of software building blocks.

1. INTRODUCTION

There are three approaches to software reusability that have achieved widespread use. The first and foremost is *subroutine libraries*. Subroutine libraries have had a significant impact on the production of mathematical software systems, and have also been applied successfully to such areas as string manipulation and I/O. However, there are several reasons why subroutine libraries alone are not sufficient to achieve an order of magnitude improvement in software productivity.

One overwhelming problem is that an individual subroutine is simply too small; another way of looking at this problem is that the glue necessary to make large numbers of subroutines work together is too large. Another serious problem with subroutine libraries is they must be written in a particular programming language, which means that decisions about primitive datatypes, constructors for structured datatypes, and subroutine linkages have already been made by the language designer or by the language implementor. A third, related problem is that the subroutines have already been written in a particular programming language, with all the details filled in; it is not possible in general to change the number or types of the parameters or to pick out part of the algorithm encapsulated in the subroutine.

The first and third problems are partially solved by the generic packages of Ada™[1], which can be considered an extension to subroutine libraries. Ada expands the size of the unit of reusability beyond the subroutine level and encapsulates some of the glue among subroutines within the package, so that the glue can also be reused. Generic packages also permit the types of selected parameters to be specified by the application, but do not support changing the number of parameters and do not aid the programmer in specializing algorithms to particular applications.

The second general approach to reusability is *software generation*. This approach has been applied successfully to certain application areas, most notably report generators, compiler-compilers[2] and language-based editors[3]. Software generation meets the important criterion of language independence; in particular, the generator can be changed to produce software in a different implementation language without significantly affecting the input notation. Software generation provides a relatively large unit of reusability, since the code produced is often several orders of magnitude larger than the specification. However, software generation has several serious problems.

The most glaring difficulty is that a new generator must be developed for each application area. This can only be done after the application area is well understood and has become relatively standardized. We do not believe that application of software generation to new areas can keep pace with expansions in software applications. Another serious problem with this approach is that it is unclear how to combine the software systems produced by different generators in meaningful ways. The obvious technique is to make large-scale patches to the generated code, but this may lead to the problem described previously for subroutine libraries: the glue holding things together is larger or more difficult to write than the original input specifications.

The third well-known approach to software reusability is *object-oriented programming*. Examples of object-oriented languages include Smalltalk-80™[4], Flavors[5], Loops[6], CommonLoops[7], C++[8] and Traits[9]. The various notions of inheritance supported by these languages all provide some simple means for composing reusable software building blocks defined as classes. This reusability is relatively flexible since a class may augment and/or substitute for the data structures and/or operations inherited from its superclass(es).

However, this approach is not powerful enough. In addition to augmenting and replacing, we have to consider the problem of combining distinct operations that are provided by different building blocks. Some object-oriented languages address this issue, but their solutions must be limited to explicit ordering because their operations are atomic procedures. In other words, there is no way to interleave methods.* Object-oriented programming also shares the second flaw of subroutine libraries. Like any other programming language, each object-oriented language implies particular decisions regarding the possible implementations of datatypes and operations. This drastically limits, *a priori*, the potential contexts in which a class can be reused.

Although none of these approaches is adequate, all three do (at least) one thing right: they treat reusability as part of the design rather than an afterthought of the implementation. It is not, in general, feasible to decompose an existing software system into reusable software building blocks that can be used to construct other systems. Reusability has to be engineered from the start.[10]

In addition to this characteristic, three other characteristics appear to be crucial in achieving an order of magnitude improvement in software reusability.

- Language-independence is necessary to prevent early implementation decisions that are not relevant to the functionality of a building block;
- Composition of components is necessary to build large software systems with complex functionality;
- Flexibility in tailoring components is necessary to expand reusability beyond the applications anticipated by the implementor of a building block.

*Some existing object-oriented languages do support 'before' and 'after' methods, but it is still impossible to interleave at a finer granularity than full procedures.

We have developed a new approach to reusable software that incorporates all four characteristics sketched above. The basic premise is to combine the advantages of object-oriented programming with the advantages of software generation. The particular way we combine ideas adopted from these approaches, plus new ideas of our own, solves many of the problems of both object-oriented programming and software generation.

However, there are certain problems that we have not (yet) solved. Our approach is no better than existing approaches at aiding programmers in determining whether or not a particular building block is suitable for reuse in a particular application; we do not suggest opportunities for reuse, we just support them. Further, our approach does not address the orthogonal problem of retrieving those existing building blocks suitable for reuse in a particular system.

2. MELD

The underlying basis for our approach consists of (1) an object-oriented notation that is independent of any particular programming language, and (2) a translator for generating executable code from the notation. Reusable software building blocks are written and composed in this notation, which provides flexible means for combining both data structures and algorithms. The translator uses techniques from software generation to produce efficient executable systems.

Our notation, called MELD,** has two essential aspects that are not found in object-oriented programming languages nor in object-oriented specification languages (such as Larch[11] and OBJ[12]). We refer to these aspects as *features* and *action equations*; these are discussed briefly below and are explained in more detail in the following section.

Features are our reusable building blocks. Features are similar to Ada packages in that they separate interface from implementation: features bundle together and provide information hiding for a collection of abstract datatypes. Features are different from packages and modules in the way in which the information exported by a feature can be used by importing features. In particular, features provide a unique mechanism for composing imported facilities <u>with other imported facilities</u> as well as with locally defined facilities. This mechanism, called *merging*, is the key to reusability.

The implementation of a feature normally consists of a collection of abstract datatypes[13]. Abstract datatypes are given as structural descriptions in a notation that provides a very general means for describing data; our notation is language-independent in the sense that it does not make any commitments to a particular concrete representation but can be implemented in terms of any conventional programming language. The abstract datatypes of features are equivalent to the classes of object-oriented languages, but using a language-independent notation based on the Interface Description Language[14, 15] (IDL). When features are merged, the corresponding

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

abstract datatypes are synthesized into composite data structures in the same way that objects consist of instance variables inherited from their defining class and from all its superclasses. Merging resembles the multiple inheritance[16] of some object-oriented languages in that both allow multiple types to be coallesced into a single type, but it differs in that coallescing is defined at the grain size of a feature rather than at the grain size of a single data type (*i.e.*, a class).

The body of a feature associates *action equations****** with each abstract datatype to describe the behavior of instances of the type. Action equations specify (1) the constraints that must hold among data structures, and (2) the dynamic interactions among data structures and between the system and external agents, such as the human user(s). As we will explain later, when features are merged, the corresponding action equations are related by the dependencies among the inputs and outputs of the equations.

Features are implemented by translating the abstract datatypes and action equations into a conventional programming language. The abstract datatypes are translated using established techniques from generation of structure editors,[17] while the translation of action equations is a simple adaptation of algorithms developed for the incremental evaluation of attribute grammars.[18] The implementation is explained in Section 4.

3. EXAMPLE

We now illustrate how reusable software building blocks are written and composed in MELD. First we implement a generic memory manager as a feature. The processing performed by the memory manager is described using action equations. Then we define a second feature, a simple environment for programming-in-the-large, which provides entities for the memory manager to manage. Finally, we merge these two reusable features into a small system.

3.1 A Memory Manager

Suppose that we would like to implement a facility for loading and storing arbitrary entities to disk. Traditionally we might modify the the implementation of the entities themselves to support memory management. Alternatively we might add the facility to do memory management for any data directly to the run-time support of the programming language, perhaps as a generic package. Neither approach induces reusability. In the first case, the memory management is specific to the entities. In the second case, the memory manager is 'reusable' in the same sense that a text editor or a compiler is reusable: 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. Using MELD, however, we take the approach of constructing a reusable software building block that can be incorporated into any system requiring memory management.

***Action equations should not be confused with other 'equations', such as algebraic equations, mathematical equations, *etc.*

The Memory Manager feature 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 together 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 timestamp representing the most recent access to it. This 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 abstract datatypes for our generic memory manager. The description is encapsulated into a feature, which has a name, an interface and an implementation. The interface lists the abstract datatypes exported by the feature in its *exports* clause and lists the other features that are imported in its *imports* clause.

In this case, the MM-ROOT and MM-ENTITY datatypes are exported. The components of MM-ENTITY are entirely hidden, but the 'maxentities' component of MM-ROOT is available to other features that import the Memory Manager feature. Any exported components are listed within the square brackets following the name of their datatype; only the listed components are accessible outside the feature.

```
Feature Memory Manager

Interface:

   Exports MM-ROOT[maxentities],
           MM-ENTITY[]

   Imports Time, DiskIO

Implementation:

   Uses Time, DiskIO

   MM-ROOT ::=
      curid: integer
      maxentities: integer
      inuse: integer
      diskid: DISK-ID
      allentities: set of MM-ENTITY
                   key uniqueid
      loaded: ordered set of MM-ENTITY
              key uniqueid
              ordered low by lastuse

   MM-ENTITY ::=
      uniqueid: integer
      incore: boolean
      lastuse: TIMESTAMP
      diskid: DISK-ID

End Feature
```

1. Feature Description
for a Memory Manager

The implementation part of the feature defines two datatypes: MM-ROOT and MM-ENTITY. The components of these data structures are listed with their types; the action equations that describe the behavior of the root and memory managed entities are given in Figures 2 and 3.

The MM-ENTITY abstract datatype is defined as a *class*, in the sense of the classes of object-oriented languages. It represents the entities managed by the memory manager. It defines four components, or *instance variables* — 'uniqueid', 'incore', 'lastuse' and 'diskid' — which contain the obvious information. Each instance variable is *typed*, and strong typing is enforced. The MM-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. This is explained later on.

The MM-ROOT class defines the memory managed root, which has six instance variables. The two most interesting are 'allentities' and 'loaded'. The 'allentities' instance variable is a *set* of objects, each an instance of the MM-ENTITY class. In MELD, the set constructor guarantees uniqueness and supports access according to a *key*, in this case the 'uniqueid' instance variable of each object. 'allentities' represents the stubs of all memory managed entities, both those that have been loaded into core and those that have not.

The 'loaded' instance variable is an *ordered set* of objects. An ordered set works in the same manner as a set, except that the objects are automatically ordered according to the value of a particular instance variable ('lastuse'). One behavior implemented by the action equations for the Memory Manager feature is to maintain 'loaded' as only those memory managed entities that are currently in core.

3.2 Equations for Memory Manager

The behavior of the generic memory manager is implemented by associating action equations with the MM-ROOT and MM-ENTITY abstract datatypes; the equations appear as *methods* for the corresponding classes.

The first method for MM-ROOT, given in Figure 2, constrains the 'inuse' instance variable to always be equal to the length of the table of loaded entities. This kind of equation ("*<address>* := *<expression>*") is called a *constraint*. A MELD constraint is unidirectional: whenever the length of 'loaded' changes, then the 'inuse' variable is automatically updated but not *vice versa*. The purpose of a constraint is to establish an invariant for all objects defined by the class.

The second method in Figure 2 sets the default value of 'maxentities' to be "100". The *default* is a special form of action equation that can be overridden by other action equations. As we shall see, the notion of defaults and the ability to override defaults them are critical for reusability.

```
MM-ROOT  ::= ...

  Methods:

  inuse := Length(loaded)

  default maxentities := 100

  allentities :=
      View: is(MM-ENTITY)

  loaded :=
      View: is(MM-ENTITY)
            and incore

  CREATE -->
      curid := 1
      diskid := NewDiskID()

  NEWOBJECT -->
      curid := curid + 1

  EXIT -->
      Send STORE To loaded[all]

  Assert inuse <= maxentities
  Exception
      Send STORE To loaded[1]
```

2. Methods for Memory Managed Root

The next two methods are also constraints. On their right hand sides, they illustrate the use of a new mechanism that we call *views*.[19] A view consists of a collection of objects that all satisfy some property, where the specification of the property is given by a *pattern*; that is, views and patterns support associative retrieval. Here the elements of the view for the 'allentities' instance variable are all instances of class MM-ENTITY. The elements of 'loaded' are the subset of memory managed entities whose 'incore' variable is set to "true". The most remarkable property of a view is that its membership is dynamically adjusted as objects are added, deleted, and modified within the system. Garlan's dissertation[20] gives a complete discussion of views and their implementation.

The four methods discussed so far are different than the methods of most object-oriented languages in that none of these methods has a name (also known as a selector). These methods are not triggered by the receipt of a message; they are *permanently active*, and are evaluated as necessary according to the dependencies between their inputs and outputs. When an input to a permanently active method changes in value, the method is automatically re-evaluated to produce a new output. For obvious reasons, there must not be any circularities among the inputs and outputs of permanently active methods.

The next three methods in Figure 2 are closer to the traditional methods of object-oriented languages. In each case, one or more action equations is attached to an *event*. An event is equivalent to the name, or selector, of a method; events are not related in any way to interprocess communication (IPC) mechanisms.**** An event can be sent to an object by the run-time support or by another equation; this is similar to the message passing of object-oriented languages. Events may have parameters, although no parameters are required for our examples. The equations attached to a particular event are evaluated only when the object receives a corresponding message.

The run-time support automatically sends a message to an object whenever any of a collection of primitive operations (such as create, destroy and access) is performed on the object. An equation for one object can send a message to another object using the *send* equation ("Send *<event>* To *<destination(s)>*"). When a new memory managed root is created, the run-time support sends the CREATE event to the root, causing the corresponding method to be evaluated. One constraint initializes 'curid' to "1" and the other sets 'diskid' to the value of the function NewDiskID (imported from the DiskIO feature). These constraints are different from the constraints discussed previously, which were not attached to events, in that they are evaluated only when their event is received. In particular, they are not re-evaluated whenever their arguments change in value — otherwise the second equation would continue re-evaluating itself forever, since NewDiskID returns a different value on each invocation.

Exit is another primitive operation. When the system terminates, the run-time support automatically sends the EXIT event to all the objects it maintains. The EXIT method for the memory managed root sends the STORE event to every entity in the 'loaded' table, causing each entity that is currently in core to be saved on disk. The STORE event does not correspond to a primitive operation; it is defined as a new event by its appearance in the Memory Manager feature.

The NEWOBJECT event is also defined by the implementor. This event is sent by the new entity whenever a new entity is created. The equation increments the value of 'curid' to produce the next unique identifier.

The final method for the MM-ROOT class is called an *assertion*. An assertion ("Assert *<boolean expression>* Exception *<action equation>*") causes the run-time support to check that a certain condition is true. If that condition is ever false, the exception can make repairs, display errors, *etc.* In this case we use an assertion to check whether our system has loaded too many entities into core and, if so, the equation sends the STORE event as needed to store the least recently accessed entities on disk. Since the assertion is not attached to any event, this activity is repeated as necessary to keep the number of loaded entities less than or equal to the value of 'maxentities'.

The methods for the MM-ENTITY class, shown in Figure 3, are similar. The functions of the methods should be self-explanatory.

****However, we are working towards a distributed implementation of action equations based on our previous work in algorithms for distributed evaluation of attribute grammars;[21] a non-distributed algorithm for attribute evaluation has been adapted for our current implementation — see Section 4.

```
MM-ENTITY  ::= ...

  Methods:

  ACCESS -->
      Assert incore
      Exception
          Send LOAD To self
      lastuse := Now()

  CREATE -->
      diskid := Undefined
      uniqueid := ^MM-ROOT.curid
      Send NEWOBJECT To ^MM-ROOT

  DELETE -->
      Assert diskid = Undefined
      Exception
          diskid := FreeDiskID(diskid)

  LOAD -->
      Assert incore
      Exception
          print := "Could not load."
      incore := Load(diskid)

  STORE -->
      Assert diskid != Undefined
      Exception
          diskid := NewDiskID()
      incore := not Store(diskid)
      Assert not incore
      Exception
          print := "Could not store."
```

3. Methods for Memory Managed Object

3.3 A Small Environment for Programming-in-the-Large

Now that we have defined a generic memory manager, we need some entities for it to manage. As part of this example, we describe a small environment for programming-in-the-large as a MELD feature. The environment illustrated in Figure 4 provides modules and implementations as entities to be memory managed. A module consists of either internal modules and/or implementations, plus additional information such as lists of imports and exports. Modules are organized into collections called projects; as we will see, a project corresponds to a memory managed root. The full description of this environment should also contain an associated operational part (for example, methods to check interfaces between modules), but this is not shown here.

```
Feature MDE

Interface:

  Exports all

  Imports Programming Language

Implementation:

  Uses Programming Language

  PROJECT ::=
      proj-name: identifier
      modules: seq of MODULE

  MODULE ::=
      mod-name: identifier
      imports: seq of IMPORT-ITEM
      exports: seq of SIGNATURE
      components: seq of COMPONENT

  IMPORT-ITEM ::=
      identifier

  COMPONENT ::=
      MODULE | IMPLEMENTATION

  IMPLEMENTATION ::=
      signature: SIGNATURE
      body: CODE

End Feature
```

4. Specification of a
Module Description Environment

Notice that the Module Description Environment (MDE) feature is also reusable. The MODULE and IMPLEMENTATION classes can be tailored to the desired programming language by importing and then using a feature that defines the appropriate structures for SIGNATURE and CODE. In the case of Ada, which defines its own module construct, the MODULE class might be merged with the imported PACKAGE class in the manner explained below.

3.4 Merging the Memory Manager and the Module Description Environment

Continuing with our example, we combine the Memory Manager and Module Description Environment features into a small system. In this system, the memory manager will manage modules and implementations of modules. We do this by establishing a connection between the MM-ROOT and PROJECT classes, on the one hand, and between the MM-ENTITY, MODULE and IMPLEMENTATION classes, on the other. Figure 5 illustrates how this is done using MELD.

```
Feature Memory Managed Module
Interface:
  Exports: ...
  Imports: MDE,
           Memory Manager
Implementation:
  Merges:
    Feature MDE
    Feature Memory Manager
    with MM-ROOT as PROJECT
         MM-ENTITY as MODULE,
                      IMPLEMENTATION
  PROJECT ::=
      maxentities: integer
      Methods:
      maxentities :=  200
End Feature
```

5. Merging Memory Manager and
Module Description Environment

A feature may combine a group of imported features. In this case the Memory Managed Module feature imports both the Memory Manager and Module Description Environment features, and the abstract datatypes from the two imported features are *merged* in the implementation. When the MM-ROOT class is merged with the PROJECT class, this means that each instance of the resulting PROJECT class of the Memory Managed Module feature has all the instance variables from both the PROJECT class of the MDE feature and from the MM-ROOT class of the Memory Manager feature. However, the only instance variable from the MM-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 value of 'maxentities' to "200", so our memory manager maintains at most 200 entities in core rather than 100. This capability for overriding defaults makes it easy to tailor features to a wide variety of applications.

The Memory Managed Module system works as follows. Consider the case where a module (or implementation) is accessed by some agent. This would happen, for example, when a human user of the environment for programming-in-the-large tried to read the text of the module. The run-time support sends the primitive ACCESS event to the module, which activates any action equations attached to the ACCESS event for the MODULE class. In this case, the only equations are inherited from the MM-ENTITY class (Figure 3). These equations update 'lastuse' to the current time and check whether the 'incore' instance variable has the value "true"; if not, the run-time support sends the LOAD event 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.

This concludes our example. A real system would probably merge a large number of such features resembling, for example, figure 6.

```
Feature A Larger System
Interface:
  Exports: ...
  Imports: ...
Implementation:
  Merges:
    Feature MDE
      ...
    Feature Memory Manager
      ...
    Feature Compilation Unit
      ...
    Feature Documentation Facility
      ...
    Feature My Error Handler
      ...
End Feature
```

6. Description of a
More Realistic, Larger System

4. IMPLEMENTATION

The implementation of MELD borrows heavily from the software generation approach to software reusability, as described in the introduction. The implementation has four parts: an environment for developing and maintaining MELD descriptions; a translator of structural descriptions; a translator of action equations; and the run-time support. The environment is itself described in MELD and implemented through a bootstrapping procedure. We briefly discuss the other three parts of the implementation here; see Garlan's[20] and Kaiser's[22] dissertations for details and discussion of complexity results. Note that the translation is independent of any particular implementation language.

A data structure in a MELD system is a synthesis of one or more abstract datatype descriptions given in different features that have been merged together. The descriptions themselves are not actually combined in the implementation. Instead, each object has several *facets*, where each facet corresponds to one of its datatype descriptions. This is necessary because only some facets of an object may be active at any given time; this was seen in our memory manager example, where the stub for an object could be loaded and manipulated independently of its content.

It is easy to translate individual structural descriptions into the corresponding datatypes 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 perhaps 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 the run-time support.

Unlike the structural descriptions, the action equations for a synthesized data structure are combined. A *local dependency graph* is constructed that represents all the action equations attached to the same event. 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 all those action equations (for the same synthesized type) that are not attached to any event. These dependency graphs are used by the run-time support to determine the order in which to evaluate active equations.

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

The run-time support provides all the necessary primitives for creating, destroying and accessing objects. It sends standard events as necessary; *e.g.*, the ACCESS event is sent to an object whenever the object is accessed. It also provides primitives to send the new events used in the MELD description and manages a queue of pending events.

The most important job of the run-time support is to order the evaluation of active action equations. This is done using an adaptation of Reps' incremental attribute evaluation algorithm,[18] which was developed for the purpose of generating language-based editors from attribute grammars.[3] The basic idea is that the local dependency graphs are combined into a *composite dependency graph* at run-time to reflect the actual connections among objects. Only the graphs for the current event, plus the graphs that are not specific to any event, 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.

A prototype implementation written in CommonLoops[7] is currently under development.

5. CONCLUSIONS

MELD meets the three fundamental criteria outlined in the introduction, and thus is superior to the three relatively accepted approaches to software reusability that we have discussed. The notation abstracts away from any particular programming language, although almost any language is suitable for implementation. MELD supports composition of components through merging and supports tailoring through renaming and default equations.

MELD is a blend of the object-oriented programming and software generation approaches to reusability that solves most of the significant problems of these two approaches. From software generation, we took the idea of a declarative notation that is independent of any particular programming language but that can be translated into an efficient implementation. From object-oriented programming, we took the concepts of inheritance and of encapsulating behavior with data structures.

To that we add our unique concept of *merging* both data structures and operations. Other object-oriented languages merge data structures, in the sense of inheriting instance variables defined by a superclass, but no other notation supports combination of algorithms on the basis of dependencies.

Our future plans include
- gaining additional experience using MELD;
- developing novel debugging aids for MELD systems;
- implementation of MELD for a multi-processor and/or distributed environment, where different facets may reside on different machines;
- synthesizing multiple visual representations of data based on MELD mechanisms;
- using MELD to build a realistic programming-in-the-large environment that can be augmented and modified by its users.

ACKNOWLEDGEMENTS

We would like to thank Charlie Krueger, David Miller and Benjamin Pierce for their useful criticisms and suggestions regarding a draft of this paper. We would also like to thank Nico Habermann for motivating our interest in reusable software.

REFERENCES

1. United States Department of Defense, *Reference Manual for the Ada Programming Language*, 1983, ANSI/Military standard MIL-STD-1815A

2. Rodney Farrow, ''Generating a Production Compiler from an Attribute Grammar'', *IEEE Software*, Vol. 1, No. 4, October 1984.

3. Thomas Reps and Tim Teitelbaum, "The Synthesizer Generator", *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, Pittsburgh, PA, April 1984, pp. 41-48, Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

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

5. David A. Moon, ''Object-Oriented Programming with Flavors'', *ACM Conference on Object-Oriented Systems, Languages, and Applications*, Portland, OR, September 1986.

6. Mark Stefik and Daniel G. Bobrow, ''Object-Oriented Programming: Themes and Variations'', *AI Magazine*, Vol. 6, No. 4, Winter 1986, pp. 40-62.

7. Danny Bobrow, *et. al.*, ''CommonLoops: Merging Common Lisp and Object-Oriented

Programming'', *ACM Conference on Object-Oriented Systems, Languages, and Applications*, Portland, OR, September 1986.

8. Bjarne Stroustrup, *The C++ Programming Language*, Addison-Wesley Pub. Co., Reading, MA, 1986.

9. Gael A. Curry and Robert M. Ayers, ''Experience with Traits in the Xerox Star Workstation'', *Workshop on Reusability in Programming*, Newport, RI, September 1983, pp. 83-96.

10. Mark Tucker, ''Private communication''. Regarding engineering of reusable software

11. John V. Guttag, James J. Horning and Jeannette M. Wing, ''The Larch Family of Specification Languages'', *IEEE Software*, Vol. 2, No. 5, September 1985, pp. 24-36.

12. Joseph Goguen, ''Parameterized Programming'', *Workshop on Reusability in Programming*, Newport, RI, September 1983, pp. 138-150.

13. Mary Shaw, ''Abstraction Techniques in Modern Programming Languages'', *IEEE Software*, No. 4, October 1984, pp. 10-26.

14. John R. Nestor, William A. Wulf and David A. Lamb, ''IDL — Interface Description Language: Formal Description'', Tech. report, Software Engineering Institute, Pittsburgh, PA, February 1986, Reprint of CMU Technical Report CMU-CS-81-139

15. Richard Snodgrass and Karen Shannon, ''Supporting Flexible and Efficient Tool Integration'', *IFIP WG 2.4 International Workshop on Advanced Programming Environments*, Trondheim, Norway, June 1986, Proceedings to appear as a book published by Springer-Verlag

16. Alan H. Borning and Daniel H. H. Ingalls, ''Multiple Inheritance in Smalltalk-80'', *AAAI-82*, Pittsburgh, PA, August 1982, pp. 234-237.

17. Raul Medina-Mora, *Syntax-Directed Editing: Towards Integrated Programming Environments*, PhD dissertation, Carnegie-Mellon University, March 1982.

18. Thomas Reps, Tim Teitelbaum and Alan Demers, ''Incremental Context-Dependent Analysis for Language-Based Editors'', *ACM Transactions on Programming Languages and Systems*, Vol. 5, No. 3, July 1983, pp. 449-477.

19. David Garlan, ''Views for Tools in Integrated Environments'', *IFIP WG 2.4 International Workshop on Advanced Programming Environments*, June 1986, Proceedings to appear as a book published by Springer-Verlag

20. David Garlan, *Views for Tools in Integrated Environments*, PhD dissertation, Carnegie-Mellon University, 198x, In progress

21. Simon M. Kaplan and Gail E. Kaiser, ''Incremental Attribute Evaluation in Distributed Language- Based Environments'', *5th ACM SIGACT-SIGOPS Symposium on Principles of Distributed Computing (PODC)*, Calgary, Alberta, Canada, August 1986, pp. 121-130, Also available as Department of Computer Science, University of Illinois at Urbana-Champaign, Technical Report UIUCDCS-R-86-1294/UILU-ENG-86-1751, September, 1986

22. Gail E. Kaiser, *Semantics of Structure Editing Environments*, PhD dissertation, Carnegie-Mellon University, May 1985, Technical Report CMU-CS-85-131

MELD: A Declarative Language for Writing Methods

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

David Garlan
Carnegie-Mellon University
Department of Computer Science
Pittsburgh, PA 15213

13 June 1986

This research 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. *This paper was completed while Dr. Kaiser was a Visiting Computer Scientist at the Software Engineering Institute, Carnegie-Mellon University, Pittsburgh, PA.

Final version to appear in **Phoenix Conference on Computers and Communications**, Phoenix, AZ, February, 1987.

Abstract

Object-oriented programs are written as collections of messages. When an object receives a message, the system attempts to find a method with the same name as given in the message. The system queries the class that defines the object. If the class provides a corresponding method, the method is performed. The method may return a value to the sender of the message. It may have side-effects on the local memory of the object. The method may send messages to additional objects as part of its computation. This notion of encapsulating operations, in the form of methods, within the definition of an object is common to essentially all object-oriented programming languages.

Messages and methods are currently written in what is fundamentally a procedural style. A message is a procedure call with several parameters, where one parameter is distinguished as the object to which the message is sent. A method is a procedure: it may return a value, have side-effects, and invoke other procedures by sending messages.

We believe that the object-oriented framework lends itself quite easily to the description of programs in a declarative language. In this paper, we propose a declarative language for writing messages and methods. Our notation retains all the important features of object-oriented programming, but adds a higher level of abstraction to the description of object behavior. Our language, called MELD, is an extension of attribute grammars. Its implementation takes advantage of algorithms developed for incremental attribute grammar evaluation in the context of language-based programming environments.

1. Introduction

In this paper we propose a new approach to the problem of writing methods for object-oriented languages. We are particularly concerned with those object-oriented languages that support multiple-inheritance, although our results apply equally well to languages that support only a single chain of ancestors for each class. Our approach derives from our previous work in the generation of programming environments [10, 12]. We show how Backus-Naur Form (BNF) [2], a standard formal notation for programming language definition, extends quite naturally to the description of classes in an object-oriented framework. We describe a declarative language based on attribute grammars [22] that we have used to specify the semantics of programming languages and programming environments [20, 16]. We demonstrate the applicability of these notations to object-oriented programming through an extended example.

Our declarative language, called MELD, combines an extension of BNF used in programming environment and compiler research [19, 25] with our extended form of attribute grammars [21, 20]. We introduce the basic concepts of MELD in the context of programming environment generation in order to ease the following discussion. We then describe intuitively the ideas behind our application of MELD to object-oriented programming. We give an overview of MELD, followed by an extended example describing the implementation of a small system using MELD. We conclude with declarative notation

A prototype implementation of MELD, written in CommonLoops [3], is currently under development at Carnegie-Mellon University. This paper does not discuss the algorithms used in the implementation; these are explained in [20] and [16].

2. Generation of Programming Environments

In recent years there has been considerable interest in systems that support the automatic generation of programming environments from formal descriptions. Representative examples include the Synthesizer Generator [29], Mentor [8], Gandalf [26] and Pecan [28]. In these systems a language-independent kernel is combined with a language-specific formal description - usually an extended form of BNF - to produce an environment that supports the construction of programs in that specific language. The kernel provides such facilities as a user interface, often in the form of a structure editor, and an interface to the host file system.

More significantly, the kernel maintains a database of program objects. This includes support for the integrity of objects and support for the invocation of operations associated with objects in the database. Program objects are typically represented as abstract syntax trees (ASTs), and the integrity support usually involves preventing modifications that would result in syntactically incorrect program objects. The operations supported by the kernel normally include primitive operations such as create an object of a particular type, destroy a particular object, and move the current focus of attention from one object to another. Additional operations may be defined for a specific programming environment; this is how language-specific type checking, code generation, interpretation, *etc.* are included in a programming environment.

For example, the formal description used for generating a Pascal programming environment would include syntactic definitions of the 'if' statement and the '=' expression similar to the illustration in Figure 2-1. These object definitions are called *productions*, after the productions of formal grammars. The IF production provides the knowledge needed by the kernel to create an IF object; the production would be augmented with pretty-printing information [13] to display an IF object. The IF and = objects are shown in Figure 2-2.

```
IF              ::= condition: EXPRESSION
                    thenpart: STATEMENT
                    elsepart: STATEMENT

=               ::= operand1: EXPRESSION
                    operand2: EXPRESSION

{* The names of built-in productions
   appear in bold italics. *}
EXPRESSION      ::= = | + | and | identifier ...

STATEMENT       ::= IF | WHILE | BEGIN-END | WITH ...
```

The IF production indicates that an IF object consists of three components, where the first is an EXPRESSION and the second and third are STATEMENTs. The = production specifies that an = object consists of two components, both EXPRESSIONs. EXPRESSION and STATEMENT are each defined by a *union*, which is a list of alternative productions.

Figure 2-1: Portion of Formal Syntax Definition for Pascal

In addition to this kind of syntactic information, the formal definition of a programming

```
if $condition
    then $thenpart
    else $elsepart

if $operand1 = $operand2
    then $thenpart
    else $elsepart
```

The user moves the focus of attention to the "$condition" component and requests the create operation with the "=" argument. The kernel creates a new = object with two components and inserts it in place of the "$condition" placeholder. If the user had supplied the "while" argument rather than the "=" argument in this situation, the kernel would have displayed an error message instead of performing the invalid operation.

Figure 2-2: Consecutive Displays of Pascal IF Statement

language also describes the semantics the language. The definition would include all the information required for type checking, code generation, run-time support for compiled code, interpretation, debugging support, and the other tools desired for the programming environment. Part of the semantics description can be written as an *attribute grammar* [22]. Reps was the first to apply attribute grammars to the generation of programming environments [6], and this approach has been adopted by other researchers [18, 7]. Figure 2-3 shows part of the attribute grammar for the Pascal 'if' statement followed by a stylistic variant of the same semantic rule. This semantic rule is applied in Figure 2-4.

In addition to type checking, attribute grammars have been applied to most other phases of compilation. They have proved very useful in the context of compiler-compilers as well as the generation of programming environments. MUG2 [11] and Linguist [9] are two representative examples of systems that have used attribute grammars to generate production-quality compilers.

However, attribute grammars alone are not sufficient for the generation of programming environments that provide dynamic tools such as interpreters and run-time support environments. Attribute grammars are not suited to the generation of such tools because of the inherently static nature of the attributes. Attribute grammars support attribute equations that *derive* the values of attributes from the components of the program and from the values of other attributes. The value of an attribute changes only if the relevant portions of the program change. It is not possible for an attribute value to reflect the history of program execution.

Because of this limitation of attribute grammars, some researchers have resorted to implementing the dynamic aspects of programming environments as collections of procedures [23, 1]. Other researchers limit their programming environments to the static tools that can be specified in the formal attribute grammar notation [29].

We have solved this problem by developing an extended form of attribute grammars, called *action equations* [21], that support the dynamic as well as the static aspects of programming environments. We have embedded traditional attribute grammars in an event-driven architecture, where *events* may represent external activities such as a user providing input for an

```
IF        ::= condition: EXPRESSION
              thenpart: STATEMENT
              elsepart: STATEMENT
              error: string

    error := if condition.type = "boolean" then ""
              else "<-- type error"


IF        ::= condition: EXPRESSION
              thenpart: STATEMENT
              elsepart: STATEMENT

    {* print is a built-in attribute implicitly
       defined as a component of every production. *}
    Assert          condition.type = "boolean"
    Exception       print := "<-- type error"
```

In the first description, the error attribute is added as a fourth component of the IF object. The value of the error attribute is defined by an attribute equation. In the second description, syntactic sugar is provided for this common case.

Figure 2-3: Portions of Two Attribute Grammars for Pascal

```
if $operand1 + $operand2   <-- type error
     then $thenpart
     else $elsepart
```

If the type attribute of the condition component has the value "boolean", then the value of the error attribute is the empty string; otherwise, this attribute is the string "<-- type error". The value of the type attribute is calculated by another attribute equation, not shown.

Figure 2-4: Display of Erroneous Portion of Pascal Program

executing program. Additional events may be generated internally as a result of events initiated externally.

Action equations work as follows. When an object receives a particular event, the attribute equations attached to that event are activated; at all other times, these equations are passive and are not considered during the attribute re-evaluations triggered by changes in the program. Attribute equations that are not attached to any event are always active, and correspond exactly to the attribute equations of attribute grammars. Figure 2-5 illustrates the action equations that describe the interpretation of the 'if' statement.

```
IF      ::= condition: EXPRESSION
            thenpart: STATEMENT
            elsepart: STATEMENT

    RUN -->
        Send RUN To condition

    RUN On condition -->
        Send RUN To
            if condition.value = true
                then thenpart
                else elsepart
```

The user requests the **run** operation when the focus of attention is an IF object, causing the kernel to send the RUN event to the IF object. This activates the *send* equation attached to the event, which in turn sends the RUN event to the **condition** component of the IF object. When the RUN event is received by the **condition** component, the **value** attribute of the component is set by an attribute equation (not shown) that calculates the value of this attribute. The send equation shown remains pending until the value attribute becomes available, and then sends the RUN event to either the **thenpart** component or the **elsepart** component, according to the value of the **value** attribute. Execution continues with the selected statement.

Figure 2-5: Portion of Action Equation Description for Pascal

3. Object-Oriented Programming

Programming environment generation systems have much in common with object-oriented programming systems such as Smalltalk80 [17]. With slight modifications, the formal description of program entities can be seen as a collection of object definitions. Productions correspond to the classes of object-oriented languages, components to instance variables, and unions to superclasses. The operations associated with program entities can be viewed as methods and the events correspond to messages. The language-independent kernel becomes an object-oriented kernel supporting an object-oriented database with facilities for browsing, instance creation and deletion, and the invocation of methods. (These correspondences will be made clearer in the following section.)

We believe there are many advantages to treating programming environment generation as a form of object-oriented programming. The most important advantage is the new ability to support *inheritance*, considered by many to be the essence of object-oriented programming. Inheritance in the context of programming environment generation behaves as follows. When a production is a member of a union, it inherits all the operations defined for the union. In this way, we can define the general behavior required for Pascal expressions once, with the EXPRESSION union, rather than repeating the definition for the = production, the + production, *etc.* We can extend the notion of union to permit the list of alternatives to include both productions and other unions. Then we can define the = and AND productions as members of the BOOLEAN-EXPRESSION union, the BOOLEAN-EXPRESSION union as a member of the

EXPRESSION union and the EXPRESSION union as a member of the root OBJECT union. The OBJECT union would define the common behavior of all Pascal language constructs. For example, the generic create operation with its syntactic validity checking would be associated with the OBJECT union and inherited by all Pascal program objects. We have developed a programming environment for Pascal, called MacGnome [4], following this object-oriented approach.

We also believe that object-oriented programming can benefit from recent research in programming environment generation. In particular, we believe that the idea of describing object behavior in a formal, declarative notation can contribute substantially to the ease of developing and maintaining object-oriented programs. We propose that current methodology of writing methods as procedures be replaced with the description of methods as event-driven constraint satisfaction.

In this scenario, the messages of object-oriented programming have the effect of sending an event. The name of the event is given in the message. When a message is received by an object, the corresponding class (a production in the language description terminology) is queried for a corresponding method. If there is an event matching the name given in the message, then the equations attached to that event become active. The active equations are evaluated in the order implied by the the dependencies among these equations. When a send equation is evaluated, it generates a message with the event given in the equation and sends the message to the destination(s) given in the equation.

We have developed a declarative language, called MELD, for writing object-oriented programs. MELD is an extension of our notation for writing language descriptions in the context of programming environment generation. MELD includes notation for describing classes, including instance variables, methods and superclasses. It also provides a modularity construct, called the *feature*, to bundle together related class definitions and support information hiding. In the next section, we give an overview of MELD. The following section demonstrates the advantages of MELD using an extended example.

4. MELD Overview

MELD[1] is an object-oriented declarative language for writing object-oriented programs. These programs are constructed from software building blocks called *features*, MELD's unit of modularity. Each feature implements a basic unit of functionality such as a menu package, a window manager, an incremental recompilation facility, or an error handler. Features can be combined with other features to produce larger systems that merge their capabilities. Summarized briefly, the most important aspects of MELD are these:

- Each feature has an implementation consisting of a collection of classes and an interface the exports some of these classes and imports other features. Other object-oriented languages such as Flavors [24] provide bundling constructs based on the Lisp packages, but do not enforce interfaces.

[1]According to the dictionary, the word "meld" is a combination of "melt" plus "weld"; thus, meld means merge. Meld also stands for Multiple Elucidation of Language Descriptions - suggested by David Barstow.

- Multiple inheritance is used to combine the class descriptions provided by one feature with those provided by others. This method of combination produces composite object definitions. The various contributing features act like Flavors mixins.
- Methods are written as systems of constraints on the values of instance variables. The most important advantage of this style of notation is that it allows the system to automatically derive dependencies and ordering relationships among methods, so that a client of a feature need not bother with the implementation of a feature in order to integrate it with mutually interacting features.
- Instance variables are strongly typed, as in Traits [5]. The system provides a rich collection of base types (integer, boolean, string, text, *etc.*) and type constructors (sequence, set, ordered-set, array, *etc.*). Most other object-oriented languages provide a single constructor such as an array; for example, Smalltalk80 supports indexed instance variables.

5. MELD: An Example

We now illustrate how object-oriented programs are written in MELD. First we describe a generic memory manager using MELD. The processing performed by the memory manager is described using our declarative notation. Then we define a simple environment for programming-in-the-large that provides entities for the memory manager to manage. Finally, we combine these two features into a small system. The discussion that follows is informal; a more detailed account of the notation is given in [14].

5.1. A Memory Manager

Suppose that we would like to implement a facility for loading and storing arbitrary entities to disk. Traditionally we might add this capability to the implementation of the entities. Alternatively we might add the facility to do memory management directly to the kernel. Using MELD, however, we take the modular approach: we define a feature that implements the memory manager; this feature can then be combined with other features that implement the entities to be managed.

The Memory Manager feature 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 together 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 timestamp representing the most recent access to it. This 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 root keeps track of the next available uniqueid, its own disk location for when the system is not in operation, and a table containing the stubs for all the memory managed entities. The root knows how many entities are currently loaded and maintains a table of loaded entities sorted by their timestamp. We assume for simplicity that the default memory management policy is to allow a maximum of *n* memory managed objects in core at one time; as new objects are accessed the least recently accessed objects will be stored on disk to make room. Later we will see how this policy can be tailored to meet the specific management policies of a system in which the Memory Manager feature is used.

Figure 5-1 gives the syntax description portion of our generic memory manager. The description is encapsulated into a feature, which has a name (Memory Manager), an interface and an implementation. 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 MM-ROOT and MM-ENTITY classes are exported. The instance variables of MM-ENTITY are entirely hidden, but the 'maxentities' instance variable of the MM-ROOT class is available to other features that import the Memory Manager feature. The exported instance variables are listed within the square brackets ("[]") following the name of their class; only the listed instance variables are accessible outside the feature.

```
Feature Memory Manager

Interface:

    Exports MM-ROOT[maxentities], MM-ENTITY[]

    Imports Time, DiskIO

Implementation:

    Uses Time, DiskIO

    {* set of and ordered-set of define
       constructors. *}
    MM-ROOT       ::= curid: integer
                      maxentities: integer
                      inuse: integer
                      diskid: DISK-ID
                      allentities: set of MM-ENTITY
                                   key uniqueid
                      loaded: ordered-set of MM-ENTITY
                              key uniqueid
                              ordered low by lastuse

    MM-ENTITY     ::= uniqueid: integer
                      incore: boolean
                      lastuse: TIMESTAMP
                      diskid: DISK-ID

End Feature Memory Manager
```

Figure 5-1: Feature Description for a Memory Manager

The Memory Manager feature imports two other features, Time and DiskIO, which provide the functionalities implied by their names. The implementation part of the Memory Manager feature lists these two imported features in its *uses* clause. Our uses clause acts like the Ada^{TM} use statement [27]: it has the effect of opening the scopes of the listed features, so the classes exported by Time and DiskIO can be named directly, without an access path. In this case, the TIMESTAMP class is imported from Time and the DISK-ID class is imported from DiskIO. If the Time feature appeared in the imports clause, but not in the uses clause, then it would be necessary to give the pathname Time.TIMESTAMP to refer to the TIMESTAMP class.

In addition to the uses clause, a feature implementation may define any number of classes and unions. The Memory Manager implementation defines two classes: MM-ROOT and MM-ENTITY. The instance variables of these classes are listed, with their types, in Figure 5-1; the methods for the MM-ROOT and MM-ENTITY classes are given in Figures 5-2 and 5-3, respectively.

The MM-ENTITY class represents the entities managed by the memory manager. It defines four instance variables, 'uniqueid', 'incore', 'lastuse' and 'diskid', which contain the obvious information. Each instance variable is *typed*. 'uniqueid' is an instance of the built-in class ***integer***, 'incore' is an instance of the built-in class ***boolean***, and 'lastuse' and 'diskid' are instances of classes imported from the Time and DiskIO features. Notice that the MM-ENTITY class represents only the stub for the entity; there are no instance variables representing its content. 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. This is explained later on.

The MM-ROOT class defines the memory managed root. It has six instance variables. The two most interesting instance variables are 'allentities' and 'loaded'. 'allentities' is a *set* of objects, where each of these objects is an instance of the MM-ENTITY class. In MELD, the set constructor guarantees uniqueness and supports access according to a *key*. In this case, the key is the 'uniqueid' instance variable of MM-ENTITYs. As we will see later on, 'allentities' represents the stubs of all memory managed entities, both those that have been loaded into core and those that have not.

'loaded' is an *ordered-set* of objects. An ordered-set works in the same manner as a set, except that the objects are automatically ordered according to the value of a particular instance variable, in this case the 'lastuse' instance variable of MM-ENTITYs. One behavior implemented by the methods for the Memory Manager feature is to maintain 'loaded' to include only the memory managed entities that are currently in core. These methods could use the notation "loaded[1]" to access the least recently used entity that is currently in core and "loaded[last]" to access the most recently used entity.

The methods defined for the class MM-ROOT are given in Figure 5-2. The first of these is an equation that causes the value of the 'inuse' instance variable to be the length of the table of loaded MM-ENTITYs, where **Length** is a primitive provided by each MELD constructor. This kind of equation ("*<address>* := *<expression>*") is called a *constraint*. It constrains the 'inuse' variable to be the same value as the length of the 'loaded' table. The constraint is unidirectional: whenever the length of 'loaded' changes, then the 'inuse' variable is automatically updated. The purpose of a constraint is to establish an invariant for all objects defined by the class (this can also be done procedurally, as in the active values of Loops [31]).

The second method in Figure 5-2 sets the default value of 'maxentities' to be "100". The *default* is a special form of action equation that can be overridden by other action equations. A default constraint is applied only if no other constraint in the final system sets the value of the instance variable on its left hand side.

```
MM-ROOT      ::= ...

     Methods:

     {* 'inuse' is the length of
        the table of loaded objects. *}
     inuse := Length(loaded)

     {* Set the default value for
        maximum number of loaded objects. *}
     default maxentities := 100

     {* Set up 'allentities' and 'loaded'
        instance variables as views.
        Views are subsets of the collection of
        objects maintained by the kernel. *}
     allentities := View: is-prod(MM-ENTITY)

     loaded := View: is-prod(MM-ENTITY) and incore

     {* Initialization. *}
     CREATE --> curid := 1
                diskid := NewDiskID()

     {* Increment uniqueid counter
        when a new object is created. *}
     NEWOBJECT --> curid := curid + 1

     {* On terminating system execution,
        store all loaded objects. *}
     EXIT --> Send STOREYOURSELF To loaded[all]

     {* If 'loaded' table overflows,
        store the least recently used object. *}
     Assert          inuse <= maxentities
     Exception       Send STOREYOURSELF To loaded[1]
```

Figure 5-2: Methods for Memory Managed Root

The next two methods (with the 'allentities' and 'loaded' instance variables) are also constraints. On their right hand sides, they illustrate the use of a new mechanism that we call *views* [15]. A view consists of a collection of objects (defined by classes), all of which satisfy some property. The specification of the property is given by a *pattern*. Here, for example, the elements of the 'allentities' instance variable are those objects of class MM-ENTITY. The elements of 'loaded' are those objects that are MM-ENTITYs and also have their 'incore' variable set to "true". The form of the collection given by the view is determined by the type of the instance variable (respectively, set and ordered-set). The most remarkable property of a view is that its membership is dynamically adjusted as objects are added, deleted, and modified within the system. See [16] for a complete discussion of views and their implementation.

The four methods discussed so far are different than the methods of most object-oriented languages in that none of these methods has a name (a name is sometimes called a selector in the literature). These methods are not triggered by the receipt of a message; they are permanently active, and are evaluated as necessary according to the dependencies between their right hand sides (arguments or inputs) and left hand sides (outputs). When an argument to a permanently active method changes in value, the method is automatically evaluated to produce a new value for its output instance variable. For obvious reasons, there must not be any circularities among the inputs and outputs of permanently active methods.

The next three methods in Figure 5-2 are closer to traditional methods. In each case, one or more equations is attached to an *event*. As explained earlier, an event corresponds to the name of a method; an event can be sent to an object by the kernel or by another equation as the name (or selector) part of a message. The equations attached to a particular event are evaluated only when the object receives a message with the matching name. Thus the equations associated with an event implement the method whose name is given by the event.

The kernel automatically sends a message to an object whenever any of a collection of primitive operations (such as create, destroy and access) is performed on the object. An equation for one object can also send a message to another object using the *send* equation ("Send *<event>* to *<destination(s)>*"). When a new memory managed root is created, the kernel sends the CREATE event to the root. This causes the corresponding method to be evaluated. The method consists of two constraints: one initializes 'curid' to "1" and the other sets 'diskid' to the value of the function NewDiskID. These constraints are different from the constraints discussed previously, which were not attached to events. These two constraints are evaluated only when their event is received. In particular, they are not re-evaluated whenever their arguments change in value (otherwise the second equation would continue re-evaluating itself forever, since NewDiskID returns a different value on each invocation).

exit is another primitive kernel operation. When the system terminates, the kernel automatically sends the EXIT event to all the objects it maintains. Most objects do not have a method for the EXIT event, and so do nothing in this situation. The MM-ROOT class does define a method for the EXIT event, so this method is performed when the memory managed root receives the EXIT event. The method sends the STOREYOURSELF event to every entity in the 'loaded' table, causing each memory managed entity that is currently in core to be saved on disk. The STOREYOURSELF event does not correspond to a primitive kernel operation: it is implicitly defined as a new event by its appearance in the Memory Manager feature.

The NEWOBJECT event is also defined by the implementor. As we will see later, this event is sent (by the new entity) whenever a new entity is created. The equation that implements the method increments the value of 'curid' to produce the next unique identifier.

The final method for the MM-ROOT class is called an *assertion.* An assertion ("Assert *<boolean expression>* Exception *<action equation>*") causes the kernel to check that a certain condition is true. If that condition is ever false, the exception can correct the situation, display an error, *etc.* In this case we use an assertion to check that our system has not loaded too many entities into core. If such a condition is detected, the equation sends the STOREYOURSELF event as needed to store the least recently accessed MM-ENTITYs to disk. Since the assertion is not attached to any event, this activity is repeated as necessary to keep the number of loaded entities less than or equal to the value of 'maxentities'.

The methods for MM-ENTITY, shown in Figure 5-3, are similar. The function of the methods should be self-explanatory. The equations involve calls to several functions provided externally, namely Load, Store, NewDiskID, FreeDiskID and Now. NewDiskID and FreeDiskID are made available by virtue of the importation of the DiskIO feature.[2] The most important of these are Load and Store, which are implemented as kernel primitives that respectively load and store objects to disk. Now is also a kernel primitive.

5.2. A Small Environment for Programming-in-the-Large

Now that we have defined a generic memory manager, we need some entities for it to manage. As part of this example, we use MELD to describe a small environment for programming-in-the-large. The environment illustrated in Figure 5-4 provides modules and implementations as entities to be memory managed. A module consists of internal modules and implementations, plus additional information such as lists of imports and exports. Modules are organized into collections called projects; as we will see, a project corresponds to a memory managed root. The full description of this environment should also contain an associated operational component (for example, methods to check interfaces between modules), but this is not shown here.

5.3. Merging the Memory Manager and the Module Description Environment

Continuing with our example, we need to combine the Memory Manager and Module Description Environment features into a system. In this system, the memory manager will manage modules and implementations of modules. We do this by establishing a connection between the MM-ROOT and PROJECT classes, on the one hand, and between the MM-ENTITY, MODULE and IMPLEMENTATION classes, on the other. Figure 5-5 illustrates how this is done using MELD.

A feature may combine a group of other features. The features are imported in the interface part of the feature. In this case the Memory Managed Module feature imports both the Memory Manager and Module Description Environment features, so it is a *client* of both of these features. The classes from the two imported features are *merged* in the implementation part of the Memory Managed Module feature. When the MM-ROOT class is merged with the PROJECT class, this means that the PROJECT class of the Memory Managed Module feature has all the instance variables from both the PROJECT class of the Module Description Environment feature

[2]Features can export functions as well as classes. Functions are defined as collections of action equations and auxiliary instance variables. These issues are not discussed further in this paper.

```
MM-ENTITY     ::= ...

     Methods:

     {* When an entity is accessed,
        load it, if necessary, and update timestamp. *}
     ACCESS -->  Assert       incore
                 Exception    Send LOADYOURSELF To self
                 lastuse := Now()

     {* When an entity is created assign it a disk
        location and a unique identifier.
        Send the NEWOBJECT event to the closest ancestor
        of the MM-ENTITY object that is an instance
        of the MM-ROOT class. *}
     CREATE -->  diskid := UndefinedDiskID
                 uniqueid := ^MM-ROOT.curid
                 Send NEWOBJECT To ^MM-ROOT

     {* When an entity is deleted free its disk space. *}
     DELETE -->  Assert       diskid = UndefinedDiskID
                 Exception    diskid := FreeDiskID(diskid)

     {* Load an entity from disk.  The Load function
        returns a boolean indicating success. *}
     LOADYOURSELF --> Assert          incore
                      Exception       print := "Could not load."
                      incore := Load(diskid)

     {* When an entity is stored, get a new
        disk location if necessary and
        store the entity on disk.
        Store returns a boolean indicating success. *}
     STOREYOURSELF --> Assert         diskid != UndefinedDiskID
                       Exception      diskid := NewDiskID()
                       incore := not  Store(diskid)
                       Assert         not incore
                       Exception      print := "Could not store."
```

Figure 5-3: Methods for Memory Managed Object

and from the MM-ROOT class of the Memory Manager feature. However, the only instance variable from the MM-ROOT class that can actually be accessed by the client is 'maxentities', since it is the only instance variable exported by Memory Manager. In this case, the client defines a method that overrides the default value of "100" and changes the value of 'maxentities' to "200".

```
Feature Module Description Environment

Interface:

    {* Export all the classes with
       all their instance variables. *}
    Exports all

    {* Imports feature(s) defining
       the desired programming language. *}
    Imports Programming Language

Implementation:

    Uses Programming Language

    {* 'seq of MODULE' indicates a sequence of
        MODULE objects. *}
    PROJECT          ::= proj-name: identifier
                         modules: seq of MODULE

    MODULE           ::= mod-name: identifier
                         imports: seq of IMPORT-ITEM
                         exports: seq of SIGNATURE
                         components: seq of COMPONENT

    IMPORT-ITEM      ::= identifier

    COMPONENT        ::= MODULE | IMPLEMENTATION

    {* SIGNATURE and CODE are imported from the
       Programming Language feature. *}
    IMPLEMENTATION ::= signature: SIGNATURE
                         body: CODE

End Feature Module Description Environment
```

Figure 5-4: Specification of a Module Description Environment

The Memory Managed Module system works as follows. Consider the case where a module (or implementation) is accessed by some external agent. This would happen, for example, when the user of the environment for programming-in-the-large tried to read the text of the module. The kernel sends the primitive ACCESS event to the MODULE object. This activates any the ACCESS method, that is, the action equations attached to the ACCESS event. In this case, the only equations are inherited from the MM-ENTITY class; these equations are repeated in Figure 5-6. The assertion checks whether the 'incore' instance variable has the value true; if not, the kernel sends a message with the LOADYOURSELF event to self, meaning the MODULE object. In any case, 'lastuse' is updated to the current time.

```
Feature Memory Managed Module

Interface:

    Exports: ...

    Imports: Module Description Environment,
             Memory Manager

Implementation:

    Merges:

        {* 'MM-ROOT' is equated with 'PROJECT' and
           'MM-ENTITY' is equated with both 'MODULE'
           and with 'IMPLEMENTATION'.  *}
        Feature Module Description Environment
        Feature Memory Manager
            with MM-ROOT as PROJECT
                 MM-ENTITY as MODULE, IMPLEMENTATION

    {* Only the 'maxentities' instance variable of the
       'PROJECT' class is exported by Memory Manager,
       and thus can be referred to here. *}
    PROJECT     ::= maxentities: integer

        Methods:

            {* The value of 'maxentities' is
               changed from 100 to 200. *}
            maxentities :=  200

End Feature Memory Managed Module
```

Figure 5-5: Merging Memory Manager and Module Description Environment

If a MODULE (or IMPLEMENTATION) object receives a message with the LOADYOURSELF event, the two equations inherited from the MM-ENTITY class are activated; these equations are also repeated in Figure 5-6. The constraint is evaluated first, since the assertion is dependent on the value of 'incore'. Remember that active action equations are always evaluated in the order implied by the dependencies among their inputs and outputs. The constraint calls the kernel Load function to load the entity from the disk location indicated by the 'diskid' instance variable. If the load fails for some reason, such as the entity not being found at the disk location, then the subsequent evaluation of the assertion and its exception causes an error message to be printed.

Recall that a newly loaded module is automatically added to the 'loaded' instance variable of

```
MM-ENTITY          ::= ...

     ACCESS -->  Assert        incore
                 Exception     Send LOADYOURSELF To self
                 lastuse := Now()

     LOADYOURSELF -->  Assert          incore
                       Exception       print := "Could not load."
                       incore := Load(diskid)
```

Figure 5-6: Some Methods for MODULEs and IMPLEMENTATIONs

the PROJECT object, because of the method inherited from MM-ROOT (repeated in Figure 5-7). The 'inuse' variable is constrained to be the length of 'loaded' because of another method inherited from MM-ROOT, also repeated. The assertion is automatically evaluated if the value of 'inuse' changes. If the new value of 'inuse' is greater than 'maxentities', the exception sends the STOREYOURSELF event to the least recently accessed entity in the 'loaded' table. This causes this module or implementation to be stored on disk, after which it is automatically removed from 'loaded'.

```
MM-ROOT            ::= ...

     loaded := View: is-prod(MM-ENTITY) and incore

     inuse := Length(loaded)

     {* 'maxentities' is actually one greater than
        the number of entities that may (transiently)
        be in core at the same time. *}
     Assert           inuse <= maxentities
     Exception        Send STOREYOURSELF To loaded[1]
```

Figure 5-7: Some Methods for PROJECTs

6. Conclusion

This behavior demonstrates the most important advantage of using action equations rather than procedures to write methods. When a message is received, all of the MELD equations attached to the corresponding event become active. This is in addition to those equations that are not attached to any event, and are thus permanently active. These equations are evaluated in the order implied by the dependencies among the equations. Thus the Memory Manager feature can describe the behavior for memory managed entities without concern for the specific behaviors defined for the particular entities by other features. The implementor of these other features can write their behaviors without concern that their methods will mask or interfere with the methods that perform memory management. (This is also the goal of the form of multiple inheritance proposed by Snyder [30].) For example, the implementor of the Module Description Environment feature can define a method for the ACCESS event knowing that this method will not block any methods defined by the Memory Manager feature for the ACCESS event.

This would not be possible in an object-oriented language where methods were defined procedurally. Most such languages require that each class provide at most one method for each relevant message. If a class defines a method with a particular name, it automatically overrides any inherited methods with the same name. If a class does not define a method with a particular name, a search strategy is applied to the superclasses to select exactly one matching method to apply. This does not permit the independent development of different behaviors in response to the same message.

Some object-oriented languages do allow multiple methods to be invoked for the same message. In these languages, the problem arises as to the order in which the various procedures should be invoked. Three solutions to this problem have been proposed: (1) require the class to explicitly invoke the desired methods inherited from superclasses, as in Loops; (2) require the class to define a primary method and require the superclasses to explicitly state whether their auxiliary methods are applied before or after the main method, as in Flavors; and (3) to define a general search strategy, such as apply the methods in the order they are found in a depth-first (or breadth-first) search of the hierarchy of classes and their superclasses. The first solution is completely general, but requires intimate knowledge of the methods provided by superclasses. The second and third solutions are not sufficient for all cases.

Our declarative notation for methods provides a solution to this problem that will in fact handle all situations. We apply all the methods that match the event given in the message. The equations that implement these methods are invoked in the order implied by the dependencies among the equations. This solution is feasible only for equations, not for procedures, because it is in general impossible to mechanically determine the dependencies among procedures.

There are situations when it is desirable for a class to provide methods that override the corresponding methods defined by its superclasses. MELD provides the default action equation for these situations. The feature that defines a method determines whether or not it is reasonable for the method to ever be overridden; if so, it is defined as a default.

References

[1] Vincenzo Ambriola, Gail E. Kaiser and Robert J. Ellison.
An Action Routine Model for ALOE.
Technical Report CMU-CS-84-156, Carnegie-Mellon University, Department of Computer Science, August, 1984.

[2] John W. Backus.
The Syntax and Semantics of the Proposed International Algebraic Language of the Zurich ACM-GAMM Conference.
In *International Conference on Information Processing.* 1959.

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

[4] Ravinder Chandhok, David B. Garlan, Dennis Goldenson, Philip L. Miller and Mark Tucker.
Structure Editing-Based Programming Environments: The GNOME Approach.
In *National Computer Conference '85.* July, 1985.

[5] Gael Curry, Larry Baer, Daniel Lipkie and Bruce Lee.
Traits: An Approach to Multiple-Inheritance Subclassing.
In *SIGOA Conference on Office Information Systems.* April, 1982.

[6] Alan Demers, Thomas Reps and Tim Teitelbaum.
Incremental Evaluation for Attribute Grammars with Applications to Syntax-directed Editors.
In *Eighth Annual ACM Symposium on Principles of Programming Languages.* January, 1981.

[7] Alan Demers, Anne Rogers and Frank Kenneth Zadeck.
Attribute Propagation by Message Passing.
In *SIGPLAN '85 Symposium on Language Issues in Programming Environments*, pages 48-59. Seattle, WA, June, 1985.
Proceedings published as *SIGPLAN Notices*, 20(7), July, 1985.

[8] Veronique Donzeau-Gouge, Gerard Huet, Gilles Kahn, and Bernard Lang.
Programming Environments Based on Structured Editors: The Mentor Experience.
Interactive Programming Environments.
McGraw-Hill Book Co., New York, NY, 1984.

[9] Rodney Farrow.
Generating a Production Compiler from an Attribute Grammar.
IEEE Software 1(4), October, 1984.

[10] Peter H. Feiler and Gail E. Kaiser.
Display-Oriented Structure Manipulation in a Multi-Purpose System.
In *IEEE Computer Society's Seventh International Computer Software and Applications Conference*, pages 40-48. Chicago, IL, November, 1983.

[11] Harald Ganzinger, Knut Ripken and Reinhard Wilhelm.
Automatic Generation of Optimizing Multipass Compilers.
In *Information Processing 77*, pages 535-540. North-Holland Pub. Co., New York, NY, 1977.

[12] David B. Garlan and Philip L. Miller.
GNOME: An Introductory Programming Environment Based on a Family of Structure Editors.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 65-72. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[13] David Garlan.
Flexible Unparsing in a Structure Editing Environment.
Technical Report CMU-CS-85-129, Carnegie-Mellon University, Department of Computer Science, April, 1985.

[14] David Garlan and Gail E. Kaiser.
MELD: An Object-Oriented Language for Describing Features.
March, 1986.
CMU Department of Computer Science.

[15] David Garlan..
Views for Tools in Integrated Environments.
In *IFIP WG 2.4 International Workshop on Advanced Programming Environments.* June, 1986.
Proceedings to appear as a book published by Springer-Verlag.

[16] David Garlan.
Views for Tools in Integrated Environments.
PhD thesis, Carnegie-Mellon University, 198x.
In progress.

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

[18] Gregory F. Johnson and Charles N. Fischer.
Non-syntactic Attribute Flow in Language Based Editors.
In *Ninth Annual ACM Symposium on Principles of Programming Languages.* January, 1982.

[19] Gail E. Kaiser and Peter H. Feiler.
Generation of Language-Oriented Editors.
In *Programmierumgebungen und Compiler*, pages 31-45. B. G. Teubner, Stuttgart, April, 1984.

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

[21] Gail E. Kaiser.
Generation of Run-Time Environments.
In *SIGPLAN '86 Symposium on Compiler Construction*, pages 51-57. Palo Alto, CA, June, 1986.
Proceedings published as *SIGPLAN Notices*, 21(7), July, 1986.

[22] Donald E. Knuth.
Semantics of Context-Free Languages.
Mathematical Systems Theory 2(2):127-145, June, 1968.

[23] Raul Medina-Mora.
Syntax-Directed Editing: Towards Integrated Programming Environments.
PhD thesis, Carnegie-Mellon University, March, 1982.

[24] David A. Moon.
Object-Oriented Programming with Flavors.
In *ACM Conference on Object-Oriented Systems, Languages, and Applications.* Portland, OR, September, 1986.

[25] John R. Nestor, William A. Wulf and David A. Lamb.
IDL — Interface Description Language: Formal Description.
Technical Report, Software Engineering Institute, Pittsburgh, PA, February, 1986.
Reprint of CMU Technical Report CMU-CS-81-139.

[26] David Notkin.
The GANDALF Project.
The Journal of Systems and Software 5(2):91-105, May, 1985.

[27] *Reference Manual for the Ada Programming Language*
United States Department of Defense, 1983.
ANSI/Military standard MIL-STD-1815A.

[28] Steven P. Reiss.
An Approach to Incremental Compilation.
In *SIGPLAN '84 Symposium on Compiler Construction*, pages 144-156. Montreal, Canada, June, 1984.
Proceedings published as *SIGPLAN Notices*, 19(6), June, 1984.

[29] Thomas Reps and Tim Teitelbaum.
The Synthesizer Generator.
In *SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 41-48. Pittsburgh, PA, April, 1984.
Proceedings published as *SIGPLAN Notices*, 19(5), May, 1984.

[30] Alan Snyder.
Object-Oriented Programming for Common Lisp.
Technical Report ATC-85-1, Hewlett Packard Company, Applications Technology Center, February, 1985.

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