Parameterized Programming and Software Architecture*

Joseph A. Goguen†
University of California, San Diego

Abstract: This paper discusses an approach to software architecture based on concepts from parameterized programming, particularly its language of "module expressions." A module expression describes the architecture of a system as an interconnection of component modules, and executing the expression actually builds the system. Language features include: modules parameterized by theories, which declare interfaces; a number of module composition operations; views for binding modules to interfaces; and both vertical and horizontal composition. Modules may involve information hiding, theories may declare semantic restrictions with axioms, and views assert behavioral satisfaction of axioms by a module. Some "laws of software composition" are given, showing how various module composition operations are related. We also show how a variety of architectural styles can be supported, and how this approach can be extended to support evolution and traceability. All this is intended to ease the development of large systems, and in particular, to make reuse more effective in practice.

1 Introduction

This paper presents an approach to software architecture based on concepts from parameterized programming. Parameterized programming [6, 7] focuses on design and module composition times, rather than on compile or run times; that is, it addresses the architectural level of software. It supports system generation, software reuse, and controlled evolution, as well as the management of configurations, versions, families, documentation, etc. The "module expressions" used in parameterized programming constitute a module connection language (abbreviated MCL, and sometimes also called an architecture description language, or ADL). An MCL may be used

1. *descriptively*, to specify and analyze a given design, or
2. *constructively*, to build a new system using existing modules, by executing design.

*Research supported in part by Office of Naval Research, Ada Joint Project Agency, ARPA Domain Specific Software Architecture project, UK Science and Engineering Research Council, the CEC under ESPRIT-2 BRA WGs 6071 (IS-CORE) and 6112 (COMPASS), and the Information Technology Promotion Agency, Japan, as part of the "New Models for Software Architecture" project sponsored by NEDO.

†On leave from Oxford University Computing Lab.

Detailed design and coding are unnecessary for construction or description if a suitable database (hereafter called a library) of specifications and relationships among them is available. Parameterized programming supports both construction and description, assuming availability of a library that contains:

1. **module expressions**, describing systems as interconnections of modules, and
2. a **module graph**, describing modules and relationships among them.

A module graph can incorporate executable code for modules, as well as specifications and other software objects. Executing a module expression in the presence of a suitable module graph yields an executable system, constructed by manipulating and linking the implementation modules.

The parameterized programming approach to software architecture has been validated by experience with LILEANNA [25, 26], an MCL using Ada for implementation and Anna [19, 20] for specification. LILEANNA was developed as part of the DSSA ADAGE project sponsored by ARPA. The implementation was done by Will Tracz of Loral Federal Systems, and has been used for helicoptor navigation software. This approach seems especially useful for "software factory" situations, such as the Loral helicopter navigation software facility, where a number of similar systems are produced over time. In such cases, systems like LILEANNA seem able to save a great deal of software development time, although significant initial investment may be needed to accumulate information for the library.

LILEANNA has a formal semantics based on category theory, following ideas developed for the Clear specification language [3, 4]; more recently, a set theoretic semantics has been given [15]. These semantics are very general, and apply to languages other than Ada and Anna, and indeed, to any implementation-specification language pair that satisfies certain axioms. The properties of an interconnection of modules are related in a straightforward way to those of its components, through the semantics of module expressions. The work reported in this paper is based on ideas developed in 1983, and first reported in [6], which suggested a design for LIL, a library interconnection language for Ada; see also [7].

1.1 What is Architecture?

The term "architecture" has been much discussed in recent literature, and there is now a large family of partially overlapping definitions. This situation suggests that instead of arguing over the meaning a single popular term, we should develop new terminology that distinguishes among the most important concepts involved. This subsection suggests one such terminology, following ideas from parameterized programming.

A narrow meaning of architecture concerns *static* aspects of systems, such as components, relationships among components, configurations, communication style, etc. A wider sense concerns the *evolutionary* aspects of software development, such as the (ever changing) design and code, and their relationships, including their traceability back to the (ever changing) requirements and other artifacts. We may call the latter **evolutionary architecture**; some aspects of evolutionary architecture are discussed in Section 5.

It is also important to distinguish the architecture of a particular system from the knowledge needed for constructing a family of related systems (or developing a large complex evolving system). We suggest calling these **system architecture** and **domain architecture**[1], respectively. In parameterized programming, a module expression captures a particular system architecture, while a module graph captures an architectural domain. Both are needed to support large scale system development efforts.

This paper is organized as follows: Section 2 discusses the basic concepts of our approach, including theories, views and the module graph; Section 3 discusses parameterization, instantiation, and the distinction between vertical and horizontal structure; Section 4 discusses module expressions, which give an easy way to describe system architectures; some software laws are also given; Section 5 sketches how evolution and traceability can be supported; and Section 6 gives some final discussion and conclusions.

Acknowledgements

I thank Will Tracz for many useful discussions and suggestions for this paper, and for his help with the examples in this paper. I also wish to thank my other collaborators, Drs. Burstall, Diaconescu, Luqi, Malcolm and Meseguer.

[1]The word "framework" is sometimes used for a collection of modules capturing common aspects of applications over a certain problem domain.

2 Hyperprogramming and Module Graphs

Parameterized programming[2] assumes that all modules have associated specifications; these serve as "headers" for other information, particularly source code and compiled code. The specifications need not be complete, but must include at least the syntax of what is exported by the module, the syntax of its interface if it is parameterized, and the names of any modules that it inherits. In parameterized programming, specification and code modules are collected together with other information to form a *module graph*, which describes the organization of the system development database, including information relevant to both system and domain architecture[3].

A module graph node may have as its header a

- theory specification,
- package specification, or
- module expression,

while an edge of the module graph may be labelled with a module relationship, such as

- inheritance,
- view,
- parameterization, or
- instantiation.

Package specifications are intended to have associated executable code. Theory specifications are not, as discussed in the next subsection. Other software objects that maybe be associated with nodes or edges are also discussed below.

Module graphs are based on ideas from what we call **hyperprogramming** [8]. The most relevant ideas can be summarized as follows:

- modules are associated with **clusters**, where
- specifications are *headers* for each cluster, and
- one or more implementations may be given for each package specification, including
  - source code, and
  - compiled code,
- plus (optionally)
  - test cases,
  - performance data,
  - documentation,

[2]The term *megaprogramming* is used for some similar ideas within the ARPA community [2, 27].

[3]The module graph is an *abstraction* of this organization. For various reasons, including efficiency and the structure of existing database systems, the information structure that is actually implemented may be quite different from that suggested by the mathematical structure of a graph.

– administrative information, such as programmer, date of last change, etc.,
– rationale, and
– links to other software objects.

Each cluster has its own node in a module graph. The other software objects referred to in the last point above might be requirements, dataflow diagrams, transcripts of interviews, review documents, etc.; these will also be attached to clusters associated with nodes in the module graph.

Here is a LILEANNA package specification for a stack of integers:

```
package INTSTACK is
   inherit INTEGER;
   type Stack;
   exception Stack_Empty;
   exception Stack_Overflow;
   function Is_Empty(S: Stack) return Boolean;
   function Is_Full(S: Stack) return Boolean;
   function Push(I: Integer; S: Stack) return Stack;
  --| where
  --|       Is_Full(S) => raise Stack_Overflow;
   function Pop(S: Stack) return Stack;
  --| where
  --|       Is_Empty(S) => raise Stack_Empty;
   function Top(S: Stack) return Integer;
  --| where
  --|       Is_Empty(S) => raise Stack_Empty;
  --| axiom
  --|       for all I: Integer, S: Stack =>
  --|          if not Is_Full(S) then
  --|             Pop(Push(I, S)) = S and;
  --|             Top(Push(I, S)) = I;
  --|          end if;
end INTSTACK;
```

Specifications need only be developed to the extent that it is practically useful to do so; often just the syntax is given. The above specification is incomplete, in that it does not define the functions Is_Full or Is_Empty.

2.1 Theories and Views

Theory specifications, called **theories** for short, are used to describe generic module interfaces; these may contain axioms, which serve as semantic constraints on the actual modules that are allowed by the interface. There are no implementation modules associated with theories.

The simplest theory just says that a type should be provided:

```
theory TRIV is
   type Element;
end TRIV;
```

Any (non-empty) module can be matched with TRIV, by designating one of its exported types.

The next theory has some axioms that provide semantic constraints on what can fit the interface that it defines, saying that the type should have a partial order structure:

```
theory POSET is
   type Element;
   function <= (X,Y: Element) return Element;
  --| axiom
  --|   for all X,Y,Z: Element =>
  --|     X <= X and;
  --|     if X <= Y then Y <= X end if; and;
  --|     if X <= Y and Y <= Z then
  --|       X <= Z end if;
end POSET;
```

Views are used to bind actual (values in) modules to generic module interfaces (i.e., theory modules), in order to instantiate generics. Because modules may involve information hiding, views only assert the *behavioral* satisfaction of the axioms in their source theory by the target module[4]. However, *proving* that such axioms are satisfied should not necessarily be part of a system intended to be practical for ordinary use. Instead, views are used for recording a programmer's belief that the axioms in the source theory hold in the target module. Support for such a belief may be provided off line on the back of an envelope, by giving a formal mechanical proof, or by anything between these extremes; the belief may even be left unsupported, although this is not recommended. The nature of the support given (e.g., a scanned image of the envelope back, or the source file of the machine proof) can be stored with the view in the module graph.

The view below asserts that the relation => on integers (as found in the Ada package Standard) satisfies the axioms of POSET; it can be used to instantiate the generic LILEANNA package SORT given later, yielding a package with a function that sorts integers in *descending* order.

```
view GEQ :: POSET => Standard is
   types (Element => Integer);
   ops ("<=" => "=>");
end GEQ;
```

Views can also be used to assert global properties of systems, by giving a view to the system as a whole (represented by a module expression saying how the system is composed from lower level modules; see Section 4) from a theory with axioms giving the properties

[4]This means that the axioms need only appear to hold under all possible "experiments" on the module, involving its externally visible operations; sometimes this is also called *observational satisfaction* [13].

to be asserted. As before, these axioms need only be behaviorally satisfied.

3 Parameterized Modules

The most characteristic feature of parameterized programming is the parameterization of modules over interface declarations: any module that "fits" the interface can be "substituted into" the parameterized module, yielding a new module that is an "instance" (or "instantiation") of the original module. The interface is described by a theory.

Below is LILEANNA code for a parameterized version of the INTSTACK module given earlier (material that is the same as in INTSTACK is indicated by ....):

```
generic package STACK[Element :: TRIV] is
   type Stack;
   ...........
   function Push(E: Element; S: Stack) return Stack;
   ...........
   function Top(S: Stack) return Element;
   ...........
  --| axiom
  --|    for all E: Element, S: Stack =>
  --| ...........
end STACK;
```

The generic package specification below is for a sorting program, parameterized by the POSET theory, which says that a partially ordered set should be supplied in order for the program to work correctly:

```
generic package SORT[Item :: POSET] is
   inherit LIST[Item];
   function Sort (X: List) return List;
   function Sorted (X,Y: List) return Bool;
   --| axiom
   --|    for all X: List =>
   --|       Sorted(Sort(X)) = true and;
   --|       ........... ;
end SORT;
```

Modules can also inherit other modules. The simple syntax for this is illustrated in the following:

```
package M42 is
   inherit SYS31;
   inherit Sort[Integer];
     ........... ;
end M42;
```

Inherited submodules are always shared, that is, new copies are not produced.

3.1 Instantiation

The SORT program specified above can be instantiated with the view GEQ simply by writing SORT[GEQ]; this results in a program for sorting integers in descending order. Similarly, we could instantiate SORT with the partial ordering relation of divisibility on natural numbers by writing SORT[DIV], where DIV is a suitable view.

Default views enable "obvious" views to be replaced by the name of their target module, or even the name of a type. For example, we would not have to write out a view from POSET to Standard that mapped the type Elt to Integer and the operation symbol => to itself, but could just write SORT[Integer]. Default views are computed using a certain set of default rules that are given in [7], and that capture many of our intuitions about what is "obvious." The following are some instantiations that use default views:

```
STACK[Integer]
STACK[LIST[Integer]]
STACK[STACK[Float]] .
```

(Default views were first implemented in OBJ3 [16], and are partially implemented in LILEANNA.)

An interface theory can call for more than one operation and/or more than one type, and views can express bindings to such interfaces. Generic modules can be parameterized by more than one interface, each defined by its own theory. Multiple views are needed to instantiate such modules.

3.2 Vertical Composition

Vertical structure describes the use of lower layers (virtual machines), whereas *horizontal* structure describes a given layer; the distinction between vertical and horizontal structure was named and formalized by Goguen and Burstall [11]; this provides direct support for layered architectural styles. Parameterized programming provides parameterization and instantiation for both vertical and horizontal structure. The following generic package specification has one horizontal and one vertical parameter:

```
generic package SORT[Item :: POSET]
      (LISTP :: LIST[Item]) is
   function Sort (X: List) return List;
   function Sorted (X,Y: List) return Bool;
   --| axiom
   --|    for all X: List =>
   --|       Sorted(Sort(X)) = true and;
   --|       ........... ;
end SORT;
```

Note that the horizontal interface theory is itself parameterized, and moreover is instantiated with the horizontal formal parameter. There is also a default view from TRIV to POSET involved in the vertical instantiation, since SORT wants a POSET whereas LIST wants a TRIV. Also note that the notation [_] is used for horizontal parameterization, while (_) is used for vertical. The same conventions apply to instantiation, as illustrated by the following module expression (where each instantiation uses a default view):

```
SORT[Natural](LIST7[Natural]) .
```

Figure 1 is a graphical evocation of the relationships among inheritance, horizontal parameterization, and vertical parameterization. Note that under horizontal instantiation, actual parameters are shared, whereas under vertical, new copies are used. Here M is a module, I is an imported module, T_H is a horizontal interface theory, and T_V is a vertical interface theory. The use of vertical or horizontal parameterization is an important architectural decision, because of its implications for sharing.

4 Module Expressions

We want to use module expressions as compact definitions of system architectures. This requires us to go beyond parameterization and instantiation, and make use of the following operations, each of which is implemented in LILEANNA:

- module aggregation, i.e., simple combination, taking account of
  - shared inherited modules, and
  - parameterization;
- deleting functionality;
- modifying functionality; and
- adding functionality.

Operations and types can be renamed, deleted or added, as can exceptions, axioms, etc.; and there are both horizontal and vertical instantiation. The following module expressions illustrate the syntax:

SYS1 + SYS2

SYS1 ∗ (**rename op** Put => Write)
+ SYS2 ∗ (**delete op** Undo)

STACK ∗ (**rename type** Stack => Stack1)[Integer]
+ STACK ∗ (**rename type** Stack => Stack1)
[LIST[Float]]

Thus + "adds" modules, i.e., forms a system containing all the summands; because summation is associative (see Section 4.2), any number of summands can be given without using parantheses. See [1] for some related work.

The make command "executes" a module expression to actually build a (sub)system, give it a name, and store it in the environment. The following illustrate this:

make SYS3 **is** STACK[STACK[Float] + SYS2
∗ (**delete op** Undo) **end** SYS3;

A make command can also be parameterized, as in the following:

make SYS4[X :: POSET] **is**
SORT[X](LIST12[X]) **end** SYS4;

Of course, a make command can use results of previous makes, as in

make SYS5 **is** SYS3 + SYS4[Float] **end** SYS5;

Furthermore, a **make** statement can be used to impose sharing of parameters on its constituents, as in the following:

make M329[X :: POSET] **is** SORT[X](LIST12[X])
+ STACK[X] **end** M329;

Whenever a module expression is introduced, it is added as a node to the module graph, and whenever a module expression is evaluated, executable code is generated and attached to the cluster of its module expression.

Module expression evaluation can be implemented by manipulating intermediate compiled code (for LILEANNA, this is Ada's intermediate compiled code language DIANA). Intermediate compiled code is easier to manipulate than object code, while source code does not have much of the required information in a sufficiently explicit form. One also gets the benefit of being able to put the manipulated code through the compiler's backend, including optimization. Module expressions were first implemented in the OBJ system [16], but using different techniques, because there is no attached code in a separate programming language. The mathematical semantics of module expression evaluation is given by the colimit of a diagram extracted from the module graph [4], or by a simpler set theoretic semantics [15].

LILEANNA provides a graphical "layout editor" [18] for module expressions, based on notation like that typically used by engineers, i.e., boxes and arrows. This helps users to edit existing module expressions to define new (sub)systems, or define them from scratch, as they wish.

4.1 Architectural Styles

Many different architectural styles [5] can be supported by parameterized programming, including different communication styles, such as shared variables, pipelining, message passing, and blackboarding. These can be described by using shared submodules in various ways. (Such a cell is a simple "object" in the sense of object oriented programming, and can be easily specified in LILEANNA.) For example, if a "cell" module C encapsulates a variable X, and if modules F and G each inherit C, then they share X, and in the system F + G, they can communicate through it. Similarly, a "post office" module can be inherited by a set of modules, and then used for passing messages among them.

Pipelining is a special case of shared variable communication, where only two modules share each cell; the cells represent the pipes, with one inheriting module reading the variable, and the other one writing it.

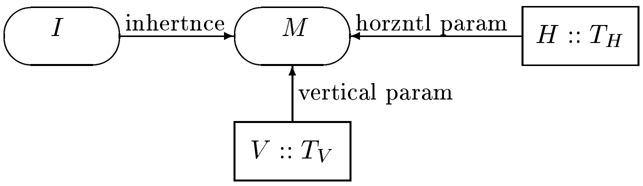

Figure 1: Horizontal and Vertical Composition

For example, if modules F and G inherit cell C1 and modules G and H inherit cell C2, then F + G + H is a (short) pipeline.

The following subsection sketches ways to describe portions of avionics architectures using module expressions. Here the modules encapsulate various digital signal processing subsystems, and shared cells represent "wires" that pass the signals.

4.1.1 Describing Avionics Systems

An interesting example of a module that takes another module as a parameter is a Kalman filter module K that needs a model A of the aircraft; this situation is expressed by the simple module expression K[A]. It makes sense to parameterize K, so that the same filter can be used for many different aircraft models.

Now consider a flight control system F that needs a guidance system G, where G needs an aircomputer A. We can describe this situation with the module expression F[G[A]]. (We omit details of the code here and hereafter.) Assuming that the modules F and G have parameter (interface) theories GS and AC respectively, then views are needed to match A to AC and G to GS; it is natural to expect that default views will work in such examples.

This simple approach based on instantiating parameterized modules works well if the system architecture is linear, or more generally, a tree; but it is not adequate for sharing among more than two modules, or for feedback loops. Feedback is necessary in avionics software, because feedback control is a crucial technique.

For example, suppose the value of a floating point variable X in A needs to be fedback into F. We can capture this by encapsulating X in C, and letting F and A each inherit C; this system is still described by the simple module expression F[G[A]]. Another approach is to provide each F and A with a new parameter for a floating point cell, in which case the system is described by the module expression F[G[A[C]], C].

This example highlights the importance of sharing for a module interconnect formalism. We have shown that parameterized programming can accomplish such sharing in several ways. Without such a capability, it would be difficult or impossible to handle feedback and variables shared among several modules.

4.2 Some Software Laws

There are many relationships among the various operations on modules; the more important of these can be considered "laws of software." Here are a few of them:

$$
\begin{aligned}
& M + M' = M' + M \ , \\
& M + (M' + M'') = (M + M') + M'' \ , \\
& M + M = M \ , \\
& M + N = M \text{ if } N \text{ inherits } M \ , \\
& M^H + M'^H = (M + M')^H \ , \\
& M^{N_1, N_2} = M^{N_2, N_1} = M^{N_1 + N_2} \ , \\
& M^{N_1^H,\ N_2^H} = M^{(N_1 + N_2)^H} \ , \\
& M^N + N = M^N \ ,
\end{aligned}
$$

where the notation M^N indicates that M inherits the module N, and M^H indicates that M inherits the set H of modules. There are many more laws, e.g., for parameterization; see [15] for these, as well as for proofs. These laws are used in the implementation of LILEANNA for simplifying module expressions.

5 Evolution and Traceability

A traditional view in software engineering is that evolution only occurs after initial development is complete. By contrast, we consider evolution to include everything involved in changing a system, plus the relationships among those things, throughout the system lifetime. The word "evolution" focuses attention on *change*, which is inevitable and unending in software development. Moreover, since large complex systems are inevitably embedded in complex evolving social contexts, they will necessarily *co-evolve* with those contexts, in the sense that each will affect the evolution of the other. (See [10, 12] for further discussion of evolution and social context.)

The ubiquity of change motivates the use of iterative lifecycle processes, and especially prototyping, i.e., quickly building and evaluating a series of prototypes, which are concrete executable models of selected aspects of a system [21]. The ability of parameterized programming to describe software architectures, in both the domain and systems senses, can greatly facilitate prototyping. In some cases, all that need be done is edit a module expression. In other cases, the module graph may need updating, e.g., writing new modules or modifying old ones. Then executing the module expression will yield a running prototype, which can be gracefully evolved into the actual system, and thereafter further evolved.

The additional information needed to cope with the dynamic evolution of (families of) software systems is provided by enriching the module graph with relevant relationships among various software objects, such as requirements and rationales. Rationales should be considered part of evolution support rather than architecture, because they must evolve along with the objects that they concern. Other subproblems include formalizing dependencies, and developing methods for calculating dependencies and for propagating the implications of changes.

5.1 Traceability

The Centre for Requirements and Foundations at Oxford has had projects to improve the acquisition, traceability, accessibility, modularity, and reusability of the numerous objects that arise and are manipulated during software development, with a particular focus on the role of requirements. An initial study [17] administered a detailed two-stage questionnaire to software engineers at a large firm, and found that traceability was considered the most important outstanding problem. Further analysis showed that there are actually several different traceability problems, which should be treated in different ways. Major distinctions are between pre-RS (Requirements Specification) traceability and post-RS traceability, and between forward and backward traceability.

Tracing back as far as requirements is important for developing large software systems. But it is also difficult because of the overhead of maintaining the huge mass of dependencies among the many objects produced by a large software development effort. Moreover, dependencies reaching far across the development cycle can be significant. Without formal representations for the objects involved, formal models for the dependencies, and tool support for managing them, it can be impossible to know what effect a change will have, and in particular, to know what other objects may have to be changed to maintain consistency. A hypergraph model for maintaining evolving dependencies, suitable as a basis for tool development, is given in [22]; this structure can be applied to give a model for evolving module graphs. Work on capturing domain knowledge has used methods from sociology, particularly ethnomethodology and its disciplines of conversastion and interaction analyses. Further information on this research may be found in [9, 10, 12].

We are also developing a system called TOOR to support tracing dependencies among evolving objects, and in particular, to show how decisions are grounded in prior objects [23]. This approach, called *hyperrequirements*, builds on parameterized programming and hyperprogrammming, and is intended to support the social context of decisions, as well as their traceability, by linking related objects, based on the broad view of context and requirements suggested in [9].

TOOR supports user-definable relations, to allow differentiating among different links between the same objects. It also allows declaring mathematical properties of relations, such as transitivity, to give additional power and flexibility in tracing links through specifiable compositions of relations. TOOR supports several different trace modes, including browsing and regular expression based search, and object and relation definition through intuitive template-driven graphical interfaces. Hyperprogramming and hyperrequirements support reuse, through the generalized notion of relation for linking objects for requirements, design, specification, coding, documentation, etc.

TOOR is built on FOOPS [14, 24], a general object oriented language with specification capabilities. This makes TOOR particularly suitable for use in an object oriented development paradigm. TOOR uses FOOPS-like modules to declare software objects and relations, and to automatically create links as objects are interconnected and evolve. TOOR also provides hypermedia facilities, based on HTML, to make its use closer to analysts' intuitions and natural activities. For example, graphs, charts, and videos can be objects in TOOR, as well as conventional documents.

6 Discussion and Conclusions

We have seen that parameterized programming can provide a systematic approach to the design and manipulation of software architectures, through its use of theories and views, its powerful methods for combining and modifying components to form new systems, and its underlying module graph data structure. The notions of module expression and module graph, we believe, add some clarity to discussions about the nature of software architecture. We have illustrated the use of module expressions to achieve several different architectural styles.

Parameterized programming extends to hyperprogramming and hyperrequirements, to support evolution by including other software objects in the module graph, such as requirements, rationales, and documentation, as well as relations to support traceability. This enables design objects and relations to be managed in a systematic way, which can have a significant impact on the reusability of code, through the reuse of design information. Support for evolution also allows integration with prototyping and traceability in a natural way. Reusability is supported by the systematic cataloging of information in an evolving module graph. It is hoped that ideas like these will help to make the development of large complex systems more reliable and efficient than it is today.

References

[1] Don Batory and Sean O'Malley. The design and implementation of hierarchical software systems with reusable components. Technical Report TR–91–22, Department of Computer Sciences, University of Texas, Austin, 1991. Revised May 1992.

[2] Barry Boehm and William Scherlis. Megaprogramming. In *Proceedings of Software Technology Conference 1992*, pages 63–82, April 1992.

[3] Rod Burstall and Joseph Goguen. Putting theories together to make specifications. In Raj Reddy, editor, *Proceedings, Fifth International Joint Conference on Artificial Intelligence*, pages 1045–1058. Department of Computer Science, Carnegie-Mellon University, 1977.

[4] Rod Burstall and Joseph Goguen. The semantics of Clear, a specification language. In Dines Bjorner, editor, *Proceedings, 1979 Copenhagen Winter School on Abstract Software Specification*, pages 292–332. Springer, 1980. LNCS 86.

[5] David Garlan and Mary Shaw. An introduction to software architectures. *Advances in Software Engineering and Knowledge Engineering*, I:41–49, 1993.

[6] Joseph Goguen. Suggestions for using and organizing libraries in software development. In Kartashev and Kartashev, editors, *Proceedings, First International Conference on Supercomputing Systems*, pages 349–360. IEEE, 1985. Also in *Supercomputing Systems*, Kartashev and Kartashev, editors, Elsevier, 1986.

[7] Joseph Goguen. Principles of parameterized programming. In Biggerstaff and Perlis, editors, *Software Reusability, Volume I: Concepts and Models*, pages 159–225. Addison Wesley, 1989.

[8] Joseph Goguen. Hyperprogramming: A formal approach to software environments. In *Proceedings, Symposium on Formal Approaches to Software Environment Technology.* Joint System Development Corporation, Tokyo, Japan, January 1990.

[9] Joseph Goguen. Requirements engineering as the reconciliation of social and technical issues. In Jirotka and Goguen, editors, *Requirements Engineering: Social and Technical Issues*, pages 165–200. Academic Press, 1994.

[10] Joseph Goguen. Formality and informality in requirements engineering. *Proceedings, International Conference on Requirements Engineering*, to appear, March 1996.

[11] Joseph Goguen and Rod Burstall. CAT, a system for the structured elaboration of correct programs from structured specifications. Technical Report Report CSL-118, SRI Computer Science Lab, October 1980.

[12] Joseph Goguen and Luqi. Formal methods and social context in software development. In Mosses, Nielsen, and Schwartzbach, editors, *Proceedings, TAPSOFT 95*, pages 62–81. Springer, 1995. LNCS 915.

[13] Joseph Goguen and Grant Malcolm. Proof of correctness of object representation. In A. William Roscoe, editor, *A Classical Mind: Essays in Honour of C.A.R. Hoare*, pages 119–142. Prentice-Hall, 1994.

[14] Joseph Goguen and José Meseguer. Unifying functional, object-oriented and relational programming, with logical semantics. In Shriver and Wegner, editors, *Research Directions in Object-Oriented Programming*, pages 417–477. MIT, 1987.

[15] Joseph Goguen and Will Tracz. An implementation-oriented semantics for module composition. In preparation.

[16] Joseph Goguen, Timothy Winkler, José Meseguer, Kokichi Futatsugi and Jean-Pierre Jouannaud. Introducing OBJ. In Goguen and Malcolm, editors, *Algebraic Specification with OBJ*. Cambridge, to appear.

[17] Orlena Gotel. Requirements traceability. Technical report, Centre for Requirements and Foundations, Oxford University Computing Lab, December 1992.

[18] J.T. Higgins. ADAGE layout editor user's manual. Technical Report ADAGE-LOR-94-04, Loral Federal Systems – Owego, July 1994.

[19] Bernd Krieg-Brückner and David Luckham. ANNA: Towards a language for annotating Ada programs. *SIGPLAN Notices*, 15(11):128–138, November 1980.

[20] David Luckham. *Programming with Specifications: An Introduction to* ANNA*, A Language for Annotating Ada Programs*. Springer, 1990.

[21] Luqi. Software evolution through rapid prototyping. *IEEE Computer*, 22(5):13–25, 1989.

[22] Luqi and Joseph Goguen. Formal methods: Problems and promises. *IEEE Software*, to appear, July 1996.

[23] Francisco Pinheiro and Joseph Goguen. An object-oriented tool for tracing requirements. *IEEE Software*, to appear, March 1996. Special issue of papers from ICRE '96.

[24] Adolfo Socorro. *Design, Implementation, and Evaluation of a Declarative Object Oriented Language*. DPhil thesis, Programming Research Group, Oxford University, 1994.

[25] Will Tracz. Parameterized programming in LILEANNA. In *Proceedings, Second International Workshop on Software Reuse*, March 1993. Lucca, Italy.

[26] Will Tracz, P. Angeline, S. Shafer and Lou Coglianese. Experience using an avionics domain-specific software architecture. In *Proceedings,* NAECON'95, pages 646–653, May 22-27 1995.

[27] Gio Wiederhold, Peter Wegner, and Stefano Ceri. Toward megaprogramming. *Communications of the ACM*, 35(11):89–99, 1992.