APPENDIX A

Previous Models

Because the idea of formalizing the concept of software or of software development has been around for so long, there are many research efforts relevant to a new formal modeling attempt. The research approach followed in this dissertation is to start with five modern “models” chosen to be representative of previous research: the 3C model of reusable software, OBJ (as realized in OBJ3), RESOLVE, Eiffel, and Standard ML. These five models will form the foundation for developing a new model.

This appendix will describe these five representatives of prior work, highlighting the issues each brings to the problem of modeling software structure. This information will be used in Appendix B to form a description of the requirements for a “universal” model of software and how it is constructed. Appendix C will show how each of the five prior models presented here can be reinterpreted within the mathematical spaces defined in ACTI.

Unfortunately, most past work on “models” of software has centered on a particular language. As a result, only one of the five models considered in this appendix is presented as a language-independent conceptual model—the 3C model. The remaining four are actually specific languages. The abstract “model” of software components that is under consideration in each of these four cases is the one implicit in each corresponding language. These four languages in particular were chosen for breadth of coverage: two of the languages are functional programming languages; one is object-oriented; one is object-based; two support formal specification; one more supports informal specification; one supports verification; three support parameterized programming; and they all aim to strongly support data abstraction and encapsulation. They also all aim to support software that is modularly constructed, and claim to be applicable to the architectural problems of large-scale software systems.

The subsections in this appendix describe each of these five models in turn. While a complete presentation of each model is beyond the scope of this dissertation, this appendix aims to present enough detail for one to grasp the unique issues that each model raises with respect to reusable subsystems. The description of each model is
divided into a basic introduction, a more detailed overview of the model or language’s key features, and a presentation of the advantages and disadvantages of that model with respect to software subsystem construction, reuse, and understanding.

A.1 The 3C Model

One of the more recent attempts to characterize reusable software components is the “3C model.” Unlike the other models discussed in this appendix, the 3C model of reusable software components was conceived as an abstract, language-independent framework for discussing the features of software parts. It was originally developed at the “Reuse in Practice” workshop, held July 11–13, 1989 in Pittsburgh, Pennsylvania [62]. The 3C model is an enhancement of the Concept/Context model initially proposed by Will Tracz in his dissertation work at Stanford [63]. This model is thus based on previous work embodied in LIL [15, 19] and OBJ [21].

A.1.1 An Overview of the 3Cs

The 3Cs stand for:

Concept—The abstraction captured in a component. This embodies what the component does. Tracz describes this term as follows:

The “concept” behind a reusable software component is an abstract canonical description of “what” a component does. Concepts are identified through requirement or domain analysis as providing desired functionality for some aspect of a system. A concept is realized by an interface specification and an (optionally formal) description of the semantics (as a minimum, the pre- and post-conditions) associated with each operation. An Ada package specification with its behavioral semantics described in Anna is an example of a reusable software concept. [62, p. 11]

Content—The implementation of that abstraction. This embodies how the component does what it does.

The “content” of a reusable software component is an implementation of a concept, or “how” a component does “what” it is supposed to do. The basic premise is that each reusable software component can have several implementations that obey the semantics of its concept. The collection of (28) stack packages found in Grady Booch’s components
is an example of a family of implementations for the same concept (a stack). [62, p. 11]

**Context**—The environment in which the component is designed to operate. This embodies the constraints that must be satisfied in the environment in which the component will be reused.

The “context” of a reusable software component is 1) the environment that the concept is defined in (“conceptual context”), and 2) the environment it is implemented under (“contentual context”). It is very important to distinguish between these two types of contexts because different language mechanisms (inheritance and genericity) apply differently to each. Furthermore, these two contexts clearly distinguish between type inheritance and code inheritance. [62, pp. 11-12]

The focus of this model is on separating the concept, content, and context of a component in order to enhance reusability.

Most programmers are already familiar with the notion of a component’s “concept”; it is usually identified with some form of module interface specification. It is important to distinguish the notion of a concept from how it is represented in a particular programming language, however [5, p. 5]. The concept, by definition, encompasses a complete description of the component’s behavior. An interface specification in a given language, on the other hand, may only capture some aspects of this description (e.g., the parameter profile) in a formal way. This is certainly useful, since it provides some measure of automatic checking, but it should not be mistaken for the richer abstraction of the component’s concept if the language does not permit a complete description of the component’s abstract behavior.

In a similar vein, software engineers usually identify the content of a component with an encapsulated module implementation. Naturally, there can be many such implementations that all provide the same externally observable functional behavior. Presumably, if the implementation language allowed, each such implementation could be checked to ensure conformance with the corresponding concept description for the component (or, as much of the concept as can be expressed in that language).

The remaining “C” in the 3C model is the context—the external environment that is relevant to the definition and operation of a given component. Most often, one might view the modules upon which another component’s implementation is built to be context. Similarly, modules that define abstract types or other information needed to describe a given component’s specification are also context. In an OO language, when one uses an existing module as the basis for the definition of a new module through inheritance, the first serves as part of the context for the second.
By making a clear distinction between external dependencies and new concepts or implementations, one can more effectively separate changeable aspects of a component from constant aspects. One can even go a step further by decoupling direct external dependencies through the use of some form of parameterization. In most procedural languages, the obvious choice for providing fixed external dependencies is through a static importation mechanism (for example, Ada's with statement, Modula-2's import, or C's #include). Alternatively, the client can be given the capability of binding in actual external dependencies through generic parameters. Potential clients of the resulting component can then configure it for use in a variety of environments (contexts). In some sense, the “parameters” that represent this context form an abstract description of the dependencies a component has on its environment.

Further, one can consider trying to separate external dependencies from either a component’s concept or from its implementation. With this in mind, the term conceptual context will be used to refer to the external dependencies of a component’s concept. The term implementation context will be used in a similar fashion to refer to the context of a particular implementation. Regardless of where a given external dependency is used, the component writer may decide to hard-code a specific reference—making this part of the context fixed—or decouple it through some mechanism (e.g., parameterization) controlled by the eventual reuser—making this part of the context variable.

The benefits of using this framework to view the problem of creating reusable software are summed up as follows:

Defining reusable components in a programming language can thus be viewed as the task of separating context from concept, concept from content, and content from context. While this does not answer the question of how to design reusable components in general, it does provide a new perspective on the question of how to represent such components. Designing the component involves forming an abstraction and identifying the concept and the context, then separating them to achieve the best change control and reusability. Once this is done, one can concretely represent this abstraction in a given programming language. [emphasis in original] [5, p. 9]

A more detailed description of the 3C model is provided by Edwards [5], who uses this model to develop a set of programming guidelines for creating reusable Ada software. This model evolved through “bottom up” efforts to characterize current knowledge about how module-level reuse is best performed today. Additional work with the 3C model has also been aimed at modeling larger grained reusable parts [38].
A.1.2 Issues Raised by the 3Cs

The 3Cs themselves form a nice conceptual framework for discussing many aspects of reusable software. Perhaps the greatest contribution of this model is the way it frames the notions of concept, content, and context as three distinct facets of a component that have equal importance. Effectively separating these parts within a given component is critical for reuse. Unfortunately, these notions have blurry definitions, in part because of the goal of language independence. The following subsections reflect the issues the 3C model raises in the context of this research by describing the advantages and disadvantages this model has with respect to reuse.

Advantages

The 3C model has the following advantages:

1. It is language-independent.

2. It recognizes the notion of an abstract description of component behavior (i.e., a concept), and advocates separating this abstraction from both the component’s implementation, and the component’s external dependencies.

3. It recognizes the notion of a component implementation (i.e., content), and advocates separating it from both the component’s abstract description of behavior and the component’s external dependencies.

4. It recognizes the notion of a component’s external dependencies, and advocates separating them from both the component’s abstract description of behavior and the component’s implementation. It also supports separating external dependencies required by a component’s abstract description from the external dependencies required by its implementation.

5. It allows for the formal specification of the behavior provided by a module (the concept).

6. It permits one to consider generalizing abstract descriptions of component behavior, although it does not specifically address such generalizations.

7. It advocates the use of multiple implementations for a given abstraction—i.e., multiple contents for one concept.

8. While it advocates separating context from concept and content, it does not specify (or limit) what language mechanisms should be used for this task.
9. While advocating generic parameter mechanisms for component abstractions and implementations, it does not prescribe a particular parameter conformance enforcement policy.

10. It does not require that concepts be simple modules—concepts could be used to encapsulate large subsystems of interconnected modules bundled in a single package, or even a component generator, where the parameters to the generator are viewed as context provided via a particularly rich formalism.

Disadvantages

Unfortunately, the 3C model also has the following limitations:

1. It is not formally defined, and there are occasional disputes over definitions for the most basic terms (e.g., “context”).

2. Because it is not well defined, it is very difficult to communicate to others.

3. Also because it is not well defined, it fails to address the needs of one who wishes to reason about the behavior of a (possibly complex) composition of components.

4. Because of its bottom-up origins, it is often criticized as being insufficient for scaling up to components of “realistic” sizes.

5. “The 3C model is a qualitative model, similar to the Greek ‘earth, air, fire, water’ model of physics.”—Don Batory, WISR’91 [37].

6. Because it does not specifically address the notion of general-purpose abstract descriptions of behavior (e.g., common interface models), it also fails to address the need for correspondence mechanisms between a given component and such a general description.

7. While it advocates many-to-one correspondences between implementations and concepts, it does not address the question of whether this relationship should be many-to-many.

8. It does not explicitly address the need for structuring the space of contextual parameters required by a concept or implementation.

9. It does not explicitly address the need for mechanisms for defining new modules as extensions or enhancements of existing ones.
A.2 OBJ

OBJ was originally designed by Joseph Goguen in 1976 [18] as a tool for extending algebraic abstract data types. Its design and early evolution were heavily influenced by Goguen’s work with Clear [3], and OBJ took on the form of a specification language that would support the testing of specifications by making them executable [17, 16]. Thus, an OBJ specification of an abstraction can be considered as a set of equations or axioms that define the semantics of the exported operations, or it can be considered as a program that can be executed by interpreting the defining equations as rewrite rules.

OBJ was also designed to support programming at a higher level of abstraction than traditional high order languages by focusing on the interconnection and composition of programming modules, particularly reusable modules. It thus borrowed from Clear many ideas for programming with parameterized modules. OBJ3, the most recent version of OBJ [21], reflects Goguen’s thoughts on the subject of parameterized programming.

Goguen describes the purpose of parameterized programming as follows:

> It may happen that there is a software part we want to reuse, but it is not in exactly the right form, or perhaps we have combined some modules and now we want to improve the efficiency of the combination. …Parameterized programming allows us to achieve such goals by modifying parameterized modules, either before or after combination or instantiation, so that they can fit a wider variety of applications. [21, p. 160]

He further characterizes the key types of modifications that can be made as the following [21, p. 160–161]:

- Extend a module by adding to its functionality.
- Rename some of its external interface.
- Restrict a module, by eliminating some of its functionality.
- Encapsulate some existing code.
- Combine (add) two or more modules.
- Modify the code inside a module.

OBJ is designed to allow all of these forms of modification while guaranteeing that some selected set of module properties are preserved by the modification.
A.2.1 An Overview of OBJ

OBJ is a functional language. All operations are functions, and their behavior is specified by sets of axioms or equations. Because OBJ specifications can be interpreted using a rewrite approach, it is very desirable that the set of axioms provided by a specification have the Church-Rosser property (i.e., independent of the order in which rewrites are applied) and be terminating. Unfortunately, the combination of a functional style with an algebraic specification approach can be bad for reuse:

In fact, because of the functional style of an algebraic specification, it is easier to define behavior in that style that cannot be implemented efficiently than to define behavior that can be. [67, p. 24]

Further, algebraic specifications do not effectively support the needs of clients or designers in reasoning about components. This is because they describe behavior, without providing a specific conceptual model of the values being manipulated. Chapter II shows why such a model is critical for supporting (human) reasoning.

OBJ’s theoretical roots lie in order-sorted algebra (OSA), which overcomes some of the expressive difficulties with many-sorted algebra approaches [21]. OSA is a rigorous mathematical theory, and it is used as the basis for an abstract denotational semantics for OBJ. The language also has a more concrete operational semantics based on order-sorted rewriting, as one might expect.

There are four high-level kinds of entities in OBJ: modules, views, module expressions, and reductions. Further, modules can be either objects or theories. These classes of entities will each be described in turn.

The name OBJ is derived from the keyword it uses to introduce a module of executable operations, called an object.

In OBJ, an object is a module, which may export types, operations, and axioms. An object is not the same as the notion of a “data object,” which is how the term is used in most other languages. Instead, an OBJ object is a description of an “abstract software component” that is available to serve as a building block when constructing larger software subsystems. Objects can thus be viewed as executable code.

An OBJ theory, on the other hand, is an abstract description of behavior or of an interface. A theory has the same structure as an object, but the difference is that objects are executable while theories simply define properties that other modules may have. The methods by which modules are manipulated, combined, and modified are applicable to both objects and theories.

Note that a particular theory is not a priori bound to any particular object. Instead, theories implicitly define families of related modules (objects and other theories) that have the specified properties. This is different than the specification for
a single object, which defines the interface of a particular module—the types, operations, and behavior exported by that module.

Most often, a theory is used to describe the semantic requirements placed on a generic parameter to another module. Such a theory would describe the semantic characteristics required of any “actual” supplied for the corresponding “formal” parameter in an instantiation. Any module that could be shown to comply with this theory—i.e., any module that meets the requirements of the parameter—could be passed in as the corresponding parameter value during an instantiation.

An OBJ view is a correspondence between an OBJ module (an object or theory) and an OBJ theory. For example, a view can be used to express the mapping between an object and a theory, showing how that object meets the requirements of that theory. Also, a view can be used to show how one theory meets the requirements of another and how everything that corresponds to the first also meets the requirements of the second. Any such view can be given a name and reused in many situations.

In OBJ, objects, theories, and views can all be generic. This allows the context of the corresponding entity to be explicitly identified: “For code to be reusable, it and anything that it relies upon (its context) must be known” [21, p. 184]. It also allows for dependencies on context to be decoupled from modules, and thus lies at the heart of parameterized programming.

An OBJ module expression is a statement which actually describes the composition of two or more modules. Module expressions are the means by which a programmer can actually specify the instantiation or aggregation of modules, and the renaming, addition, elimination, or replacement of entities exported by a module. Theories can be composed to form more sophisticated composite theories, and collections of objects and theories can be composed to form composite objects. The evaluation process includes ensuring that any formally stated program properties, specified through theories and views, are maintained by the product of the composition.

Finally, a reduction is a statement that evaluates a given OBJ expression in the context of a particular object. The expression is evaluated by interpreting the axioms within the object(s) as rewrite rules. Thus, a reduction is analogous to a top-level program execution, where the body of the program is simply an expression—as would be expected in a functional language.

A.2.2 Issues Raised by OBJ

Goguen’s work with OBJ has provided the theoretical foundations behind many of the component level reuse research efforts in the last decade. It provides programming language concepts that support reuse and which have a mathematically rigorous basis.
The advantages and disadvantages of OBJ with respect to reuse are described in the following subsections.

**Advantages**

As expected, OBJ has a huge number of advantages for supporting reuse and the manipulation of subsystems, mostly stemming from the use of parameterized programming. Among these advantages are:

1. It allows for the formal specification of the behavior provided by a module (the concept).
2. Theories support the notion of generalizing properties shared by many modules.
3. It supports the notion of a formally verifiable correspondence between a given module and a given theory—called a view.
4. It supports many-to-many mappings between modules and the theories that they satisfy.
5. It provides a mechanism for separating context from concept (e.g., generic parameterization).
6. It provides an intensional parameter conformance mechanism for use with generic parameters \([7]\)—any module that can be mapped to the requirement theory defining a formal parameter can be substituted as the corresponding actual.
7. It provides a mechanism for structuring the space of contextual parameters required by a concept (e.g., theories).
8. It provides a mechanism to define new modules as extensions or enhancements of existing ones (importation/inheritance).

**Disadvantages**

Unfortunately, OBJ also has the following limitations:

1. It appears to be poorly understood, except in theoretical circles.
2. It does not clearly separate specification from implementation, since an OBJ object defines both the specification and implementation simultaneously.
3. OBJ views allow behavior to be added to a given module in order for it to conform to a theory. While this added behavior simply takes the form of additional axioms in OBJ, it would also necessitate implementation code in a more conventional language. This code would need to be encapsulated in some structure.

4. It lacks a distinct notion of “context”—it indirectly addresses this need instead by defining some language mechanisms that can be used to provide context (e.g., generic parameterization and importation).

5. It is based on algebraic rather than model-based specification techniques. Weide et al. discuss why this is not the best choice for reuse [67, p. 10–20, 24].

6. It fails to provide the support necessary for dealing with module-level state and objects that have their own local state [21, p. 180]. This is a natural outgrowth of the fact that OBJ is a functional language. In LIL [15, 19], Goguen remedies this situation to some extent.

7. OBJ does not allow nested modules in its current version, although Goguen recognizes the need for them and envisions them as part of OBJ in the future [21, p. 199].

8. Goguen hints at the need for “dependent types”—i.e., nested generic modules—but implies that there is not yet a developed formal semantics for such objects [21, pp. 209–210].

9. OBJ3 includes a built-in module called TUPLE that is an n-ary parameterized module. This is one example of the need for “variable-length” module parameterization, a need that has arisen in other languages as well. Basically, one would like to have a “vector” of modules, all matching the same requirement theory, be a parameter to another module. Unfortunately, OBJ3 only provides this capability for this one built-in module, and there is no generalized way for users to parameterize their own modules in this fashion.

A.3 RESOLVE

RESOLVE [67, 33, 27, 24, 57, 31], the REusable SOftware Language with Verifiability and Efficiency, is the product of ongoing research efforts by the Reusable Software Research Group (RSRG) at The Ohio State University. This language consolidates many of the features and concepts that support reuse from existing programming languages into a single uniform framework. As noted by Weide et al., RESOLVE
supports many of the basic notions from the 3C Model [67, p. 4–8, 60–61]. The language constructs, along with their associated proof rules [33], implicitly describe a formal model of reusable software.

A.3.1 An Overview of RESOLVE

The structure of RESOLVE components nicely fits into the framework of the 3C model:

- The abstract functional behavior of a piece of software is explicitly separated from the implementation of that behavior; i.e., concept is separated from content.

- For a particular abstract behavior there may be multiple implementations that differ in time/space performance or in price, but not in functionality; i.e., a given concept may have more than one content that realizes it.

- The external factors that contribute to the explanation of behavior are separated from that explanation; i.e., the context of concept is separated from the concept itself.

- The external factors that contribute to the implementation of behavior are separated from implementation code; i.e., the context of content is separated from the content itself.

[67, p. 7]

In addition, RESOLVE adds some contributions that are not described in previous work.

The reasoning behind many of the contributions of RESOLVE is highlighted by considering the following question: “What could be worse than not reusing software?”

1. “An inappropriate component might be chosen—one whose actual behavior is misunderstood by the client programmer” [67, p. 10].

2. “An apparently reusable abstract component may be designed poorly from the standpoint of reuse.” In particular, it might be prohibitively inefficient when applied by some clients [67, p. 21].

3. “A not-quite-suitable component might be available—one with no hope of adaptation to the specific needs of the client” [67, p. 45].
4. “An incorrect concrete component might be chosen—one that is not a correct realization of the corresponding abstraction” [67, p. 53].

RESOLVE addresses each of these ways that reuse can go awry.

To address the first type of mistake, choosing an incorrect component, RESOLVE was designed with clarity and abstractness of specifications in mind. In this case, clarity means that a component specification must be “clear, unambiguous, and understandable to a potential client and to a potential implementer” [67, p. 11]. Abstractness means that the specification “must be free of implementation details in order to support a variety of concrete [implementations].” To encourage specifications with these properties, RESOLVE has a sublanguage for writing formal specifications using a model-based rather than an algebraic approach. The idea that a single abstract specification may have many alternative concrete implementations is central to RESOLVE’s notion of a module.

The second type of mistake involves poorly designed components, which may be excessively inefficient in some circumstances.

An abstract component with only inefficient concrete components to implement it tempts a client to “roll his/her own.” A poorly designed abstract component may even inherently rule out efficient realizations. [67, p. 21]

Further, Weide et al. [67, p. 21] claim there is no theoretical reason to believe that there is an intrinsic tradeoff between generality and efficiency. RESOLVE has many features aimed at eliminating design choices that restrict efficient component implementations. This is one reason why RESOLVE uses model-based specifications rather than algebraic techniques—the functional style of algebraic specifications make it easier to define specifications that cannot be implemented efficiently [67, p. 24]. Further, RESOLVE advocates the use of swapping rather than copying as the basic mechanism for moving data objects around in programs [25].

To address the third type of mistake, inflexible components, RESOLVE provides language mechanisms to parameterize context through genericity. The language recognizes the distinction between fixed and variable (e.g., parameterized) context, and the distinction between the context of the concept and the context of the implementation (content). Thus, RESOLVE allows components to have separate generic parameters for specifications and for implementations. This generic mechanism allows instances of concepts to be parameters to other concepts or implementations. For maximum reusability, the RESOLVE discipline encourages fully parameterized components that have no fixed context [51, p. 25]. Finally, RESOLVE supports a limited
form of type or specification inheritance originally called “enhancement,” and later [9] termed “re-exporting.”

To address the fourth type of mistake, incorrect components, RESOLVE requires that it be possible to certify that any component implementation correctly supplies the functionality described in the corresponding specification. The proof rules for RESOLVE [33] ensure that the verification process factors well along component boundaries, so that components can be independently verified. This modularizes the verification of large software systems, and contradicts earlier beliefs that “there is no reason to believe that a big verification can be the sum of many small verifications” [4].

The net effect of this framework is that RESOLVE requires complete information hiding and encapsulation, rather than the leaky encapsulation supported or even encouraged by other conventional programming languages, such as Ada. This works to the programmer’s benefit, whether she be a client or an implementer of a component, by eliminating many of the problems caused by “chasing efficiency at the expense of correctness” [67, pp. 43–45].

A.3.2 Issues Raised by RESOLVE

RESOLVE stands out among the languages discussed in this appendix in that it was specifically designed to support software reuse above all other concerns. Further, its goal of modular verifiability has brought forward many additional insights. It meshes extremely well with the framework of the 3C model, and arguably provides a model of software construction that is scalable to very large systems. The following subsections describe its advantages and disadvantages in more detail.

Advantages

RESOLVE has the following advantages for constructing reusable software:

1. The abstract functional behavior of a piece of software is explicitly separated from the implementation of that behavior.

2. A particular abstract description of behavior may have multiple implementations.

3. The external factors that contribute to the explanation of behavior are separated from that explanation.

4. The external factors that contribute to the implementation of behavior are separated from implementation code.
5. RESOLVE requires descriptions of behavior to be formally specified.

6. Functional specifications are provided using a model-based rather than an algebraic approach. This is important for both efficiency and human factors reasons, and naturally supports RESOLVE’s emphasis on data objects that have their own local state.

7. In RESOLVE, all modules are generic, and parameterization mechanisms are provided for expressing context dependencies.

8. The language recognizes the distinction between fixed and variable (e.g., parameterized) context, and the distinction between the context of the concept and the context of the implementation (content).

9. RESOLVE’s generic mechanism allows instances of concepts to be parameters to other concepts or implementations, allowing for the structuring of the space of contextual parameters.

10. A limited form of inheritance is provided to allow new modules to be defined as enhancements or extensions of existing modules.

11. The correspondence between a component implementation and a concept is clearly defined and formally verifiable.

12. The verification rules for the language ensure that the verification process can be truly modular, where component implementations can be verified once independently of any given contextual environment.

13. The language implicitly enforces complete information hiding and encapsulation, discouraging the weaker encapsulation that leads to “leaky” abstractions in more conventional languages.

**Disadvantages**

The RESOLVE work does have the following limitations, however:

1. It appears to be poorly understood outside of the RSRG.

2. It is difficult to communicate to others.

3. Because of its bottom-up origins, the techniques used in RESOLVE are often criticized as being insufficient for scaling up to components of “realistic” sizes. This is a particularly prevalent (albeit invalid) criticism of the verification techniques.
4. It lacks a way to describe general properties shared by many module concepts.

5. The definition of conformance for generic parameters is restrictive compared to that provided in OBJ—module parameters are provided via an extensional mechanism while more primitive parameters are provided intensionally. This can be a critical issue when it comes to composability requirements [6].

6. The component model is implicitly defined as part of the language and verification rules rather than being explicitly separated. This means that it is difficult to extract a language-independent model with the same formal basis. Further, entangling the language features and the component model in this way makes it difficult to use the model for many of the purposes to which it might be applied.

7. The use of inheritance for defining new modules in terms of existing ones is conservative. This is related to the requirement for independent formal verifiability of the resulting modules, but a less conservative approach to inheritance could still permit modularity.

8. RESOLVE does not allow nested modules, generic or otherwise.

9. “Variable-length” contextual parameters are not supported.

A.4 Eiffel

Eiffel is one of the more recent object-oriented (OO) languages to be used on industrial development projects. OO approaches in general, and Eiffel in particular, bring important additional insights to the software reuse problem that are not highlighted by the other approaches described here.

As with most OO languages, Eiffel is aimed at achieving quality software by focusing on the extendibility, flexibility, and adaptability of software components. Bertrand Meyer observes that over 40% of maintenance costs result from changes in user requirements, remarking that:

The magnitude of this proportion seems to reflect the lack of extendibility of commonly implemented software: systems are much harder to change than they should be. [46, p. 8]

His proposed approach to software construction focuses on achieving the key qualities of correctness, robustness, extendibility, reusability, and compatibility. These qualities are intended both to improve the overall quality of software and to reduce the cost of maintaining it.
While the OO paradigm embodied in Eiffel is not formally defined, the language itself can be taken as an expression of an implicit model of software construction.

A.4.1 An Overview of Eiffel

Bertrand Meyer [46, p. 12] defines five criteria for evaluating a software construction method with respect to modularity:

- **Decomposability**—a problem can be decomposed into several subproblems that can be solved independently.
- **Composability**—the method encourages producing software elements that may be freely combined with each other, possibly in very different environments, to produce new systems.
- **Understandability**—the method helps in producing software elements that can be separately understood by people.
- **Continuity**—a small change in the problem to be solved results in a change to one, or just a few modules in the system.
- **Protection**—error conditions that arise at run-time in one module are contained within that module without being propagated throughout the system.

These five criteria reflect the properties of modular code that Meyer believes contribute to achieving the key qualities of correctness, extendibility, reusability, and so on. Further, those key qualities also contribute to more maintainable code.

Another principle that Meyer considers to be crucial for good software is the “open-closed” principle. Meyer states that modules should be both open and closed [46, p. 23–25]:

**Open**—“A module is said to be open if it is still available for extension” [46, p. 23]. This is directly aimed at planning for the unplanned—preparing for the eventuality of changing requirements.

**Closed**—A module is closed if it is ready for use by other modules. Meyer equates this with having a stable, well-defined interface. For program modules, a compiled implementation available for use by clients is also necessary.

The open-closed principle ensures that software objects are always ready for use, but also always ready for adaptation, enhancement, or change.
Meyer [46] describes the rationale behind Eiffel, and how it and the OO techniques he proposes measure up as a software construction method against these criteria. Much of the emphasis on extendibility and flexibility led Meyer to design Eiffel as a class-based OO language that relies on inheritance as a primary mechanism for relating modules (classes).

In Eiffel, classes are the central programming concept. A class is a combination of a module and a type: it defines both a type and the operations available on that type. Further, a class can inherit from other classes, allowing modules to be defined by difference. In this case, the superclass is termed the “ancestor” class, while the subclass is termed the “heir.”

Each Eiffel class defines certain “features”: attributes or operations that are defined for that class. Only features that are explicitly exported are available for use by clients, although all features are available to any heir classes. Eiffel further allows features to be constrained by “assertions,” which can be used to record the pre- and postconditions of operations, class (module) invariants, and loop invariants. To encourage correctness, class-level assertions apply to all heirs of the given class (and thus all redefinitions of class features).

Because Meyer realized the necessity of protecting software components from variability of the types that are operated on, he included a form of genericity in Eiffel. Eiffel classes can be generic, with one or more formal class parameters. Meyer refers to this as simple, “unconstrained” genericity, and describes ways that inheritance might be used to simulate other forms of genericity. There is no provision for any direct form of procedures as generic parameters.

Note that in Eiffel, there are two kinds of types: “simple” types (one of the built-in scalar types INTEGER, BOOLEAN, CHARACTER, or REAL), and class types. While variables of “simple” types are defined in the language with the usual semantics, variables of non-simple types are defined to be references to objects to support explicit sharing. Thus, there is a deliberate mixture of reference and value semantics inherent in the language.

Meyer also highlights two other features of Eiffel as being critical for an effective OO language: polymorphism and dynamic binding. Polymorphism allows a given program entity—e.g., a variable—to refer to instances belonging to different classes at run-time. Dynamic binding is an implementation technique whereby the run-time system automatically selects the version of an operation (that may have multiple definitions or implementations) necessary for operating on the current object. Meyer argues for these features in order to make software more extendible and representation-independent. By making software components immune to implementation changes, even at run-time, client software can use multiple implementations of
a given data structure without needing to know which implementation will be used for any particular object of that data type.

A.4.2 Issues Raised by Eiffel

The following subsections provide an overview of the advantages and disadvantages of this language, which capture the issues that it raises with respect to reuse.

Advantages

Eiffel has the following advantages:

1. It provides for polymorphic variables.

2. It provides inheritance for defining new concepts in terms of old, defining new implementations in terms of old, enforcing subtype relationships between data abstractions, defining polymorphic relationships, and more.

3. It explicitly supports unplanned, adaptive reuse through inheritance.

4. It provides for semi-formal specification of the behavior of abstractions.

5. It provides strong encapsulation of the implementation details of an abstraction.

Disadvantages

Eiffel also has several limitations:

1. In Eiffel, the notion of a user-defined type is tightly tied to the notion of a module.

2. Polymorphism in Eiffel is tied to both the typing system and to the (single) inheritance mechanism.

3. The many conflicting uses of inheritance [8] are co-mingled in a single language mechanism.

4. Variables of user-defined types are explicitly defined to be references to objects. Explicit aliasing is not only allowed, it is encouraged. Further, built-in assignment and equality operations have reference semantics. The language also defines \texttt{Clone} and \texttt{Equal} features for every object, which do not have reference semantics, although they only implement shallow copying or equality testing\footnote{Newer versions of Eiffel that post date [46] may have “deep” versions of the \texttt{Clone} operation.}. 

\footnote{Newer versions of Eiffel that post date [46] may have “deep” versions of the \texttt{Clone} operation.}
5. Eiffel supports automatic initialization of objects using programmer-defined features from the corresponding class. It does not, however, support any finalization of objects.

6. Eiffel provides automatic garbage collection for unreachable objects as the primary means of storage reclamation.

7. Eiffel’s specification-like assertion slots were not designed with formal specification or verification in mind, leading to many limitations. Eiffel’s assertions are boolean expressions that must be evaluable at run-time. There is no provision for universal or existential quantification in assertions. Assertion expressions must be written in terms of an object’s state (i.e., attributes of the implementation), so there is no provision for abstraction of specifications if one wishes to use these assertions as a specification tool.

8. The specification and implementation of a class are defined together, making it difficult to effectively separate them. It is possible to separate them by placing the specification in a deferred class from which the implementation inherits, but this is simply a programmer convention.

9. Although classes can be generic, only other classes are allowed to be generic parameters.

10. Class-level invariants are only enforced after object creation and after any call to a routine of that class [46, p. 479]. Since routines of a class can call other routines of the same class, there is no guarantee that invariants will hold before a given routine is called.

11. Eiffel does not distinguish the notion of context, and provides little support for separating it from either concept or implementation.

A.5 Standard ML

ML was introduced in 1977 to be used as the Meta Language for use with a theorem prover called Edinburgh LCF (Logic for Computable Functions) [54, p. 9], which was more akin to a programmable proof checker. The inference rules and proof methods used by Edinburgh LCF were represented as functions, and ML was intended as a vehicle to allow users to express their own proof strategies in the same functional style. In addition to the full power of higher order functions, ML was given an innovative module system for structuring code. ML was also given imperative features
for pragmatic reasons, although its main focus is on declarative programming. Now, ML has a life of its own as a general purpose programming language.

Although several dialects of ML have evolved, the ML community has banded together to promote a common dialect now known as Standard ML. The formal definition [48] includes the denotational semantics of the language, which constitutes a rigorous description of a model of software component construction. Within this dissertation, the name “ML” will always refer to Standard ML.

### A.5.1 An Overview of Standard ML

The presentation of ML is often split into the presentation of the “core” language used to define types and values, and the module system [54, 48, 47]. Note that because ML is a functional language, its notion of “value” includes functions, so that defining values in the core language subsumes the notion of defining functions, and defining types subsumes the notion of defining classes of functions. In this section, however, it is the module system that provides the majority of ML’s contributions to the reuse problem: “ML’s modules may be the most advanced of any language” [54, p. 1], or at least of any non-academic, general purpose programming language.

The terminology for ML’s module system is derived from abstract algebra [54, pp. 275–276]. In abstract algebra, one frequently packages a set of objects together with its operations to form a more abstract object—an algebraic structure. Groups, rings, fields, and so on are common examples of such structures. Further, one can simply describe the names of the required operations and the laws that they must satisfy, forming an algebraic signature. For example, the typical definition of what it means to be a group (a non-empty set along with a binary operation on its elements, where the set is closed under the operation, the operation is associative, the operation has an identity element in the set, and so on) defines an algebraic signature that all groups must meet. Finally, one can consider functors that map algebraic structures to other algebraic structures.

Proceeding from this analogy, an ML module is either a structure or a functor. A structure is simply a collection of declarations, usually of items that are related. These declarations may include types, values, and other structures. This means that structures can be grouped into larger structures, forming complex but self-contained subsystems that can be manipulated as a single unit.

Similarly, an ML functor is a mapping from structures to structures. The body of the functor defines a structure parametrically in a manner similar to the generics of OBJ or RESOLVE. In the abstract, applying such a functor means substituting an actual structure into the functor’s body in place of the corresponding formal
parameter. Laurence Paulson [54, p. 1] describes functors as an “extension” of the usual notion of generic modules as it exists in languages like Ada, since the parameters to a functor may be arbitrarily complex structures, rather than simple types and operations.

In addition to modules, ML also offers signatures, which specify type-checking information about a set of exported declarations. A signature lists the types, the name and type of each value, and the signature of each substructure that is exported. A given structure can then be an instance of this signature if it defines all of the exported entities described in the signature by using conforming declarations. An instance of a signature can declare items that are not specified in the signature, and just as different values can have the same type, different structures can have the same signature. Thus, signatures intensionally define entire classes of structures that share a given syntactic interface.

Note that structures are only a means of grouping declarations, and do not provide any means of hiding information. Viewing a structure through a specific signature provides a way to hide extra details. When a structure is declared, it may be given an explicit signature to hide some of its details, or a signature constraint may be applied later to restrict access to those details. In fact, different signatures could be applied to the same structure at different points to provide views of that structure at different levels of abstraction. A structure declaration need not explicitly mention a signature, however. In this case, ML will infer a signature for the structure that contains the maximum amount of type-checking information and that hides no details.

Structure, functor, and signature declarations in ML are all made in the context of some “environment” of currently active definitions. Simplistically, this environment is the set of declarations that have been made prior to the current declaration. Hence, all three entities can make direct references to external definitions in that environment. To aid in modularity, the original designer of ML’s module system proposed the “Signature Closure Rule”: signatures can only refer to structures specified within themselves, but not to free-standing external structures [54, p. 247]. Every signature that obeys this rule is completely self-contained, carrying around inside itself the definitions on which it depends. While this rule has been relaxed in most implementations to allow more freedom for programmers, it is enforced by some compilers on separately compiled modules to simplify the compilation process.

While functors seem to be an elegant implementation of generics, they are interesting because of the novel ways in which they are used by ML programmers. Because a functor takes a structure as an explicit parameter, it is common for all of a functor’s external dependencies to come in through that parameter, promoting a very modular form of programming. This also allows modules to be developed and compiled in-
A.5. STANDARD ML

dependently. Once the functors are constructed, they can be linked together simply by applying them. Because parameters to functors are specified through signatures, there are no direct dependencies between program units, and it is easy to plug together alternative configurations. This all-functors style of programming [54, p. 270] has very powerful benefits for reuse.

ML also provides sharing constraints, which allow functors to require that certain subparts of their parameters are identical—for example, requiring that a list abstraction and a tree abstraction both operate over the same type. This is very critical for complex contexts, where a functor imports many subsystems through its parameter mechanism. If the functor expects the same key building block to be used by several of the subsystems in its context, it must be able to express this restriction.

Finally, ML also supports a form of polymorphism based on type schemes [54, p. 55]. A type scheme is a template or pattern that describes a relationship among one or more parameters that are also types. For example, the identity function in ML has the type scheme \( \alpha \to \alpha \), where \( \alpha \) can be considered as a variable representing some ML type. This identity function could be applied to an object of any type, and from the type scheme of the function, we know the range of the function will always be the same type as its domain. This form of polymorphism is interesting because it is amenable to static type checking and prevents run-time type errors, while at the same time allowing a great deal of the flexibility programmers have in polymorphic languages that do not support static type checking.

A.5.2 Issues Raised by Standard ML

Like OBJ, ML provides an elegant and powerful module abstraction for packaging functional definitions. ML also provides support for a novel form of polymorphism that is distinct from the usual OO interpretation. Further, the denotational emphasis of the Standard ML definition ensures that all of these features have a well-documented mathematical meaning which can serve as the basis for language-independent developments. Taken together, ML provides many insights for constructing reusable software, and the issues it brings to this research are reflected in the advantages and disadvantages presented in the following subsections.

Advantages

Standard ML provides the following advantages for constructing reusable software components:

1. It supports the notion of an abstract description of a component's interface through signatures.
2. Because the formal semantics of ML define a subsumption relationship over signatures, signatures can also be used to represent more generalized descriptions of component interfaces.

3. It supports the notion of a component implementation through structures.

4. It allows multiple structures to match the same signature, thereby permitting multiple implementations for the same concept.

5. It supports many-to-many mappings between structures and the signatures that they satisfy.

6. It allows contextual dependencies to be separated from a component through generic parameters.

7. The Signature Closure Rule advocates complete separation of contextual dependencies from modules.

8. The all-functors programming style supported by ML requires the complete separation of contextual dependencies from all modules.

9. Because generic parameters to an ML functor can be structures, a mechanism is provided to structure the contextual space.

10. ML provides an intensional parameter conformance mechanism for use with generic parameters [7]—any structure that can be mapped to the signature defining a formal parameter can be substituted as the corresponding actual.

11. It provides direct support for nested modules, although it does not allow nested functors.

Disadvantages

Standard ML also has the following disadvantages:

1. Signatures only capture the syntactic aspects of a component's interface.

2. The mechanism by which structures correspond to signatures is based only on syntactic information, and ignores the behavior of the types, operations, and substructures involved.

3. Because of the way signature-to-structure conformance is defined, it is not possible to separate the generic parameters representing conceptual context from those that represent context of the implementation.
4. ML allows structures to serve as a description of their own (implicit) signature, so that the two notions of concept and content can be co-mingled in the same entity.

5. It lacks a distinct notion of “context”—it indirectly addresses this need instead by defining some language mechanisms that can be used to provide context (e.g., generic parameterization and importation).

6. It does not provide any mechanisms for defining new modules as extensions or enhancements of existing ones (inheritance).