CHAPTER V
Conclusion

This chapter summarizes the research conducted for this dissertation, and presents the conclusions drawn from this work. Some open issues and future work are discussed, and the chapter concludes with a presentation of contributions to the field.

5.1 Summary and Conclusions

The following subsections summarize and draw conclusions from the three primary divisions of this work — reusability issues, definition of RESOLVE, and interaction between programming language and environment design. These subsections essentially summarize chapters 2, 3, and 4, respectively.

5.1.1 Software Reusability

Software quality and programmer productivity are two of the biggest challenges facing the software engineering community. Reusability, a mainstay of other engineering disciplines, is an approach to software development that addresses both of these issues. A designed-for-reuse software component is economically efficient to design and build, it most likely is of a higher quality than a "scavenged" part, and reusing it increases the productivity of client programmers. Despite these advantages, there are both technical and non-technical impediments to widespread software reuse.

A reusable software component is defined as one that can be incorporated into a variety of programs without modification (except possibly parameterization); furthermore, reusing a software component should not compromise other software engineering principles such as information hiding and data abstraction.

A reusable component has several important characteristics. First, it is formally specified by mathematical statements that provide a complete and unambiguous description of the component. Formal specification is important because it tells a potential client programmer exactly what the part does, it tells an implementor of the part

exactly what the implementation must accomplish, it is necessary if formal verification is to succeed, and the process of writing a formal specification often helps reveal ambiguities and inconsistencies in the part's design. In addition to formal mathematical specification, it is important to include an informal natural language description of the component because it can be instrumental in helping a programmer (both client and implementation) understand the part's function.

A second characteristic of a well-designed reusable component is the separation of the part's specification and implementation into separately-compilable units. This is an effective method of realizing information hiding, it permits designers and programmers to consider a part's abstract function and performance as separate issues, it allows a specification to have multiple implementations, and it permits a client program to be compiled and verified before any implementation exists.

Third, a reusable component should be generic if possible. A generic part is one whose definition is parameterized somehow (e.g., by a component type). A generic part is a template for a family of parts, and must be instantiated to create a usable part. Generic parts achieve reusability in an obvious way.

Fourth, it should be possible for a specification to have multiple implementations, each with possibly different performance characteristics. This increases the likelihood that a part will be reused because clients place different performance requirements on a part. A client programmer should be allowed to choose an implementation for one part, and another implementation for a different instance of that same part. Also, it should be possible for a client programmer to select a different implementation of a part without having to recompile or reverify the client program.

A fifth and final characteristic of a well-designed reusable part is that efficient implementations must be possible, because a part that has no efficient implementation will be neither used nor reused. Perhaps the biggest implication of this is that an implementation should not rely on copying as a primary means of moving data.

Using these five characteristics, the features of a programming language can be characterized by how well they encourage and facilitate the design and implementation of reusable software components. For example, it is difficult to formally specify a program written in a language having implicit pointers and aliasing, and some parts have

no efficient implementation in a language where copying is the only data movement primitive. A survey of nine representative modern programming languages — Anna/Ada, Modula-2, Euclid, Gypsy, Alphard, C++, Eiffel, Larch/CLU, and Z — reveals that none adequately supports the design and implementation of reusable components. This supports the first point of the thesis, namely that no modern programming language has all of the constructs necessary to encourage and facilitate the design of reusable software parts; in fact, existing languages have constructs (e.g., implicit pointers and copying as the only data movement primitive) that actually thwart the design of reusable software.

5.1.2 RESOLVE Programming Language

A language called RESOLVE (an acronym for REusable SOftware Language with Verifiability and Efficiency) was developed to demonstrate that it is possible to have a practical language incorporating constructs that encourage the design and implementation of reusable components. RESOLVE is an imperative language, with control structures similar to those found in most structured languages such as Pascal and Ada. A program is organized as a collection of separately-compiled modules. The behavior of each module is formally specified in a conceptualization, with the structures and code implementing each conceptualization contained in a realization. A conceptualization typically defines a type and a set of operations with one or more parameters of that type. Conceptualizations may be generic, and several realizations may exist for any conceptualization.

RESOLVE distinguishes between mathematical entities (types, domains, and variables) and program entities (types, domains, and variables). A mathematical domain is an anonymous set of anonymous values defined by an instance of a theory module, which defines a set of axioms that implicitly define the domain and a notation for expressing functions and relations among the values in this domain. A mathematical type is the name given to a mathematical domain by the client that instantiates the theory module. A mathematical variable is a symbol that stands for some value of its type's domain. Two mathematical types are equivalent if and only if they are names for the same domain.

Similarly, a program domain is defined by an instance of a conceptualization. Each element in a program domain has a concrete representation (which may be as low-level

as a configuration of bits in memory) and is modelled by an element from a corresponding mathematical domain. The elements in a program domain are those values that are "reachable" by executing the operations defined in the conceptualization. When a conceptualization is instantiated by a client, a marker is created for each program domain defined. A program type is the name for a marker, which in turn designates a program domain. Two program types are equivalent if and only if they are names for the same marker. A program variable is a symbol that stands for some value from the domain of its type; however, because every value in a program type's domain has a mathematical model, it is possible to *reason* about the value of a program variable as a mathematical value.

Program type equivalence and mathematical type equivalence are defined differently. Effectively, the mathematical types defined by two identical instances of a theory are equivalent, whereas the program types defined by two identical instances of a conceptualization are not equivalent; in fact, markers are introduced in the discussion of program types solely to make these two program types inequivalent. The justification for this is based in part upon our design philosophy that realizations determine performance characteristics of the operations and do not affect the contexts where those operations can be legally invoked. In other words, changing the realization of a facility should not alter the syntactic correctness of the module. Thus, program types defined by two identical instantiations are inequivalent, so that if the realization of one instance changes, the syntactic legality of the program can't change.

Every program type has an initial value specification defined for it. Every (program) variable is guaranteed to have a value that meets the initial value specification for its type before the variable's first reference. Initial values are important for the formal specification of a program, and to guarantee that variables always have some legitimate value from the domain of its type.

The effect of an operation is formally defined using pre- and post-conditions, which are assertions about the values of the operation's parameters before and after the operation invocation, written in first-order predicate calculus. Within these assertions parameters are considered to be values from the mathematical domain of the parameter's program type's model, and the assertions involve functions and relations in the theories defining those types. The pre-condition is specified in a requires clause, and is an assertion the

operation assumes to be true when it is invoked. The post-condition is specified in an ensures clause, and is an assertion the operation guarantees to be true when it returns, provided the requires clause was true when the operation was invoked.

A realization contains the data structures and code that implement the program types and operations specified in a conceptualization. A correspondence is defined that maps values of a type's representation (considered as mathematical values) to the type's model. To accomplish initialization, an initialization routine is defined for every program type specified in the conceptualization, which is automatically invoked for every variable of that type. Also, variables whose representations involve dynamically-allocated memory should have that memory released when it is no longer needed. To accomplish this, a type has a finalization routine, which is automatically invoked for every variable of that type at the end of the block in which the variable is declared.

An interesting feature of RESOLVE is that no types are built-in to the language. Instead, every type is provided by some conceptualization, including those normally built-in such as integer, character, and boolean. Likewise, structured types such as arrays and records are not defined in RESOLVE, but are defined by conceptualizations. A similar approach is taken with respect to pointer variables. This design makes RESOLVE very regular. However, it also involves defining control operations, which are special in that they can be invoked only within if and while statements.

Another feature of RESOLVE is that data is moved by swapping the contents of two variables, rather than copying the contents of one variable to another. The ability to make a copy of a data value is not a built-in operation in RESOLVE, and RESOLVE does not define the traditional "copy" assignment statement. If it should be possible for a client to make a copy of a variable of a particular type, it is the responsibility of the writer of the conceptualization providing that type (or one that uses it) to specify an operation that accomplishes this; a copy operation defined this way is invoked just like any other operation. All parameter passing is defined in terms of swapping actual parameters with corresponding formal parameters.

Defining swapping as the only data movement primitive offers two advantages over traditional languages defining copying as the data movement primitive — it is possible to implement swapping so it takes constant time to execute, and swapping guarantees that implicit aliasing never occurs. The former property is important for efficient

implementation of generic components, and the latter for formal specification and verification.

RESOLVE is a programming language that has the constructs necessary to encourage and facilitate the design of reusable software parts, and is the "proof by demonstration" supporting the second point of the thesis, namely that it is possible to have a usable language incorporating constructs that encourage the design of reusable software components.

5.1.3 Interaction of Programming Language and Environment Design

A programming language and its program development environment are interrelated in the sense that the ease with which programs can be developed depends upon the facilities provided by the environment, which in turn depend upon the constructs defined in the programming language. Some influences that programming language design and environment design have on each other were explored. Structure and syntax directed editors, hardware platforms, and the development of environment components such as librarians are recent examples of the influence that programming language design has on programming environments.

Many recent programming languages were not significantly influenced by program environments. However, an editing environment was designed concurrently with RESOLVE, and influenced the language in several significant ways. In particular, RESOLVE keywords are not abbreviated, parameter and argument lists are such that statements do not become arbitrarily wide, and several complex syntactic constructs such as operator overloading and infix notation are handled as alternate syntax by the editor rather than complicating the syntax of the language. This is an example of a programming language whose design was influenced in positive ways by the environment, and supports the third point of the thesis, namely that when a programming language and editing environment are designed at the same time, each can influence the other in positive ways.

5.2 Future Work

There are several areas that hold promise for productive future work involving both the language design and environment. Four of these are discussed here.

5.2.1 Enhancements

There are instances where a conceptualization, say A, is very similar to another one, say B, in the following sense: A provides identical definitions for all items defined in B, in addition to defining one or more new items. We could say that A enhances B. For example, a conceptualization defining type Stack with operations Push, Pop, Is_Empty, and Copy is similar to (i.e., an enhancement of) one defining everything but Copy.

Although this feature is included in many "object-oriented" languages, it is not currently in RESOLVE. The challenge is to define enhancements without sacrificing other desirable features of RESOLVE (e.g., formal specification and verifiability).

One proposal is to extend RESOLVE to allow a definition such as the following:

```
conceptualization Stack_With_Copy_Template
  enhances Stack_Template

  interface
    procedure Copy
      parameters
        preserves s1 : Stack
        produces s2 : Stack
      end  parameters
      ensures "s2 = #s1"
  end  interface

end Stack_With_Copy_Template
```

Conceptualization Stack_With_Copy_Template provides a client with all items defined in Stack_Template (i.e., type Stack and operations Push, Pop, and Is_Empty), plus operation Copy. An instance of Stack_With_Copy_Template can be used in any context where an instance of Stack_Template is required (e.g., passed as an actual facility parameter to an instantiation); this is allowed because Stack_With_Copy_Template is a "superset" of Stack_Template.

Realizations of Stack_With_Copy_Template are obligated to implement *all* operations, including those defined in Stack_Template. Of course, a realization may want to implement some of these operations using a realization of Stack_Template, and syntax for this should be defined. Note, however, that new syntax is not required, as demonstrated here:

```
realization of Stack_With_Copy_Template by Stack_Copy_Real_1
```

```
realization  auxiliary
  facilities
    Stack_Fac is Stack_Template (Item)
      realized by Stack_Real_1
  end  facilities
end  realization  auxiliary

interface
  type Stack is represented by Stack_Fac.Stack
    exemplar s_rep
    correspondence "s_rep = s"
  end Stack

  procedure Push
    parameters
      alters s : Stack
      consumes x : Item
    end  parameters
    begin
      Stack_Fac.Push (s,x)
  end Push
    .
    .   similarly for Pop and Is_Empty
    .
  procedure Copy
    parameters
      preserves s1:Stack
      produces s2 : Stack
    end  parameters
    begin
      .
      .   code to implement Copy in terms of Push, Pop, and
      .   Is_Empty
  end Copy
end  interface
end Stack_Copy_Real_1
```

Note that this realization has no more access to Stack_Real_1's representation of Stack_Template than any other client (i.e., information hiding is enforced).

There are several advantages of allowing enhancements. First, enhancements permit secondary operations to be encapsulated with the primary ones, allowing clients to reuse code for secondary operations. Second, very efficient implementation of some secondary operations may be possible if the realization implements all operations — both primary and secondary. For example, [Weide 86b] and [Pittel 90] describe an implementation of Copy for stacks that executes in constant time, but is not a secondary operation in the sense that it is not coded in terms of primary operations.

Enhancements appear to offer many of the advantages of inheritance, without adversely affecting the prospects for formal specification and verification.

5.2.2 Accessors

Currently in RESOLVE, accessing items within a structure (e.g., an array) is accomplished by invoking an operation that swaps an actual parameter with an item in the structure. For example, '`Access (a,i,x)`' swaps the ith element of array a with variable x. Swapping the jth element of a with the ith element requires three invocations of Access.

It might be useful to introduce a fourth kind of operation — an 'accessor' — into RESOLVE. An accessor would swap an item within a structure with some other value, and would be invoked within a swap statement. For example, an accessor called Arr_Access could be invoked by '`Arr_Access(a,i) :=: x`', which has the same effect as '`Access (a,i,x)`', or by '`Arr_Access(a,i) :=: Arr_Access(a,j)`', which swaps the ith with the jth element of a, or by '`Push(s,Arr_Access(a,i))`', which pushes the original value of the ith element onto stack s, leaving it with an initial value. In other words, an accessor could be invoked anywhere a variable is allowed.

5.2.3 Synoptic Comments

Comments often play a less-than-prominent role in programming language design, yet their importance in understanding and explaining a program should not be underestimated. It might be useful to extend the notion of comment to be more than just white space. Specifically, the inclusion of "synoptic comments" that are explicitly associated with program code, similar to Kaelbling's scoped comments [Kaelbling 88], are proposed.

A synoptic comment would be bound to program structures as the program is constructed in the editor. A programmer would have several options regarding the display of synoptic comments and their associated code — both the comment and code could be displayed, only the comment and not the code, or only the code and not the comment. This would allow a programmer to hide irrelevant code when a comment summarizing the action of the code is sufficient.

Given an editor such as the one developed for RESOLVE, it might be possible to bind code to the synoptic comment by dragging a tool of some sort over the code. In this case, the binding is not textual, but visual.

5.3 Contributions

The focus of this research has been to investigate programming language features that either encourage or discourage the design of reusable software components — in other words, the influence of reusable software on programming language design. The following are the main contributions to the field:

Characterization of Reusable Software Components — Five characteristics of a well-designed reusable software component are: it is formally specified, its specification and implementation are in separately-compilable units, it is parameterized (i.e., generic) if possible, it may have multiple implementations, and it is possible to construct an efficient implementation. Evaluation of several modern programming languages with respect to how well they support components exhibiting these characteristics suggests that none has all of the features necessary to encourage and facilitate the design and implementation of reusable software components.

Definition of RESOLVE — A new programming language, called RESOLVE, is defined that has all of the necessary constructs to encourage and facilitate the design and implementation of reusable components. The next two contributions are actually features of RESOLVE.

Swapping as a Data Movement Primitive — Copying the contents of one variable to another is the only data movement primitive defined in modern programming languages, impacting the ability to develop efficient implementations of generic parts. In RESOLVE, the only data movement primitive is swapping the contents of one variable with another. Swapping is efficient to implement for all types (i.e., a constant time operation), and is safe in the sense that it does not introduce aliasing.

Simple Yet Powerful Definition of Types and Type Equivalence — Types and type equivalence (both mathematical and program) are defined in terms of abstract sets and functions. These definitions are unambiguous and powerful, yet easy to understand and implement.

Alternate Syntax — The alternate syntax mechanism allows the programmer to benefit from syntactic constructs such as overloading and infix notation without complicating the design of the programming language or its parser. This is accomplished by having the syntax directed editor process alternate syntax.