CHAPTER I

Introduction

Computers and the software that control them have a profound impact on our everyday lives. They control our microwave ovens when we cook, allow us access to our checking accounts at automatic tellers, assist doctors in diagnosing our maladies, and play an integral role in our nation's defense, to name a few. In fact, our current standard of living and lifestyle (and for some, literally their lives) would not be possible without computers and computer software.

The reliability of these computer systems is by no means perfect. Stories of computer failure abound in the popular press and trade publications, and "computer error" is a contemporary phrase everyone knows. For example, the first scheduled launch of the space shuttle *Columbia* was postponed because of a synchronization problem among the five on-board computers, caused by a software bug in which a delay factor was reset from 50 to 80 milliseconds [Joyce 85]. After five years and twenty space flights, the *Columbia* astronauts were still being supplied with a book, *Program Notes and Waivers*, containing a list of known software problems, ranging from interleaved messages on the display, to a bug where the contents of a ground communication buffer was lost if an astronaut happened to be typing while a program was being uploaded [Joyce 85]. In 1986 a software bug in a linear accelerator exposed a cancer patient to a massive radiation dose of 16,500 rads in less than a second, orders of magnitude over the normal 2-8 rads per second, resulting in the patient's death [Fitzgerald 87]. More recently, a bug in switching software resulted in a nine-hour failure of the AT&T long-distance network in January 1990 [Fitzgerald 90]. The list goes on and on.

A direct consequence of these sorts of software quality problems is the fact that software engineers are beginning to be held legally liable for their software products. Computer litigation is among the fastest growing areas of law, and computer "lemon laws" are a real possibility in the near future [Carver 88]. This provides the software engineering

community even more incentive for developing the tools and techniques for building error-free software.

Why is it so difficult to design software that works? It seems that many software engineers regard design of error-free software as software engineering's holy grail. Articles such as [Poston 87] treat zero-defect software as an unrealistic goal, while [Currit 86] and [Lipow 82] assume that software *will* fail, and define estimates for mean time to failure (MTTF) of software and number of faults per line of code, respectively. It is interesting to note that other engineering disciplines don't seem to need these measures. For example, civil engineers don't calculate the mean time to failure of the *design* for a bridge (although they may calculate the MTTF of the actual bridge structure due to deterioration of materials or negligent maintenance).

Although the software engineering community realizes that error-free software is very difficult to design, not all software engineers are as pessimistic as those just mentioned. For example, Hamilton [Hamilton 86, Hamilton 76] believes that developing error-free software is possible, though very difficult. An *a priori* assumption of the work presented here is that zero-defect software is a viable goal in any software project.

Another problem facing software engineers is the productivity of the people building software. In recent years the productivity of computer hardware engineers has increased dramatically, while software productivity seems to be holding its own at best, with code being produced at the same old rate of one to two delivered lines of code per person-hour [Boehm 87]. Even relatively small increases in software productivity would be economically significant. For example, it is estimated that a mere 20% improvement in software productivity would be worth $45 billion in 1995 for the US and $90 billion worldwide [Boehm 87].

Designing reusable software components is an approach to software engineering that attacks both of these problems (i.e., software reliability and productivity). Reusability is a mainstay of other engineering disciplines [Stovsky 90], and its applicability to software engineering is promoted by many software engineers (e.g., [Meyer 88], [Brooks 87], [Biggerstaff 87], and [Hamilton 86]). Surprisingly, the state-of-the-practice suggests that designing reusable software parts is presently not encouraged [Meyer 88]. This is not to say that software is not reused, but rather that software

components in general are not presently *designed* specifically for reuse within larger systems.

The Reusable Software Research Group at Ohio State has been investigating issues relevant to the design of reusable software components. The issues addressed by our group include programming language design, program development methodology, programming environment design, program specification and verification, program implementation, and program testing. These issues are not being addressed in isolation, but in the larger context of design of reusable software parts.

The work presented here is the identification of the salient features of a programming language and environment that encourage the development of reusable software components. The programming language and system developed from this research is called RESOLVE, for REusable SOftware Language with Verifiability and Efficiency.

Programming languages have a profound impact on software reuse, since they strongly influence the way a programmer designs and reasons about a program. In this context the following three points comprise the thesis of this work:

- One of the biggest technical impediments to the widespread design and (re)use of software is that no modern programming language has an adequate set of constructs to support the design of reusable software parts. In fact, existing languages are based on some constructs that actually thwart the design of reusable software.

- It is possible to design a practical programming language based entirely upon constructs that support and even encourage the design of reusable software parts.

- When a programming language and editing environment are designed at the same time, each can influence the other in positive ways.

The approach taken in supporting this thesis is to define the characteristics of "good" reusable parts, and to identify the programming language features necessary to support the design and development of such parts. Several important contemporary programming languages are then evaluated with respect to these language features, demonstrating that none has all of the features necessary to support the development of

reusable parts. Next, a programming language is defined that does have an adequate collection of features; this language is called RESOLVE, and is the primary contribution of this research. Finally, a prototype RESOLVE editing environment is discussed that points out some ways the environment influenced the design of RESOLVE, and also compares this environment to several popular programming environments.

The dissertation is organized as follows. Chapter 2 contains background definitions and discussion of the issues surrounding reusable software components and the influence of programming languages on their design. Also included in this chapter is a presentation of features of several programming languages and a demonstration that none has all of the constructs necessary to encourage and facilitate the development of reusable software parts. Chapter 3 contains the definition of RESOLVE and argues why it is appropriate for the design and implementation of reusable software components. Chapter 4 contains a description of the prototype RESOLVE program development environment, and Chapter 5 discusses possible future work and conclusions drawn from the work so far.