Semantic Spaces for Specifications and Templates

David S. Gibson
Bruce W. Weide

Department of Computer and Information Science
The Ohio State University
Columbus, OH 43210-1277 USA
Tel: +1-614-292-1517
Fax: +1-614-292-2911
Email: {dgibson|weide}@cis.ohio-state.edu
URL: http://www.cis.ohio-state.edu/rsrg

Abstract

Component-level maintenance involves changing the behavior of a system in useful and predictable ways by removing and replacing entire components. In order to perform component- level maintenance, an engineer must understand not only the structural relationships but the *behavioral* relationships between the component to be replaced and the potential replacement. Using mathematical modeling to help achieve this understanding entails having a single formal framework that captures the behaviors of components and systems *and* the meanings of various associated artifacts such as behavioral specifications. There is a rather straightforward and quite general way to do this.

Keywords: Component-based systems, formal semantics, formal specifications, maintenance, object-oriented programming, parameterized programming, templates.

Workshop Goals: Learning about related work, feedback, networking.

1 Background

As members of the Reusable Software Research Group (RSRG) at The Ohio State University, we have been exploring various aspects of software component engineering for over ten years. One recent focus of our research has been on the identification, formalization, and expression of dependency and behavioral relationships between software components in particular, and analogous components of arbitrary systems in general. Our ideas about component relationships are based on the results of RSRG research which have been incorporated into the RESOLVE framework, language, and discipline for software component engineering [SW94].

To demonstrate the application of these ideas for software component relationships, we have developed the RESOLVE/Ada, RESOLVE/C++, and RESOLVE/Ada95 disciplines for software component engineering. These disciplines use the language mechanisms of C++ and Ada to encode language-independent software component relationships. But as suggested above, we also are interested in exploring the relationships between software systems and other engineered systems, especially in terms of the mathematical modeling techniques that support good design. This interest is one of the motivating forces behind the work outlined here.

2 Position

One of the most striking differences between typical physical systems and software systems is the difficulty involved in replacing one component with another, and predictably achieving a desired effect. For example, consider a common table lamp whose components are a base, a switched socket, an electrical cord, a shade, and a bulb. We can change the behavior of the lamp simply by replacing one light bulb with another (say, of different wattage) and we have no trouble predicting what effect will be observed. Why can't component-based software be maintained in a similar way? Simply and gently put, software engineers generally do not know how to build software in which the behavioral effects of changes are localized. This is not to say it is impossible or unknown how to do this; quite the contrary. In fact, we hereby assume when discussing software systems the use of a component-based software design discipline and framework — such as the RESOLVE approach [SW94] — in which it is technically and practically possible to reason modularly about the behavioral effects of software changes in general, and component substitution in particular [EHMO91, EHO94, Hey95, Gib97].

A central problem in the maintenance of a component-based system is the ability to understand the *meaning* of a component. Consider a software system implemented in C++. For a typical *operational program component*, such as a class, "meaning" generally is taken to be the operational behavior arising from the component at run time. However, there are at least two other kinds of artifacts involved in modern component-based software systems:

- *specifications* of operational program components, such as Larch [GH93], RESOLVE, or $\mathcal{Z}$ [Spi88] formal specifications, which are used to record the required or intended functionality of operational program components; and
- *templates*, or generic (parameterized) components, such as Ada generic packages, C++ class templates, or RESOLVE realization modules, which are *instantiated* to generate what amount to operational program components.

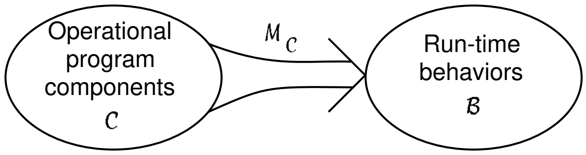

Figure 1: $\mathcal{M}_\mathcal{C} : \mathcal{C} \to \mathcal{B}$

Although specifications and templates do not have behaviors in the same sense as ordinary operational components, they must and do have meanings. What are these meanings, and how are they related to the meanings of operational program components? Software systems are symbolic entities so the standard notion of semantics makes these questions sensible. Other physical systems are not symbolic, but like software systems they do exhibit behaviors — and their design documents are symbolic. Is there a way to develop a mathematical modeling framework for system behavior that applies equally well to physical systems and to software systems? Such a framework should prove useful, for instance, in dealing with formal models of embedded systems, where the physical system being controlled and the software inside it must be analyzed and/or designed together. It also should help illuminate deeper connections between well-engineered physical systems and well-engineered software systems.

Our position is:

> There is a rather straightforward and general framework for *integrating* two related notions: modeling the behavior of systems, and modeling the meanings of various artifacts associated with those systems (including specifications and templates).

We focus attention here on component-based software systems, generally leaving to the reader the challenge of making the leap to other physical systems — with only the slightest hints from us as to how to do so. During workshop discussions we hope to be able to pursue these connections.

2.1 Problem Being Addressed

The purpose of defining a formal semantics for a language is to give precise meanings to the strings in that language. In principle, it is relatively well understood how to define the semantics of operational program components in a programming language such as C or Pascal. Of course as a practical matter, defining the formal semantics for most programming languages is an enormous chore, but this is unimportant for the current paper. What we seek is simply a framework in which such a semantics *might* be defined, as illustrated in Figure 1. For each string which is an operational program component in the language (i.e., an element of $\mathcal{C}$), the semantic function $\mathcal{M}_\mathcal{C}$ assigns to it a run-time behavior (i.e., an element of $\mathcal{B}$). In this paper we constrain neither the structure of $\mathcal{B}$ nor the semantic function $\mathcal{M}_\mathcal{C}$, except to stipulate that $\mathcal{M}_\mathcal{C}$ is total.

Now consider a combined specification-and-implementation language such as RESOLVE, or a hybrid that combines a specification language (e.g., Larch or RESOLVE or $\mathcal{Z}$) with a modern implementation programming language with templates (e.g., Ada or C++ or RESOLVE). As shown in Figure 2, such a language also contains strings which are specifications and templates. As noted above, the meanings of these strings might not be elements

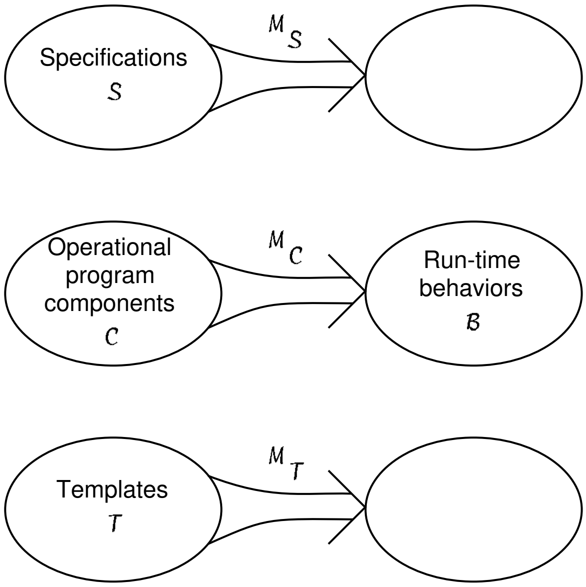

Figure 2: $\mathcal{M}_{\mathcal{S}} : \mathcal{S} \rightarrow ???$ and $\mathcal{M}_{\mathcal{T}} : \mathcal{T} \rightarrow ???$

of $\mathcal{B}$. What *do* specifications and templates mean? And more important, how are these meanings connected to those of operational program components?

2.2 Significance for Component-Based Software Engineering

Answers to these questions are important for component-based system engineering because they are a necessary link in defining precisely certain *relations* involving operational program components, specifications, and templates. These relations are central to understanding component-based systems and to performing component-level maintenance activities [EGWZ97, Gib97] because they allow a software engineer to reason about the behavioral effects of substituting one operational program component for another. Here is a quick intuitive overview of some of the most important relations, with C/C_i, S/S_i, and T/T_i taken to be operational program components, specifications, and templates, respectively:

- C_2 ***uses*** C_1: the behavior of C_2 depends on the behavior of C_1.
- C ***implements*** S: the behavior of C conforms to the behavior specified by S.
- S_2 ***extends*** S_1: any operational program component that ***implements*** S_2 also ***implements*** S_1.
- T ***needs*** S: T requires as an actual parameter some (any) operational program component that ***implements*** S.

The first relation, ***uses***, is the usual connection between software components. This connection allows $\mathcal{M}_{\mathcal{C}}$ to be "constructed" bottom-up [Edw95]. That is, suppose C_2 mentions C_1 in its context (i.e., brings it into scope, in the programming language sense). If C_2 invokes some operation of C_1, then $\mathcal{M}_{\mathcal{C}}(C_2)$ is defined in terms of $\mathcal{M}_{\mathcal{C}}(C_1)$. The meaning of the ***uses*** relation as defined above can be formalized independently of the meanings of specifications and templates. But the meanings of the other relations above cannot be so formalized; yet these other relations lie at the heart of component-based system design and maintenance [Gib97].

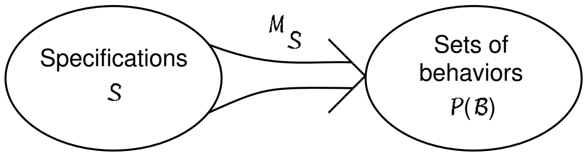

Figure 3: $\mathcal{M}_\mathcal{S} : S \to \mathcal{P}(\mathcal{B})$ (where $\mathcal{P}(\mathcal{B})$ is the powerset of $\mathcal{B}$)

2.3 A Semantic Space for Specifications

Figure 1 can be interpreted as arising from the scientist's view of the world, as opposed to the engineer's view [Cha97]. That is, the scientist faces this situation: "I need to explain this artifact or phenomenon ..." The scientist's attack is to build a mathematical model of the artifact or phenomenon in question. For software the result of this activity is denoted in Figure 1 by $\mathcal{M}_\mathcal{C}$, but entirely parallel diagrams arise for other sets of physical artifacts and/or phenomena and their associated model spaces. The engineer faces a slightly different situation: "I need some artifact or phenomenon such that ..." The engineer's task is not so much to model something which exists and needs to be explained — though that is certainly part of it — but something which *ought* to exist. Possibly many particular implementations can satisfy the engineer's specifications.

Figure 3 illustrates how we may use this observation to view the semantics of specifications. The meaning of a specification is the set of behaviors that we wish to allow as those of conforming implementations. Indeed this directly becomes the definition of the ***implements*** relation:

$$C \textit{ \textbf{implements} } S \equiv \mathcal{M}_\mathcal{C}(C) \in \mathcal{M}_\mathcal{S}(S) \tag{1}$$

Similarly, it leads directly to the definition of the ***extends*** relation:

$$S_2 \textit{ \textbf{extends} } S_1 \equiv \mathcal{M}_\mathcal{S}(S_2) \subseteq \mathcal{M}_\mathcal{S}(S_1) \tag{2}$$

Here is how this approach might pan out for a software example. Suppose operational program component C is a procedure (with possibly relational, i.e., non-deterministic behavior) and specification S is a procedure specification given by a precondition and a postcondition. We might consider a point in $\mathcal{B}$ to be a binary relation on a state space Σ (i.e., a set of ordered pairs from $\Sigma \times \Sigma$), where (σ_1, σ_2) is in $\mathcal{M}_\mathcal{C}(C)$ iff C is permitted to start with its environment in state σ_1 and, when doing so, might change its environment to state σ_2. The precondition and postcondition from S also define a binary relation $\rho(S)$ on Σ, where the precondition says where $\rho(S)$ is defined and the postcondition defines it there. We would define $\mathcal{M}_\mathcal{S}(S)$ as the *set* of binary relations on Σ which are at least as defined as $\rho(S)$ and which are consistent with it where both are defined.

Notice that this framework lets us consider the meanings of specifications for systems other than programs. For example, Figure 3 and the definitions of ***implements*** and ***extends*** make sense even if $\mathcal{C}$ is the set of possible cars or TV sets (or even mere descriptions thereof), $\mathcal{B}$ is the set of possible behaviors of cars or TV sets, and $\mathcal{S}$ is the set of instruction manuals for cars or TV sets, respectively.

Our suggested semantic space for specifications might seem so simple as to be obvious or trivial. But the apparent simplicity is deceiving. In fact, we do not know of any other work

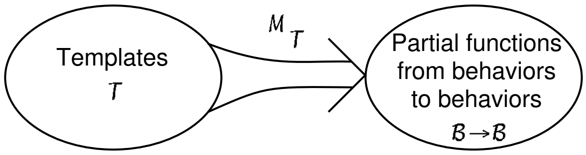

Figure 4: $\mathcal{M}_{\mathcal{T}} : \mathcal{T} \rightarrow (\mathcal{B} \rightarrow \mathcal{B})$

that addresses the question of how to define the semantics of specifications in a domain-independent way.[1] Even limiting the focus to software specifications, we have seen little previous work on the question of tying specifications' formal semantics to those of operational program components. Stand-alone specification languages such as Larch and $\mathcal{Z}$ have formal semantics, but in them a specification such as S above apparently would denote the single relation $\rho(S)$ — which is in $\mathcal{B}$, not in $\mathcal{P}(\mathcal{B})$ as in our framework. RESOLVE semantics are based on the ACTI formal model of component-based software [Edw95], which defines semantic spaces for several kinds of software components. Among these are operational software components ("concrete instances", tantamount to $\mathcal{C}$) and specifications ("abstract instances", tantamount to $\mathcal{S}$) for modules exporting types and/or operations. The ACTI semantic space for abstract instances is, however, essentially structurally identical to that for concrete instances. What we propose here is different. It is a general construction where the semantic space for specifications is a higher-order space, obtained directly and mechanically from $\mathcal{B}$, which is unrelated to the internal structure of $\mathcal{B}$.

2.4 A Semantic Space for Templates

Figure 4 illustrates how we may view the semantics of templates. The meaning of a template is a partial function from behaviors to behaviors. This leads us to define the dependency relationship that is most important for a template, as follows:

$$T \textbf{\textit{ needs }} S \equiv \mathbf{dom}(\mathcal{M}_{\mathcal{T}}(T)) = \mathcal{M}_{\mathcal{S}}(S) \tag{3}$$

The traditional view of a template is that it is a "transformation" which "generates" an operational program component when it is *instantiated*, i.e., when it is given another operational program component as an actual parameter. The actual parameter C cannot be just anything; it must implement some specification S. The resulting component C then implements some (usually different) specification S'. This is what we might call the "macro expansion" view of a template: C can be obtained by substituting C for the template formal parameter wherever the latter occurs in the string T. In fact, this is how most compilers deal with templates.[2]

The view suggested by Figure 4 differs subtly in that a template transformation is one of behaviors (on the diagram's right side), not of strings (on the left side) [Edw95]. When a software engineer creates a template T, whose meaning is some partial function $\mathcal{M}_{\mathcal{T}}(T)$, part of the job is to state precisely the domain of this partial function. This is tantamount to characterizing which components T could accept as an actual parameter when instantiated.

[1]But there is some related work, discussed in Section 3, in which the need for such a framework is mentioned.

[2]A template, in general, can have several formal parameters. But we limit our model to the case of one parameter, without loss of generality, by postulating a simple packaging mechanism that combines several parameters into a unit [Edw95].

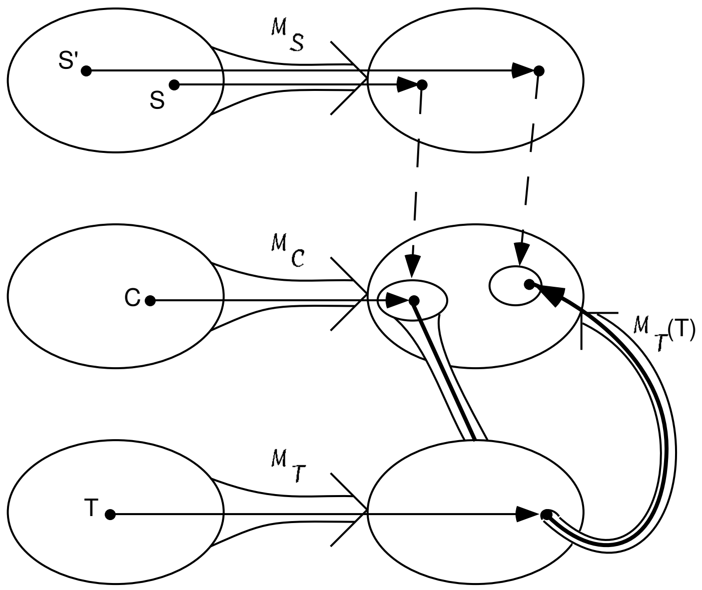

Figure 5: The meaning of template instantiation: applying $\mathcal{M}_{\mathcal{T}}(T)$

Defining the domain of $\mathcal{M}_{\mathcal{T}}(T)$ is now accomplished by choosing a specification S whose meaning $\mathcal{M}_{\mathcal{S}}(S)$ is exactly the subset of behaviors for which $\mathcal{M}_{\mathcal{T}}(T)$ is defined.

Figure 5 illustrates how template instantiation works in this framework. Determining the meaning of the instance of T with C as the actual parameter does not involve textual substitution of C for the formal parameter of T. The instantiation simply means $\mathcal{M}_{\mathcal{T}}(T)(\mathcal{M}_{\mathcal{C}}(C))$.

For example, one way of building a symbol table is to store entries in a binary search tree. We might write code that uses binary search tree functionality as a template T, for which each instance ***implements*** a symbol table specification S'. This defers, until system integration time, the choice of implementation C for binary tree specification S. T simply demands the availability of some — any — implementation of S. When a particular C that ***implements*** S has been selected, the system builder provides it as the actual parameter when instantiating T. The resulting instance has a meaning which lies in $\mathcal{M}_{\mathcal{S}}(S')$, i.e., the meaning of the symbol table specification S'.

Most software engineers and programming language designers seem to view templates from the macro expansion viewpoint. So generally it might be appropriate if all the semantic functions involved were consistent with the macro expansion view, to keep the latter from being misleading. In other words, if C' is the result of the string substitution of C for the formal parameter of T, then it should be the case that $\mathcal{M}_{\mathcal{T}}(T)(\mathcal{M}_{\mathcal{C}}(C)) = \mathcal{M}_{\mathcal{C}}(C')$. However, a string transformation resulting in C' technically is not required to define the meaning of T or any instance of T.

The reader should find it a good test of understanding of this framework to be able to identify not just symbolic but physical artifacts that arise in the engineering of physical systems, and which play the same role as templates play in the software world. This emphasizes why it is significant that instantiating a template should not, in general, be viewed in terms of string substitution.

What about specification templates? Although space limitations preclude discussion here, we see the appropriate semantic space for these units being $\mathcal{B} \to \mathcal{P}(\mathcal{B})$. See [Gib97] for details.

3 Comparison With Related Work

One of the most important objectives we have for attending the workshop is to find out more about related work. We have seen little other work addressing the general question of semantic spaces for general system artifacts, or seeking an integrated formal treatment of the mathematical modeling of physical systems and the semantics of symbolic representations for physical systems and software systems. The closest thing we know of is a body of work in artificial intelligence called *functional representation* [Cha94]. This work explores understanding of "causal processes" and "device behavior", where a "device" might involve physical hardware or software or both. It has influenced our thinking by emphasizing the significance of an integrated mathematical modeling framework that deals with not just software systems but the physical systems in which they are embedded. In fact, a recent paper describes a "smallest ontological framework" [CJ96] which (except for vocabulary) is consistent with our starting point in Figure 1. However, though specifications and templates (by other names, of course) are central issues there, too, the functional representation community has concentrated on the structure of $\mathcal{B}$ and languages for $\mathcal{C}$. So we view our present contribution as complementary and orthogonal.

Most work on integrating formal specifications into the software life-cycle relies on using one language for writing specifications (e.g., Larch or $\mathcal{Z}$) and another language for writing implementations (e.g., Ada or C++) [KH96]. We have concentrated on a single integrated language (RESOLVE) because it is difficult to pursue an agenda that includes formal verification unless the semantics of specifications and implementations are defined together. Perhaps our ideas about the semantics of specifications can suggest new ways for those who work with separate specification and implementation languages to integrate their semantic spaces. An integrated semantic treatment seems essential in order to avoid pitfalls in reasoning about software systems which several authors have exposed [Coo78, EHMO91, Lea91, EHO94, LW94, SWO97].

Our ideas about templates also might encourage the software engineering formal methods community to consider templates more seriously. While RSRG (like [Gog84]) has concluded that templates are inevitably at the very heart of component-based systems [Gib97] and templates have appeared in more and more modern programming languages, most of the formal methods community has hardly mentioned them. We should note that the centrality of templates also seems clear to some, but not all, of the functional representation community in AI.

RESOLVE is unusual in that it is a combination specification-and-implementation language with modern features such as templates, some forms of inheritance, etc. The semantic spaces developed to date for RESOLVE component varieties do not directly follow the approach suggested here. They are based on the ACTI formal model of software subsystems [Edw95], which has heavily influenced this paper. In particular the ACTI semantic space for templates ("concrete templates", tantamount to $\mathcal{T}$) suggests our proposed view of templates, and the ACTI semantic space for "abstract templates" suggests our proposed view of specification templates mentioned at the end of Section 2.4.

References

[Cha94] B. Chandrasekaran. Functional representation and causal processes. In M. C. Yovits, editor, *Advances in Computers*, volume 38, pages 73–143. Academic Press, 1994.

[Cha97] B. Chandrasekaran. Personal communication, August 1997.

[CJ96] B. Chandrasekaran and J. R. Josephson. Representing function as effect: Assigning functions to objects in context and out. In *Working Notes AAAI-96 Workshop on Modeling and Reasoning with Function*, Portland, OR, August 1996.

[Coo78] S. A. Cook. Soundness and completeness of an axiom system for program verification. *SIAM Journal on Computing*, 7(1):70–90, February 1978.

[Edw95] Stephen Hilary Edwards. *A Formal Model of Software Subsystems*. PhD thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1995. Also available as Ohio State University Technical Report OSU-CISRC-3/95-TR14.

[EGWZ97] S. H. Edwards, D. S. Gibson, B. W. Weide, and S. Zhupanov. Software component relationships. In *Proceedings 8th Annual Workshop on Software Reuse*, Columbus, Ohio, March 1997.

[EHMO91] George W. Ernst, Raymond J. Hookway, James A. Menegay, and William F. Ogden. Modular verification of Ada generics. *Computer Languages*, 16(3/4):259–280, 1991.

[EHO94] George W. Ernst, Raymond J. Hookway, and William F. Ogden. Modular verification of data abstractions with shared realizations. *IEEE Transactions on Software Engineering*, 20(4):288–307, 1994.

[GH93] John V. Guttag and James J. Horning. *Larch: Languages and Tools for Formal Specification*. Springer-Verlag, 1993.

[Gib97] David S. Gibson. *Behavioral Relationships Between Software Components*. PhD thesis, The Department of Computer and Information Science, The Ohio State University, Columbus, Ohio, 1997. To appear.

[Gog84] Joseph A. Goguen. Parameterized programming. *IEEE Transactions on Software Engineering*, 10(5):528–543, September 1984.

[Hey95] Wayne D. Heym. *Computer Program Verification: Improvements For Human Reasoning*. PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1995. Also available as Ohio State University Technical Report OSU-CISRC-12/95-TR55.

[KH96] H. Kilov and W. Harvey, editors. *Specification of Behavioral Semantics in Object-Oriented Information Modeling*. Kluwer Academic Publishers, 1996.

[Lea91] Gary T. Leavens. Modular specification and verification of object-oriented programs. *IEEE Software*, 8(4):72–80, July 1991.

[LW94] Barbara H. Liskov and Jeannette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16(6):1811–1841, November 1994.

[Spi88] J. M. Spivey. *Understanding $\mathcal{Z}$: A Specification Language and Its Formal Semantics*. Cambridge University Press, UK, 1988.

[SW94] Murali Sitaraman and Bruce W. Weide, editors. Special feature: Component-based software using RESOLVE. *ACM SIGSOFT Software Engineering Notes*, 19(4):21–67, 1994.

[SWO97] Murali Sitaraman, Bruce W. Weide, and William F. Ogden. On the practical need for abstraction relations to verify abstract data type representations. *IEEE Transactions on Software Engineering*, 23(3):157–170, March 1997.

Biography

David S. Gibson is a Major in the United States Air Force and a Ph.D. candidate in the Department of Computer and Information Science at The Ohio State University. He received a B.S. in Physics and Computer Science from Duke University in 1983 and his M.S. in Computer and Information Sciences from Trinity University in San Antonio in 1986.

Bruce W. Weide is Professor of Computer and Information Science at The Ohio State University. He received a B.S. in Electrical Engineering from the University of Toledo in 1974 and a Ph.D. in Computer Science from Carnegie-Mellon University in 1978. He is a co-founder (with Bill Ogden and Stu Zweben) of the Reusable Software Research Group at The Ohio State University.

We gratefully acknowledge financial support from the National Science Foundation under grants CCR-9311702, DUE-9555062, and CDA-9634425, and the Fund for the Improvement of Post-Secondary Education under project number P116B60717. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation or the U.S. Department of Education.