SOFTWARE ENGINEERING NOTES 19, 4 (OCT. 1994), 23-28.

Part I: The RESOLVE Framework and Discipline — A Research Synopsis

William F. Ogden
Murali Sitaraman
Bruce W. Weide
Stuart H. Zweben

RESOLVE is an overloaded name that refers to three related things:

- a conceptual *framework* to guide thinking about component-based software systems;
- a specific *language* to support easy description of components and systems within that framework; and
- a general *discipline* for using that language (or others with comparable features) to design high-quality software components and systems.

This paper provides a summary of only the RESOLVE framework and discipline, while supplying pointers to the literature for details and comparisons to related work. Details of the RESOLVE language are deferred to companion papers [Parts II and III].

In a nutshell, RESOLVE software components are just *parameterized modules*. A typical specification module defines formally the structural interface and functional behavior of an encapsulated abstract data type (ADT) and associated operations whose parameters are of that type and other types. A typical implementation module describes how such a type is represented as a composition of other ADTs and how the associated operations are effected by invoking sequences of operations associated with the types used in the representation.

So what's new here? At first glance, the basic RESOLVE framework and notation resemble those of modern formal specification languages and imperative sequential object-based (if not object-oriented) programming languages. If you are familiar with formal specification in Z, VDM, or Larch, and implementation in Ada or C++, then you should have no trouble understanding the RESOLVE work. What RESOLVE provides beyond its language is an integrated framework in which all the important, but sometimes conflicting, aspects of software design can be considered at once. In this framework, component engineering can exploit new perspectives that differ from conventional approaches in many important ways. The full compass of advantages can best be appreciated by examining specific reusable software components. So we present and explain a few examples in the companion papers [Parts II, III, and IV] and indicate where to find other and more complex examples [Part V].

1. Summary of the RESOLVE Framework

The basic RESOLVE framework steps out of the straight jacket of classical component design. It starts by distinguishing between *abstract components* (specifications) and *concrete components* (implementations). The former are represented graphically as circles and the latter as rectangles in Figure 1, which is adapted from an explanation of the "3C reference model" [Weide 91]. Distinguishing between these two kinds of components allows each abstract component to be realized using any of several concrete components that correctly achieve the intended functionality but differ in performance characteristics, as well as providing other important advantages over traditional frameworks for component-based software.[1] The thin lines in Figure 1 depict the "is implemented by" relation that connects an abstract component and the (possibly numerous) concrete components that realize its behavior. (The heavy lines depict a "uses" relation discussed below.)

An abstract component is described using an implementation-neutral behavioral specification that captures the *concepts* involved: the structural and functional properties of an abstract component interface. Simple concepts include typical abstract data types, e.g., queues, lists, maps, graphs, etc., but of course there are also more complex concepts. For variety, we use the terms "abstract component", "specification", and "concept" interchangeably.

A concrete component has two parts. The first is a description of implementation-related information that tells a client programmer what he or she needs to know about the component in order to use it in executable code, and which is called a *realization header*. The second is the implementation code itself, or *realization body.* For variety, we use the terms "concrete component", "implementation", and "realization" interchangeably.

Each component of either kind has a *context*, which consists of a set of identifiers and their meanings that are either declared locally or obtained from other components. Some of the context may be *fixed*, i.e., invariant over all uses of the component, and some may be *parametric*, i.e., supplied by the client only when an *instance* of the component is created. Figure 1 shows (a portion of) the context of each concrete component. Each heavy line depicts a "uses" relation between a concrete component and a lower-layer abstraction it is built upon and which therefore is part of its context. The diagram, however, does not show any context for abstract components.

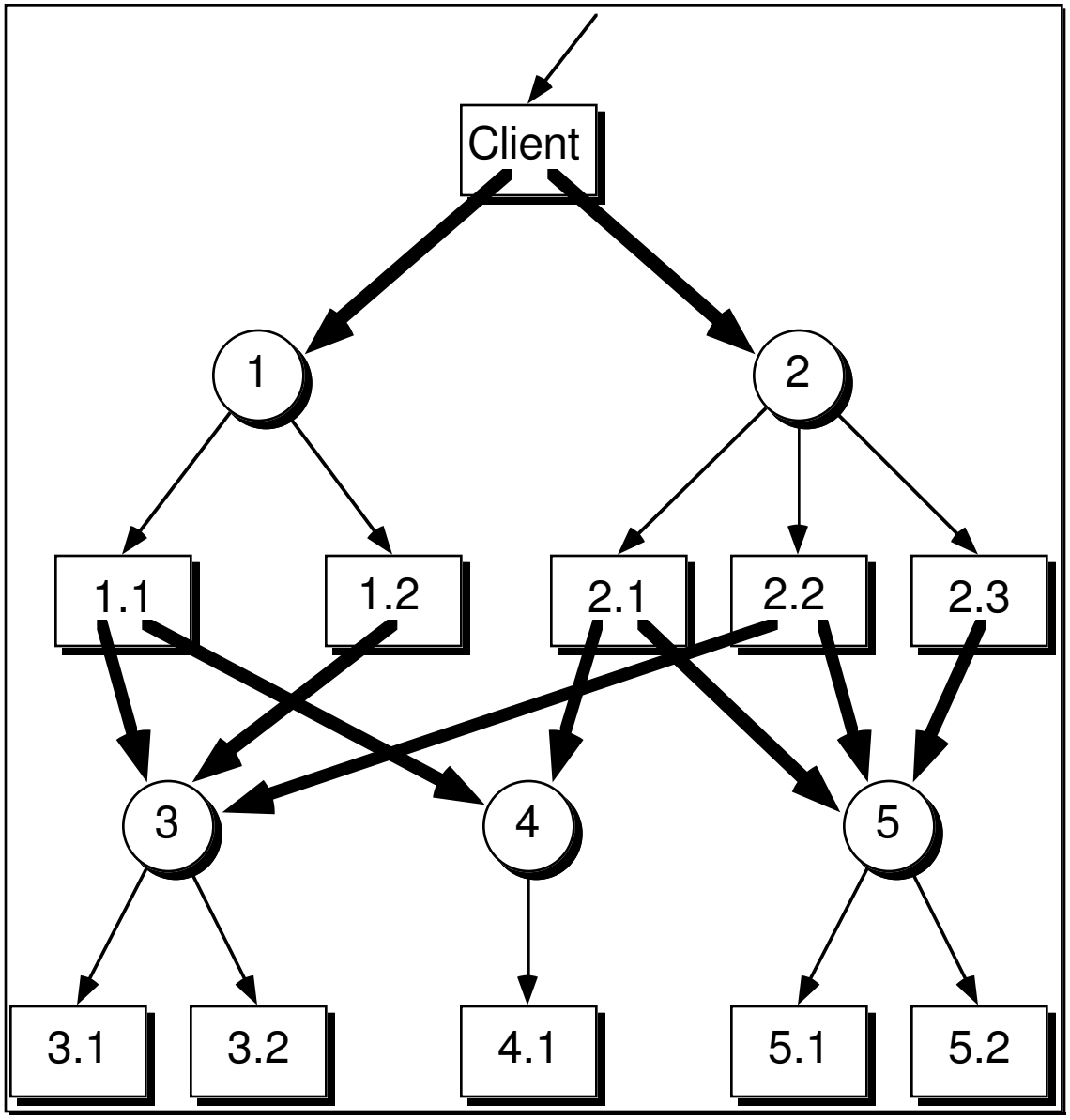
---

[1] Most software engineering texts and papers define a "component" as a specification plus *an* implementation. This restricted view — which has been codified in nearly every programming language that has a "module" construct — is remarkably deceptive. It masks out many of the most important and interesting problems (and solutions) connected with component-based software. For the origins of the 3C model, which closely resembles the RESOLVE framework, see Tracz, W., "Where Does Reuse Start", *Softw. Eng. Notes 15*, 2 (1990), 42-46.

To reuse an abstract component, a *client* component (which itself can be either an abstract or concrete component and in Figure 1 is concrete) declares an *instance* of the component by choosing a particular concept C and by fixing C's parameters. If the client is a concrete component, it completes the instance by selecting some realization R which has been certified to implement the specified behavior of C, and by fixing any additional parameters in R's header.

Thus, as a client you adapt a selected component to suit your needs by choice of appropriate generic parameters and by choice of a realization. These mechanisms, as manifested in the RESOLVE language, are very powerful [Hollingsworth 92a, Hollingsworth 92b, Sitaraman 92a], and parallel the tried-and-true methods underlying component design and composition in more mature engineering fields.

Figure 1: A Window on the Framework for Component-Based Software



2. Summary of the RESOLVE Discipline

Certain key software engineering goals such as information hiding can be fostered and even enforced within the above framework, given a suitable companion programming language. But the central intellectual problem of choosing a suitable design from a vast design space cannot be solved merely by adopting clever notation. Moreover, with any notation, the specification of a component must still be carefully crafted. In other words, how a notation is used is at least as important as the notation itself.

RESOLVE includes formal notations [Parts II, III, and IV] for supporting each element of the above framework. These language features differ in superficially subtle but fundamentally important ways from those used in other work on component-based software [Sitaraman 93]. Notational differences aside, though, one significant characteristic that sets the RESOLVE effort apart is the development of a discipline that prescribes how the framework and language *should be used* to design and compose high-quality component-based software systems [Hollingsworth 92b]. That is, the RESOLVE discipline goes beyond the fashionable normative and descriptive models of software reuse. It tells — rather precisely — what to do and what not to do, and *why*, when designing, specifying, implementing, certifying, and composing software components.

Furthermore, the RESOLVE discipline has been applied to constructing components and applications not just in the RESOLVE language (which we view primarily as a research vehicle), but in popular practical languages such as Ada and C++ [Part V]. The requisite linguistic mechanisms for supporting the RESOLVE discipline are one for separating specifications from implementations, and one for providing flexible generic parameterization. Other languages (e.g., ML and Eiffel) have similar basic features, but we are not working on adapting the RESOLVE discipline to them.

In the following sections we highlight some of the most important features and implications of the RESOLVE discipline. If you are looking for a single source that discusses these and many other principles for component-based software in more detail, the description of the RESOLVE/Ada discipline [Hollingsworth 92b] is recommended.

2.1. Concepts

A fundamental part of the RESOLVE discipline deals with design of abstract components. Among the issues it addresses in this regard are: how to design and reuse heavily parameterized concepts; how to create collections of abstract components that are highly composable with each other; how to make formal specifications understandable to potential clients; and how to design concepts that admit highly efficient and performance-flexible implementations. One characteristic all these considerations have in common is that if they succeed in promoting reuse, then the amortized cost of good design becomes negligible. With a discipline that constructively narrows the design space, and with the potential to explore much more of that space than would ever be thinkable for a single-

use design, you have a decent chance of discovering — sooner rather than later — the sort of clean and elegant abstract components that, in retrospect, seem so inevitable.

2.1.1. Parameterization

Every RESOLVE concept has a context section and an interface section [Part II]. The former is effectively the import list and the latter the export list, in traditional terms. The RESOLVE discipline suggests that, for maximum reusability, a component's context should be *fully parameterized*. That is, if context can be made parametric (as opposed to fixed), then it should be.

Because adaptation of concepts is accomplished using these parameters, rather than by the quite seductive but ultimately costly method of source-code modification, RESOLVE components tend to push the frontiers of parameterization well beyond current practice.[2] Specifically, careful parameterization makes it possible to extend the functionality of a concept C in numerous directions and have these extensions automatically work for all alternative implementations of C [Part II]. This seems to be a kind of inheritance, but there is no need for a typical inheritance mechanism in order to achieve it. In fact, the RESOLVE discipline adapted to C++ uses templates (C++'s generic parameterization mechanism), not inheritance, as the basis for such extensions.

Another role for parameterization is in the *specialization* of a concept [Part II]. The RESOLVE discipline recommends that you achieve specialization by *partial instantiation* of a fully parameterized concept. This creates a less adaptable but less complex concept and one that, by taking advantage of the assumptions that restrict its generality, sometimes admits a more efficient implementation than the fully general version.

The overall effect of carefully observing these principles is the creation of highly cohesive groups of concepts. Typically, there is a fully parameterized *kernel concept* with basic reusable functionality at the center of each group, surrounded by numerous extensions and variants [Hollingsworth 92b].

---

[2] Some mechanisms and uses of parameterization in RESOLVE are similar to those pioneered by Goguen with OBJ and LIL; see Goguen, J.A., "Parameterized Programming", *IEEE Trans. on Software Eng. SE10*, 9 (Sept. 1984), 528-543. But we see no evidence that sophisticated parameterization techniques have made their way into common practice. This is one reason we have adapted RESOLVE-based techniques to Ada and C++, where the required linguistic features are not quite available but the user community is large and the potential impact on software engineering practice much greater. Time will tell whether this approach has any more hope of success than sticking with a research language.

2.1.2. Composition

Following the RESOLVE discipline results in abstract components with quite uniform interfaces [Edwards 93b, Edwards 94]. This permits them to be readily composed with each other using little or none of the "glue" that is required when components are designed in an *ad hoc* fashion. The RESOLVE discipline is so exacting about many details of abstract component design [Hollingsworth 92b] that, for certain kinds of functionality, it sometimes seems hard to design concepts that are not automatically composable with others in the library. When the RESOLVE discipline was applied to the long-standing problem of designing a general iterator concept, for example, the design space it delineated was surprisingly small and a path through it to a clean abstract interface was fairly easy to find [Weide 94b]. We note that this space did not include any previously published iterator designs.

Not all components are composable with all others, of course. If concept C restricts not just the form but the properties of other concepts it might be composed with, then these (semantic) restrictions must be explicitly recorded in C's context. That way, when C is instantiated, the client's environment can be carefully checked to see whether it satisfies C's composition restrictions. If these restrictions are not satisfied, then the client might not work because C's behavior guarantee becomes void.

2.1.3. Comprehensibility

A component must be understandable to a potential client if it is to be reusable. There is an apparent paradox here: The only way to specify a concept unambiguously is to do it formally; however, formal specifications are by reputation generally obscure. Our approach is to try to make formal specifications straightforward and readable for an ordinary programmer who has the modest mathematical background typical of a computer science graduate. An important rationale for this approach is that specifications are read far more often than they are written (assuming component reuse is achieved). This means that a concept designer can afford to be quite deliberate when specifying a concept, in an attempt to simplify as much as possible both the behavioral model and its formal explanation. In fact, the objective of producing formal specifications that are concise and easy to comprehend turns out to help produce components that are easy to work with on an informal level, too. The issues here include how to select a mathematical model for a type, how to choose operations to package with a type, and how to build on a reader's knowledge of related concepts in explaining new ones.

One important principle of the RESOLVE discipline involves the choice of a mathematical model for a programming type (ADT). Even when alternative formulations are sufficiently powerful to specify the desired behavior, some result in more understandable explanations than others [Sitaraman 93]. So we provide a specific recommendation, the principle of *full abstraction*, as a criterion for concept specification. The basic idea is simple: There should be a one-to-one mapping between the intuitive

states of an ADT and the possible mathematical model states. Examples of this principle in action illustrate why it is important, and give evidence that it is not commonly followed [Weide 94b, Weide 94c].

Another principle of design-for-comprehensibility is that the primary operations exported by a kernel concept should be minimally sufficient and potentially complete [Weide 91]. Continued extension is inevitable for software components. It is simply an unattainable goal to try to design a reusable component once-and-for-all to provide every conceivable variant of the basic functionality. The RESOLVE discipline suggests a way to keep kernel component designs simple and understandable, yet capable of supporting arbitrary extensions of functionality in a uniform way. Although the catalog of possible additions and enhancements is endless, there are some important kinds that arise frequently in practice. For example, we have described how to enhance (systematically) any concept defining a collection type by adding provably correct iterator capabilities [Weide 94b].

Specialization, combined with a method for *re-exporting* a concept's interface [Part II], makes it possible to specify and construct groups of closely related concepts quickly and easily. Features common to group members are explicit and well-defined (not simply relying on naming conventions, for example). This improves a reader's ability to understand new components by leveraging understanding of related ones [Edwards 93].

2.1.4. Performance

The design of a concept's intended functional behavior should not rule out efficient implementations. Surprisingly (to us), this statement strikes some people as counter to the party line about worrying about functionality before worrying about performance; it strikes others as going without saying. We reconcile the two views by noting that we do not claim that a concept should *reveal* anything about its implementations. In fact, a good mathematical model is implementation-neutral. Rather, we argue that there are easy-to-apply design principles which can help you avoid designing concepts that preclude certain efficient implementations and that you ought to observe these principles especially carefully when designing reusable components.

One such principle concerns data movement. Consider an abstract component that defines some kind of collection type. In traditional designs, an operation that inserts an item into a collection is specified so that a *copy* of the item has to be constructed by the operation's implementation. Similarly, there is normally an operation to get an item out of a collection, and this supplies a copy of the item to the caller. While such designs might be appropriate when the participating items are scalars (e.g., 32-bit integers), they are not suitable when the items may themselves be (large) objects defined by other sophisticated components. Copying pointers to such objects is an unacceptable solution because it impedes modular reasoning about program behavior [Harms 91]. So, in the RESOLVE discipline, an insertion operation "consumes" the caller-supplied item and *moves* it into a collection; the caller-supplied item is replaced by an easily-obtained value of its type. A removal operation moves the item in the other direction, from collection to

caller. This kind of design frees implementations from the cost of copying when it is not needed, but still supports copy-based behavior, when appropriate, via simple extensions to the kernel concept.

For this and several related reasons, *swapping* — not copying/assignment — is the basic data movement mechanism in RESOLVE. Every component that exports a type also provides a swap operation for that type. The default use of swapping for data movement results in abstract component designs which are subtly different from traditional ones, and which admit more efficient implementations [Harms 91, Weide 91].

The explicit distinction between abstract and concrete components allows flexibility in one major aspect of performance, by hiding how computations take place in the various representations and implementations available for a single abstraction [Sitaraman 92a]; see Section 2.2 for related uses of parameterization. But additional performance flexibility at the concept design level is attainable by extending the object-oriented design paradigm to include *recasting* of single large-effect operations into objects [Weide 94c]. When an algorithm is recast as an object, its specification hides not only how but *when* computations actually take place. For example, you can recast the operation of sorting — usually a single procedure — into a "sorting machine" concept. This abstract component exports a type (a sorting machine state) plus a few operations to manipulate that type. It hides from the client knowledge about if and when sorting actually takes place. The usual sorting algorithms provide the basis for many implementations, but amortized-cost realizations generally dominate them in flexibility and overall performance [Weide 94c].

2.2. Realizations

RESOLVE concept specifications are truly devoid of implementation details. Among the consequences of this is the appearance of a small set of highly reusable kernel concepts, each of which permits a variety of implementations providing different performance profiles. Language support for multiple implementations of a concept is one of the distinguishing aspects of the RESOLVE work [Sitaraman 92a]. Two other important aspects of concrete components are discussed below: the need for separate realization headers and bodies, and the use of realization-specific parameters.

2.2.1. Realization Headers

Admitting that there might be multiple implementations for a single concept C raises many interesting questions [Sitaraman 92a]. How and where should we record non-functional properties and indicate how they differ among implementations of C? How and where should parameters to realizations appear, and when and where should they be bound? A concrete component might make additional non-functional guarantees (e.g., on performance) and place additional requirements on the context of the client program (e.g., by using implementation-specific parameters). Clearly these should not be associated with C, which captures the *common* features of all its implementations. On the other

hand, they should not be associated directly with the code that implements C, because clients need to see the former but not the latter. So it seems clear that a concrete component must be split into two parts: a client-visible portion (the "realization header") that lists requirements and guarantees that are specific to an implementation; and the actual code in the body of that implementation (the "realization body").

The additional requirements and guarantees seem to belong in the context and interface sections, respectively, of a realization header. In fact, it seems it should be possible to have a realization header that serves more than one realization. As described in the current RESOLVE language [Part III], however, a realization header has only a context section, and there is one realization body per header. The reason is that we have yet to work out the precise form and meaning of a realization header's interface section — or that of another module to augment it. These details depend on solutions to some important problems in expressing and verifying performance and other non-functional properties that might require different "intermediate" mathematical models for different classes of realizations [Sitaraman 94].

2.2.2. Parameterization

Realization-specific context may be fixed or parametric [Part III]. The RESOLVE discipline encourages full parameterization of most implementation aspects. This makes implementations more portable and more flexible (in terms of performance) than they can be with traditional software designs. Parameterization of a realization is powerful because RESOLVE realizations usually are implemented by *layering* on other abstract components. In Figure 1, realization $R_{1.1}$ is layered on concepts C_3 and C_4. This means that the performance of $R_{1.1}$ depends partly on how instances of C_3 and C_4 are used in the data representations and algorithms of the $R_{1.1}$, and partly on the specific realizations chosen for the instances of C_3 and C_4. In particular, C_3 has two implementations, $R_{3.1}$ and $R_{3.2}$, which presumably differ in performance. Full parameterization of $R_{1.1}$ can defer the choice between these to a client, offering "tunable" performance without the need for the client to touch so much as one line of code in $R_{1.1}$ [Sitaraman 92a, Hollingsworth 92b].

2.3. Modular Certification

The need for *modular reasoning* about the functional and performance correctness of realizations with respect to their concepts and realization headers is the single most important driving force behind the RESOLVE discipline. Components designed or implemented without following a discipline such as ours generally cannot support modular reasoning, and this makes sound reasoning about the behavior of large systems completely intractable [Hollingsworth 92b, Weide 92, Weide 93, Weide 94a]. Here we outline some issues related to modular reasoning about correctness, including the nature of a proof system and logical questions about it.

2.3.1. Certification of Correctness

One objective of the RESOLVE research effort is to show how to achieve modular verification of correctness of functionality, i.e., *local* or *component-wise* certification that a realization achieves the specified functional behavior of its concept [Weide 93, Weide 94a]. Each realization's correctness argument is modular and local, since it makes no reference to any particular client of the component and is "relative" to the correctness of all lower-layer constituent component realizations. Individual component verification also assumes that the client environment satisfies the restrictions described in its concept and realization header contexts.

Consider Figure 1 again. Once a component implementation, say $R_{1.1}$, is verified to be correct, verification of a client of C_1 involves first checking that the client environment satisfies C_1's context requirements. If these are satisfied, then the proof of the client relies only on abstract functionality information about C_1. Crucially, it does *not* call for re-verification of $R_{1.1}$. The need for this property has long been understood in some circles, but we know of no other practical component-based discipline that actually achieves it. The reason is that *strict* personal discipline is required beyond the mere use of the encapsulation mechanisms of modern programming languages [Weide 93, Weide 94a].

Authors of RESOLVE realizations are required to include module-level and statement-level formal assertions to be used in correctness arguments [Part III, Krone 88, Ernst 94]. Disciplined designs of realizations in languages such as Ada and C++ also should include these assertions (formal if possible, but rigorous in any case) in stylized comments. To assist in this, the RESOLVE discipline includes guidelines to help you write the three main kinds of assertions required: loop invariants, representation conventions, and abstract-concrete type correspondences [Krone 88, Hollingsworth 92b, Heym 94b].

We have developed a formal proof system for modular verification of RESOLVE components [Krone 88, Ernst 94, Heym 94b]. While Hoare-style systems concentrate on statement-level verification, our efforts have focused on modular verification of both functional and non-functional behaviors of parameterized components and compositions. For example, we have examined the role of verification context in modular verification of components with generic parameters [Ernst 91] and the complex issues in verification of realizations where object representations share common storage [Ernst 94].

Though the foundational basis for the RESOLVE efforts is formal verification, testing techniques for specification-based certification also have been explored [Parrish 91, Zweben 92]. The RESOLVE discipline includes several specific principles for organizing a group of concepts for testability. It results in a uniform and convenient way of dealing with testing and debugging of realizations of any abstraction [Hollingsworth 92b] without violating modularity.

2.3.2. Soundness, Relative Completeness, and Expressiveness

We have shown the logical soundness and relative completeness of the RESOLVE proof system [Krone 88, Heym 94b]. To accomplish this, we have developed a denotational semantics for most of RESOLVE, and have defined validity in these terms [Ernst 94, Heym 94b, Edwards 94]. Because the semantics and the proof system are based on the same foundations, the structure of soundness and relative completeness proofs is rather straightforward. Unfortunately, the details are still intricate.

There are a number of things that make our approach novel. For example, the definition of validity depends not merely on the pre- and postconditions of an operation's realization, but also on those of the operations invoked by it; a valid realization requires preconditions of called operations to be true at invocation. Another complication is introduced by operations that manipulate abstract objects. The semantics of these operations can, in general, be modeled only using relations even when every statement called by the operation is deterministic. Part of the reason is that data abstraction itself has the potential to introduce non-determinism [Ernst 94, Heym 94b].

Completeness questions with respect to tight specification and verification of non-functionality constraints, such as performance, also raise interesting issues. So techniques for improving expressiveness of specifications also have been addressed in the RESOLVE work [Krone 88, Ernst 91, Sitaraman 94].

3. Status

The RESOLVE effort to date includes work on a wide variety of fundamental issues of component-based software engineering, from formal methods for specification and verification to practical techniques for building scalable systems of Ada and C++ components.

There is statistically significant experimental evidence (from studies of student programmers) that adhering to certain major tenets of the RESOLVE discipline reduces software cost and improves its quality [Zweben 94]. But practical validation of the approach still needs to be obtained by careful empirical studies of additional and larger-scale programming efforts. These cannot be carried out in a purely academic setting, but will require close collaboration with industrial partners.

In addition to such empirical studies, we plan to continue to pursue foundational issues. For example, various improvements to the RESOLVE language dealing with non-functional behavior (especially execution-time performance) [Sitaraman 92, Sitaraman 94] and with common interface models [Edwards 93b, Edwards 94] need to be addressed. Before we do this, however, we need to be sure that the extensions fit as seamlessly as possible into the formal basis for semantics and modular verification.

Although much of the work on RESOLVE involves formal approaches, we have spent a lot of effort "translating" foundational results into terms suitable for students and practicing programmers [Hollingsworth 92b, Heym 94b]. For several years RESOLVE principles for component-based software engineering have been taught in various undergraduate and graduate courses (e.g., at The Ohio State University, West Virginia University, Indiana University Southeast, Denison University, Muskingum College, and New Jersey Institute of Technology) and in courses for industrial practitioners (e.g., at AT&T Bell Laboratories, and at TRI-Ada tutorials in 1993 and 1994). It is now time for a major effort that takes an integrated component-based view of software engineering, as the focus for teaching important software engineering concepts starting from CS1 and continuing through several subsequent courses for CS majors where software design and development are emphasized.