

# CHAPTER V

## Conclusion

In this chapter, we summarize the research conducted for this dissertation and present conclusions drawn from the work. Next we present contributions to the field and conclude with a discussion of some open issues and possible future work.

### 5.1 Summary and Conclusions

The primary parts of this work were the definition of a software discipline, and the formulation of a specific discipline for constructing Ada components.

#### 5.1.1 *Definition of a Software Discipline*

Chapter 1 defined the notion of a *discipline* for constructing software components. Briefly, a discipline includes goals, properties, principles, and local certification techniques.

One important aspect of the definition is its requirement of principles for guiding the construction of the components. Without specific principles to guide the engineer, we tend to get collections of vague rules or guidelines such as, “Exploit the features of the Ada language to write general purpose, adaptable code that has the maximum potential for future reuse” [SPC 89, page 174]. Rules such as these are nice, but do not really aid the component designer, nor do they help potential clients evaluate or differentiate among available components.

Another important aspect of the definition is its requirement for feasible certification that a component possesses the desired properties. Even when an engineer applies well-defined principles, it is usually possible to design components that do not exhibit all these properties. Section 1.3.3 discussed the importance of requiring feasible certification of the properties by providing an example of a component that allowed implementation leaks across its abstraction barrier. Components such as these, if allowed to be part of a software system, can lead to situations where the entire system must be considered as a monolithic whole when certifying some important properties, such as correctness. This is not feasible for even modest-sized systems because of the high level of effort required to find and understand all related components. This established the need for feasible certification of the properties, and the implication that feasibility of certification demands *locality* of certification.

Additionally, Chapter 1 provided a survey of related work showing that other researchers and practitioners have, at best, only implicitly formulated a definition for a software discipline. Usually they include goals and principles, or goals and properties. All of these implicit formulations lacked a requirement for, or techniques to accomplish, any kind of certification, feasible or not. This is important because feasible certification is a crucial requirement for a successful attack on software productivity and quality concerns.

Finally, Chapter 1 formulated a language-independent, component-type-independent discipline (based on our definition of a software discipline) for constructing high-quality software components. This discipline introduced a goal (i.e., high quality), properties (composability, correctness, reusability, and understandability), and one principle. This supports the first point of the thesis, namely that there is a programming language-independent, component-type-independent engineering discipline for the construction of high-quality software components. However, the single principle is not of much help by itself; the discipline must be refined and specialized to a component type and language in order to be practically valuable.

### *5.1.2 Formulation of a Specific Software Discipline for Constructing Ada Components*

Chapter 1 introduced a language-independent, component-type-independent discipline for constructing high-quality software components. In Chapter 2, we refined this formulation by fixing the programming language to be Ada and the component type to be abstract data types, and by introducing 33 specific principles for constructing high-quality components in Ada.

Section 2.1's presentation began with an example of a component whose structure is similar to components found in the literature. This component was then refined by the application of the principles as each one was introduced. Throughout this refinement process, we repeatedly referenced components having similar structure from the literature, thus assuring the reader that we were not using contrived examples. Ultimately, after applying the first 18 interface principles of Section 2.1, the resulting component bore little resemblance to any components found in the literature (except components published by the author or others from the author's research group). We view this as proof that work previously done in this area of software engineering and component design has not produced similar results.

Specifically, we compared RESOLVE/Ada components with two of the most widely known and highly regarded published component libraries: the Booch Components [Booch 87], and components developed by Musser and Stepanov [Musser 89]. RESOLVE/Ada components share few characteristics with components from these libraries, and for good reason. It can be easily shown that their components are not locally certifiable for composability and correctness. For example, all Booch generic packages that are parameterized by a private type, not limited private. On the other hand, Booch's components almost always export their type as limited private. This alone precludes direct composition of these components. Furthermore, as the example in Section 2.1.6 showed, the use of private types can lead to implementation leaks across abstraction barriers, thus jeopardizing local certification of correctness. Similarly, Musser and Stepanov develop components having the same characteristics as the Booch Components. For example, on page 236 of [Musser 89], they provide a Stack generic package that imports a private type and exports a limited private type.

To illustrate further the difference between RESOLVE/Ada components and components developed by others, we present the following table. It shows which of the 18 component interface principles from Section 2.1 are followed by guidelines and/or components from other component libraries. An “x” in a particular column indicates that column’s principle is followed, at least most of the the time, by that row’s component library and/or guidelines. We do not provide a table for the rest of the principles (from the rest of Chapter 2 and 3) because that table would be even sparser than this one.

Table 1 — Component Interface Principles (Section 2.1) Followed by Published Component Libraries and/or Guidelines

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
A		x		x	x		x						x					x
B		x	x	x	x								x					
C		x		x									x		x		x	x
D													x					
E	x	x	x		x		x						x					x

A — [Booch 87]

B — [Hibbard 83]

C — [Musser 89]

D — [St. Dennis 86]

E — [Wallis 90]

The principles of Section 2.1 included the component interface principles, the principles of Section 2.2 the client implementation principles, and the principles of Section 2.3 the component implementation principles. All told, Chapter 2 introduced 33 principles governing the construction of RESOLVE/Ada components.

Section 2.4 demonstrated how the principles lead to components that can be locally certified for the high-quality properties (i.e., composability, correctness, reusability and understandability). By being locally certifiable, we mean there is always a fixed extent in the “local area” of the component within a larger system that must be examined in order to certify its possession of a particular property. Local certification of composability for a component C depends on C’s package specification and only one package specification of a component corresponding to each of C’s facility parameters. Local certification of correctness for C’s implementation depends on C’s package specification, its package body and the package specifications of the components used by C’s implementation. Local certification of reusability and understandability depend on C’s package specification and its package body. This demonstration of local certifiability of RESOLVE/Ada components supports the second point of the thesis, that the discipline can be refined to a specific programming language and component type (such as Ada and abstract data types) and can be observed to lead to the development of locally certifiable high-quality components.

Chapter 3 discussed several practical issues concerning RESOLVE/Ada components, with special emphasis on *layering* of components. For example, the mathematical model-based formal comments appearing with each RESOLVE/Ada component can be used to develop Display operations for the exported type (useful during component testing) and for the exported type’s representation (useful during component implementation and debugging). Furthermore, since preconditions are specified, components that check their preconditions can be developed (useful for client program testing). Finally, since RESOLVE/Ada components are fully parameterized, Chapter 3 discussed how to do partial instantiation,

which fixes some or all of a component's generic parameters (possibly useful for prototyping).

Chapter 4 discussed the problems associated with bootstrapping from “raw” Ada. The implementation principles of Sections 2.2 and 2.3 do not allow the programmer to use all of Ada's constructs, e.g., the use of access types is prohibited. However, in order to develop non-trivial programs, many of these prohibited constructs must somehow be available. Chapter 4 introduced encapsulations for Ada's record, array and access type constructors. Each encapsulation was provided in the form of a generic package whose package specification conforms to the component interface principles of Section 2.1 but whose package body is implemented in raw Ada. These encapsulations are built so that they can be used by and composed with RESOLVE/Ada components without jeopardizing local certification.

## 5.2 Contributions

The focus of this research has been to define a detailed engineering discipline for constructing software components and to investigate its utility by using it as a basis for formulating a specific discipline for Ada. The following are the main contributions to the field:

*Definition of “software discipline.”* A discipline for constructing software components includes a goal, properties that the components should possess, principles guiding the construction of the components, and local certification techniques for demonstrating component possession of the desired properties. Central to the definition is the requirement for *local* certification techniques and *specific* principles for guiding component construction.

*Formulation of a particular software discipline.* Our specific discipline (called the RESOLVE/Ada Discipline) has a goal (high-quality components), four properties (composability, correctness, reusability and understandability), 41 detailed principles, and has been demonstrated to lead to locally certifiable components.

Contributions within the RESOLVE/Ada Discipline itself include:

- A set of specific principles for Ada.
- Each of the following RESOLVE/Ada Principles: 9, 10, 11, 12, 18, 19, 20, 23, 25, 26, 27, 28, 29, 30, 31, 32, 37, 38, 39, 40, 41.
- Actual RESOLVE/Ada components (see Appendix A for a list of package specifications and Appendix B for a list of package bodies).

*New Abstractions.* During this research we have developed some new abstractions, including `Permutation_Template`, `N_Way_Nilpotent_Template`, `String_Model_Template`, `Tuple_Model_Template`, and `Set_Model_Template`.

## 5.3 Future Work

Several areas look promising for productive future work involving extensions to the RESOLVE/Ada Discipline and alternative “raw” Ada implementations for the encapsulated Ada type constructors. A few of the possibilities are discussed here.

*Storage management issues.* Close examination of the implementations of `One_Way_Nilpotent_Template`, `Permutation_Template`, `Static_Array_Template`, and `Record2_Template` (i.e., the bootstrapped components), shows that all of these components dynamically allocate storage from the system heap. There are many issues related to this which need to be investigated. For example, many engineers feel that dynamic storage allocation is not practical for real-time applications because they cannot depend on the performance the run-time system's storage management routines. One avenue of research might be to investigate alternative implementations for the bootstrapped components which minimize the use of the run-time system's storage management routines. This might be done by having each component pre-allocate a storage pool at package initialization, followed by maintaining a free list of returned storage throughout execution. Preliminary work suggests this approach will succeed

*Dealing with concurrency.* Ada has the tasking construct which permits multiple threads of control to be created. But Principle 19 prohibits the use of Ada's task construct. Is there a way to allow RESOLVE/Ada components to be implemented using Ada tasks, and still maintain local certifiability of the RESOLVE/Ada Discipline's properties? Could tasking be used at least in the implementations of the bootstrapped components in such a way as to improve storage management (e.g., reduce memory fragmentation) or to improve their performance (e.g., allow the client of `Permutation_Template` to continue on while `Permutation_Template` performs bookkeeping).

*How to display types modeled by functions.* In Chapter 3 we discussed how displaying mathematical functions presents a problem because in our specifications they are total; it might be difficult to display a function with an infinite domain. Currently, constructing the Display operation for this type of function has to be handled on a case-by-case basis, and is based on the particular function and what part of it needs to be seen by the user. Is there a general way to handle displaying mathematical functions? Or must it always be handled on a case-by-case basis?

*How to write locally certifiable performance specifications for a component's operations.* An extension to the RESOLVE/Ada Discipline might be a property stating that the performance of each operation is specified. When components are layered there seems to be a problem with writing an operation's performance specification without including specific information about the performance of lower level components. If there is no solution for this problem, then the property requiring performance specifications will not be locally certifiable. That is, writing the performance specification of an operation will not be possible until a system is created and furthermore may require examining the entire system.