CHAPTER VI

Conclusions

This chapter summarizes the research conducted for this dissertation and presents the conclusions drawn from it. It continues by presenting the contributions of this research to the field of computer science, and concludes with a discussion of future research directions.

6.1 Summary

This dissertation defends the thesis that conventional programming languages are inadequate for constructing large, complex software systems that are “understandable.” Chapter II explains the basis for this claim, namely that modern programming languages:

- Treat module-like structures as syntactic mechanisms whose primary purpose is grouping declarations and controlling visibility; and
- Give other program elements, such as procedures, meanings that are hierarchically constructed on the basis of their implementations.

While these are reasonable choices when designing a language to instruct computers, they do not address the cognitive limitations humans face when trying to understand complex artifacts. In order to address these concerns, it is necessary to assign independent meanings to individual software building-blocks, to separate the abstract description of a software part’s behavior from its implementation, and to provide a mechanism for explaining how the implementation of the part provides behavior consistent with that abstract description.

Chapter III introduced a new language-independent model of software called ACTI. ACTI stands for “Abstract and Concrete Templates and Instances,” the four principle classes of software subsystems it defines. ACTI addresses the limitations of conventional programming languages described above because:
• In ACTI, each software subsystem (building-block) is given an intrinsic meaning; it is not just a syntactic construct. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.

• The meaning of a software subsystem is not synthesized from the meanings of the lower-level subsystems it is built upon, and is completely independent of how the subsystem is implemented.

The introduction to the model presented in Chapter III was centered around a running code example, and was presented in informal, intuitive terms.

Chapter IV then provided a formal definition of ACTI. This model’s development followed the approach of denotational semantics, defining mathematical spaces of objects intended to represent the meanings of program structures such as specifications, implementations, and their relationships. ACTI was developed with the phase distinction between compile-time and run-time semantics in mind, and it only addresses the dynamic, execution-time aspects of module meanings.

Finally, Chapter V continued the running example that was begun in Chapter III to demonstrate some of the more advanced aspects of ACTI. It showed the implementation of a subsystem (a concrete instance), and the interpretation mapping explaining why that implementation was consistent with the abstract “cover story” described in the corresponding specification (an abstract instance).

The remainder of the research presented in this dissertation consists of three principle components: a discussion of five previous software models; a check list that represents an operational definition of the necessities for effectively structuring software; and an analysis of the ACTI model. Together, these parts form a defense of the claim that ACTI is a mathematically formal, language-independent model of software components that captures the underlying conceptual view of software architecture embedded in modern module-structured languages.

Appendix A gives overviews of the five prior models of software chosen as representative of previous work. The five are: the 3C model of reusable software, OBJ, RESOLVE, Eiffel, and Standard ML. Of these five “models,” four are actually particular programming languages, of which two are functional programming languages, one is object-oriented, one is object-based, two support formal specification, one more supports informal specification, one supports formal verification, and three support parameterized programming. These five models were chosen because they represent the breadth of what researchers in computer science have said about software construction, and how to best address the architectural problems of large-scale software systems.
6.2. CONCLUSIONS

Appendix B then takes these five models and uses them to devise an operational means of measuring the extent to which any model addresses the intuitive notions about software construction and reuse that have been voiced. These intuitive notions are captured qualitatively, but in the more rigorous form of a structured check list of 22 properties that any universally applicable model of software construction should have. Each property in the list is justified, and cross-checked against each of the previous models, and also against the viewpoints of various software engineering professionals who build, use, and manage software components. Finally, a summary of how each of the prior five models measures up against the check list is presented in Table 3.

Appendix C then puts ACTI to the same test, determining how well it measures up against the structured check list developed in Appendix B. Unlike prior models, ACTI possesses all of the 32 properties in the check list. It provides support for abstractions by modeling types, instance subsystems, and template subsystems. It faithfully supports separation of concerns by making all subsystems independent, requiring explicit context interfaces, and elevating the “is-implemented-by” relationship to a first-class object. Through interpretation mappings, it provides an intensional mechanism for binding subsystems together and defining behavioral subtyping relationships between types or subsystems. It also provides convenient operators for programming by difference, or deriving new software objects from existing ones, while completely separating this mechanism from interpretation mappings. Appendix C then continues, explaining how each of the previous models could be mapped into the framework of the ACTI model.

6.2 Conclusions

The focus of this dissertation has been the development of a programming language-independent, but formal, model of software construction that gives meaning to the notion of a software “building-block.” Throughout this research effort, it has become apparent how profoundly the notations programmers use can affect the way they view the world. ACTI was developed to encompass the best of previous attacks on the software structuring problem while adding the capabilities necessary to capture abstract models of software parts, and it strives to strictly separate the distinct concepts involved. It does this to better support language designers by giving a more flexible, capable shape to their world view, so they can in turn help programmers the same way.

ACTI is general enough to encompass a wide variety of program semantics, especially the meanings of architectural structuring features. This means that both good and bad strategies of software structuring can be modeled. It follows that ACTI
might provide a useful framework for talking about what strategies and mechanisms are most beneficial from the perspectives of software engineering in general and reuse in particular. Three of the most obvious architectural issues affecting reuse and software engineering are:

1. Appropriate separation of specification and implementation.
2. Support for parameterized programming.
3. Separating subtyping from programming by difference.

Each of these issues are discussed in the sections below.

6.2.1 Separation of Specification and Implementation

Perhaps the most critical insight about software structure embodied in ACTI is that a module is not simply a specification plus an implementation. At its most general, the specification-to-implementation relationship is many-to-many, and each specification and each implementation can have an independent meaning on its own.

The relationship between specifications and implementations is also a unique feature of ACTI. It encompasses both type-level and module-level representation invariants and abstraction functions (or, more properly, abstraction relations). Over and above this relationship, both specifications and implementations may contain their own constraints, at both the type and the module levels.

Finally, ACTI provides true abstraction for both types and modules, in the sense described by Harms [24]. In other words, ACTI provides more than the ability to hide representations and implementations; it allows a module to provide an abstract “cover story” describing how a type should be logically viewed, or how an operation behaves. The manner in which an implementation fulfills such a cover story is captured through interpretation mappings. All of these characteristics are in strong support of good software engineering.

6.2.2 Parameterized Programming

ACTI follows the work of Goguen [21] in providing complete support for parameterized programming. Abstract and concrete templates, or generic specifications and implementations, provide the cornerstone of this support. Both specifications and implementations can be parameterized independently, as in RESOLVE [51, pp. 23–24]. Further, ACTI’s capabilities for modeling nested modules and templates means that ACTI has true high-order generics. Finally, the way template parameter requirements
are represented in ACTI allows for generics with variable-length parameter lists and correspondingly variable output instances.

6.2.3 Subtyping and Derivations (Inheritance)

While ACTI was developed to incorporate all of the beneficial aspects of OOP strategies, it takes a distinctly nontraditional approach to inheritance. It clearly distinguishes between its specification derivation mechanisms, implementation derivation mechanisms, and is-a relationships, unlike most traditional OO languages which use a single inheritance mechanism for all three [8]. By also providing support for modeling polymorphic type representations and dynamic binding behavior, ACTI manages to provide all of the OO features while avoiding the problems than can arise if they are all co-mingled in a single mechanism.

6.3 Contributions

The primary contribution of this research to the field of computer science is ACTI itself. Secondarily, the structured check list developed and described in Appendix B is also a contribution, and can be extended on its own to form a more refined operational definition of the requirements for a general purpose model of software.

As a model of software construction, ACTI is unique because:

1. It promotes the creation of complex software systems that are understandable. This is achieved by both supporting the formation of mental models of software parts, and addressing the inherent human (cognitive) limitations in dealing with those mental models.

2. It gives a real semantic denotation to “modules,” which are the building-blocks from which software is composed. This denotation identifies just what “modules” should be so that they are meaningful, allow sophisticated composition, and present simple (not synthesized) conceptual models.

3. It identifies “interpretation mappings” as the mechanism that allows one to clearly represent why an abstraction (a specification with a simple model) correctly describes the composite behavior of a complex combination of lower-level software parts (an implementation).

4. It is language-independent, and unifies the concepts behind object-oriented programming and more traditional module-based programming. Thus, it solves the
problem of understandable software composition across module- and class-based languages.

5. It solves the problems normally associated with “inheritance” [8] by separating “programming by difference” from type-to-type and module-to-module subtyping relationships.

6. The research approach started with an explicit analysis of needs, and development of the model was then driven by these stated requirements, rather than by the features of a given language.

7. It strives for complete separation of concerns, which was not achieved by any of the prior models (see Table 3).

Further, ACTI builds on previous work:

1. It gives software modules a real semantic denotation.

2. It treats software module specifications as complete behavioral descriptions, not just “types for modules.”

3. It treats relationships between modules as more than simple extension relationships. Interpretation mappings effectively tell one how to “interpret” lower-level module structures at the more abstract specification level.

4. It provides complete support for parameterized programming.

5. It supports modularly verifiable software.

6. It focuses on modularity of software construction.

These contributions are important for three reasons. First, ACTI can form the basis of a denotational semantic model for any module-based language, including an OOPL. Second, if ACTI is used as the starting point for such a semantic model, it helps to ensure that the lessons from past ventures in this area will be carried into new development. Third, and most importantly, if ACTI is used as such a starting point, it will help to ensure the resulting language has a vision for the meaning of software parts and how they contribute to the understandability of a complex software system.

6.4 Future Research

There are many promising possibilities for future work based on ACTI. A few of the most interesting areas are discussed here.
6.4. FUTURE RESEARCH

RESOLVE’s Denotational Semantics. Since ACTI was designed using the denotational approach, the next logical step is to use it as (part of) the semantic definition for a language designed to support effective software construction. RESOLVE is just such a language, and a worthwhile extension of this research would use ACTI as the dynamic semantics for RESOLVE. All that is necessary is to define a corresponding static semantics to form a complete language definition. Some facets of ACTI might also instigate refinements of some of RESOLVE’s language features, or even spur the addition of new features.

Language Features. Some of the mathematical spaces in ACTI are more expressive than any of the corresponding features the author has observed in programming languages. For example, consider an abstract template, which takes an abstract instance as a parameter. The allowable abstract instances to which this template may be applied are described by a domain predicate, the AIDP component of the template. This is a boolean-valued function over the space of all abstract instances. Thus, in ACTI it is possible to model generics that are applied to a variety of parameters, not all of which have to have the same structure. This means that the same generic specification (or implementation) could operate on modules that contain different numbers of types, operations, etc., with different names, and so on.

The simplest ramification of this expressiveness is that ACTI templates can operate on “variable length” parameter lists—a single template could operate on a module with one type or with ten types and generate an appropriate output in each case, and similarly for variable numbers of operations, nested modules, or even nested templates. Further, in ACTI it is possible to define an abstract template that, given any abstract instance that exports at least one type, generates another abstract instance that is identical, but also exports Copy and IsEqual operations for that type (or even for every type in the parameter). No programming language at the time of this writing allows one to define generics that are parameterized this flexibly.

The generality of such mathematical spaces in ACTI may point at new language features, or variations on old ones, for exploration.

Software Architecture Issues. Over time, it seems that some software practices have turned out to be more beneficial than other at managing complexity and dealing with problems of scale. Since ACTI is merely a model of how software can be put together, it provides the necessary descriptive ability to discuss how software should be put together. A potentially fruitful avenue of research is to look at some of the issues discussed in Section 6.2, and determining which architectural practices are “good” and which are “bad,” from a software engineering perspective.
CHAPTER VI. CONCLUSIONS

Exceptions. For simplicity, ACTI does not attempt to represent exceptions or exception propagation behavior in any way. Many languages, including PL/I, Ada, CLU, Eiffel, and Standard ML, include some form of exception mechanism that programmers may use for handling erroneous conditions that arise at run-time. Exceptions are simply a control flow mechanism, however, not a software structuring mechanism. As a result, it seemed reasonable to leave them outside the scope of this research project. Incorporating exceptions into ACTI would be provide an interesting future research topic, however.

Concurrency. Again for simplicity, ACTI was developed assuming a single thread of control. Of course, if one cannot do the single-threaded case correctly, it is pointless to consider the more complex concurrent case, so this was a reasonable first step. Note, however, that since ACTI only maps out the mathematical spaces of meaning for program objects, it does not embody any explicit model of computation. Thus, the problem of how to effectively model the environment of a concurrent program within ACTI, or an extension of it, is a particularly exciting research direction. The question of whether this would entail any additional software modularization mechanisms is also open.

Polymorphism in OOPLs. Because ACTI judiciously separates its derivation mechanisms from its representation of type-to-type and module-to-module relationships, it has a noticeably different flavor than the programming model of a traditional OOPL. Looking at the effects of using mappings, rather than inheritance, to determine polymorphic behavior in OOPLs is another possible research topic.

Performance Specifications. While specifying the functional behavior of software operations is fairly well understood, specifying their time and space performance in a modular fashion is an area of open research. Promising work on incorporating support for modular verification of performance specifications within the RESOLVE framework has already begun to appear in the literature [34, 35, 58]. It would be interesting to examine whether ACTI can effectively capture the specification structures needed for this task, or how it would need to be extended to do so.

Conjoining Specifications. The need for performance specifications opens up a more general problem in software specification. That is, it is very desirable to permit semi-independent “facets” of a specification to be described independently. The theoretical foundations necessary to do this are uncertain, however. For example, usually one considers specification of the time performance of operations to be “independent”
of specification of the functional behavior of those same operations. Unfortunately, both specifications are dependent on the abstract model of the data being manipulated by the operations. Typically, performance specifications require that more detail about the way data is actually represented should be visible, so that details of performance variations can be adequately described. It is possible that through the use of interpretation mappings, distinct “facets” of a specification could be written using differing abstract models of data, and later conjoined to form a composite specification for a subsystem. Exploration in this direction could make the job of specification much easier.