Modeling Modular Software Structure for Human Understanding

Stephen H. Edwards
Dept. of Computer and Information Science
The Ohio State University
2015 Neil Avenue
Columbus, Ohio 43210–1277
E-Mail: edwards@cis.ohio-state.edu
URL: http://www.cis.ohio-state.edu/~edwards

Abstract

People form internal mental models of the things they interact with in order to understand those interactions. This psychological insight has been used by the human-computer interaction (HCI) community to build software systems that are more intuitive for end users, but it has only been informally applied to the problems of software designers, programmers, and maintainers. Conventional programming languages still do little to help client programmers develop good mental models of software subsystems.

To address this problem, we have developed the Abstract and Concrete Templates and Instances (ACTI) model of modular, parameterized software subsystems. This model of software structure addresses the needs of human software engineers who must reason about collections of interacting software parts during design, maintenance, and evolution.

ACTI is different from other module systems and models of software in several ways. In ACTI, a subsystem never *has any implicit dependencies, and never depends directly on any external definitions—all external dependencies are described through an explicit interface. In addition, a subsystem specification is meaningful by itself, even without respect to any implementation. Finally, a subsystem is more than just a collection of types and operations; it also includes: an explicit model of behavior, an explicit model of all external dependencies, a collection of definitions used to construct and describe these models, and (potentially complex) substructure. There are strong parallels between ACTI and other research on the understanding of modularly structured physical devices, particularly Functional Representation.*

Keywords: *Mental model, model-based specification, interfaces, bindings, generics*

1 Introduction

Modern programming languages have evolved from their predecessors with the primary purpose of describing instructions to computers. Generally, these languages were not designed to help explain to people the meaning of the software that they can describe. This has led to two significant problems with programming languages today: modules are considered to be purely syntactic constructs with no independent meaning, and those parts of programs that are deemed meaningful (usually procedures, in imperative languages) have "hierarchically constructed" meanings.

To address these deficiencies, here we outline a new model of component-based software that provides concrete support for recording critical information about each software structure, information that can form the basis for a programmer's own mental model of that structure. This new model, termed the ACTI model (for "Abstract and Concrete Templates and Instances") is both mathematically formal and programming language-independent. It captures and formalizes the underlying conceptual view of software architecture embedded in modern module-structured languages while simultaneously providing support for forming mental models. As a result, it can serve as a general-purpose theory of the nature of software building-blocks and their compositions.

1.1 Why Conventional Languages Fail

Most modern programming languages have some construct that is intended to be the primary "building-block" of complex programs. This building-block may be called a "module," a "package," a "structure," or a "class." Unfortunately, these constructs are rarely given meaningful semantic denotations. Conventional wisdom in the computer science field is that these constructs are primarily for grouping related defini-

tions, controlling visibility, and enforcing information hiding. For example, when considering module-structured languages like Ada or Modula-2, Bertrand Meyer writes:

> In such languages, the module is purely a syntactic construct, used to group logically related program elements; but it is not itself a meaningful program element, such as a type, a variable or a procedure, with its own semantic denotation. [1, p. 61]

In this view, there is no way for one to make such building-blocks contribute directly to the understandability of the software comprising them. While object-oriented languages usually give a stronger intuitive meaning to the notion of a "class," they also fail to provide any vision of how the meaning of individual classes can contribute to a broader understanding of the software systems in which they are embedded.

In addition, those program elements that *are* given a semantic denotation are often given a meaning that is "hierarchically constructed," or synthesized. In other words, the meaning of a particular program construct, say a procedure, is defined directly in terms of its implementation—a procedure "means" what the sequence of statements implementing it "means." The meaning of its implementation is defined in terms of the meanings of the lower-level procedures that it calls. Thus, a procedure's meaning is constructed from the meanings of the lower-level program units it depends on, and the meanings of those lower-level units in turn depend on how they are implemented, and so on.

This simple synthesis notion of how meaning is defined bottom-up is adequate from a purely technical perspective. It is also very effective when it comes to describing the semantics of layered programming constructs. Unfortunately, it is at odds with the way human beings form mental representations of the meanings of software parts [2].

The result of these two features of existing programming languages is that they are inadequate for effectively communicating the meaning of a software building-block to people (programmers, in particular). The semantic denotations of programming constructs in current languages only relate to *how* a program operates. They fail to capture *what* a program is intended to do at an abstract level, or *why* the given implementation exhibits that particular abstract behavior. In order to address these concerns, it is necessary to assign meaning to software building-blocks, to separate the abstract description of a software part's intended behavior from its implementation, and to provide a mechanism for explaining why the implementation of the part achieves behavior consistent with that abstract description.

1.2 Toward Understandable Software

The ACTI model directly addresses these deficiencies of current programming languages by giving a software subsystem a well-defined meaning of its own, independent of how it may be implemented. This meaning includes an explicit model of behavior, which can serve as a reference to help a client programmer understand the subsystem—form an effective mental model of it. Further, the constant presence of such a behavioral description acts as a continual cue to aid the programmer in maintaining the consistency and correctness of her own understanding.

Because a person's internal mental representations are so critical for comprehension, supporting the formation and maintenance of effective mental models is important if one wishes to support complex software structures that are understandable by humans. Understandable software is vital for software designers, who must design in the context of reusable software parts; for testing and maintenance personnel, who often must understand software written by others; and for reverse engineers or re-engineers, who want to gain as much value from previous work as possible.

Although a full treatment of ACTI's formal definition is beyond the scope of this article because of space considerations, the following sections provide a general overview of the model at an intuitive level. Section 2 introduces the main entities in ACTI. Next, Section 3 highlights the unique and novel features of the model. Section 4 then outlines how ACTI provides support for software understanding. Relationships to previous work, particularly AI-based work on the understanding of physical devices, is discussed in Section 5.

2 An Overview of ACTI

The ACTI model [2] is centered around the notion of a "software subsystem," a generalization of the idea of a module or a class that serves as the building-block from which software is constructed. A subsystem can vary in grain size from a single module up to a large scale generic architecture. ACTI is designed specifically to capture the larger meaning of a software subsystem in a way that contributes to human understanding, not just the information necessary to create a computer-based implementation of its behavior.

The ACTI model is based on four different kinds of subsystems:

Abstract Instance—A disembodied subsystem specification or interface description. There is *no*

Table 1: The Four Kinds of Subsystems

Subsystem Varieties		
	Abstract (Specification)	Concrete (Implementation)
Template	RESOLVE concept SML functor signature	RESOLVE realization SML functor
Instance	Ada package spec. SML signature Eiffel class interface	Ada package body SML structure Eiffel class impl.

implementation associated with anything defined in the specification.

Concrete Instance—A subsystem that provides implementations for its types and operations. All of the defined types and operations in the subsystem are represented and/or implemented.

Abstract Template—A subsystem-to-subsystem function that, when applied to its argument, which is some abstract instance, will generate another abstract instance. Effectively, an abstract template is a form of generic subsystem specification.

Concrete Template—A subsystem-to-subsystem function that, when applied to its argument, which is some concrete instance, will generate another concrete instance. Thus, a concrete template is a form of generic subsystem implementation.

The terms used for this classification are all based on the work of Weide *et al.* [3, p. 23], and the same ideas appear in the 3C model [4]. The name "ACTI" is an acronym derived from these four terms: "Abstract and Concrete Templates and Instances."

This view of the world allows software subsystems to be partitioned along two orthogonal dimensions, as shown in Table 1. The distinction between "abstract" and "concrete" embodies the separation between a specification or interface, and an implementation or representation. The distinction between "template" and "instance" allows one to talk about both generic subsystems, and the product of fixing (binding) the parameters of such a generic subsystem: an instance subsystem.

Formally, ACTI is a collection of mathematical spaces, together with relations and functions on those spaces, that can be used in explaining (or defining) the denotational semantics of program constructs. In spirit, the model was developed in accordance with the denotational philosophy, as described by E. Robinson:

> In the denotational philosophy inspired by Strachey the program, or program fragment, is first given a semantics as an element of some abstract mathematical object, generally a partially ordered set, the semantics of the program being a function of the semantics of its constituent parts; properties of the program are then deduced from a study of the mathematical object in which the semantics lives. [5, p. 238]

ACTI is not a programming language, however. Instead, it is a mathematical model that is useful for programming language designers, or researchers studying the semantics of programming languages. It is a formal, theoretical model of the structure and meaning of software subsystems. It is rich enough to be used as the denotational semantic modeling space when designing new languages, and has been shown to subsume the run-time semantic spaces of several existing languages chosen to be representative of the modern imperative, OO, and functional philosophies [2].

ACTI has two features that specifically address the inadequacies described in the introduction:

1. In ACTI, a software subsystem (building-block) has an intrinsic meaning; it is not just a syntactic construct used for grouping declarations and controlling visibility. This meaning encompasses an abstract behavioral description of all the visible entities within a subsystem.

2. The meaning of a software subsystem is *not*, in general, hierarchically constructed. In fact, it is completely independent of all the alternative implementations of the subsystem.

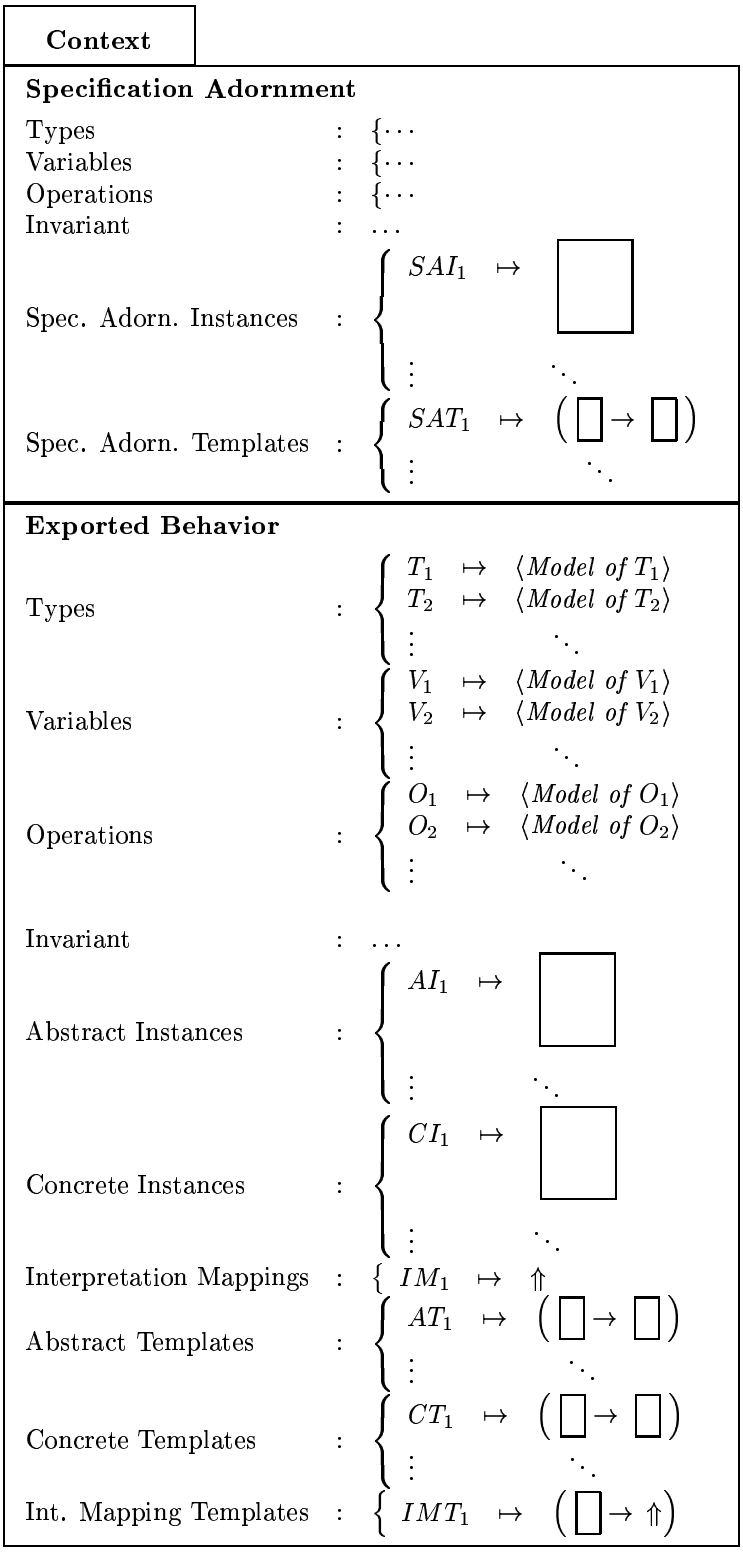

Figure 1: The Details of an Abstract Instance

Thus, ACTI provides a mechanism for describing *what* a subsystem does, not just how it is implemented. The meaning provided for a subsystem is a true abstraction—a "cover story" that describes behavior at a level appropriate for human understanding without explaining how the subsystem is implemented. Further, ACTI provides a formally defined mechanism, called an interpretation mapping, that captures the explanation of *why* an implementation of a subsystem will give rise to the more abstractly described behavior that comprises the meaning attributed to the subsystem—in short, an explanation for why the cover story works.

2.1 Abstract Instances

Table 1 gives examples of some programming language structures that might typify each of the four kinds of subsystems. We begin with abstract instances.

An abstract instance is a subsystem specification or interface description. There is *no* implementation associated with anything defined in the abstract instance. Further, like all other ACTI subsystems, an abstract instance cannot directly refer to any entities outside of itself—it is completely self-contained. If a given abstract instance relies on external definitions, they must be imported through an explicit interface that expresses exactly what expectations the instance places on its environment—an explicit "context" interface.

To briefly give the flavor of the mathematical spaces in ACTI, Figure 1 schematically depicts an abstract instance object. The abstract instance is divided into three parts, the most familiar of which is the **Exported Behavior**. The exported behavior portion of the abstract instance defines all of the services provided by the instance:

- All types that it provides, including a mathematical model space for each;
- All variables, including their types;
- All operations, including a pre- and postcondition-oriented model of their behaviors;
- An invariant over the entire instance;
- Nested abstract instances;
- (Specifications of) nested concrete instances;
- Nested interpretation mappings (interpretation mappings are described below); and
- Nested templates (templates are described below).

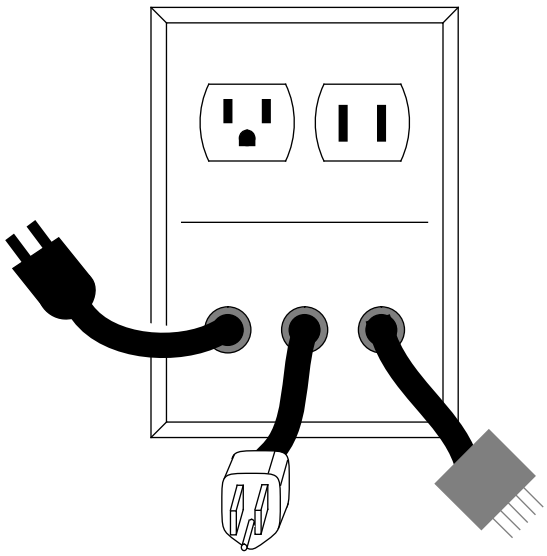

Figure 2: An Abstract Instance Is A "Face Plate"

All of these components have values taken from some complete partial order (CPO) space defined in the ACTI model.

In addition to providing a behavioral model of all exported features, ACTI includes complete behavioral descriptions of all imported features [2]. The **Context** section shown in Figure 1 is actually one (nested, possibly empty) abstract instance that is used to completely define all of the external dependencies of the main instance shown in the figure.

The remaining section of the abstract instance, the **Specification Adornment** section, is rarely manifested in programming languages. It is an area where purely mathematical definitions of types, operations, or other entities can be made, purely for use as tools in creating more understandable behavioral descriptions. While the **Exported Behavior** describes what we would normally consider to be programming-level properties of a subsystem, and while **Context** describes programming-level external dependencies, **Specification Adornment** describes mathematical specification tools.

Intuitively, we can think of an abstract instance as a "face plate" that describes an explicit interface at both the syntactic and behavioral levels, as shown in Figure 2. Here, the "sockets" on the upper half of the face plate symbolize the explicit interface to external dependencies, while the "plugs" on the lower half symbolize the features provided by this subsystem.

In ACTI, abstract instances can have

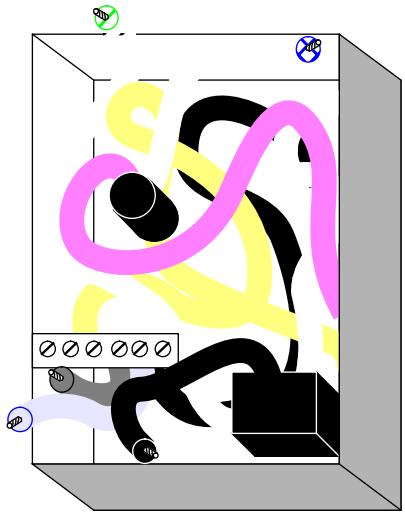

Figure 3: A Concrete Instance Is An "Open Box"

substructure—that is, abstract instances can be nested within other abstract instances. This provides a mechanism for describing cohesive groups of related features within one "face plate" as a unit, as well as for modularizing complex specifications.

2.2 Concrete Instances

A concrete instance is a subsystem that provides implementations for all of its exported features—types, operations, etc. This is much closer to the usual programming language notion of a "module." Just as with abstract instances, a concrete instance has a meaning in isolation, and cannot implicitly depend on any external entities—all external dependencies must be explicitly described in its context interface. The behavior of all features is also included in the meaning of the concrete instance. Just as with abstract instances, concrete instances can have substructure, or be nested within other concrete instances.

Unlike most programming languages, ACTI imposes no predetermined relationships between abstract and concrete instances. Implementations are meaningful in their own right (and in isolation), even without respect to any particular specification to which they may conform. While an abstract instance is in essence an "implementation-free" subsystem specification, a concrete instance is a "specification-free" implementation. The traditional notion of "conformance" between an implementation and a specification is thus many-to-many in this model.

In Figure 3, a concrete instance is shown as an electrical junction box without a face plate. Just like

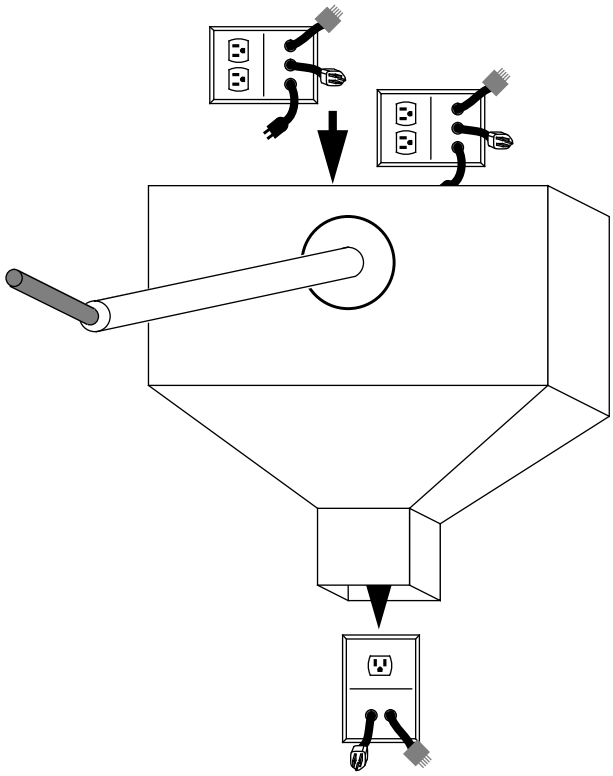

Figure 4: A Template Is A "Generator"

the abstract instance, the concrete instance has exported features (loose wires) and an explicit interface to its environment (a terminal block). Of course, one might expect the behavior of these features to be described in terms at a different level of abstraction from those used in the specification(s) to which this concrete instance conforms.

2.3 Templates

Templates in ACTI are subsystem generators. One can think of a template as a "function" that takes a subsystem as a parameter and produces a new subsystem as its result. An abstract template is a subsystem-to-subsystem function that can be applied to an abstract instance to generate another abstract instance. Effectively, an abstract template is a generic subsystem specification. A concrete template is a subsystem-to-subsystem function that can be applied to a concrete instance to generate another concrete instance. Thus, a concrete template is a generic subsystem implementation.

Figure 4 gives an intuitive impression of an abstract template. One or more abstract instances are provided as actual parameters, and the template is

applied to them (by turning the crank) to generate a new abstract instance.

2.4 Interpretation Mappings: Relationships Between Subsystems

In ACTI, the relationships between different subsystems are expressed explicitly via *interpretation mappings*. Intuitively, one can think of an interpretation mapping as being an "impedance matcher" between different abstract models of behavior. An interpretation mapping explains how one set of features can be "interpreted as" or "mapped into" another set of features in a behaviorally consistent way. This is shown in Figure 5 as a cable with differently shaped plugs. As with the other ACTI entities, there are also interpretation mapping templates for describing parameterized families of mappings between families of subsystems.

Interpretation mappings are used for several purposes:

- To explain how one abstract instance conforms to another (how one specification is a generalization of another).
- To explain how a concrete instance conforms to an abstract instance (how an implementation fulfills a specification).
- To explain how one or more external subsystems conform to the explicit context interface of an abstract or concrete instance.

Interpretation mappings are at heart explicit representations of *bindings* between subsystems. Because an ACTI subsystem intentionally includes a detailed description of its behavior, however, such a binding necessarily involves more than just a name-to-name correspondence—it must also include the equivalent of an *abstraction function* (or, more generally, *abstraction relation*) in order to bridge the gap between descriptions presented in completely different abstract terms.

While this paper can only give an overview of the concepts involved in ACTI, a strong intuitive grasp of the abstract versus concrete and template versus instance distinctions gives one an effective understanding of the heart of ACTI.

3 What Is Different Here?

All of the varieties of subsystems modeled in ACTI have already appeared in modern programming languages in one form or another—although rarely do all four appear together, and there is much disagreement about their details. The contributions of ACTI and its

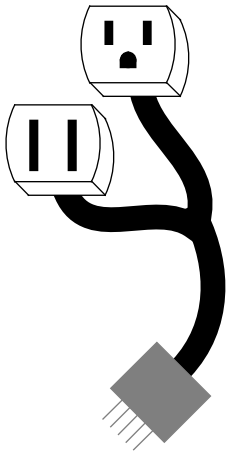

Figure 5: Interpretation Mappings Are Connectors

crucial differences are in the details of subsystems and how they fit together.

In ACTI, subsystems are meaningful by themselves. In particular, a specification has a well-defined meaning, including a complete picture of the behavior of the features it describes, even without respect to any implementation. An implementation also has meaning, without reference to any specification to which it might conform.

As a result, subsystems never depend directly on anything outside of themselves. All external dependencies are described through an explicit context interface, and there is no notion of "implicit" dependencies or hidden coupling between subsystems.

The "meaning" of a subsystem (a specification or implementation) is more than just a collection of types and operations. This is critically different than the notion of "module" in most programming languages. Instead, the meaning of a subsystem includes:

- An explicit model of behavior (independent of implementation details).
- An explicit model of all external dependencies (the context interface).
- A collection of definitions used to construct and describe behavioral models.
- Substructure (which is potentially complex).

In addition to the emphasis on subsystems, ACTI also includes explicit representation of correspondence

relationships between subsystems. In ACTI, all *bindings* are explicitly represented through these interpretation mappings:

- Binding an external unit to a subsystem's explicit context interface.
- Mapping a concrete instance to the abstract instance(s) it conforms to.
- Mapping an abstract instance to another abstract instance it conforms to.

Finally, in ACTI all entities can be parameterized. Implementations can be parameterized independently of specifications, and interpretation mappings can be parameterized independently of the subsystems they relate. Examples of the utility of this flexibility are surprisingly plentiful [2].

4 Support For Software Understanding

It is well-accepted that people form *mental models*—internal representations of external artifacts—for devices and other bits of technology with which they interact [6, p. 241]. From psychology, we understand that people do this naturally, and that such models help individuals in two ways [6, p. 241]:

1. A mental model allows one to *predict* the behavior of the person or thing with which the interaction takes place.
2. A mental model allows one to *explain why* the behavior arises.

Both of these benefits are important for a person to understand how to interact effectively with another person, a physical device, or a piece of complex software. Hence, a mental model that is *effective* is one that provides sufficient predictive and explanatory power, and which a person can reasonably internalize and use to understand an interaction.

Here, we are concerned with a "programmer-user's" interactions with a software subsystem, rather than with an end-user's interactions with a complete application. Following Norman's terminology [6], *target system* will be used to refer to the component subsystem with which a person is interacting. The *system image* is the entire visible "programmer interface" to the software component seen by a(nother) software professional. It may include a system specification, complete source code, manuals and instructions accompanying the software, and even the way the software behaves and responds under operating conditions.

Mental models evolve naturally through interaction with the target system [6, p. 241]. Over time, people reformulate, modify, and adapt their mental models whenever these models fail to provide reasonable predictive or explanatory power. For most purposes, the models need not be completely accurate, and usually they are not, but they must be functional. Norman documents the following general observations about mental models [6, p. 241]:

1. Mental models are incomplete.
2. People's abilities to simulate or mentally execute their models are severely limited.
3. Mental models are unstable: People forget the details of the system they are using, especially when those details (or the whole system) have not been used for some period.
4. Mental models do not have firm boundaries: similar devices and operations get confused with one another.
5. Mental models are "unscientific": People maintain "superstitious" behavior patterns even when they know they are unneeded because they cost little in physical effort and save mental effort.
6. Mental models are parsimonious: Often people do extra physical operations rather than the mental planning that would allow them to avoid those actions.

These observations indicate that mental models are inherently limited. These limitations stem from human cognitive limitations, a person's previous experiences with similar systems, and even misleading system images [6, p. 241]. As Norman points out:

> In making things visible [in the system image], it is important to make the correct things visible. Otherwise people form explanations for the things they can see, explanations that are likely to be false. ... People are very good at forming explanations, at creating mental models. It is the designer's task to make sure that they form the correct interpretations, the correct mental models: the system image plays the key role. [7, p. 198]

Given this information, how can one support the formation and maintenance of effective mental models for complex software systems? Norman points out

that the designer of a software subsystem already has a *conceptual model* of the system that is (presumably) accurate, consistent, and complete [6, p. 241][7, pp. 189-190]. The goal, in the best of all possible worlds, is to ensure that the system image is completely consistent with the conceptual model of the designer, and that from this system image the user forms a mental model consistent with the designer's conceptual model. Further, the system image should help alleviate the inherent human limitations of mental models. A well-designed system image can help to make the user's mental model more complete, to record details so the user's model has firm boundaries, and to present those details in such a way that the user's model is more stable.

ACTI was designed with these issues in mind, and as a result, explicitly incorporates model-based explanations of behavior in the meaning of each software subsystem. This behavioral description captures what the component designer wishes to impart about the conceptual model he intends for the client software engineer to acquire—it is a semantic (rather than purely syntactic) system image.

Because an ACTI subsystem has a meaning that contains the designer's conceptual model, it can serve as a cue to help the programmer *form and maintain* a mental model of the subsystem. By serving as a point of reference, it also helps to address the natural limitations of the programmer's internal model. This is the basis for supporting software that is understandable by humans, not just executable by machines.

5 Relation to Previous Work

In the realm of computer languages, Section 3 delineates the primary ways in which ACTI is unique with respect to previous work. For a discussion of previous efforts to effectively capture modular structuring techniques, Edwards [2] presents a thorough comparison of several programming languages that are representative of best current practices: RESOLVE [8, 9], OBJ [10], Standard ML [11], and Eiffel [1]. As typical of current efforts, most of these languages inadequately support the formation or maintenance of effective mental models of software parts. The complete analysis, including a detailed check list of software structuring and composition properties supported by these languages, is available electronically [2]. Other efforts to provide better support for mental models have concentrated primarily on adding (possibly structured) comments to component specifications, the inadequacy of which is explained in [12].

Interestingly, there are other areas of computer science where similar work has been and is being carried out. Current work on Functional Representation [13, 14, 15, 16] (FR) is closely related. FR grew out of artificial intelligence work on reasoning about physical systems, partly motivated by diagnostic and design problem solving. As with other AI work on these problems, FR originally focused on the functions of devices—that is, the effects objects have on their environment. More recently, B. Chandrasekaran has documented an ontological framework within which the notion of "function" can be explained, and which provides a unified technical vision underlying the various approaches to device understanding [17].

This more general framework has allowed Chandrasekaran to present FR as a general theory of comprehension, along with a specific language that serves as a corresponding representation mechanism [17]. He defines comprehension to be the task of producing one or more of the following descriptions of a given artifact:

- The intended **function** of the artifact, i.e., its behavior.
- The **structure** of the artifact, i.e., a specification of its components and how they are put together.
- A **causal account** of how the artifact achieves its function, and the roles played by the components in achieving it.

The result is that FR can be considered to be a domain-independent theory for understanding how system-level behaviors emerge from a system's structure.

ACTI is aimed at exactly the same goal within the domain of software systems. As a result, it is not surprising that ACTI and FR share a number of critical features:

- Support for human understanding is a primary goal.
- Behavior is described independently of structure and implementation. In particular, behavioral descriptions are meaningful in isolation, without respect to any particular device that may provide such behavior.
- There is the potential for multiple realizations of the same behavior.
- An explicit bridge must be constructed between the (more abstract) vocabulary of a system-level behavioral description and the (lower level) functions achieved by the system's component parts.

- Similarly, when devices are composed to form larger structures, one must explicitly describe the bridge between them.

- The "context interface" through which external forces can act on a given artifact is explicitly defined.

Earlier FR work has even been applied to software for diagnostic and explanation purposes [18, 19]. To date, this has been at the "programming-in-the-small" level of individual statements and their interactions, while ACTI addresses "programming-in-the-large" issues of subsystem meaning and composition.

The similarities between ACTI and FR, two efforts that were arrived at independently, strengthen the claim that they both make progress toward supporting human understanding effectively. Further, there are areas where the two complement each other. FR, for example, provides for explicit representation of a "causal account" of how aggregate behavior arises from the functions of individual parts, which can contribute to human understanding for modification or diagnostic purposes. Similarly, ACTI provides explicit support for specification adornments—a place for designers to capture model-building tools and useful subparts of behavioral descriptions. The complementary nature of these parallel efforts provides fruitful ground for future work on integrating the two frameworks.

6 Conclusions

ACTI addresses the problem of *understandable* software composition across module- and class-based languages. To achieve this, it gives a real semantic denotation to subsystems (modules) that includes a simple (i.e., not "bottom-up") model of behavior. It allows one to clearly describe why abstractions (simple models) correctly capture the behavior of complex combinations of lower-level parts, using interpretation mappings.

Because each ACTI subsystem has a meaning that reflects its designer's conceptual model, it supports the formation of mental models by client programmers, and also addresses their limitations. This is a critical missing link in the support of understandable software that has been ignored by previous efforts in programming language design.

ACTI is universal, in that it is not tied to any particular programming language. It unifies module-structured and object-oriented notions of software, and can serve as a general theory of software structure and meaning.

Acknowledgements

The author gratefully acknowledges financial support from the National Science Foundation (grant number CCR-9311702) and the Advanced Research Projects Agency (contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714). B. Chandrasekaran deserves special thanks for recognizing and cultivating the ties between Functional Representation and ACTI.

References

[1] B. Meyer, *Object-Oriented Software Construction.* New York, NY: Prentice Hall, 1988.

[2] S. Edwards, *A Formal Model of Software Subsystems.* PhD thesis, Dept. of Computer and Information Science, The Ohio State University, Columbus, OH, 1995. Also available as technical report OSU-CISRC-4/95-TR14, by anonymous FTP from `ftp://ftp.cis.ohio-state.edu/pub/tech-report/1995/TR14-DIR`, or through the author's home page.

[3] W. F. Ogden, M. Sitaraman, B. W. Weide, and S. H. Zweben, "The RESOLVE framework and discipline—a research synopsis," *ACM SIGSOFT Software Engineering Notes*, vol. 19, pp. 23–28, Oct. 1994.

[4] W. Tracz, "Implementation working group summary," in *Reuse in Practice Workshop Summary* (J. Baldo, Jr., ed.), (Alexandria, VA), pp. 10–19, IDA Document D-754, Institute for Defense Analyses, Apr. 1990.

[5] E. Robinson, "Logical aspects of denotational semantics," in *Category Theory and Computer Science* (D. H. Pitt, A. Poigné, and D. E. Rydeheard, eds.), vol. 283 of *Lecture Notes in Computer Science*, pp. 238–253, New York, NY: Springer-Verlag, 1987.

[6] D. A. Norman, "Some observations on mental models," in *Readings In Human-Computer Interaction: A Multidisciplinary Approach* (R. M. Baecker and W. A. S. Buxton, eds.), pp. 241–244, San Mateo, CA: Morgan Kaufmann Publishers, Inc., 1987.

[7] D. A. Norman, *The Design of Everyday Things.* New York, NY: Doubleday/Currency, 1990.

[8] M. Sitaraman and B. W. Weide, editors, "Special feature: Component-based software using RESOLVE," *ACM SIGSOFT Software Engineering Notes*, vol. 19, pp. 21–67, Oct. 1994.

[9] B. W. Weide, W. F. Ogden, and S. H. Zweben, "Reusable software components," in *Advances in Computers* (M. C. Yovits, ed.), vol. 33, pp. 1–65, Academic Press, 1991.

[10] J. A. Goguen, "Principles of parameterized programming," in *Software Reusability, Volume I: Concepts and Models* (T. J. Biggerstaff and A. J. Perlis, eds.), pp. 159–225, New York, NY: ACM Press, 1989.

[11] R. Milner, M. Tofte, and R. Harper, *The Definition of Standard ML.* Cambridge, MA: MIT Press, 1990.

[12] S. H. Edwards, "Good mental models are necessary for understandable software," in *Proceedings of the Seventh Annual Workshop on Software Reuse* (L. Latour, ed.), Aug. 1995.

[13] B. Chandrasekaran, "Functional representation and causal processes," in *Advances in Computers* (M. C. Yovits, ed.), vol. 38, pp. 73–143, Academic Press, 1994.

[14] Y. Iwasaki, R. Fikes, M. Vescovi, and B. Chandrasekaran, "How things are intended to work: Capturing functional knowledge in device design," in *Proceedings of the 13th International Joint Conference on Artificial Intelligence*, pp. 1516–1522, Morgan KAufmann, 1993.

[15] M. Vescovi, Y. Iwasaki, R. Fikes, and B. Chandrasekaran, "CFRL: A language for specifying the causal functionality of engineered devices," in *Eleventh National Conference on AI*, pp. 626–633, AAAI Press/MIT Press, 1993.

[16] Y. Iwasaki and B. Chandrasekaran, "Design verification through function- and behavior-oriented representation," in *Artificial Intelligence in Design '92* (J. S. Gero, ed.), pp. 597–616, Kluwer Academic Publishers, 1992.

[17] B. Chandrasekaran, "An explication of function." The Ohio State University, Laboratory for AI Research, draft, 1996. Also available electronically from `http://www.cis.ohio-state.edu/~chandra`.

[18] D. Allemang, "Using functional models in automatic debugging," *IEEE Expert*, vol. 6, pp. 13–18, 1991.

[19] B. Liver and D. T. Allemang, "A functional representation for software reuse and design," *International Journal of Software Engineering and Knowledge Engineering*, vol. 5, pp. 227–269, 1995.