APPENDIX C

How Does the ACTI Model Measure Up?

Given the formal description in Chapter IV, the ACTI model will be analyzed using the check list developed in Appendix B to determine how comprehensive it is. Section C.1 presents this analysis. Subsequent sections illustrate how each of the prior models described in Appendix A can be interpreted within ACTI to demonstrate its generality.

C.1 Comparing ACTI with the Check List

Section 3.1 presented three overriding concerns that guided the development of ACTI: defining it in mathematical terms, maintaining language independence, and achieving all of the properties described in the structured check list of Appendix B. While ACTI’s form addresses the first two, the third is addressed by ACTI’s content. This section provides a point by point explanation of how ACTI satisfies all of the properties described in the structured check list of Appendix B. Since this check list serves as an operational definition of the “underlying conceptual view of software architecture embedded in modern module-structured languages” described in the thesis, this section is the heart of defending the claim that ACTI does in fact capture this information.

C.1.1 Abstractions

Type Abstractions

A Type Environment, which appears in every abstract or concrete instance, maps type names to mathematical domains representing the allowable values for that type. Because any arbitrary domain can be used as the associated value of a type name in a Type Environment, it is possible for arbitrarily abstract or concrete sets of values to be associated with a type.
Typically, one would expect abstract characterizations of the domain associated with a type or data abstraction to appear in an abstract instance representing a subsystem specification. Similarly, a more concrete characterization of the representation of the object-level state present in a data abstraction to be captured within a concrete instance representing a subsystem implementation. The two could then be combined using a suitable interpretation mapping to form an encapsulated unit, as described in Section 4.13.3, combining the abstract description of behavior with their concrete but encapsulated representations.

Note that in some programming languages, type information is completely relegated to the typing system and considered part of the static semantics of the language. Since ACTI must deal meaningfully with the relationships between subsystem specifications and implementations, which necessarily includes dealing with the relationship between an abstract type specification and its low-level representation, the domain information associated with a type must be included in the model.

**Types May Have Invariant Properties**

In addition to associating a mathematical domain with a type name, each *Type Environment* also associates a *CNSTR* with the type. This predicate over the corresponding domain can be used to describe type invariants.

**Types Have Abstract Models**

As described above, the mathematical domain *MD* associated with a type name by a *Type Environment* can be any arbitrary domain. This allows the flexibility for choosing abstract domains for characterizing the values of a type, and such a domain serves as an explicit model of the type.

**State Abstractions**

The *Variable Environment* within each concrete instance effectively captures the notion of state. It is an association between names (variable names) and their respective types and values. For an abstract instance, its *Variable Environment Signature* simply describes the type of its state by associating names with type information only.

**Operation Abstractions**

Every concrete instance contains an *Operation Environment* component, which associates operation names with a corresponding *Operation Meaning*. This *Operation Meaning* includes an explicit representation of the input-to-output relation computed
by the operation, as well as the effects of the operation on the state of the concrete instance.

**Operations Entail Behavior**

Every *Operation Meaning* associated with an operation includes an *Operation Model*, which is made up of an abstract characterization of when the operation is legitimately applicable (*DP*), and a relational description of the effects of the operation if it is applied legitimately (*EP*). As one would expect, the *Operation Environment Signature* in an abstract instance associates only an *Operation Model* with each operation name, rather than a complete *Operation Meaning*.

**Subsystems: Collections of Definitions**

Concrete instances serve as cohesive bundles of type, state, operation, and subsystem definitions that can be treated as a single composite structure.

**Subsystems May Have Invariant Properties**

Every abstract and concrete instance includes an *Environment Invariant*, which is a predicate over the entire subsystem. This predicate can be used to capture subsystem-level invariant properties.

**Nested Subsystems**

Every concrete instance has four subcomponents for modeling nested subsystems:

2. A *Concrete Instance Environment* for nested concrete instances.

Abstract instances, of course, may only contain abstractions of concrete instances.

**Subsystem Abstractions**

An *Abstract Instance* is an abstraction over a collection of concrete instances. It is also an abstraction over a collection of other, more detailed abstract instances. In this way, abstract instances are similar to OBJ theories, which are abstractions over both OBJ objects and other OBJ theories.
Parameterized Subsystems

Templates fulfill this role. A Concrete Template is a function taking a single concrete instance as an argument and producing another concrete instance as its result. Since a concrete instance can contain arbitrarily many types, variables, operations, and nested subsystems, the template effectively has access to an unlimited number of arguments. Similarly, because the resulting concrete instance can contain arbitrarily many types, variables, operations, nested instances, and nested templates, a given template can simultaneously produce an unlimited number of results, including other templates. ACTI templates are higher-order functors, in the Standard ML sense [44].

Similarly, an Abstract Template is a function taking a single abstract instance as an argument and producing another abstract instance as its result. Together, abstract and concrete templates provide the heart of ACTI’s support for parameterized programming.

Explicit Context Interfaces

The CTXT component of every abstract and concrete instance serves as an explicit declaration of external dependencies.

C.1.2 Separation of Concerns

Specification Versus Implementation

The space of Abstract Instances is separate and distinct from that of Concrete Instances. Great pains have been taken to ensure that appropriate separation here is observed.

The Specification/Implementation Relationship

All relationships between specifications and implementations (or between pairs of specifications) are captured in the space of Interpretation Mappings. A Interpretation Mapping describes the correspondence between two instances independently of both of the instances involved.

Information Hiding

As described in Section 4.13.3, given an appropriate abstract instance and interpretation mapping $M$, a concrete instance can be “cut down” using the $\downarrow M$ operator. Intuitively, this operator simply removes all information from the concrete instance
that is not mapped through the abstract interface via $M$. This provides a useful way to capture information hiding and encapsulation.

**Context Versus Specification**

Every abstract instance has a $CTX$ component which serves as an explicit declaration of external dependencies. All reference to entities that are not defined within the abstract instance must be imported through this context interface, thus effecting a separation between those external dependencies and the subsystem specification itself.

**Context Versus Implementation**

Every concrete instance has a $CTX$ component which serves as an explicit declaration of external dependencies. All reference to entities that are not defined within the concrete instance must be imported through this context interface, thus effecting a separation between those external dependencies and the subsystem implementation itself.

**Specification Information Versus Program Code**

Both abstract and concrete instances contain $Math Environments$, where all definitions made expressly for simplifying the specification can be placed. Here, they will be kept separate from the definitions of programmatic entities in the remainder of the subsystem.

**Derivations Versus “Is-A” Relationships**

In ACTI, is-a relationships are captured by $Interpretation Mappings$, as discussed below in Section C.1.3. Derivations, however, are captured by operators that produce new subsystems by combining, extending, or restricting existing ones. Thus, these two classes of relationships are separated in ACTI.

**C.1.3 Mappings and Bindings**

**Specification-to-Implementation Mappings**

Specification-to-implementation relationships in ACTI are expressed as elements in the space of $Interpretation Mappings$. 
Specification-to-Specification Mappings

Specification-to-specification relationships in ACTI are expressed as elements in the space of Interpretation Mappings.

Binding Context

Bindings between the CTXT component of an abstract or concrete instance and its immediately containing subsystem (its environment) are defined as Interpretation Mappings. This, combined with module embedding, provides an expressive way of explaining how a given environment fulfills the external dependency requirements of a given subsystem as described in Section 4.13.6.

Intensional Contextual Binding

Every Interpretation Mapping is intensional, since the mapping itself is independent of both instances involved. Thus, the CTXT component of a given abstract or concrete instance is simply a statement about the minimum properties any corresponding external environment must have, and any environment possessing these properties, regardless of its name or how it was derived, can provide an effective parent environment for the given instance.

Binding Subsystem Parameters

Templates are essentially functions from instances to instances. Providing the actual parameter values corresponding to the formal subsystem parameters is simply mathematical function application.

Intensional Subsystem Parameter Binding

Each template has an associated “domain predicate” which is true for all instances to which that template may legitimately be applied. Both intensional and extensional parameter binding mechanisms can be modeled by such a domain predicate.

“Is-A” Type Relationships

In ACTI, is-a relationships are captured by Interpretation Mappings, which express how one abstract or concrete instance can be “interpreted as” another abstract instance. As part of expressing this relationship, the mapping must also explain how types in the first can be interpreted as types in the second in a behavior-preserving
way. Thus, interpretation mappings effectively capture is-a relationships at the both the type and the subsystem levels.

**Types With Many Representations**

Because the mathematical domain of values associated with a type can be any arbitrary domains, a disjoint union of domains could be used as the space of values for a given type. This domain could be used to model the many alternative representations for a given type.

Further, one could also provide a domain of uniform values as a model of the same type, at a higher level of abstraction. If the necessary type interpretation mapping between the two spaces can be defined, then a interpretation mapping between the two instances can be defined, and the $\downarrow_M$ operator can be used to encapsulate the lower-level representation details.

**Late Binding**

The domains associated with types are flexible enough to encompass operations—the space of *Operation Meanings*, for example, is just another domain which could be used to define the values associated with some type. Types which are modeled by highly structured domains containing one or more operations as part of each object value can be used to turn operation binding into a behavior modeled at run-time. Further, type-level or module-level tags attached to values can also be used for dynamic dispatch purposes.

**C.1.4 Derivations**

**Specification Derivation**

Derivation of one specification in terms of an already existing specification is easily defined as an operator over the space of abstract instances, or even abstract templates. This is described in Section 4.13.4.

**Implementation Derivation**

Derivation of one implementation in terms of an already existing implementation is easily defined as an operator over the space of concrete instances, or even concrete templates. This is described in Section 4.13.4.
Specification-to-Implementation Relationship Derivation

Derivation of one interpretation mappings in terms of an already existing interpretation mappings is easily defined as an operator over the space of interpretation mappings, just as for abstract and concrete instances.

Simultaneous Specification/Implementation Derivation

Given an abstract instance $AI$, a concrete instance $CI$, and a interpretation mapping $M$ describing their relationship, it is simple to define a composite derivation operator from the more primitive derivations for abstract instances, concrete instances, and interpretation mappings. The typical inheritance mechanism in an OOP language should be interpreted as just such a composite operator.

C.2 Interpreting the 3Cs within ACTI

Given that ACTI possesses all 22 properties in the structured check list, the next step is to see if each of the previous models can be “reinterpreted” within the framework of ACTI’s mathematical spaces. This will also give the reader a better intuitive understanding of what can be modeled ACTI, and how.

The 3C model is centered around the notions of concept, content, and context, and appropriately separating them. Since these terms are not formally defined, it is difficult to talk about whether or not they are perfectly captured in ACTI. Nonetheless, these notions all have clear representatives in ACTI:

\[
\begin{align*}
\text{Concept} & = \text{Abstract Instance} & (C.1) \\
\text{Generic Concept} & = \text{Abstract Template} & (C.2) \\
\text{Content} & = \text{Concrete Instance} & (C.3) \\
\text{Generic Content} & = \text{Concrete Template} & (C.4) \\
\text{Context of Concept} & = CTXT \text{ of Abstract Instance} & (C.5) \\
\text{Context of Content} & = CTXT \text{ of Concrete Instance} & (C.6)
\end{align*}
\]

Clearly, ACTI is concerned with strict separation of concept from content, context from concept, and context from content.

C.3 Interpreting OBJ within ACTI

Interpreting OBJ within the mathematical spaces of ACTI is more challenging. This is particularly true because OBJ was designed as an executable specification language,
and thus has no real concept of an “implementation.” OBJ objects are the natural analog of implementations in that language, though, an analogy supported by the language’s denotational semantics [21, p. 178].

The core OBJ constructs and their ACTI representatives are:

\[
\begin{align*}
\text{Object} &= \text{Concrete Instance} \quad \text{(C.7)} \\
\text{Parameterized Object} &= \text{Concrete Template} \quad \text{(C.8)} \\
\text{Theory} &= \text{Abstract Instance} \quad \text{(C.9)} \\
\text{Parameterized Theory} &= \text{Abstract Template} \quad \text{(C.10)} \\
\text{Axioms} &= \text{Collection of Operation Models} \quad \text{(C.11)} \\
\text{Views} &= \text{Interpretation Mappings} \quad \text{(C.12)} \\
\text{Module Composition} &= \text{Derivation Operators} \quad \text{(C.13)}
\end{align*}
\]

The only case in which these identities are only approximate is that of views. Ostensibly, OBJ views are type-to-type and operation-to-operation name mappings [21, p. 191]. Unfortunately, because OBJ is based on an algebraic semantic model and all operations are functions in the mathematical sense, views may also map a composition of operations to an operation. Two descendants of OBJ, LIL and LILEANNA, carry this a step further, allowing arbitrary code (implementation) fragments to be introduced as part of a view. Because implementations should be encapsulated, and introducing code fragments as part of an abstract interpretation map opens up conflicting goals, this feature of views is (purposefully) not captured by ACTI interpretation mappings.

### C.4 Interpreting RESOLVE within ACTI

RESOLVE is a fairly natural fit for ACTI, since it’s denotational semantics played a prominent role in the initial formulation of the model. As a result, ACTI is more general than RESOLVE, but still retains structures that can clearly model the basic RESOLVE constructs:

\[
\begin{align*}
\text{Concept} &= \text{Abstract Template} \quad \text{(C.14)} \\
\text{Realization} &= \text{Concrete Template} \quad \text{(C.15)} \\
\text{Realization Parameters} &= \text{Parameters to Concrete Template} \quad \text{(C.16)} \\
\text{Type Correspondence} &= R \text{ in Type Interpretation Mapping} \quad \text{(C.17)} \\
\text{Type Convention} &= C \text{ in Type Interpretation Mapping} \quad \text{(C.18)} \\
\text{Module Correspondence} &= MCORR \text{ in Interpretation Mapping} \quad \text{(C.19)}
\end{align*}
\]
Module Convention = $MCONV$ in Interpretation Mapping \hspace{1cm} (C.20)
Math Definitions = Math Environment in an Instance \hspace{1cm} (C.21)
State Variables = $MVE$ in a Math Environment \hspace{1cm} (C.22)

C.5 Interpreting Eiffel within ACTI

Eiffel, the only OO language among those presented in Appendix A, identifies the ideas of module and type. This gives the language a different flavor, but ACTI models the two concepts differently. However, it is easy to interpret an Eiffel class as an ACTI concrete instance with a single type defined in its $Type\ Environment$. Given this interpretation, the other Eiffel constructs can be interpreted as follows:

\begin{align*}
\text{Class} & = \text{Concrete Instance plus Type} \hspace{1cm} (C.23) \\
\text{Generic Class} & = \text{Concrete Template} \hspace{1cm} (C.24) \\
\text{Class Attribute} & = \text{Entry in} \ Variable\ Environment \hspace{1cm} (C.25) \\
\text{Class Feature} & = \text{Operation Meaning} \hspace{1cm} (C.26) \\
\text{Feature Assertions} & = \text{Operation Model} \hspace{1cm} (C.27) \\
\text{Class Invariant} & = EI \text{ in Concrete Instance} \hspace{1cm} (C.28) \\
\text{Inheritance} & = \text{Combined Derivation Operator and} \hspace{1cm} (C.29) \\
& \hspace{1cm} \text{Interpretation Mapping} \hspace{1cm} (C.30)
\end{align*}

Eiffel polymorphism can be interpreted in terms of interpretation mappings, which explain how to translate between module and type abstractions with is-a relations. Eiffel’s dynamic binding can also be modeled explicitly in ACTI, for example by associating “tag” values with concrete instances and recording these values in each object belonging to the type defining in that type defining in that concrete instance. The model of computation built on top of these math spaces can then use these tag values in selecting the appropriate $Operation\ Meaning$ to apply to a given object.

C.6 Interpreting Standard ML within ACTI

Just like RESOLVE, Standard ML is also a natural fit for ACTI because of the role its denotational semantics played in the initial formulation of the model. The ACTI structures that clearly model the basic Standard ML constructs are:

\begin{align*}
\text{Structures} & = \text{Concrete Instance} \hspace{1cm} (C.31) \\
\text{Signatures} & = \text{Abstract Instance} \hspace{1cm} (C.32)
\end{align*}
C.6. INTERPRETING STANDARD ML WITHIN ACTI

Functors = Concrete Template \hspace{0.5cm} (C.33)
Types = Types \hspace{0.5cm} (C.34)
Function Values = Operation Meanings \hspace{0.5cm} (C.35)
Values = entries in a Variable Environment \hspace{0.5cm} (C.36)

Interestingly, Standard ML functions can be modeled within either the Operation Environment or the Variable Environment of a subsystem, since they are just values of a different type in that language. In the analogies above, they have been identified with ACTI operation meanings, however, since operations have attached behavioral descriptions, which values do not.
APPENDIX C. HOW DOES THE ACTI MODEL MEASURE UP?