CHAPTER IV

The ACTI Definition

This chapter presents the formal definition of ACTI. For the reader interested in the development of ACTI, Appendices A through C document the relevant background material, the method used to develop it, its requirements, and brief comparisons with four existing programming languages.

4.1 Foundational Concepts

The ACTI model is defined as a collection of mathematical spaces, together with a collection of operations on those spaces. To simplify the formal definition of the model, it relies on several foundational concepts. These basic concepts are central to understanding ACTI and how to use it, and they are described in this section.

4.1.1 Complete Partial Orders

Because ACTI is defined using the concepts of denotational semantics, the reader should be familiar with the idea of a complete partial order. A partial order is a pair \((D, \sqsubseteq)\) of a set \(D\) and a relation \(\sqsubseteq\) on \(D\) that is reflexive, antisymmetric, and transitive. Briefly, a complete partial order (CPO) is a partial order \((D, \sqsubseteq)\) where the following conditions hold:

1. The set \(D\) has a “least” element with respect to \(\sqsubseteq\). This least element is denoted by \(\sqsubseteq_D\), or simply \(\bot\) (which is read “bottom”).

2. For every chain \(S\) of elements in \(D\) related by \(\sqsubseteq\), there exists a least upper bound of the chain, denoted \(\sqcup S\).

For formal definitions of chains, least upper bounds, and other concepts related to CPOs, consult a text on denotational semantics (e.g., [42]).
CHAPTER IV. THE ACTI DEFINITION

Because CPOs play such a central role in denotational semantics, it is not surprising that all of the mathematical spaces in ACTI are CPOs. More formally, the universe of discourse in which ACTI is defined is the category CPO of CPOs; all of the mathematical spaces defined in ACTI are objects in this category.

For the purposes of this dissertation, the defined mathematical spaces can be appreciated with a limited knowledge of CPOs, including two important facts. First, any ordinary set of values can be turned into a simple CPO; and second, the collection of all CPOs is closed under Cartesian product and function formation operations.

An ordinary set $S$ can be turned into a flat CPO [42, p. 74–75]. To do this, pick any value that is not in the set $S$; call this element $\bot_S$. Now it is easy to form the set $S_\bot = S \cup \\{\bot\}$. Then the relation $\sqsubseteq_S$ can be defined:

$$\forall a, b \in S : a \sqsubseteq_S b \iff (a = \bot_S \text{ or } a = b) \quad (4.1)$$

Clearly, $(S_\bot, \sqsubseteq_S)$ is a CPO.

Next, if $(D_1, \sqsubseteq_{D_1}), \ldots, (D_n, \sqsubseteq_{D_n})$ are all CPOs, then $(D_1 \times \cdots \times D_n, \sqsubseteq_x)$ is also a CPO [42, p. 75]. Here, the relation $\sqsubseteq_x$ is the relation formed by point-wise application of the component $\sqsubseteq_{D_i}$ relations. Similarly, the least element of the new CPO is $\bot_{D_1 \times \cdots \times D_n}$.

Finally, if $(D, \sqsubseteq_D)$ and $(E, \sqsubseteq_E)$ are both CPOs, then $((D \rightarrow E), \sqsubseteq_{(D \rightarrow E)})$ is also a CPO [42, p. 75]. The least element in the new CPO is the function that maps every element in $D$ to $\bot_E$. Once again, the relation $\sqsubseteq_{(D \rightarrow E)}$ is formed by point-wise comparison of the range values of two functions across all possible domain values.

In the examples above, $\bot$ and $\sqsubseteq$ have been subscripted to indicate which sets they are intended to operate on, for clarity. Typically, however, these symbols are written without subscripts where their intended meaning is clear from context. This convention will be followed in the remainder of this chapter.

The majority of the mathematical spaces in ACTI are constructed using cartesian product formation or function formation, so it should be clear at each point how the individual spaces are interpreted as CPOs.

4.1.2 Name-to-Value Mappings

Section 3.4 discusses how run-time program states can be viewed in general terms as name-to-value mappings. Similarly, subsystems are collections of name-to-value associations that can be manipulated as a whole.
Within this dissertation, the term *mapping* will be used as another name for a function $f : D \rightarrow E$ from one CPO to another. The notation\(^1\) $\text{Dom } f$ will be used to refer to the “meaningful” domain of $f$; that is, the set of all $d \in D$ where $f(d) \neq \bot$. Similarly, $\text{Ran } f$ will be used to refer to the “meaningful” range of $f$; that is, the set of all $e \in E$ not equal to $\bot$ such that there is some $d \in D$ where $f(d) = e$. We will say a particular $d \in D$ is *defined* in $f$ when $f(d) \neq \bot$. Otherwise, when $f(d) = \bot$, we say that $d$ is *undefined* in $f$.

Because the mappings associated with typical software subsystems usually have small meaningful domains, they are sometimes written out explicitly. The notation $f = \{d_1 \mapsto e_1, \ldots, d_n \mapsto e_n\}$ will be used to explicitly define a mapping with $\text{Dom } f = \{d_i : 1 \leq i \leq n\}$ and $\text{Ran } f = \{e_i : 1 \leq i \leq n\}$. Note that $f$ is still a total function, and it is understood that for all $d \not\in \text{Dom } f$, $f(d) = \bot$. Thus, the shorthand $\{\}$ refers to the mapping for which $f(d) = \bot$ for all $d \in D$.

Given two maps $f$ and $g$ both from $D$ to $E$, we can also talk about combining them. The map $f + g$, read *$f$ modified by $g$* \cite[p. 17]{48}, is the map with meaningful domain $\text{Dom } f \cup \text{Dom } g$ defined as follows:

$$
(f + g)(d) = \begin{cases} 
g(d) & \text{if } g(d) \neq \bot \\
f(d) & \text{otherwise} \end{cases}
$$

The term *name-to-value mapping* will be used for any mapping $f : \text{Names} \rightarrow E$ from the space of *Names* to another CPO $E$. The space *Names* is defined in Section 4.3.3.

### 4.1.3 Environments

Name-to-value mappings are intuitively easy to understand, especially when related to their roots in program states. In traditional operational or denotational semantics, a program state simply records the value of every named object at a particular point in the execution history of a program. One can also think of such a collection of name-to-value associations as an “execution environment,” or simply an “environment” that gives particular bindings for some set of names. As a result, the term *environment* will be used for any ACTI mathematical space that is a name-to-value mapping.

\(^1\)All of the notation presented in this section is adapted from that in the Standard ML definition \cite[pp. 16–17]{48}. Note, however, that here all mappings are total functions, while the Standard ML notation is used with partial functions having finite domains.
4.1.4 Signatures

Because ACTI respects the central concerns of software engineering, its design was affected by the need for appropriately separating specifications from implementations. As a result, many of the mathematical spaces in the model are related in pairs—one space to represent actual software objects, and the other to represent specifications, or conceptual models, of those objects. A space of specification-only values is called a space of signatures. Such a space contains behavioral signatures, each of which describes the externally observable behavior of some class of software objects. This is a richer notion that contrasts with the usual use of the term “signature” in computer science, where it is often used to mean a purely syntactic description, such as just an operation’s parameter profile.

For example, the space of Abstract Templates described in Section 4.8 captures the denotations of generic specifications. It is closely tied to the space of Abstract Template Signatures, which capture the designer’s conceptual model of a generic specification—its parameter profile and an abstract description of its behavior expressed as pre- and postconditions. All ACTI spaces labeled as spaces of signatures serve this purpose. As one would expect from the informal overview presented in Chapter III, the signature for a concrete instance is an abstract instance.

4.1.5 Well-Formed Objects

Unfortunately, simply defining the structure of values in the mathematical spaces of the ACTI model is not enough. Many kinds of objects in ACTI contain redundant or mutually dependent information, and the consistency of this information cannot be assured simply by examining the structure of the object.

Thus, meaningful objects that can model actual program structures form a subset of the values that are contained in ACTI’s spaces. We call these objects well-formed objects, indicating that they are internally consistent. As each new mathematical space is introduced in the ACTI definition, complete criteria for all well-formed objects within that space are also defined. For spaces where no explicit criteria are given, all objects within the space are considered well-formed.

For many spaces, we are concerned with more than just internal consistency—we are also concerned with mutual consistency in the context of some other object, say the containing subsystem. Thus, some objects can only be judged as well-formed with respect to another object. Since subsystems are the cornerstone of the ACTI world, the well-formedness of any particular subsystem is absolute, independent of any other objects. This is reflected in the criteria for well-formed abstract instances, concrete instances, abstract templates, and concrete templates. For most other spaces, how-
ever, the notion of a well-formed object is explicitly defined with respect to an object from another space, often an abstract or concrete instance.

Here, we are only concerned with collections of meaningful objects. Throughout the definition we assume that all specific objects under discussion are well-formed.

### 4.1.6 Assertive Programming

Some programming languages, such as RESOLVE and OBJ, already allow formal behavioral descriptions of program elements. In order to support the formal semantics of assertions used in these descriptions, ACTI incorporates the ideas of *assertive programming*. Assertive programming is centered around the idea of giving meaning to behavioral assertions, such as pre- and postconditions, the same way meaning is given to procedures or statements—in the formal semantics, assertions affect the run-time state of execution.

To record the effects the formal assertions have on a program’s state within the formal semantics, we follow Ernst *et al.* [12, 10] and add the notion of an *Assert Status* to the ACTI definition. An assert status component is added to the denotation of a run-time state to reflect the cumulative effect of “executing” the assertions in a program.

An *Assert Status* value is one of \{CF, VT, NL, \perp\}. Certain assertions in a program must be true in order for the program to be considered correct, and the violation of such an assertion results in an assert status of *categorically false* (CF). Similarly, some assertions are assumed to be true, and if such an assertion fails to hold, it results in an assert status of *vacuously true* (VT). The NL assert status value stands for *neutral*, and is used in program states where no assertions have been violated. The value “bottom” (\perp) is used when the assert status of a particular program state is indeterminate. The curious reader is encouraged to explore the concepts of assertive programming further ([12], [10, pp. 267-268]).

### 4.1.7 Intuitive Explanations

Continuing the approach begun in Chapter III, the definition of the ACTI model is written to appeal as much as possible to one’s intuition and common sense. This is as much to ease the understanding of the formal definition as it is to make the material accessible to a wider audience. To this end, the formal definitions of all but the simplest mathematical spaces are followed by an “intuitive explanation” of the ideas that are captured in the formal definition. Most frequently, this means relating the abstract formal definition to more concrete programming language terms, but the additional explanatory material also may contain rationale for some choices made in
CHAPTER IV. THE ACTI DEFINITION

the formal definition. The reader who is more interested in getting a quick grasp of
the general ideas behind ACTI’s structure rather than a perfect mastery of the formal
definitions themselves should focus attention toward these intuitive explanations.

4.2 Outline of the Mathematical Spaces in ACTI

Because of the four-way partitioning of the kinds of subsystems in the ACTI world,
the formal definition of ACTI is centered around four distinct mathematical spaces:
one for each kind of subsystem. For simplicity, these spaces are named after each of
the subsystem classes: abstract instances, concrete instances, abstract templates,
and concrete templates.

The remainder of this chapter presents the definitions of these mathematical
spaces, along with the details of the smaller spaces from which they are constructed.
These are presented in the following order:

- Names (\(\mathcal{N}\))
- Modeling Domains (\(\mathcal{MD}\))
- Environments or Concrete Instances (\(\mathcal{E}\))
  - Type Environments (\(\mathcal{T E}\))
    * Type Models (\(\mathcal{T M}\))
  - Variable Environments (\(\mathcal{V E}\))
    * Variable Models (\(\mathcal{V M}\))
    * Variable Signatures (\(\mathcal{VS}\))
  - Operation Environments (\(\mathcal{O E}\))
    * Operation Meanings (\(\mathcal{OM}\))
    * Operation Models (\(\mathcal{OMOD}\))
    * Parameter Profiles (\(\mathcal{PP}\))
  - Environment Invariants (\(\mathcal{EI}\))
  - Abstract Instance Environments (\(\mathcal{AIE}\))
    * Context-Bound Abstract Instances (\(\mathcal{CBAL}\))
  - Concrete Instance Environments (\(\mathcal{CIE}\))
    * Context-Bound Concrete Instances (\(\mathcal{CBCI}\))
  - Interpretation Mapping Environments (\(\mathcal{IME}\))
4.2. OUTLINE OF THE MATHEMATICAL SPACES IN ACTI

- Abstract Template Environments (ATÉ)
- Concrete Template Environments (CTÉ)
- Interpretation Mapping Template Environments (IMTÉ)

• Abstract Instances (AI)
  - Variable Environment Signatures (VES)
  - Operation Environment Signatures (OES)
  - Abstract Template Environment Signatures (ATÉS)
  - Concrete Template Environment Signatures (CTÉS)

• Concrete Templates (CT)
  - Concrete Template Signatures (CTS)

• Abstract Templates (AT)
  - Abstract Template Signatures (ATS)

• Specification Adornment Environments (SAÉ)
  - Specification Adornment Variable Environments (SAVE)
  - Specification Adornment Environment Invariants (SAÉI)
  - Specification Adornment Instance Environments (SAIÉ)
  - Specification Adornment Template Environments (SAＴÉ)
  - Specification Adornment Templates (SAT)
  - Specification Adornment Template Signatures (SATＳ)

• Interpretation Mappings (IM)
  - Type Interpretation Mappings (TIM)
  - Variable Interpretation Mappings (VIM)
  - Operation Interpretation Mappings (OIM)
  - Abstract Template Interpretation Mappings (ATIM)
  - Concrete Template Interpretation Mappings (CTIM)
CHAPTER IV. THE ACTI DEFINITION

- Interpretation Mapping Templates (\(\mathcal{IMT}\))
  
  - Interpretation Mapping Template Signatures (\(\mathcal{IMTS}\))

- Basis Environments (\(\mathcal{BE}\))

By convention, the plural form of a space name is used when referring to the entire space (e.g., \(\text{Name}\)), while the singular form is used when referring to a particular element within that space (e.g., a \(\text{Name}\)). Further, upper case script abbreviations are often used (indicated above in parentheses after the complete name).

After the definitions for ACTI’s mathematical spaces are presented, this section continues with a discussion of operations on subsystems. Chapter V then uses a brief example to help illustrate the spaces.

4.3 Names (\(\mathcal{N}\))

The mathematical space \(\text{Name}, \mathcal{N}\), is the CPO of all identifiers available for labeling program entities, including types, variables, operations, adornment definitions, and all kinds of subsystems. A \(\text{Name}\) from \(\mathcal{N}\) should not be confused with the notion of a textual identifier in a programming language, as discussed below. The only assumptions made about this space are that it is a CPO, and that it contains a countably infinite number of unique elements (i.e., has cardinality \(\aleph_0\)). In applying the ACTI model to a specific programming language, any convenient mathematical space satisfying these assumptions can be used for \(\text{Name}\).

Throughout this chapter, we assume without loss of generality that all defined objects are given distinct names, and there is no notion of hiding by scope or overloading. Handling these cases is a natural extension, and is usually defined as part of the static (compilation phase) semantics of a particular programming language.

An Intuitive Explanation of Names

When writing a program, it is natural for one to give names to each declared entity. \(\mathcal{N}\) helps to flesh out the formal analog of this idea. In ACTI, the world is made up of subsystems, and every declared entity within every subsystem must be associated with some (unique) identifier. These identifiers are drawn from \(\mathcal{N}\), and are used when it is necessary to refer to specific objects within a particular subsystem. The requirement that \(\mathcal{N}\) be countably infinite ensures that there will always be enough unique identifiers to label every object in a subsystem (or program), no matter how large.
4.4 Modeling Domains (MD)

Since we are working in the realm of mathematics, our view of identifiers can be simplified in comparison with textual names in a programming language. In particular, since ACTI is only concerned with the dynamic (execution phase) semantics of programs, syntactic conveniences like allowing distinct objects in distinct scopes to carry the same name, or allowing the same name to denote distinct operations in different contexts, are not relevant. For the purposes of this definition, we can consider all of these issues to be handled effectively by an appropriate static semantics defining the relevant naming rules for a given programming language syntax. The rules for a particular programming language might even restrict the set of possible textual names to be finite (e.g., no names longer than 32 characters), but this usually only puts a limit on the number of textual names visible within a given scope, not on the number of distinct objects that may exist within a program.

Within the dynamic semantics, the concept of a Name becomes a unique, logical identifier for some program object, and this identifier need not be related to the actual textual name by which the programmer might refer to that object in source code. In the remainder of this chapter, whenever the term “identifier” is used, it will always refer to some element of $\mathcal{N}$.

The mathematical space Modeling Domains, $\mathcal{MD}$, is the CPO of all CPOs available for use in abstractly modeling data types contained in other ACTI structures. The only assumptions made about this space are that it is a CPO, and that it contains a countably infinite number of CPOs (i.e., has cardinality $\aleph_0$). Although $\mathcal{MD}$ itself must be denumerable, there is no restriction on the cardinality of the elements of $\mathcal{MD}$. In applying the ACTI model to a specific programming language, any convenient mathematical space satisfying these assumptions can be used for $\mathcal{MD}$.

A programming language designer can construct a useful candidate for $\mathcal{MD}$ by performing the following steps:

1. Select a basis set $B$ of CPOs from the category CPO. Ensure that $B$ is at most countably infinite, and ensure that it encompasses reasonable elements for modeling the critical data types at hand.

2. Form the closure of $B$, say $B'$, under finite application of the cartesian product operator ($\times$) and function formation operator ($\to$) on CPOs.

3. Form a flat CPO from the set $B'$, and use it as $\mathcal{MD}$.

The space resulting from this process will be denumerable.
An Intuitive Explanation of Modeling Domains

\(\mathcal{MD}\) is the realm of mathematical domains available for use as the abstract model of any data type. It is important that this space be a CPO itself in order to guarantee the existence of least fixed points, a guarantee needed if ACTI will be used as part of a language’s denotational semantics.

Fortunately, because \(\mathcal{N}\) is denumerable, no software construct (modeled in ACTI) can refer to more than a countably infinite number of distinct types. In practice, no program can refer to more than a finite number of distinct types. As a result, we can be sure that, given the correct \(\mathcal{MD}\), ACTI will be able to reasonably model all of the types relevant to any given program. The choice of the exact \(\mathcal{MD}\) to use, however, is left open in this definition. The question of how to best choose \(\mathcal{MD}\) for any specific programming language is beyond the scope of this work, but is not essential to understanding the ACTI model.

4.5 Environments or Concrete Instances (\(\mathcal{E}\))

The space of Environments is a cartesian product:

\[
\mathcal{E} = \mathcal{AI} \times \mathcal{SAE} \times \mathcal{TE} \times \mathcal{VE} \times \mathcal{OE} \times \mathcal{EI} \times \mathcal{AIE} \times \mathcal{CIE} \times \mathcal{IME} \times \mathcal{ATE} \times \mathcal{CTE} \times \mathcal{IMTE}
\]

Thus, an Environment (or, alternatively, a Concrete Instance) is a 12-tuple \(E = (\text{CTXT}, \mathcal{SAE}, \mathcal{TE}, \mathcal{VE}, \mathcal{OE}, \mathcal{EI}, \mathcal{AIE}, \mathcal{CIE}, \mathcal{IME}, \mathcal{ATE}, \mathcal{CTE}, \mathcal{IMTE})\), where:

CTXT is an Abstract Instance (Section 4.6, p. 78) declaring all external references appearing in this concrete instance.

SAE is a Specification Adornment Environment (Section 4.9, p. 85) containing all of the specification adornment definitions provided by \(E\).

TE is a Type Environment (Section 4.5.1, p. 69) containing all of the types provided by \(E\).

VE is a Variable Environment (Section 4.5.2, p. 70) containing all of the variables provided by \(E\).

OE is an Operation Environment (Section 4.5.3, p. 71) containing all of the operations provided by \(E\).

EI is an Environment Invariant (Section 4.5.4, p. 74) that always holds for \(E\).
4.5. ENVIRONMENTS OR CONCRETE INSTANCES ($\mathcal{E}$)

$\text{AIE}$ is an Abstract Instance Environment (Section 4.5.5, p. 74) containing all of the Abstract Instances provided by $E$.

$\text{CIE}$ is an Concrete Instance Environment (Section 4.5.6, p. 75) containing all of the Concrete Instances provided by (i.e., sub-environments contained in) $E$.

$\text{IME}$ is an Interpretation Mapping Environment (Section 4.5.7, p. 77) containing all of the Interpretation Mappings (Section 4.10, p. 90) provided by $E$.

$\text{ATE}$ is an Abstract Template Environment (Section 4.5.8, p. 77) containing all of the Abstract Templates provided by $E$.

$\text{CTE}$ is an Concrete Template Environment (Section 4.5.9, p. 77) containing all of the Concrete Templates provided by $E$.

$\text{IMTE}$ is an Interpretation Mapping Template Environment (Section 4.5.10, p. 77) containing all of the Interpretation Mapping Templates (Section 4.11, p. 97) provided by $E$.

In order to state the well-formedness criteria for an environment, it is necessary to define cycle-freedom. We can say an environment $E_1$ directly contains another environment $E_2$ if $E_2 \in \text{Ran } E_1$ CIE. We can further say that $E_1$ indirectly contains $E_2$ if any environment in $\text{Ran } E_1$ CIE contains $E_2$ (directly or indirectly). An environment $E$ is cycle-free if it does not contain itself, either directly or indirectly.

An Environment $E$ is well-formed if it is cycle-free, each of its twelve components is well-formed with respect to $E$, and $E.EI(E)$ is true—i.e., the invariant within the concrete instance holds.

For convenience, there are several notational shorthand forms we will adopt when discussing concrete instances. To talk about the set of all identifiers defined in a concrete instance $E$, we will use Dom $E$:

$$\text{Dom } E = \text{Dom } E.CTXT \cup \text{Dom } E.SAE \cup \text{Dom } E.TE \cup \text{Dom } E.VE \cup$$
$$\text{Dom } E.OE \cup \text{Dom } E.AIE \cup \text{Dom } E.CIE \cup \text{Dom } E.IME \cup$$
$$\text{Dom } E.CTE \cup \text{Dom } E.ATE \cup \text{Dom } E.IMTE$$

We also will use TypeNames $E$ to refer to the set of all identifiers from $\mathcal{N}$ that are mapped to a Type Model by $E$ or some concrete subsystem nested in $E$, excluding specification adornment definitions:

$$\text{TypeNames } E = \text{Dom } E.TE \cup \text{TypeNames } E.CIE \cup$$
$$\text{TypeNames } E.CTXT$$
CHAPTER IV. THE ACTI DEFINITION

The TypeNames operator only reflects the names of types that are provided by some concrete instance—i.e., that belong to an implementation. While $E.AIE$ also provides type definitions, no programmatic implementations of those types are provided.

TypeName is really an overloaded set of operators that are mutually defined: one for abstract instances, one for concrete instances, one for abstract instance environments, and one for concrete instance environments. We will consider VariableNames and OperationNames to be defined similarly. Also, we will define LocalTypeNames $E$ to be shorthand for TypeNames $E - TypeNames E.CTXT$, with LocalVariableNames and LocalOperationNames defined similarly.

We know that Dom $E.TE$ has no elements in common with TypeNames $E.AIE$, TypeNames $E.CIE$, or TypeNames $E.CTXT$, since all defined objects are assumed to have unique identifiers. Thus, we can also define a shorthand expression for the type model associated with a type name $t$ defined somewhere in $E$:

$$
\text{TypeModel} (t, E) = \begin{cases} 
E.TE(t) & \text{if } t \in \text{Dom } E.TE \\
\text{TypeModel} (t, E.CIE) & \text{if } t \in \text{TypeNames } E.CIE \\
\text{TypeModel} (t, E.CTXT) & \text{if } t \in \text{TypeNames } E.CTXT 
\end{cases} \quad (4.6)
$$

As with TypeNames, TypeModel is really an overloaded set of operators that are mutually defined: one for abstract instances, one for concrete instances, one for abstract instance environments, and one for concrete instance environments. The operators VariableModel and OperationModel are both defined the same way.

The signature of a concrete instance is an abstract instance. It can be generated directly from the concrete instance by the abstract operator described in Section 4.13.1, page 100.

An Intuitive Explanation of Concrete Instances

An Environment (or a Concrete Instance) is the run-time denotation of an executable subsystem (or module). Often, it is useful to think of a concrete instance as the implementation of a subsystem, such as a package body, a class implementation, or a realization.

The $CTX$ component of an environment $E$ is an explicit declaration of all entities outside $E$ that are referred to directly in the values of any objects within $E$. This is the “context” introduced in Section 3.4.3. Effectively, $CTX$ provides names and abstract specifications for the external objects needed to fully define $E$. Items within $E$ can only “see” the outside world through this explicit interface. In the Partial.Map.Facility example (Figure 11, p. 34), external RESOLVE facilities such as Standard_Character_String_Facility, Standard_Integer_Facility, and Standard_Boolean_Facility were all imported as context.
4.5. ENVIRONMENTS OR CONCRETE INSTANCES (E)

The SAE component of E collects all of the specification adornment definitions present in the subsystem (see Section 3.4.5). TE, VE, OE, AIE, CIE, ATE, and CTE represent collections of sub-objects within E of various kinds, and are all the direct formal analogs of the module substructures shown in Figure 26. The predicate EI is a formal representation of a subsystem-level invariant which must always hold for E.

Finally, the IME and IMTE components of a concrete instance E represent all of the Interpretation Mappings (Section 4.10, p. 90) and Interpretation Mapping Templates (Section 4.11, p. 97) provided by that subsystem, respectively. Briefly, interpretation mappings capture relationships between subsystems—they allow one to express how a subsystem (either abstract or concrete) can be “interpreted” as providing the behavior described by a particular abstract instance. By allowing interpretation mappings to be given names and exported by subsystems, they become first-class citizens of the programming world that can be identified and reused independently of any pair of subsystems to which they might be applicable.

4.5.1 Type Environments (TΣ)

A Type Environment is a name-to-value mapping from N to Type Models. A Type Model is a two-tuple \( <MD, CNSTR> \), where:

- \( MD \) is a Modeling Domain, which is simply a CPO from the space of Modeling Domains.
- \( CNSTR \) is a predicate over \( MD \), which acts as a “constraint” or further restriction on allowable values of the corresponding type.

A Type Model is well-formed if its \( CNSTR \) is applicable to all values in its \( MD \). A Type Environment \( te \) is well-formed if all of the type models in \( \text{Ran } te \) are well-formed.

As a notational convenience, we will use \( CNSTR(MD) \) to refer to the set of all values from \( MD \) for which the predicate \( CNSTR \) is true. This set represents the collection of all legitimate values for a program object interpreted as belonging to a type associated with a given Type Model.

Because a type environment contains only modeling information and not any runtime value information, a type environment is its own signature.

An Intuitive Explanation of Type Models

As indicate by the name, a Type Model is a direct mathematical expression of the designer’s conceptual model for a type. Effectively, any well-defined set of values (any CPO) can serve as the basis for such a conceptual model. The designer then
has the option of further constraining this set to indicate which values are allowable or meaningful for program objects of a given type. The ability to express such a constraint is not strictly needed for expressiveness, but does allow designers to provide simpler conceptual models.

In Partial.Map.Facility (Figure 7, p. 30), the exported type Partial.Map is the only item in its type environment. This type environment would map the name from $\mathcal{N}$ associated with the program identifier Partial.Map to a type model formalizing the contents of Figure 8 (p. 31).

### 4.5.2 Variable Environments ($\forall \mathcal{E}$)

A Variable Environment is a name-to-value mapping from $\mathcal{N}$ to Variable Models. A Variable Model is a two-tuple $(VSIG, VAL)$, where:

- $VSIG$ is a Variable Signature.
- $VAL$ is some value from the domain of the type named by $TY$.

A Variable Signature is simply a type name from $\mathcal{N}$.

A Variable Environment $ve$ is well-formed with respect to a concrete instance $E$ when all of the variable models in $\text{Ran} \ ve$ are well-formed with respect to $E$. A Variable Model $vm$ is well-formed with respect to a concrete instance $E$ if:

1. $vm.VSIG$ is well-formed with respect to $E$.

2. $vm.VAL \in tm.CNSTR(tm.MD)$, where $tm \equiv \text{TypeModel}(vm.VSIG, E)$. This means that the variable’s value comes from the mathematical domain associated with the model of its type and satisfies the constraint associated with that model.

A Variable Signature $VSIG$ is well-formed with respect to an (abstract or concrete) subsystem $S$ if $VSIG \in \text{TypeNames} \ S$, meaning the variable’s type is defined in $S$ or in some subsystem nested in $S$.

A Variable Environment $ve$ is well-formed with respect to a specification adornment environment $SAE$ and the subsystem $S$, either an abstract or concrete instance it is contained in, when all of the variable models in $\text{Ran} \ ve$ are well-formed with respect to $SAE$ and $S$. A Variable Model $vm$ is well-formed with respect to a specification adornment environment $SAE$ and a subsystem $S$ if:

1. $vm.VSIG$ is well-formed with respect to $SAE$.

2. $vm.VAL \in tm.CNSTR(tm.MD)$, where $tm \equiv \text{TypeModel}(vm.VSIG, SAE)$.
4.5. ENVIRONMENTS OR CONCRETE INSTANCES ($\mathcal{E}$)

A Variable Signature VSIG is well-formed with respect to a specification adornment environment SAE if VSIG $\in$ SATypeNames SAE.

An Intuitive Explanation of Variable Models

A Variable Model, just like a Type Model, is a mathematical expression of the designer’s conceptual model for a variable. It encompasses both the type and the value of the variable. The variable environment includes all of the variables provided by a subsystem (not including variables provided by nested subsystems).

4.5.3 Operation Environments ($\mathcal{OE}$)

An Operation Environment is a name-to-value mapping from $\mathcal{N}$ to Operation Meanings. An operation environment oe is well-formed with respect to a concrete instance $E$ when all of the operation meanings in Ran oe are well-formed with respect to $E$.

Operation Meanings ($\mathcal{OM}$)

An Operation Meaning is a three-tuple $\text{om} = \langle OMOD, PR, SF \rangle$, where:

OMOD is an Operation Model (Section 4.5.3, p. 72) describing the parameter profile and abstract behavior of the operation.

$PR$ is a “procedure relation,” a relation over the input arguments, the containing concrete instance before execution, the output values, and the containing concrete instance after execution $(\text{Args}(om.OMOD.PP) \times \mathcal{E} \times \text{Args}(om.OMOD.PP) \times \mathcal{E} \rightarrow \mathcal{B})$, which represents the actual relation computed by the operation.

$SF$ is a “status function,” an Assert Status-valued function over the input arguments and the containing concrete instance $(\text{Args}(om.OMOD.PP) \times \mathcal{E} \rightarrow \mathcal{AS})$, representing the assert status computed by the operation.

An Operation Meaning om is well-formed with respect to an environment $E$ when the following conditions are met:

1. $om.OMOD$ is well-formed with respect to $E$.

2. For all input arguments and concrete instance values satisfying $om.OMOD.DP$, all allowable output arguments related to these inputs by $om.PR$ must also satisfy $om.OMOD.EP$. In other words, the actual relation computed by the operation must be consistent with the postcondition in the operation signature.
3. For all input arguments and concrete instance values satisfying $om.OMOD.DP$, $om.SF$ must not produce the value $CF$.

An Intuitive Explanation of Operation Meanings

An *Operation Meaning* records the designer’s conceptual model of a program operation as an operation model, described below. In addition, an operation meaning describes in $PR$ the actual relation computed by a program operation for use in the run-time semantics. In simple terms, one can imagine $PR$ as just a relation from input values to output values. However, it is defined to be slightly richer, so that the entire environment $E$ containing a particular operation is also part of the space on which we define the relation $PR$. This allows ACTI to capture program operations that affect the visible state of a subsystem, by making the subsystem itself an explicit parameter to every program operation.

The $SF$ component of an operation meaning is similar to the $PR$ component, but instead of computing the output values of the corresponding program operation given the input values, it computes the “assert status” value that results from executing the operation. This is important for supporting assertive programming (Section 4.1.6, p. 61).

**Operation Models (OMOD)**

An *Operation Model* is a three-tuple $(PP, DP, EP)$, where:

$PP$ is a *Parameter Profile* describing the number and types of parameters that must be passed into an operation conforming to this signature.

$DP$ is a “domain predicate,” a boolean-valued function over the arguments to the operation and the containing concrete instance $(\text{Args}(PP) \times \mathcal{E} \rightarrow \mathcal{B})$ that represents the precondition for the operation.

$EP$ is an “effect predicate,” a boolean-valued function over the input arguments, the containing concrete instance before execution, the output values, and the containing concrete instance after execution $(\text{Args}(PP) \times \mathcal{E} \times \text{Args}(PP) \times \mathcal{E} \rightarrow \mathcal{B})$, that represents the postcondition for the operation.

An *Operation Model* $omod$ is well-formed with respect to a subsystem $S$ (either an abstract or concrete instance) when $omod.PP$ is well-formed with respect to $S$. 
4.5. ENVIRONMENTS OR CONCRETE INSTANCES (E)

An Intuitive Explanation of Operation Models

An Operation Model records the designer’s conceptual model of a program operation. This includes both the parameter profile of the operation, represented in PP, and an abstract description of the operation’s behavior. This behavioral description is recorded as separate precondition and postcondition assertions. The precondition is represented by DP, the “domain predicate” that is true for input values that meet the precondition it defines. The postcondition is represented by EP, the “effect predicate” that, given a particular set of input values, is true for output values that meet the postcondition it defines.

Note that both DP and EP are defined so that the (abstract or concrete) instance containing this particular operation signature is an explicit parameter accounted for in both the pre- and postconditions. This allows operation models to completely encompass the behavior of operations that depend on or affect visible state variables, either in the subsystem where the operation is defined, or in that subsystem’s explicitly declared context. Often, programmers loosely think of such state variables as “global variables.”

Parameter Profiles (PP)

A Parameter Profile is a string of identifiers from N. A string is simply a finite sequence of elements, which will be written as <t₁, t₂, ..., tₙ>. The empty string will be written as ε. For a thorough treatment of basic string theory, refer to an introductory book on discrete mathematics [22, pp. 251–264] or the theory of computation [39, pp. 29–33].

Since the space of Parameter Profiles is a CPO, it must have an ordering relation. For strings, we will use the relation ⊑ defined as follows:

\[
\forall \text{ strings } a, b : a \sqsubseteq b \text{ iff } a \text{ is a prefix of } b
\]  

(4.7)

Thus, ε is the least element (⊥) for a string CPO using this ordering relation.

A Parameter Profile pp is well-formed with respect to a subsystem S (either an abstract or concrete instance) when all of the identifiers appearing in pp are elements of TypeNames S.

For a parameter profile pp = <t₁, t₂, ..., tₙ> that is well-formed with respect to a subsystem S, let <MD₁, C₁> = TypeModel(t₁, S), the Type Model associated with t₁. An argument tuple conforming to pp is an element of the cartesian product:

\[
C_1(MD_1) \times C_2(MD_2) \times \ldots \times C_n(MD_n)
\]  

(4.8)

We will use Args(pp) as shorthand for this cartesian product, the set of all argument tuples conforming to the parameter profile pp.
An Intuitive Explanation of Parameter Profiles

A Parameter Profile describes the parameter signature of an operation. For a parameter profile \(<t_1, t_2, \ldots, t_n>\), \(t_i\) is the name of the type of the operations \(i\)th argument. This part of the ACTI model is common among virtually all modern imperative languages.

4.5.4 Environment Invariants (\(EI\))

An Environment Invariant is a boolean-valued function over an Environment (Section 4.5, p. 66). All environment invariants are well-formed.

4.5.5 Abstract Instance Environments (\(AI\))

An Abstract Instance Environment is a name-to-value mapping from \(N\) to Context-Bound Abstract Instances (\(CB\)). An abstract instance environment \(aie\) is well-formed with respect to a subsystem \(S\), either an environment or an abstract instance, if all of the context-bound abstract instances in \(\text{Ran} aie\) are well-formed with respect to \(S\).

A Context-Bound Abstract Instance is a two-tuple \(<AI, IM>\), where:

\(AI\) is an Abstract Instance (Section 4.6, p. 78).

\(IM\) is an Interpretation Mapping (Section 4.10, p. 90).

A context-bound abstract instance \(cbai\) is well-formed with respect to a subsystem \(S\) if the following conditions hold:

1. Both \(cbai.AI\) and \(cbai.IM\) are well-formed.

2. \(IM\) can be used to interpret \(S\) as fulfilling the context required by \(cbai.AI\). Formally, if \(S\) is an abstract instance, then \(cbai.IM.DP(S, cbai.AI.CTXT)\) must be true; if \(S\) is a concrete instance, then \(cbai.IM.DP(\text{abstract}(S), cbai.AI.CTXT)\) must be true.

The operators TypeNames, VariableNames, OperationNames, TypeModel, VariableModel, and OperationModel introduced on page 67 are all defined on abstract instance environments. The definition for TypeNames is:

\[
\text{TypeNames } aie = \bigcup_{cbai \in \text{Dom } aie} \text{TypeNames } cbai.AI \tag{4.9}
\]
4.5. ENVIRONMENTS OR CONCRETE INSTANCES (E)

The definitions of VariableNames and OperationNames are similar. The definition for TypeModel \((t, aie)\) is:

\[
\text{TypeModel} \ (t, aie) = \begin{cases} 
\text{TypeModel} \ (t, AI) & \text{if } \exists AI \in \text{Ran} \ aie \text{ where } t \in \text{TypeNames} \ AI \\
\bot & \text{otherwise}
\end{cases}
\] (4.10)

The definitions of VariableModel and OperationModel are similar.

An Intuitive Explanation of Abstract Instance Environments

An Abstract Instance Environment, say \(aie\), is just a collection of named (abstract) subsystems that are contained within some other (abstract or concrete) subsystem, say \(S\). Each Context-Bound Abstract Instance \(cbai\) within the abstract instance environment represents a single nested subsystem within \(S\). See Section 4.13.6, page 104, for more information about binding the context of a subsystem.

Since \(S\) “contains” every context-bound abstract instance in \(aie\), it really forms the environment, or “context,” in which those instances live. Thus, each \(cbai\) contains not only a smaller abstract instance \((cbai.AI)\), but also an explanation for how the outer containing environment \(S\) meets the context requirements of that inner instance. This binding between the containing environment and the explicit context interface of the abstract instance \(cbai.AI\) is expressed as an Interpretation Mapping, which explains how to “interpret” \(S\) as fulfilling the context interface of \(cbai.AI\). As a result, \(cbai.AI\) is no longer independent of the environment in which it lives—its context is bound to the containing subsystem \(S\). This is the meaning behind the name “context-bound.”

In order for a context-bound abstract instance \(cbai\) to be well-formed, both of its subcomponents must be well-formed. Further, the interpretation mapping contained in \(cbai\) must be applicable to both the containing subsystem \(S\) and the nested abstract instance \(cbai.AI\). This is why one cannot tell whether a context-bound abstract instance is well-formed in isolation; this property is always relative to the containing (abstract or concrete) subsystem \(S\). This is what is formally expressed in the requirement (2) for well-formed context-bound abstract instances.

4.5.6 Concrete Instance Environments (CIE)

A Concrete Instance Environment is a name-to-value mapping from \(N\) to Context-Bound Concrete Instances \((CBCI)\). A concrete instance environment \(cie\) is well-formed with respect to a concrete instance \(E\) if all of the context-bound concrete instances in \(\text{Ran} \ cie\) are well-formed with respect to \(E\).
A *Context-Bound Concrete Instance* is a two-tuple $\langle CI, IM \rangle$, where:

$CI$ is an *Concrete Instance* (Section 4.5, p. 66).

$IM$ is an *Interpretation Mapping* (Section 4.10, p. 90).

A context-bound concrete instance $cbei$ is well-formed with respect to a concrete instance $E$ if the following conditions hold:

1. Both $cbei.CI$ and $cbei.IM$ are well-formed.

2. $cbei.IM,DP(absttact(E), cbei.CI, CTXT)$ is true; that is, $IM$ can be used to interpret $E$ as fulfilling the context required by $cbei.CI$.

The operators TypeNames, VariableNames, and OperationNames introduced on page 67 are all defined on concrete instance environments. The definition for TypeNames is:

$$ \text{TypeNames } cie = \bigcup_{cbei \in \text{Dom } cie} \text{TypeNames } cbei.CI \quad (4.11) $$

The definitions of VariableNames and OperationNames are similar. The definition for TypeModel $(t, cie)$ is:

$$ \text{TypeModel } (t, cie) = \begin{cases} \text{TypeModel } (t, CI) & \text{if } \exists CI \in \text{Ran } cie \text{ where } \\ t \in \text{TypeNames } CI & t \in \text{TypeNames } CI \\ \bot & \text{otherwise} \end{cases} \quad (4.12) $$

The definitions of VariableModel and OperationModel are similar.

**An Intuitive Explanation of Concrete Instance Environments**

The structure and meaning of a *Concrete Instance Environment* exactly parallel that of an *Abstract Instance Environment*. There are only two critical differences: a *Concrete Instance Environment* contains concrete, rather than abstract, instances; and a *Concrete Instance Environment* can only appear in a concrete instance. Since abstract instances contain no implementation details, their CIE component is really just another *Abstract Instance Environment*.

A *Concrete Instance Environment*, say $cie$, is just a collection of named concrete instances that are contained within some other concrete instance, say $E$. Each *Context-Bound Concrete Instance* $cbei$ within the concrete instance environment represents a single nested subsystem within $E$. 

*CHAPTER IV. THE ACTI DEFINITION*
4.5. **ENVIRONMENTS OR CONCRETE INSTANCES (E)**

Just as with an *Abstract Instance Environment*, since $E$ “contains” every context-bound concrete instance in $\text{cie}$, it forms the environment, or “context,” in which those instances live. Thus, each $\text{cbbcI}$ contains not only a smaller concrete instance ($\text{cbbcI}.$), but also an explanation for how the outer containing environment $E$ meets the context requirements of that inner instance. This binding between the containing environment and the explicit context interface of the concrete instance $\text{cbbcI}.$ is expressed as an *Interpretation Mapping*, which explains how to “interpret” $E$ as fulfilling the context interface of $\text{cbbcI}.$ Now, $\text{cbbcI}.$ is no longer independent of the environment in which it lives; its context is *bound* to the containing concrete instance $E$.

The requirements for well-formed context-bound concrete instances are exactly analogous to those for context-bound abstract instances.

### 4.5.7 Interpretation Mapping Environments ($\text{IME}$)

An *Interpretation Mapping Environment* is a name-to-value mapping from $\mathcal{N}$ to *Interpretation Mappings* (Section 4.10, p. 90). An *Interpretation Mapping Environment* ime is well-formed when all of the interpretation mappings in $\text{Ran ime}$ are well-formed.

### 4.5.8 Abstract Template Environments ($\text{ATE}$)

An *Abstract Template Environment* is a name-to-value mapping from $\mathcal{N}$ to *Abstract Templates* (Section 4.8, p. 83). An *Abstract Template Environment* ate is well-formed when all of the abstract templates in $\text{Ran ate}$ are well-formed.

### 4.5.9 Concrete Template Environments ($\text{CTE}$)

A *Concrete Template Environment* is a name-to-value mapping from $\mathcal{N}$ to *Concrete Templates* (Section 4.7, p. 82). A *Concrete Template Environment* cte is well-formed when all of the concrete templates in $\text{Ran cte}$ are well-formed.

### 4.5.10 Interpretation Mapping Template Environments ($\text{IMTE}$)

An *Interpretation Mapping Template Environment* is a name-to-value mapping from $\mathcal{N}$ to *Interpretation Mapping Templates* (Section 4.11, p. 97). An *Interpretation Mapping Template Environment* imte is well-formed when all of the interpretation mapping templates in $\text{Ran imte}$ are well-formed.
4.6 Abstract Instances (AI)

The space of Abstract Instances is a cartesian product:

$$
\text{AI} = \text{AI} \times \text{SAE} \times \text{TE} \times \text{VES} \times \text{OES} \times \text{EI} \times \text{AIE} \times \text{IME} \times \\
\text{ATES} \times \text{CTES} \times \text{IMTE}
$$

(4.13)

Thus, an Abstract Instance is a 12-tuple $\text{AI} = \langle \text{CTX}, \text{SAE}, \text{TE}, \text{VES}, \text{OES}, \text{EI}, \\
\text{AIE}, \text{CIE}, \text{IME}, \text{ATES}, \text{CTES}, \text{IMTE} \rangle$, where:

- $\text{CTX}$ is an Abstract Instance declaring all external references appearing in this abstract instance. Here, it is worth pointing out that since the space of Abstract Instances is a CPO, this component may be $\bot$, indicating there are no external dependencies.

- $\text{SAE}$ is a Specification Adornment Environment (Section 4.9, p. 85) containing all of the math definitions provided by AI.

- $\text{TE}$ is a Type Environment (Section 4.5.1, p. 69) containing all of the types provided by AI.

- $\text{VES}$ is a Variable Environment Signature (Section 4.6.1, p. 80) containing all of the variables provided by AI.

- $\text{OES}$ is an Operation Environment Signature (Section 4.6.2, p. 81) containing all of the operations provided by AI.

- $\text{EI}$ is an Environment Invariant (Section 4.5.4, p. 74) that always holds for any Environment that can be interpreted as satisfying AI.

- $\text{AIE}$ is an Abstract Instance Environment (Section 4.5.5, p. 74) containing all of the Abstract Instances provided by AI.

- $\text{CIE}$ is another Abstract Instance Environment containing Abstract Instances describing the sub-environments contained in AI.

- $\text{IME}$ is an Interpretation Mapping Environment (Section 4.5.7, p. 77) containing all of the Interpretation Mappings provided by AI.

- $\text{ATES}$ is an Abstract Template Environment Signature (Section 4.6.3, p. 81) containing all of the Abstract Template Signatures provided by AI.
4.6. ABSTRACT INSTANCES (AI)

CTES is a Concrete Template Environment Signature containing all of the Concrete Template Signatures provided by AI.

IMTE is an Interpretation Mapping Template Environment (Section 4.5.10, p. 77) containing all of the Interpretation Mapping Templates (Section 4.11, p. 97) provided by AI.

We can say an abstract instance \( AI_1 \) directly contains another abstract instance \( AI_2 \) if \( AI_1.CTXT = AI_2 \), if \( AI_2 \in \text{Ran } AI_1.AIE \), or if \( AI_2 \in \text{Ran } AI_1.CIE \). We can further say that \( AI_1 \) indirectly contains \( AI_2 \) if any environment in the set \( \{AI_1.CTXT\} \cup \text{Ran } AI_1.AIE \cup \text{Ran } AI_1.CIE \) contains \( AI_2 \) (directly or indirectly). An abstract instance \( AI \) is cycle-free if it does not contain itself, either directly or indirectly.

An Abstract Instance \( AI \) is well-formed if it is cycle-free, and each of its twelve components is well-formed with respect to \( AI \).

Just as with concrete instances, for notational convenience we will use \( \text{Dom } AI \) as a shorthand for the union of the meaningful domains of the ten name-to-value mappings contained in \( AI \). TypeNames, introduced on page 67, will be defined in the obvious way:

\[
\text{TypeNames } AI = \text{Dom } AI.TE \cup \text{TypeNames } AI.CIE \cup \\
\text{TypeNames } AI.CTXT
\]  

(4.14)

VariableNames and OperationNames also have parallel definitions. LocalTypeNames, LocalVariableNames, and LocalOperationNames are also applicable to abstract instances, with the same definitions as for concrete instances. The definition of TypeModel \( (t, AI) \) is:

\[
\text{TypeModel } (t, AI) = \begin{cases} 
AI.TE(t) & \text{if } t \in \text{Dom } AI.TE \\
\text{TypeModel } (t, AI.CIE) & \text{if } t \in \text{TypeNames } AI.CIE \\
\text{TypeModel } (t, AI.CTXT) & \text{if } t \in \text{TypeNames } AI.CTXT
\end{cases}
\]  

(4.15)

VariableModel and OperationModel also have parallel definitions.

An Intuitive Explanation of Abstract Instances

An Abstract Instance is the run-time denotation of a subsystem (or module) specification. Its structure exactly mirrors that of a Concrete Instance, but while a concrete instance contains the designer’s conceptual models for objects and corresponding runtime values, an abstract instance contains only the designer’s conceptual models for the objects it contains. Usually, such a conceptual model is called a “signature,” since it defines the conceptual shape of a program object without defining its actual value.
In Section 3.4, both Partial\_Map\_Facility and Communal\_PMap\_Facility were specifications. In ACTI, the natural denotation for such a programming construct is an abstract instance.

Just as in a concrete instance, the CTXT component of an abstract instance AI is an explicit declaration of all entities outside AI that are referred to directly in the signatures of any objects within AI. Effectively, CTXT provides names and abstract specifications for the external objects needed to fully define AI. Items within AI can only “see” the outside world through this explicit interface.

The SAE component of AI collects all of the specification adornment definitions present in the abstract instance (see Section 3.4.5). TE, VES, OES, AIE, CIE, IME, ATES, CTES, and IMTE represent collections of sub-objects within AI of various kinds, and are all the direct counterparts of the similarly named subcomponents of a concrete instance. Now, however, since AI does not contain any run-time values for variables, operations, or nested subsystems, several of these components hold only signatures. Note that despite its name, CIE holds an Abstract Instance Environment. This is because abstract instances play the role of “signatures” for concrete instances.

Finally, the predicate EI is a formal representation of a subsystem-level invariant which must always hold for any concrete instance that satisfies the abstract description AI. In no sense can EI “hold” for AI, since EI is a predicate over the space of Concrete Instances. Further, since AI contains only abstract models of program entities, not their actual run-time values, it would not make sense to ask whether or not it satisfies such an invariant.

4.6.1 Variable Environment Signatures (VES)

A Variable Environment Signature is a function from \( \mathcal{N} \) to \( \mathcal{N} \). A Variable Environment Signature ves is well-formed with respect to an abstract instance AI if every identifier in Ran ves is in TypeNames AI.

An Intuitive Explanation of Variable Environment Signatures

A Variable Environment Signature is the “signature” of a Variable Environment. It gives the type associated with each variable name defined in the environment, but leaves out the value information. One can think of it as performing the same function as a plain Variable Environment, but instead of mapping each variable identifier to a complete Variable Model (Section 4.5.2, p. 70), it maps each variable identifier to just the projection of the TY component of its Variable Model.
4.6. ABSTRACT INSTANCES (AI)

4.6.2 Operation Environment Signatures (OES)

An Operation Environment Signature is a function from $\mathcal{N}$ to Operation Models. An Operation Environment Signature $oes$ is well-formed with respect to an abstract instance $AI$ if every operation signature in $\text{Ran } oes$ is well-formed with respect to $AI$.

An Intuitive Explanation of Operation Environment Signatures

An Operation Environment Signature is the “signature” of an Operation Environment. It gives the pre- and postcondition information associated with each operation name defined in the abstract instance, but leaves out information about the actual relation and status function computed by the operation.

In Partial Map Facility (Figure 7, p. 30), the exported operations Define, Undefine, Undefine_Any_One, Is_Defined, Size, Initialize, Finalize, and Swap make up the meaningful domain of its operation environment signature. For example, this operation environment signature would map the name from $\mathcal{N}$ associated with the program identifier Define to an operation model formalizing the contents of Figure 9 (p. 31).

4.6.3 Abstract Template Environment Signatures (ATES)

An Abstract Template Environment Signature is a function from $\mathcal{N}$ to Abstract Template Signatures (Section 4.8.1, p. 84). An Abstract Template Environment Signature $ates$ is well-formed with respect to an abstract instance $AI$ if every abstract template signature in $\text{Ran } ates$ is well-formed with respect to $AI$.

An Intuitive Explanation of Abstract Template Environment Signatures

An Abstract Template Environment Signature is the “signature” of an Abstract Template Environment. It gives the pre- and postcondition information associated with each abstract template name defined in the abstract instance, but leaves out information about the actual instance-to-instance function and status function computed by the template when it is applied.

4.6.4 Concrete Template Environment Signatures (CTES)

A Concrete Template Environment Signature is a function from $\mathcal{N}$ to Concrete Template Signatures (Section 4.7.1, p. 83). A Concrete Template Environment Signature $ctes$ is well-formed with respect to an abstract instance $AI$ if every concrete template signature in $\text{Ran } ctes$ is well-formed with respect to $AI$. 
An Intuitive Explanation of Concrete Template Environment Signatures

A Concrete Template Environment Signature is the “signature” of an Concrete Template Environment. It gives the pre- and postcondition information associated with each concrete template name defined in the abstract instance, but leaves out information about the actual instance-to-instance function and status function computed by the template when it is applied.

4.7 Concrete Templates (CT)

A Concrete Template is a three-tuple \( \langle CTS, CICIF, SF \rangle \), where:

- \( CTS \) is a Concrete Template Signature, defined below.
- \( CICIF \) is the “concrete instance-to-concrete instance function” \( (\mathcal{E} \rightarrow \mathcal{E}) \) actually computed by instantiating the template.
- \( SF \) is a “status function,” an Assert Status-valued function over the input concrete instance \( (\mathcal{E} \rightarrow \mathcal{AS}) \) representing the assert status actually computed by instantiating the template.

A Concrete Template \( ct \) is well-formed when the following two conditions are met:

1. For all input concrete instance values \( E \) satisfying \( ct.CTS.CIDP \), the effect predicate \( ct.CTS.CIEP(E, ct.CICIF(E)) \) must be true. In other words, the actual instance-to-instance function computed by the template must be consistent with the postcondition described by \( ct.CTS.CIEP \).

2. For all input concrete instance values \( E \) satisfying \( ct.CTS.CIDP \), \( ct.SF(E) \) must not produce the value \( CF \).

An Intuitive Explanation of Concrete Templates

A Concrete Template is a generic subsystem realization, which should be thought of as a function from concrete instances to concrete instances. The characteristics of this function are captured in the three-tuple described above. The structure of a template mirrors that of an Operation Meaning (Section 4.5.3, p. 71): \( CTS \) is the signature, which describes both the precondition for applying the template, and the (relational) postcondition describing the effects of applying the template; and \( CICIF \) and \( SF \) describe the actual computation carried out when the template is applied. For more information on applying template to generate new instances, see Section 4.13.7, page 106.
Note that a conceptual template only has a single concrete instance argument. This is all that is needed, since a given concrete instance can contain many nested concrete instances as subcomponents.

Further, notice that the acceptable concrete instances to which this template may be applied are defined by the precondition \( CIDP \), not simply by conformance with some particular abstract instance. As a result, it is possible to construct templates in ACTI that can be applied to several structurally different concrete instances, generating an appropriate new instance for each one. The most interesting application of this feature is in generalizing the notion of generics to encompass a variable number of parameters.

With the appropriate precondition, an ACTI concrete template could be constructed that could be applied to a concrete instance containing one, two, or \( n \) nested subsystems. For each of these possible inputs, this template could generate an appropriate output instance. Similarly, generics that build on a variable number of type parameters or operation parameters can also be defined.

### 4.7.1 Concrete Template Signatures (\( CTS \))

A Concrete Template Signature is a two-tuple \( \langle CIDP, CIEP \rangle \), where:

- \( CIDP \) is a “concrete instance domain predicate,” a boolean-valued function over the input concrete instance \( \langle E \rightarrow B \rangle \) that represents the precondition for the template.

- \( CIEP \) is a “concrete instance effect predicate,” a boolean-valued function over the input and output concrete instances \( \langle E \times E \rightarrow B \rangle \) that represents the postcondition of instantiating the template.

All concrete template signatures are well-formed.

### 4.8 Abstract Templates (\( \mathcal{AT} \))

An Abstract Template is a three-tuple \( \langle ATS, AIAIF, SF \rangle \), where:

- \( ATS \) is an Abstract Template Signature, defined below.

- \( AIAIF \) is the “abstract instance-to-abstract instance function” \( \langle \mathcal{AT} \rightarrow \mathcal{AT} \rangle \) actually computed by instantiating the template.

- \( SF \) is a “status function,” an Assert Status-valued function over the input abstract instance \( \langle \mathcal{AT} \rightarrow \mathcal{AS} \rangle \) representing the assert status actually computed by instantiating the template.
CHAPTER IV. THE ACTI DEFINITION

An Abstract Template at is well-formed when the following two conditions are met:

1. For all input abstract instance values AI satisfying at.ATS.AIDP, the effect predicate at.ATS.AIEP(AI, at.AIIF(AI)) must be true. In other words, the actual instance-to-instance function computed by the template must be consistent with the postcondition described by at.ATS.AIEP.

2. For all input abstract instance values AI satisfying at.ATS.AIDP, at.SF(AI) must not produce the value CF.

An Intuitive Explanation of Abstract Templates

An Abstract Template is a generic module specification, which should be thought of as a function from abstract instances to abstract instances. Its structure exactly parallels that of a Concrete Template. Just like a concrete template, the nested structure of the single abstract instance parameter allows an abstract template effectively to be parameterized by many instances. Because the requirements on the single parameter are expressed through a predicate, that parameter may contain variable numbers of nested instances, allowing variable-length generic parameter lists.

In Section 3.4.6, Communal_PMap_Template was a parameterized component specification. In ACTI, the natural denotation for such a programming construct is an abstract template.

4.8.1 Abstract Template Signatures (ATS)

An Abstract Template Signature is a two-tuple \langle AIDP, AIEP \rangle, where:

AIDP is an “abstract instance domain predicate,” a boolean-valued function over the input abstract instance ($\mathcal{AI} \to \mathcal{B}$) that represents the precondition for the template.

AIEP is an “abstract instance effect predicate,” a boolean-valued function over the input and output abstract instances ($\mathcal{AI} \times \mathcal{AI} \to \mathcal{B}$) that represents the postcondition of instantiating the template.

All abstract template signatures are well-formed.
4.9 Specification Adornment Environments (SAE)

A Specification Adornment Environment (or Specification Adornment Instance) is a five-tuple $SAE = \langle SATY, SAVE, SAIE, SATE \rangle$, where:

$SATY$ is a Type Environment (Section 4.5.1, p. 69) containing all of the types provided by $SAE$.

$SAVE$ is a Specification Adornment Variable Environment (Section 4.9.1, p. 87) containing all of the variables provided by $SAE$.

$SAEI$ is a Specification Adornment Environment Invariant (Section 4.9.2, p. 88) that always holds for $SAE$.

$SAIE$ is a Specification Adornment Instance Environment (Section 4.9.3, p. 88) containing all of the nested Specification Adornment Instances provided by $SAE$.

$SATE$ is a Specification Adornment Template Environment (Section 4.9.4, p. 88) containing all of the Specification Adornment Templates (or theories) provided by $SAE$.

A Specification Adornment Environment $SAE$ is well-formed with respect to a subsystem $S$, either an abstract or concrete instance, if each of its six components is well-formed with respect to $S$ and $SAE.SAEI(SAE)$ is true (i.e., the invariant within $SAE$ holds).

Just as we defined the operator $\text{TypeNames}$ for abstract and concrete instances, we will use $\text{SATypeNames} \ SAE$ to refer to the set of all identifiers from $\mathcal{N}$ that are mapped to a Type Model by $SAE$ or some specification adornment environment nested within $SAE$:

$$\text{SATypeNames} \ SAE = \text{Dom} \ SAE.SATY \cup \text{SATypeNames} \ SAE.SAIE \ (4.16)$$

$\text{SATypeNames}$ is an overloaded set of operators that are mutually defined: one for specification adornment instances, one for specification adornment instance environments, one for abstract instances, and one for concrete instances. The definition of $\text{SATypeNames}$ when applied to specification adornment instance environments is given in Section 4.9.3, page 88. For a subsystem $S$ that is either an abstract instance or a concrete instance:

$$\text{SATypeNames} \ S = \text{SATypeNames} \ S.SAE \cup \text{SATypeNames} \ S.CTXT \quad (4.17)$$

We will consider $\text{SAVariableNames}$ to be defined similarly.
In addition, we will use SATypeModel \((t, SAE)\) as shorthand for the type model associated with a type name \(t\) defined somewhere in \(SAE\):

\[
\text{SATypeModel}(t, SAE) = \begin{cases} 
SAE.SATY(t) & \text{if } t \in \text{Dom } SAE.SATY \\
\text{SATypeModel}(t, SAE.SAIE) & \text{if } t \in \text{SATypeNames}
\end{cases}
\]

The notation SATypeModel \((t, S)\), where \(S\) is an abstract or concrete instance, will be used with the following meaning:

\[
\text{SATypeModel}(t, S) = \begin{cases} 
\text{SATypeModel}(t, S.SAE) & \text{if } t \in \text{SATypeNames } S.SAE \\
\text{SATypeModel}(t, S.CTXT) & \text{if } t \in \text{SATypeNames } S.CTXT
\end{cases}
\]

The operation SAVariableModel on specification adornment environments, abstract instances, and concrete instances is defined in the same way.

**An Intuitive Explanation of Specification Adornment Environments**

“Specification adornments” are auxiliary definitions that a subsystem designer creates to make it easier to describe the conceptual models of program objects. Basically, these definitions are purely mathematical constructs that can be used as building blocks in describing any *Type Model, Operation Model, Environment Invariant*, or other subsystem structure. These definitions are called specification adornments because they do not describe any of the actual behavior of a subsystem, but are merely “adornments” that make it easier to write down a specification of that behavior.

A *Specification Adornment Environment* is a module-like structure for organizing a collection of specification adornment definitions. These definitions may include type models, variable models whose values are constrained by an invariant, or nested specification adornment environments or templates. Note that mathematical functions or other structures useful in defining predicates, invariants, preconditions, and so on, can easily be defined as particular variables within the specification adornment environment simply by ensuring that the variable has a type with the appropriate mathematical domain in its type model.

The second version of *Partial_Map_Facility* presented in Section 3.4.5 defines the math type *PARTIAL_FUNCTION* and the math operation *DEFINED_IN* in its local context (Figure 14, p. 38). In ACTI, these definitions would make up the specification adornment environment in the abstract instance denoted by *Partial_Map_Facility* (Figure 17, p. 41).
4.9. SPECIFICATION ADORNMENT ENVIRONMENTS (SAE)

4.9.1 Specification Adornment Variable Environments (SAE)

A Specification Adornment Variable Environment is simply a Variable Environment. The criteria for well-formedness are different for specification adornment variable environments, however.

A Specification Adornment Variable Environment save is well-formed with respect to a specification adornment environment SAE and the subsystem S, either an abstract or concrete instance, when all of the variable models in Ran save are well-formed with respect to SAE and S. A Variable Model vm is well-formed with respect to a specification adornment environment SAE and a subsystem S if:

1. vm.VSIG is well-formed with respect to SAE and S.

2. vm.VAL ∈ tm.CNSTR(tm.MD), where tm ≡ SATypeModel (vm.VSIG, S).

A Variable Signature VSIG is well-formed with respect to SAE and S if VSIG ∈ SATypeNames S.

An Intuitive Explanation of Specification Adornment Variable Environments

A specification adornment variable environment save associates names with variable models, just as a variable environment does. The only difference is that the variables represented by associations in save are present for specification purposes only. As a result, the types of these variables are limited to specification adornment types by the rules for well-formedness given above.

Also, it is interesting to note that specification adornment variable environments can also be used to model specification-level operations as well as specification-level variables. A mathematical operation being used for specification purposes can be represented as a specification adornment variable whose type is the appropriate space of functions. As a result, there is no need for a “specification adornment operation environment.”

In the Communal_PMap_Facility introduced in Section 3.4.5, total_size is an example of a specification adornment variable. The specification adornment variable environment in the abstract instance denoted by Communal_PMap_Facility would map the name from \( \mathcal{N} \) associated with the program identifier total_size to an appropriate variable model (see Figure 20, p. 20). In addition, both of the math operations DEFINED_IN and MAX_TOTAL_LENGTH declared in Communal_PMap_Facility’s local context would also be present in its specification adornment variable environment.
4.9.2 Specification Adornment Environment Invariants (SAEI)

A Specification Adornment Environment Invariant is a boolean-valued function over the space of Specification Adornment Environments (Section 4.9, p. 85). All specification adornment environment invariants are well-formed.

4.9.3 Specification Adornment Instance Environments (SAIE)

A Specification Adornment Instance Environment is a name-to-value mapping from \( \mathcal{N} \) to Specification Adornment Environments. A Specification Adornment Instance Environment \( saie \) is well-formed when all of the specification adornment instances in \( \text{Ran} \ saie \) are well-formed.

The operators SATypeNames, SAVariableNames, SATypeModel, and SAVariableModel, introduced on page 85 are all defined on specification adornment instance environments. The definition for SATypeNames is:

\[
\text{SATypeNames}\ saie = \bigcup_{SAE \in \text{Dom} \ saie} \text{SATypeNames}\ SAE
\]  

(4.20)

The definition of SAVariableNames is similar. The definition for SATypeModel \((t, saie)\) is:

\[
\text{SATypeModel}\ (t, saie) = \begin{cases} 
\text{SATypeModel}\ (t, SAE) & \text{if } \exists SAE \in \text{Ran} \ saie \text{ where } t \in \text{SATypeNames}\ SAE \\
\bot & \text{otherwise}
\end{cases}
\]  

(4.21)

The definition of SAVariableModel is similar.

4.9.4 Specification Adornment Template Environments (SATTE)

A Specification Adornment Template Environment is a name-to-value mapping from \( \mathcal{N} \) to Specification Adornment Templates. A Specification Adornment Template Environment \( sate \) is well-formed when all of the specification adornment templates in \( \text{Ran} \ sate \) are well-formed.
4.9. SPECIFICATION ADORNMENT ENVIRONMENTS (SAE)

4.9.5 Specification Adornment Templates (SAT)

A Specification Adornment Template is a three-tuple \( \langle ATS, AIAIF, SF \rangle \), where:

- \( SATS \) is a Specification Adornment Template Signature (Section 4.9.6, p. 90).

- \( SAISAIF \) is the “specification adornment instance-to-specification adornment instance function” \( (SAE \rightarrow SAE) \) actually computed by instantiating the template.

- \( SF \) is a “status function,” an Assert Status-valued function over the input specification adornment instance \( (SAE \rightarrow AS) \) representing the assert status actually computed by instantiating the template.

A Specification Adornment Template \( sat \) is well-formed when the following two conditions are met:

1. For all input specification adornment environment values \( SAE \) satisfying the domain predicate \( sat.SATS.SAIDP, sat.SATS.SAIEP(SAE, sat.SAISAIF(SAE)) \) must be true. In other words, the actual instance-to-instance function computed by the template must be consistent with the postcondition described by \( sat.SATS.SAIEP \).

2. For all input specification adornment instance values \( SAE \) satisfying the domain predicate \( sat.SATS.SAIDP, sat.SF(SAE) \) must not produce the value \( CF \).

An Intuitive Explanation of Specification Adornment Templates

A Specification Adornment Template is the natural analog of a concrete (or abstract) template in the world of specification adornment definitions. A Specification Adornment Template is a generic specification adornment module, which should be thought of as a function from specification adornment instances to specification adornment instances. Its structure exactly parallels that of a Concrete Template or an Abstract Template. Just like those two kinds of templates, the nested structure of the single specification adornment instance parameter allows a specification adornment template effectively to be parameterized by many instances. Because the requirements on the single parameter are expressed through a predicate, that parameter may contain a variable number of nested instances, allowing variable-length generic parameter lists.
CHAPTER IV. THE ACTI DEFINITION

4.9.6 Specification Adornment Template Signatures (SATŚ)

A Specification Adornment Template Signature is a two-tuple \( \langle SAIDP, SAIEP \rangle \), where:

\( SAIDP \) is a “specification adornment instance domain predicate,” a boolean-valued function over the input specification adornment instance \( \langle SA \rangle \rightarrow \mathcal{B} \) that represents the precondition for the template.

\( SAIEP \) is a “specification adornment instance effect predicate,” a boolean-valued function over the input and output specification adornment instances \( \langle SA \rangle \times \langle SA \rangle \rightarrow \mathcal{B} \) that represents the postcondition of instantiating the template.

All specification adornment template signatures are well-formed.

4.10 Interpretation Mappings (IM)

An Interpretation Mapping between abstract instances describes how one can be interpreted as providing the behavior described by the other. For ease of communication, we will speak of an interpretation mapping as being from a more specific abstract instance, say \( AI_S \), to a more general abstract instance, say \( AI_G \), describing how \( AI_S \) can be “interpreted as” providing behavior wholly consistent with \( AI_G \). For an interpretation mapping \( IM \), we can write this symbolically as:

\[
IM \models AI_S \text{ is-interpretable-as } AI_G
\]

Such an interpretation mapping is a nine-tuple \( IM = \langle DP, CTXTmap, TYmap, VARmap, OPmap, MCORR, MCONV, ATEmap, CTEmap \rangle \), where:

\( DP \) is a “domain predicate,” a boolean-valued function over both of the abstract instances \( \langle AI \times AI \rightarrow \mathcal{B} \rangle \) which indicates the applicability of \( IM \). For the remainder of this section, we assume without loss of generality that \( AI_S \) and \( AI_G \) are the two abstract instances under consideration and that \( DP(AI_S, AI_G) = \text{true} \).

\( CTXTmap \) is an Interpretation Mapping.

\( TYmap \) is a Type Interpretation Mapping (Section 4.10.1, p. 92).

\( VARmap \) is a Variable Interpretation Mapping (Section 4.10.2, p. 93).

\( OPmap \) is an Operation Interpretation Mapping (Section 4.10.3, p. 94).
4.10. **INTERPRETATION MAPPINGS (IM)**

**MCORR** represents a “module-level correspondence,” a relation over two abstract instances \( \mathcal{AI} \times \mathcal{AI} \rightarrow \mathcal{B} \).

**MCONV** represents a “module-level convention,” a predicate over the more specific abstract instance \( \mathcal{AI} \rightarrow \mathcal{B} \).

**ATmap** is an Abstract Template Interpretation Mapping (Section 4.10.4, p. 95).

**CTmap** is a Concrete Template Interpretation Mapping (Section 4.10.5, p. 96).

An **Interpretation Mapping** \( IM \) is well-formed if the following conditions are met for every pair of abstract instances \( AI_s, AI_g \) that satisfy \( IM.DP \):

1. All of the components of \( IM \) are well-formed with respect to \( AI_s, AI_g, \) and \( IM \).

2. \( IM.CTXT \models AI_s.CTXT \text{-is-interpretable-as } AI_g.CTXT \) (see the intuitive explanation for an explanation of this notation).

3. \( MCONV \) holds for \( AI_s \). Furthermore, every operation in \( AI_s \) that is “interpreted as” some operation in \( AI_g \) must respect \( MCONV \). More formally, the following assertion must be true:

\[
\forall O \in \text{Ran } \text{OP} \map \cap \text{LocalOperationNames } AI_s : ( \exists \text{OMOD } \in \text{OMOD} : ( \\
\text{OMOD} = \text{OperationModel}(O, AI_s) \text{ and } \\
\forall \text{args, args}' \in \text{Args } \text{(OMOD.PP)} : ( \\
\forall \text{AI}_s' \in \text{AI} : ( \\
(\text{OMOD.DP(args, AI_s)} \text{ and } \\
\text{OMOD.EP(args, AI_s, args', AI_s'}) \\
\Rightarrow MCONV(AI_s')))))) \tag{4.23}
\]

**An Intuitive Explanation of Interpretation Mappings**

An **Interpretation Mapping** is a correspondence between two abstract instances that shows how one of them can be “interpreted as” satisfying the abstract behavior described by the other. This relationship is illustrated in Figure 27, and is written symbolically as:

\[
IM \models A \text{-is-interpretable-as } B \tag{4.24}
\]

Thus, an interpretation mapping shows how one abstract instance (\( A \)) can be used in place of another (\( B \)), because the externally visible behavior the first describes is perfectly consistent with the behavior described by the second.
Since an interpretation mapping $IM$ shows how to interpret an $A$ as a $B$, it is natural to visualize the interpretation mapping as being “directed” from $A$ to $B$. This interpretation can be considered “lossy,” since there may be many features provided by $A$ that are not needed in order to provide the (usually simpler, more abstract) behavior described by $B$.

Note, however, that the name-to-value mappings defined within $IM$ go in the opposite direction. These maps associate each name in $B$ to the corresponding entity in $A$ that is “interpreted as” it. This is a mathematical convenience that makes it easier to ensure that every entity is $B$ has some analog in $A$. It also helps provide a place to hang additional information about properties of the interpretation (i.e., there is more to an interpretation map than just a one-to-one association of names between $A$ and $B$).

For practical purposes, an interpretation mapping can be used to define a “subtype”-like relationship between subsystem specifications. Knowing that $IM$ correctly explains how an $A$ can be interpreted as a $B$ means knowing that any place a subsystem satisfying specification $B$ is required, one satisfying $A$ will do. Thus, $B$ is “more general” than $A$, or “more weakly specified” than $A$. Similarly, the class of concrete instances that provide the behavior described in $A$ also necessarily provide the behavior described in $B$, so one can consider $A$ “a (subsystem) subtype of” $B$.

4.10.1 Type Interpretation Mappings ($TIM$)

A Type Interpretation Mapping $tim$ is a name-to-value mapping from $N$ (type names in LocalTypeNames $AI_C$) to three-tuples $\langle N, R, C \rangle$, where:

$N$ is a name, presumably from LocalTypeNames $AI_S$ (e.g., a type defined in the more specific abstract instance).
4.10. INTERPRETATION MAPPINGS (IM)

\( R \) represents the “correspondence” between the two types. It is a relation over the math domains associated with the two type names, along with both \( AI_G \) and \( AI_S \) (for an identifier \( t \in \text{Ran} \, \text{tim} \), \( R \) is an element of \( \text{TypeModel} (t, AI_G). MD \times AT \times \text{TypeModel} (N, AI_S). MDAT \rightarrow B \).

\( C \) represents the “convention” over the implementation type. It is a predicate over the math domain associated with \( N \) (e.g., \( \text{TypeModel} (N, AI_S). MD \)).

A Type Interpretation Mapping \( \text{tim} \) is well-formed with respect to an abstract instance \( AI_S \), and abstract instance \( AI_G \), and an interpretation mapping \( IM \) if the following conditions hold:

1. The type interpretation mapping \( \text{tim} \) must completely cover the set of LocalTypeNames \( AI_G \), and must use a non-⊥ element from LocalTypeNames \( AI_S \) as the interpretation for each entity in LocalTypeNames \( AI_G \). More formally, for every such \( AI_S \) and \( AI_G \), \( \text{tim} \) must satisfy two conditions:

\[
\begin{align*}
\text{LocalTypeNames } AI_G & \subseteq \text{Dom } \text{tim} \\
\text{LocalTypeNames } AI_S & \supseteq \bigcup_{\langle N, R, C \rangle \in \text{Ran } \text{tim}} \{ N \}
\end{align*}
\]

Condition (4.25) requires that \( \text{tim} \) provide an interpretation for every type exported by \( AI_G \). Condition (4.25) requires that each such interpretation be meaningful (i.e., non-⊥).

2. For every \( \langle N, R, C \rangle \in \text{Ran } \text{tim} \), every operation used in the interpretation respects the “type convention” \( C \) for all output values it produces of type \( N \). In other words, for every operation name \( O \in \text{Ran } IM. \text{OPmap} \) (i.e., operation in \( AI_S \) being interpreted as some operation in \( AI_G \)), the postcondition in the operation model associated with \( O \) by \( AI_S \) (OperationModel \( (O, AI_S). EP \)) implies that \( C \) holds for all output values from \( O \) of type \( N \).

4.10.2 Variable Interpretation Mappings (VIM)

A Variable Interpretation Mapping is a name-to-value mapping from \( N \) (variable names in LocalVariableNames \( AI_G \)) to \( N \) (variable names in LocalVariableNames \( AI_S \)). A Variable Interpretation Mapping \( vim \) is well-formed with respect to an abstract instance \( AI_S \), and abstract instance \( AI_G \), and an interpretation mapping \( IM \) if for every variable name \( N \) in LocalVariableNames \( AI_G \), \( T_N = \text{VariableModel} (N, AI_G) \) (a variable signature, which is a type name) is mapped to \( T_{vim(N)} \text{VariableModel} (vim(N), AI_S) \) by \( IM. \text{TYmap} \). In other words, \( vim \) must respect the Type Interpretation Mapping in \( IM \).
CHAPTER IV. THE ACTI DEFINITION

4.10.3 Operation Interpretation Mappings ($\mathcal{OIM}$)

An Operation Interpretation Mapping $oim$ is a name-to-value mapping from $\mathcal{N}$ to $\mathcal{N}$. $oim$ is intended to map operation names in LocalOperationNames $AI_G$ to (a subset of) the operation names in LocalOperationNames $AI_S$.

An Operation Interpretation Mapping is well-formed with respect to an abstract instance $AI_S$, and abstract instance $AI_G$, and an interpretation mapping $IM$ if the following conditions hold:

1. The operation interpretation mapping $oim$ must completely cover the set of names in LocalOperationNames $AI_G$, and must use a non-$\perp$ element from LocalOperationNames $AI_S$ as the interpretation for each operation in LocalOperationNames $AI_G$. More formally, for every such $AI_S$ and $AI_G$, $oim$ must satisfy two conditions:

\[
\text{LocalOperationNames } AI_G \subseteq \text{Dom } oim \tag{4.27}
\]
\[
\text{LocalOperationNames } AI_S \supseteq \text{Ran } oim \tag{4.28}
\]

Condition (4.27) requires that $oim$ provide an interpretation for every operation exported by $AI_G$. Condition (4.28) requires that each such interpretation be meaningful (i.e., $nn-\perp$).

2. For each operation $O \in \text{LocalOperationNames } AI_G$, let $O_S$ and $O_G$ be defined as follows:

\[
O_S = \text{OperationModel} \left( oim(O), AI_S \right) \tag{4.29}
\]
\[
O_G = \text{OperationModel} \left( O, AI_G \right) \tag{4.30}
\]

The lower level (more specific) operation model $O_S$ must be consistent with the behavior described by the higher level (more general) operation model $O_G$. Essentially, we want to ensure that the precondition in $O_S$ is the same or weaker as that of $O_G$, while the postcondition is the same or stronger. Unfortunately, these pre- and postconditions may be based on different type models for the parameters.

The type correspondences defined in $IM.TYmap$ for each type visible in $AI_G$ provide the basis for “interpreting” the lower level pre- and postconditions of $O_S$ to the higher level conceptual models so that we can properly compare the behavior described by the two operation models. First, we require that both $O_S$ and $O_G$ have the same number of parameters in the same order, and that for each parameter type $T$ in $O_G.PP$, the corresponding parameter type in $O_S.PP$ is $IM.TYmap(T).N$. 
4.10. INTERPRETATION MAPPINGS ($\mathcal{IM}$)

Next, we can “translate” $O_S.DP$ and $O_S.EP$ by composing them with the type correspondence relations for each parameter type $T$ (i.e., $IM.T\text{Ymap}(T).R$). Once this is done, we require that $O_G.DP$ imply the translated version of $O_G.DP$. Similarly, the translated version of $O_S.EP$ must imply $O_G.EP$.

4.10.4 Abstract Template Interpretation Mappings ($\mathcal{ATIM}$)

An Abstract Template Interpretation Mapping is a name-to-value mapping from $\mathcal{N}$ (abstract template names in $AI_G$) to $\mathcal{N}$ (abstract template names in $AI_S$). To describe the conditions necessary for an abstract template interpretation mapping to be well-formed, we need to be able to talk about the identifiers of abstract template signatures defined in an abstract instance. For this purpose, we now define an overloaded operator AbstractTemplateNames.

The operator AbstractTemplateNames can be applied to abstract instances or abstract instance environments. For an abstract instance $AI$, AbstractTemplateNames $AI$ is defined as follows:

$$\text{AbstractTemplateNames } AI = \text{Dom } AI.ATES \cup \text{AbstractTemplateNames } AI.CIE \cup \text{AbstractTemplateNames } AI.CTXT \quad (4.31)$$

For an abstract instance environment $AIE$, the set of AbstractTemplateNames $AIE$ is defined as follows:

$$\text{AbstractTemplateNames } AIE = \bigcup_{AI \in \text{Dom } AIE} \text{AbstractTemplateNames } AI \quad (4.32)$$

As expected, LocalAbstractTemplateNames $AI$ is defined as AbstractTemplateNames $AI$—AbstractTemplateNames $AI.CTXT$. We can also give AbstractTemplateModel $(x, AI)$ the obvious analogous definition for $x \in \text{AbstractTemplateNames } AI$.

An Abstract Template Interpretation Mapping $atim$ is well-formed with respect to an abstract instance $AI_S$, and abstract instance $AI_G$, and an interpretation mapping $IM$ if the following three conditions hold:

$$\text{LocalAbstractTemplateNames } AI_G \subseteq \text{Dom } atim \quad (4.33)$$

$$\text{LocalAbstractTemplateNames } AI_S \supseteq \text{Ran } atim \quad (4.34)$$
∀x ∈ LocalAbstractTemplateNames AI_G : (∀AT ∈ AT : (AbstractTemplateModel (x, AI_G).AIDP(AT) implies (AbstractTemplateModel (IM.ATmap(x), AI_S).AIDP(AT)
and (AbstractTemplateModel (IM.ATmap(x), AI_S).AIEP(AT)
implies
AbstractTemplateModel (x, AI_G).AIEP(AT)))))) \hspace{1cm} (4.35)

Condition (4.33) requires that \textit{atim} provide an interpretation for every abstract template signature exported by \textit{AI}_G. Condition (4.34) requires that each such interpretation be meaningful (i.e., \textit{mi-⊥}). Condition (4.35) requires that each such interpretation respect the pre- and postconditions of the abstract template signatures under consideration (i.e., the domain predicates of abstract template signatures in \textit{AI}_S must be the same or weaker than those in \textit{AI}_G, and the effect predicates in \textit{AI}_S must be the same or stronger).

### 4.10.5 Concrete Template Interpretation Mappings (\textit{CTIM})

A \textit{Concrete Template Interpretation Mapping} is a name-to-value mapping from \textit{N} (concrete template names in \textit{AI}_G) to \textit{N} (concrete template names in \textit{AI}_S). To describe the conditions necessary for a concrete template interpretation mapping to be well-formed, we need to be able to talk about the identifiers of concrete template signatures defined in an abstract instance. For this purpose, we now define an overloaded operator \textit{ConcreteTemplateNames}.

The operator \textit{ConcreteTemplateNames} can be applied to abstract instances or abstract instance environments. For an abstract instance \textit{AI}, the set of \textit{ConcreteTemplateNames} \textit{AI} is defined as follows:

\[
\text{ConcreteTemplateNames } \textit{AI} = \text{Dom } \textit{AI}.CTES \cup \\
\text{ConcreteTemplateNames } \textit{AI}.CIE \cup \\
\text{ConcreteTemplateNames } \textit{AI}.CTXT \hspace{1cm} (4.36)
\]

For an abstract instance environment \textit{AIE}, \textit{ConcreteTemplateNames} \textit{AIE} is defined as follows:

\[
\text{ConcreteTemplateNames } \textit{AIE} = \bigcup_{\textit{AI} \in \text{Dom } \textit{AIE}} \text{ConcreteTemplateNames } \textit{AI} \hspace{1cm} (4.37)
\]
4.11. INTERPRETATION MAPPING TEMPLATES (IMT)

As expected, LocalConcreteTemplateNames $A_I$ is defined as ConcreteTemplateNames $A_I$—ConcreteTemplateNames $A_I.CTXT$. We can also give ConcreteTemplateModel $(x, A_I)$ the obvious analogous definition for $x \in$ ConcreteTemplateNames $A_I$.

A Concrete Template Interpretation Mapping $ctim$ is well-formed with respect to an abstract instance $A_I_S$, and abstract instance $A_I_G$, and an interpretation mapping $IM$ if the following three conditions hold:

\[
\text{LocalConcreteTemplateNames } A_I_G \subseteq \text{Dom } ctim \quad (4.38) \\
\text{LocalConcreteTemplateNames } A_I_S \supseteq \text{Ran } ctim \quad (4.39)
\]

\[
\forall x \in \text{LocalConcreteTemplateNames } A_I_G : ( \\
\forall CT \in CT : ( \\
\quad \text{ConcreteTemplateModel } (x, A_I_G).CIDP(CT) \text{ implies } ( \\
\qquad \text{ConcreteTemplateModel } (IM.CTmap(x), A_I_S).CIDP(CT) \\
\qquad \text{and } ( \\
\qquad \quad \text{ConcreteTemplateModel } (IM.CTmap(x), A_I_S).CIEP(CT) \text{ implies } \\
\qquad \quad \text{ConcreteTemplateModel } (x, A_I_G).CIEP(CT)))))) \\
(4.40)
\]

Condition (4.38) requires that $ctim$ provide an interpretation for every concrete template signature exported by $A_I_G$. Condition (4.39) requires that each such interpretation be meaningful (i.e., $nn_{-\bot}$). Condition (4.40) requires that each such interpretation respect the pre- and postconditions of the concrete template signatures under consideration (i.e., the domain predicates of concrete template signatures in $A_I_S$ must be the same or weaker than those in $A_I_G$, and the effect predicates in $A_I_S$ must be the same or stronger).

4.11 Interpretation Mapping Templates (IMT)

An Interpretation Mapping Template is a three-tuple $\langle IMTS, AIIIMF, SF \rangle$, where:

$IMTS$ is an Interpretation Mapping Template Signature, defined below.

$AIIIMF$ is the “abstract instance-to-interpretation mapping function” $A_I \to IM$ actually computed by instantiating the template.

$SF$ is a “status function,” an Assert Status-valued function over the input abstract instance $A_I \to AS$ representing the assert status actually computed by instantiating the template.
CHAPTER IV. THE ACTI DEFINITION

An Interpretation Mapping Template \( int \) is well-formed when the following two conditions are met:

1. For all input abstract instance values \( AI \) satisfying \( int.IMTS.AIDP \), the effect predicate \( int.IMTS.IMEP(AI, int.AIIMF(AI)) \) must be true. In other words, the actual abstract instance-to-interpretation mapping function computed by the template must be consistent with the postcondition described by \( int.IMTS.IMEP \).

2. For all input abstract instance values \( AI \) satisfying \( int.IMTS.AIDP \), \( int.SF(AI) \) must not produce the value \( CF \).

**An Intuitive Explanation of Interpretation Mapping Templates**

An Interpretation Mapping Template is a generic or parameterized interpretation mapping, which should be thought of as a function from abstract instances to interpretation mappings. The characteristics of this function are captured in the three-tuple described above. Its structure exactly parallels that of a Concrete Template or an Abstract Template. Just like a concrete or abstract template, the nested structure of the single abstract instance parameter allows an interpretation mapping template effectively to be parameterized by many instances. Because the requirements on the single parameter are expressed through a predicate, that parameter may contain variable numbers of nested instances, allowing variable-length generic parameter lists.

**4.11.1 Interpretation Mapping Template Signatures (IMTS)**

An Interpretation Mapping Template Signature is a two-tuple \( \langle AIDP, IMEP \rangle \), where:

\( AIDP \) is an “abstract instance domain predicate,” a boolean-valued function over the input abstract instance \( \langle AI \rightarrow B \rangle \) that represents the precondition for the template.

\( IMEP \) is an “interpretation mapping effect predicate,” a boolean-valued function over the input abstract instance and the output interpretation mapping \( \langle AI \times IM \rightarrow B \rangle \) that represents the postcondition of instantiating the template.

All interpretation mapping template signatures are well-formed.

**4.12 Basis Environments (BE)**

A Basis Environment is a top-level run-time environment for a program. A Basis Environment is a two-tuple \( \langle AS, E \rangle \), where:
4.13. OPERATIONS ON SUBSYSTEMS

$AS$ is an Assert Status, one of $\{NL, VT, CF, \bot\}$.

$E$ is an Environment (Section 4.5, p. 66).

**An Intuitive Explanation of Basis Environments**

A basis environment environment is simply a single concrete instance bundled together with an assert status (see Section 4.1.6). The concrete instance component of the basis environment is intended to model the traditional notion of a run-time program state—a name-to-value mapping that assigns appropriate values to all of the software objects in a program. Using a concrete instance for this purpose gives structure to the program state, allowing the value assignment the state represents to be compartmentalized along the subsystem boundaries within the program. Adding an assert status value supports formal verification in the “assertive programming” style. This combination is directly analogous to the program state model presented by Ernst et al. [10, p. 268] [12, pp. 2–3, 8–9], and is necessary for supporting modular verification of software [12, pp. 2–3].

**4.13 Operations on Subsystems**

All of the mathematical spaces in ACTI support the basic mathematical operations one would expect from their structure: projection and injection for cartesian products, application and composition for functions, and so on. Given these operations, however, there are several more sophisticated operations on the spaces which can help describe what happens to the current execution environment when subsystem-changing statements are executed (e.g., when instantiations occur, when inheritance is applied, or when subsystems are combined). For example, when instantiating a component in RESOLVE the following tasks are all performed (not necessarily in this order):

- An abstract template (concept) is selected, say $C$.

- Values for the RESOLVE conceptual context parameters from the current execution environment (e.g., a parameter value for the abstract template) are selected.

- A concrete template (realization), say $R$, is selected that implements the chosen abstract template (i.e., there exists some interpretation mapping $IM$ such that $IM \models R$ is-interpretable-as $C$).
• Values for the RESOLVE realization context parameters from the current execution environment (e.g., a parameter value for the concrete template) are selected.

• All definitions provided by the resulting module instantiation are added to the execution environment.

The operations described in this section make it easier to talk about the primitives involved in higher-level operations like instantiation of a RESOLVE-style component.

4.13.1 Abstracting an Environment

Because Concrete Instances and Abstract Instances are so similarly structured, it is useful to define an operation that, given a concrete instance, will produce a corresponding abstract instance which contains “as much information as possible.” Intuitively, one might think of this generated abstract instance as a “maximal specification” for the corresponding concrete instance, in that it contains exactly the same conceptual models, with all of the run-time value information removed.

This operation, called \texttt{abstract}, has the following signature:

\[
\texttt{abstract} : \mathcal{E} \rightarrow \mathcal{AI}
\]  

(4.41)

It works simply by eliminating some information from the environment as follows:

• Removing all value information from the Variable Environment, turning it into a Variable Environment Signature.

• Converting each Operation Meaning in the Operation Environment to an Operation Model by removing the procedure relation and status function components. This turns the Operation Environment into an Operation Environment Signature.

• Converting each Concrete Instance in the Concrete Instance Environment into an Abstract Instance by recursively applying the \texttt{abstract} operator.

• Converting each Abstract Template in the Abstract Template Environment to an Abstract Template Signature by removing its AIAIF and SF components.

• Converting each Concrete Template in the Concrete Template Environment to a Concrete Template Signature by removing its CICIF and SF components.

• Leaving the CTXT, SAE, TE, EI, AIE, IME, and IMTE components unchanged.
4.13. OPERATIONS ON SUBSYSTEMS

4.13.2 Reinterpreting an Environment

Given the abstract operator, it is now possible to talk about interpretation mappings that show how a particular interface (abstract instance) is a interpretation of an environment. For a given interpretation mapping $IM$, abstract instance $AI$, and environment $E$, if:

$$IM \models \text{abstract}(E) \text{ is-interpretable-as } AI$$  \hspace{1cm} (4.42)

then we can say that the concrete instance $E$ can be interpreted as meeting the behavioral description in $AI$ under $IM$ (or, alternatively, that $E$ is an implementation of $AI$). Thus, the following is shorthand for the above expression, with application of the abstract operator omitted but understood:

$$IM \models E \text{ is-interpretable-as } AI$$  \hspace{1cm} (4.43)

4.13.3 Cutting Down an Environment by an Interface

Given that we can talk about an interface generalizing an environment, say $IM \models E \text{ is-interpretable-as } AI$, we can then talk about a “cut” operator that can use such an interface along with the corresponding interpretation mapping to “reduce” the environment so that only what is “visible through the interface” remains. This is useful because $E$ may contain many objects that are not used in the interpretation. A cut operator allows one to formally model the effect of “encapsulating” the implementation $E$, so that only the features explicitly exported by the interface $AI$ are visible.

The cut operator has the following signature:

$$\downarrow : \mathcal{E} \times IM \times AI \rightarrow \mathcal{E}$$  \hspace{1cm} (4.44)

Intuitively, this operator simply removes all information from the input environment that is not mapped through the interface via $IM$. For a given $E$, $IM$, and $AI$, applying this operator produces a new environment $E_2 = (E \downarrow IM \downarrow AI)$. Essentially, this new environment is $AI$, extended with information about the variable values, procedure relations, status functions, and concrete instances from $E$, as viewed through the module and type correspondences defined in $IM$. As a result, abstract $(E_2) \approx AI$. The only place where abstract $E_2$ and $AI$ differ is in their CTXT components.

Since $E$ may have more external dependencies in its context than $AI$ does, these extra dependencies must be preserved in order for the implementation of $E$ to remain
valid. As a result, $E_2.CTXT$ is defined as:

$$E_2.CTXT = \begin{align*}
CTXT &= E.CTXT.CTXT \\
SAE &= E.CTXT.SAE \\
TE &= E.CTXT.TE \\
VES &= E.CTXT.VES \\
OES &= E.CTXT.OES \\
EI &= E.CTXT.EI \\
AIE &= E.CTXT.AIE \\
CIE &= E.CTXT.CIE + \\
IME &= E.CTXT.IME \\
ATES &= E.CTXT.ATES \\
CTES &= E.CTXT.CTES \\
IMTE &= E.CTXT.IMTE
\end{align*}$$

Here, $unused$ represents some name from $\mathcal{N}$ that is not in $\text{Dom } E.CTXT.CIE$. This modification creates an empty subsystem with the same context as $AI$, and then embeds (see Section 4.13.6 below) it within the context of $E$, and uses the result as the context for $E_2$. This ensures that all external dependencies from $E$ are preserved, while their interpretation under $IM$ as the more abstract context of $AI$ is also recorded.

### 4.13.4 Derivation of Instances by Difference

Given two abstract instances $AI_1$ and $AI_2$, it is natural to begin thinking about module composition operators. For example, one would expect:

$$AI_1 + AI_2 = AI_3$$

(4.46)

to produce a third abstract instance which defines the “sum” of the types, operations, sub-instances, etc., of $AI_1$ and $AI_2$ (assuming that the sets of names defined in $AI_1$ and $AI_2$ are disjoint). In fact, this is a basic generalization of the “modifies” operator for name-to-value mappings introduced in Section 4.1.2. An abstract (or concrete) instance contains only three components that are not name-to-value mappings: $CTXT$, $SAE$, and $EI$. Specification adornment environments contain only name-to-value mappings and an invariant $SAEI$. Thus, we can recursively define the
4.13. OPERATIONS ON SUBSYSTEMS

+ operator on specification adornment environments as:

\[
SAE_1 + SAE_2 = \begin{cases} 
SATY = SAE_1.SATY + SAE_2.SATY \\
SAVE = SAE_1.SAVE + SAE_2.SAVE \\
SAEI = SAE_1.SAEI \land SAE_2.SAEI \\
SAIE = SAE_1.SAIE + SAE_2.SAIE \\
SATE = SAE_1.SATE + SAE_2.SATE 
\end{cases}
\] (4.47)

From this, we can now define the + operator on abstract instances:

\[
AI_1 + AI_2 = \begin{cases} 
CTXT = AI_1.CTXT + AI_2.CTXT \\
SAE = AI_1.SAE + AI_2.SAE \\
TE = AI_1.TE + AI_2.TE \\
VES = AI_1.VES + AI_2.VES \\
OES = AI_1.OES + AI_2.OES \\
EI = AI_1.EI \land AI_2.EI \\
AIE = AI_1.AIE + AI_2.AIE \\
CIE = AI_1.CIE + AI_2.CIE \\
IME = AI_1.IME + AI_2.IME \\
ATES = AI_1.ATES + AI_2.ATES \\
CTES = AI_1.CTES + AI_2.CTES \\
IMTE = AI_1.IMTE + AI_2.IMTE 
\end{cases}
\] (4.48)

Modification of concrete instances is defined similarly.

Given this basic modification operation, all of Goguen’s module expression calculus [21, pp. 194–198] can be easily adapted to abstract and concrete instances. It can even be applied to templates and interpretation mappings. As a result, Goguen’s entire “horizontal composition” strategy, which amounts to inheritance used for programming by difference—purely as a definitional mechanism without any effect on the typing system—can be easily interpreted in the framework of the ACTI model.

4.13.5 Composing Interpretation Mappings

Given two interpretation mappings, \( IM_1 \) and \( IM_2 \), and an abstract instance \( AI \), it is possible to define the composite mapping:

\[
IM_1 \circ AI \circ IM_2 = IM_3
\] (4.49)

This is a natural result of the fact that interpretation mappings form a preorder over abstract instances.
4.13.6 Binding Context and Composing Instances

Because every abstract and concrete instance in ACTI has an explicit context interface (recorded in its $CTX$ component), the question arises of how such an interface is tied to specific external definitions in the subsystem’s surrounding environment. As indicated in Section 4.5.5 (pages 74–75), “context-bound” abstract and concrete instances describe subsystems with context explicitly tied to their surrounding environment. To “fix” the context of a subsystem $S$, one must literally embed $S$ into some larger subsystem that represents the environment in which $S$ lives, say $OUTER$. This outer environment may be either an abstract or a concrete instance.

Before binding such a subsystem’s context, one must have a subsystem $S$ in isolation, i.e., with free context, as well as some outer environment $OUTER$ that will provide the surrounding medium providing actual objects matching the context requirements of $S.CTX$. This situation is depicted in Figure 28.

![Diagram of Subsystem $S$ and Outer Environment $OUTER$]

Figure 28: A Subsystem $S$ and the Outer Environment That Will Provide Its Context
The only missing component is an explanation of how $OUTER$ actually fulfills the context dependencies of $S$. All that is required is an interpretation mapping $IM$ such that $IM \models OUTER \text{ is-interpretable-as } S.CTX$. Thus, $IM$ formally defines how $OUTER$ (or abstract($OUTER$) if $OUTER$ is concrete) can be interpreted as the context of $S$. This is shown in Figure 29.

Given that $S$, $OUTER$, and $IM$ are all well-formed, and $IM.DP(OUTER, S.CTX)$, the precondition for $IM$, is true, all that is left is to embed the pair $\langle S, IM \rangle$ in the appropriate (abstract or concrete) instance environment in $OUTER$ with a new name, say $S_{name}$. Suppose both $S$ and $OUTER$ are abstract instances. Embedding $\langle S, IM \rangle$ in the concrete instance environment of $OUTER$ is accomplished by modifying $OUTER.CIE$ to be:

$$OUTER.CIE + \{S_{name} \mapsto \langle S, IM \rangle\}$$

(4.50)

Figure 29: $IM$ Shows How $OUTER$ Can Be Interpreted as the Context for $S$
4.13.7 Instantiating a Template

Intuitively, instantiating a template is just like applying a function. A template's parameter is a single subsystem, and applying the template to that parameter produces another subsystem. Templates are only meaningfully applied to subsystems that meet their preconditions—i.e., satisfy their domain predicates.

We can use normal functional notation to represent the application of a template. Suppose $AT$ is an abstract template, and $S$ is an abstract instance such that $AT.AIDP(S)$ is true. Then $AT.AIAIF(S)$, or simply $AT(S)$, is also an abstract template, which may later be embedded in another subsystem given an appropriate interpretation mapping to bind its context.

4.14 Connection with Traditional Denotational Semantics

Most conventional work on denotational semantics for particular programming languages has focused on the meaning of lower-level programming constructs: types, variables, control flow constructs, procedures, and so on. Work has also been done on intra-module assertions like correspondence relations and conventions [33]. From this perspective, a view of a program state as a simple mapping from variable names to variable values has been adequate. As suggested by Ernst et al., however, the notion of program state must expand to include information about procedure parameter profiles and behavioral specifications (i.e., *Operation Models*) and even information about module specifications, if modular verification is to be possible [12, p. 2]. This is exactly what has been done in ACTI.

Fortunately, adding this additional information and a corresponding structure to the notion of a program state does not invalidate previous work on the meaning of lower-level programming constructs. ACTI is geared toward integration with a procedure-level denotational semantic model similar to that described by Ernst et al. [12, 10, 13, 11]. In fact, if operation meanings included a procedure function (instead of a procedure relation), no changes would be necessary to this prior work in order for it to be used in combination with ACTI.

However, because of the relational nature of *Operation Models*, and the relational nature of type correspondence relations and subsystem state correspondence relations in interpretation mappings, a strictly functional notion of procedural semantics is not viable. This is due to the fact that in ACTI, a procedure’s meaning can only be defined in terms of the context of the subsystem it is contained in. This context contains only operation models, not operation meanings, so that the meaning of the procedure in question can be defined independently of any other external operations upon which it relies. Since the behavioral description in an operation model can be relational,
4.15 STRENGTHS AND WEAKNESSES

regardless of the actual computation carried out by the corresponding code, in general a procedure depending on such context cannot be given a strictly functional meaning.

Interestingly, John Sanderson points to a solution to this problem [56]. He describes a relational calculus of computation suitable for modeling the denotational semantics of lower-level program constructs. His calculus has the compositional and least fixed point properties necessary for use in modeling the semantics of conventional languages. This calculus could be adopted for integration with ACTI to provide a complete denotational semantic model for a programming language, such as RESOLVE.

4.15 Strengths and Weaknesses

In addition to the contributions listed in Section 1.3, the ACTI model as defined here has several important strengths:

- It provides detailed support for modeling the behavior of all software objects. Models of software objects, if written down formally or informally are, critical for supporting the formation of effective mental models. As a result, they are critical for supporting understandability.

- It provides strong support for separation of specification (or abstract modeling) from implementation in all of the details of programming. The prevalence of spaces of “signatures” in the ACTI definition shows that this separation is reflected in many more ways than simply the separation of module specifications from module implementations.

- It meets all criteria for a “comprehensive” model of software structure and meaning presented in Appendix B.

- It combines the advantages of the previous models of software described in Appendix A, while avoiding their disadvantages.

Notably, however, the generality achieved in this model brings with it some traits that might be considered weaknesses from the perspective of more traditional work in modeling software meaning:

- The ACTI model is not “fully abstract” in the sense described by Allen Stoughton [60]. Full abstractness is based on the idea of behavioral equivalence. Two programs are behaviorally equivalent if the compute the same input-output function
(or relation). They are indistinguishable in terms of their externally visible behavior. For a model of software to be fully abstract, it must model behaviorally equivalent software constructs using the same denotation.

In ACTI two subsystems that are behaviorally equivalent need not be identical. Even though their behavior is the same, the models used to describe their behavior, their specification adornment definitions, and so on, may all be different. This can pose problems for formal reasoning about the equivalence of software parts.

It may be possible to address this limitation by formalizing the notion of a "reversible" or "two-way" interpretation mapping. Effectively, such a mapping would capture the idea of a behavioral isomorphism between the two subsystems.

- Similarly, The fact that two software objects are identical in ACTI does not mean that are always legitimately interchangeable. For example, the Finalize operation for Partial_Map introduced in Section 3.4.2 has a behavioral specification that says it has no externally visible effect. This does not mean, however, that its execution can be eliminated by an optimizer—substituted by a null operation that has the same external behavior. This is typically only a problem with infrastructure-related functions, like Initialize and Finalize, that are built-in to a specific language or execution model.

- Finally, interpretation mappings do not define a partial ordering over abstract instances; they define a preordering. That is, for two abstract instances $A$ and $B$, $A$ is-interpretable-as $B$ and $B$ is-interpretable-as $A$ does not necessarily imply $A = B$. Much of the past work on relationships between specifications (subtyping, refinement, and so on) relies on a partial ordering of specifications. This problem is partly due to the fact that interpretation mappings are more general than other proposed specification relations, and also partly due to the fact that ACTI is not a fully abstract model of software.

4.16 Chapter Summary

This chapter presents the formal definition of the ACTI software model. This model is centered around the mathematical spaces modeling the four types of subsystems: Abstract Instances, Concrete Instances, Abstract Templates, and Concrete Templates. The model also includes Interpretation Mappings and Interpretation Mapping Templates as central ideas. All of the spaces in ACTI are complete partial orders (CPOs), so that the model can be used as the starting point for the denotational semantics of a programming language.
Many of the mathematical spaces in ACTI contain name-to-value mappings. These mappings associate identifiers from the space of Names with objects from other ACTI spaces. This is analogous to the traditional idea of a program state. In ACTI, however, a program's state can be modeled with a Basis Environment, which is a highly structured collection of name-to-value mappings, rather than a single flat state function.

In addition to the notion of name-to-value mappings, the ACTI definition also uses the concept of signatures. Many of the mathematical spaces in the ACTI model are related in pairs—one space to represent actual software objects, and the other to represent specifications, or conceptual models, of those objects. Such a space of specification-only values is called a space of signatures.

The ACTI definition presented in this chapter provides the foundation necessary to understand the intuitive concepts introduced in Chapter III at a formal level. Chapter V continues the example introduced in Chapter III in the context of this formal definition.
CHAPTER IV. THE ACTI DEFINITION