Characterizing Observability and Controllability of Software Components

Bruce W. Weide

Stephen H. Edwards

Wayne D. Heym[*]

Timothy J. Long

William F. Ogden

Department of Computer and Information Science

The Ohio State University

2015 Neil Avenue

Columbus, OH 43210

{weide,edwards,heym,long,ogden}@cis.ohio-state.edu

Technical Report OSU-CISRC-9/95-TR37

September 1995

[*] Also with Department of Mathematical Sciences, Otterbein College, Westerville, OH 43081.

Abstract

Two important objectives when designing a specification for a reusable software component are understandability and utility. For a typical component defining a new abstract data type, a significant common factor affecting both of these objectives is the choice of a mathematical model of the (state space of the) ADT, which is used to explain the behavior of the ADT's operations to potential clients. There are subtle connections between the expressiveness of this mathematical model and the functions computable using the operations provided with the ADT, giving rise to interesting issues involving the two complementary system-theoretic principles of "observability" and "controllability". Previously we recommended a practical way to test compliance of a proposed design with these informally-defined principles: it should be possible to construct layered implementations of operations to test equality of and to copy variables of an ADT. This paper discusses problems associated with formalizing intuitively-stated observability and controllability principles in accordance with these tests. Although the example we use for illustration is simple, the analysis has implications for the design of reusable software components of every scale and conceptual complexity.

Keywords

abstraction, controllability, design for reuse, formal specification, full abstraction, model-based specification, observability, reuse, software component

1. Introduction

Specifying the behavior of a software component — especially one that is meant to be reused — is a challenging task. Some important "quality" objectives of design in this area include avoiding implementation bias [Jones 90] and achieving understandability for potential component clients [Sitaraman 93]. How can the specifier's design space be limited so high quality reusable component designs are allowed while low quality ones are ruled out? And how can proposed design principles be made effectively checkable and not merely slogans?

Surely no general guidelines can succeed completely, but experience shows that some do constrain the design space in the right ways. In prior work we surveyed several specification principles that were intuitively described in the literature and proposed practical tests for compliance [Weide 91]. In this paper we report on some interesting problems associated with two of these principles, observability and controllability, which deal with the relationship between the expressiveness of the mathematics used in a specification and the computational power of the specified component. Informally, they (together) provide a test for "minimality" of the specified state space of an ADT.

Our contributions here are:

- We show why it is important to make careful and unambiguous definitions of these principles, because superficially reasonable interpretations of the informal definitions can easily lead to compliance tests that admit poor designs.
- We illustrate unexpected difficulties in making careful and unambiguous definitions.
- We lay out a road map of possible ways to formalize observability and controllability. At each fork in the road (marked in the text with ψ) this paper takes a particular branch in concert with folklore about specification design, leading toward and beyond fairly specific principles proposed in the literature [Weide 91]. This gives a depth-first view of the landscape of Figure 1. A more comprehensive paper in preparation will discuss the paths we do not follow here.

1.1. The Principle of Observability

One of the most important design decisions facing a reusable component specifier is the selection of an appropriate mathematical model (also called "conceptual model" or "abstract model" or "mental model" [Norman 90]) for the state space of values for variables (or "objects") of a new abstract data type (ADT) [Edwards 94, Guttag 93, Spivey 89, Weide 91, Wing 90]. This model is used to explain the abstract behavior of a component's operations, so the choice of model directly influences the understandability of the concept and the ease of reasoning about its implementations and clients that are layered on top of it [Sitaraman 93, Edwards 95]. Typically, the specification designer must consider a variety of candidate mathematical models before identifying the "best" one(s). There are many options because both standard and newly-conceived mathematical models — and compositions and combinations thereof — are candidates.

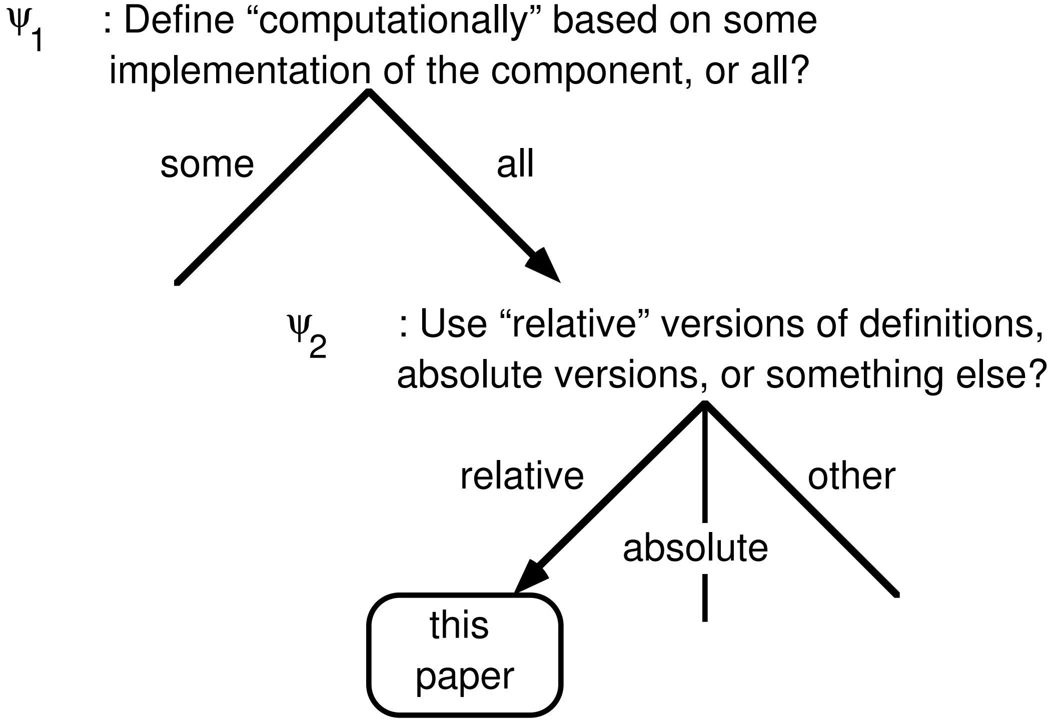
ψ_1 : Define "computationally" based on some implementation of the component, or all?
some
all
ψ_2 : Use "relative" versions of definitions, absolute versions, or something else?
relative
absolute
other
this paper

Figure 1 — Major Decision Points in Formalizing Observability and Controllability

An intuitively pleasing ideal that helps limit the design space in this dimension is the *principle of observability:*

$\mathbf{O_0}$ A specification S defining the program type ADT is observable iff every two unequal values in ADT's state space are "computationally distinguishable" using some combination of operations of S.

An appropriate way to view observability is in terms of the connection between the structure of the state space imposed on it by its mathematical operators and predicates, and the computational structure imposed on it by the specified programming operations. Observability dictates that the model should define a state space which makes distinctions that are just sufficient to specify the intended behavior of the operations — and no more; i.e., the model does not distinguish values that are indistinguishable by the operations. One predicate that is available in nearly every useful mathematical state space is equality. Basing observability on equality makes the principle generally applicable, although it is possible to refine it to other predicates particular to individual mathematical theories.

A prime motivation for demanding observability is psychological. In trying to understand a specification, a client naturally assumes that distinctions in the state space are important. If a specification makes distinctions (two model values are unequal mathematically) without differences (variables with those two distinct values are computationally indistinguishable), confusion is inevitable. The conceptual model the specifier is trying to give the client fails to convey the true situation, and the client is likely to look for another model of the component's behavior and to translate mentally between the official specification and this alternate view — a situation virtually certain to result in miscommunication between the parties [Norman 90].

1.2. The Principle of Controllability

A complementary objective to understandability is utility: a reusable component should be useful to a variety of clients whose particular needs for variants of a basic functionality are perforce unknown at component design time. Another way to view this notion of utility is in terms of "functional completeness". This suggests that the combination of operations being specified should be at least powerful enough to construct any value in the state space defined by the model.

An intuitive statement of this property is the *principle of controllability:*

C_0 A specification S defining the program type ADT is controllable iff every value in ADT's state space is "computationally reachable" using some combination of operations of S.

A prime motivation for seeking controllability is technical, although it might be argued that observability is technically even more crucial. An example illustrates their combined importance. Suppose a client programmer using the specified component S wants to show that a code segment preserves the abstract value of some ADT variable, although perhaps the value is temporarily modified within the segment. If S is not both observable and controllable then most likely it is impossible to do this — either because it is impossible to predictably reconstruct the original value in the first place (e.g., because the original value resulted from non-deterministic behavior that is not certain to be repeated), or because it is impossible to know that a proposed reconstructed value is really equal to the original and not simply computationally indistinguishable from it.

1.3. The Need for Practical Compliance Tests

How are observability and controllability applied in practice? Typically a designer has an informal notion of what basic functionality is sought. An initial set of operations is postulated, and the next question is what model to use to explain the state space over which these operations work. The principles of observability and controllability lead the designer to seek a state space for the specified behavior without redundant values that cluster into non-singleton congruence classes of computationally indistinguishable points, and without values that are not even reachable. A first attempt at specifying the operations is made using a "natural" model that is thought (hoped) to lead to a specification which is both observable and controllable. But sometimes it is not, in which case there are two repair strategies: try another model, or modify the behavior of some operations and perhaps add and/or remove some. In this paper we use an example that illustrates only the second approach. But in either case the designer checks again for observability and controllability. With luck, the process eventually terminates with a design that satisfies both of these design principles (and presumably others of simultaneous interest).

In order to carry out this iterative process, then, a designer has to have effective practical tests for whether a specification complies with the two principles. This requires making clear, unambiguous definitions of the principles, which is the focus of this paper.

We begin in Section 2 by reviewing related work and outlining a working example. In Section 3 we discuss ambiguities in, and possible formalizations of, $\mathbf{O_0}$ and $\mathbf{C_0}$; then in Section 4 we explain how these definitions break down when applied to parameterized components that typify reusable software components (e.g., Ada generic packages and C++ class templates). Finally, in Section 5 we draw conclusions and again relate the path of this paper to the road map in Figure 1.

2. Background and Working Example

The principles of observability and controllability, as defined here, are meaningful only in the context of model-based specifications where mathematical theory and program specification are separate, as in Larch [Guttag 93] and RESOLVE [Edwards 94]. The question addressed by observability and controllability is essentially whether the mathematical model of an ADT is in some sense "minimal" in size and structure for specifying a programming concept. This is not a well-formed question for true algebraic specifications, in which a mathematical theory and a programming component being specified are treated as inseparable. The closely related taxonomy of mathematical functions of a theory into "observers" and "constructors" (e.g., [Liskov 86, Guttag 93]) is clearly related in spirit, but these notions are one level removed as they pertain to the design of mathematical theories and not to the design of model-based specifications that use those theories.

A related issue that received much attention in the late 1970's in the algebraic specification community is when two mathematical values should be considered equal. Some authors [Liskov 75, Goguen 78] considered two values to be different unless demonstrably equal based on the axioms. Others [Guttag 78] considered two values to be equal unless provably different. While the first group took a traditional view and insisted that the smallest congruence relation defined by the axioms be used, the latter group allowed any congruence relations (including the smallest) consistent with the axioms. In general, for well-defined theories that are typically used as models (e.g., the Larch set trait [Guttag 93]) the two notions converge. Our consideration of observability and controllability is independent of this question, because we simply assume equality in the mathematical spaces as a given predicate with the requisite properties.

To our knowledge, the model-based specification community has not systematically considered the problem of choosing an appropriate mathematical model for specifying an ADT. There is the notion of an "unbiased" or "sufficiently abstract" or "fully abstract" model [Jones 90], which is similar to observability in the sense that it is defined almost exactly like $\mathbf{O_0}$. But this informal definition leaves open the possibility of various interpretations, along the lines suggested in Sections 3 and 4, which is precisely the confusion we wish to clear up.

To illustrate these difficulties we use the example in Figure 2 of a possible specification for a Set ADT. Here the appropriate mathematical model seems clear. The question is what operations need to be provided in order to achieve observability and controllability. The specification language is RESOLVE [Bucci 94, Edwards 94, Ogden 94], but the issues involved arise in any model-based specification language [Wing 90].

In RESOLVE, the mathematical model of an ADT is defined explicitly, as with **finite set**; or by reference to a program type, as with **math**[Item], which denotes the mathematical model type of the program type Item. Every program type in RESOLVE carries with it initialization and finalization operations (invoked in a client program through automatically-generated calls at the beginning and end of a variable's scope, respectively), and a swap operation (invoked in a client program using the infix ":=:" operator). The effect of initialization is specified in the **initialization ensures** clause. The effect of finalization usually is not specified because it has no abstract effect; in any event this aspect is unimportant here. The effect of swapping is to exchange the values of its two arguments.

Operation specifications are simplified by using abstract parameter modes **alters**, **produces**, **consumes**, and **preserves**. An **alters**-mode parameter potentially is changed by executing the operation; the **ensures** clause says how. A **produces**-mode parameter gets a new value that is specified by the **ensures** clause, which may *not* involve the parameter's old value (denoted using a prefix "#") because it is irrelevant to the operation's effect. A **consumes**-mode parameter gets a new value that is an initial value for its type, but its old value *is* relevant to the operation's effect. A **preserves**-mode parameter suffers no net change in value between the beginning of the operation and its return, although its value might be changed temporarily while the operation is executing.

The example is simple but it helps to illustrate the nature of the problems facing a specification designer. Is the specification in Figure 2 observable and controllable? What does it mean for two Set values to be "computationally distinguishable", or for a Set value to be "computationally reachable"?

```
concept Set_Template

  context

    global context

      facility Standard_Boolean_Facility
      facility Standard_Integer_Facility

    parametric context

      type Item

  interface

    type Set is modeled by finite set of math[Item]
      exemplar s
      initialization
        ensures  s = empty_set

    operation Insert
        (
          alters s: Set
          consumes x: Item
        )
      requires  x is not in s
      ensures   s = #s union {#x}

    operation Remove
        (
          alters s: Set
          preserves x: Item
        )
      requires  x is in s
      ensures   s = #s - {x}

    operation Is_Member
        (
          preserves s: Set
          preserves x: Item
        ): Boolean
      ensures   Is_Member  iff  (x is in s)

    operation Size
        (
          preserves s: Set
        ): Integer
      ensures   Size = |s|

end Set_Template
```

Figure 2 — Possible Model-Based Specification of a Set ADT

3. Formalizing the Principles

In this section we consider possible interpretations of $\mathbf{O_0}$ and $\mathbf{C_0}$, especially hoping to pin down the phrases “computationally distinguishable” and “computationally reachable”.

3.1. Stating the Principles More Precisely

A big problem with the informal definitions $\mathbf{O_0}$ and $\mathbf{C_0}$ has to do with the possibility of relationally-specified behavior. Although every operation in Figure 1 has functional behavior — the results of each operation are uniquely determined by its inputs — there are many situations where it is appropriate to define an operation so its post-condition can be satisfied in more than one possible way [Weide 94]. A correct implementation might exhibit functional behavior, but a client of the specification cannot count on any particular function being computed — only on the results of each operation satisfying the relation specified in the post-condition.

The practical difficulty this causes in applying $\mathbf{O_0}$ and $\mathbf{C_0}$ is that code layered on top of such a component appears to be non-deterministic, in the sense that it might do something with one implementation of the component but quite another with a different implementation. This is so even when the layered operation is specified to have functional behavior; among other things, the code implementing the layered operation might always terminate with some implementations of the underlying component, but not with others.

ψ_1 When we say “computationally distinguishable” or “computationally reachable”, do we mean for *some* implementation of the specified component, or for *all*?

A strong version of observability is that it should be possible to write a client program that can decide equality of two variables for *every* implementation of the underlying component specification; similarly for controllability. We can formalize this by stipulating the total correctness of certain code layered on top of the specified concept. An implementation of specified behavior is totally correct if it is partially correct (i.e., correct if terminating) *and* terminating, for any totally correct implementations of the components it uses.

We select this path because it leads to the specific principles identified in earlier work [Weide 91], and thereby come to the following possible formalization of observability:

$\mathbf{O_1}$ A specification S defining the program type ADT is observable iff there is a totally correct layered implementation of:

```
operation Are_Equal
     (
       preserves x1: ADT
       preserves x2: ADT
     ): Boolean
```

```
        ensures   Are_Equal  iff  (x1 = x2)
```

Controllability is slightly different in flavor, since as expressed in $\mathbf{C_0}$ it seems to say something about an entire family of operations. It might be formalized as follows:

$\mathbf{C_1}$ A specification S defining the program type ADT is controllable iff for every constant c: **math**[ADT], there is a totally correct layered implementation of:

```
      operation Construct_c
          (
            produces x: ADT
          )
        ensures   x = c
```

3.2. Making the Principles Symmetric: Relative Observability and Controllability

A hint that something lurks below the surface here is the disturbing asymmetry between the definitions $\mathbf{O_1}$ and $\mathbf{C_1}$, the first involving a two-argument program operation and the second a quantified mathematical variable and a one-argument program operation.

ψ_2 Should observability and controllability be defined in terms of relationships between two program variables, or in terms of a program variable and a universally quantified mathematical variable, or perhaps in some other way?

Here we choose the first path, which we took in deriving the principles published earlier [Weide 91] and which *a priori* seems as reasonable as any other. The revision needed for controllability, however, makes it clear that the definition is contingent, or relative, in the following sense. "Computationally reachable" does not mean (as in $\mathbf{C_1}$) that every value in the state space can be constructed from scratch, i.e., starting from an initial value of the ADT. It means that every value in the state space can be reached from every other — even if the given starting point could not itself have been constructed from scratch. Because the meaning of $\mathbf{C_2}$ is now apparently quite different from that of $\mathbf{C_1}$, we add the modifier "relatively" in defining both principles this way:

$\mathbf{O_2}$ A specification S defining the program type ADT is *relatively* observable iff there is a totally correct layered implementation of:

```
      operation Are_Equal
          (
            preserves x1: ADT
            preserves x2: ADT
          ): Boolean
        ensures   Are_Equal  iff  (x1 = x2)
```

$\mathbf{C_2}$ A specification S defining the program type ADT is *relatively* controllable iff there is a totally correct layered implementation of:

```
      operation Get_Replica
```

```
    (
      preserves x1: ADT
      produces x2: ADT
    )
  ensures   x2 = x1
```

These definitions match practical compliance tests of prior work [Weide 91]. But they still have some technical problems.

3.3. Making the Principles More Independent

By definitions $\mathbf{O_2}$ and $\mathbf{C_2}$, relative observability is not entirely independent of relative controllability, since it demands that the arguments to Are_Equal should be preserved and this apparently requires some degree of controllability. Similarly, the first argument to Get_Replica must be preserved and proving this seemingly requires observability, as noted in Section 1.2. Is it possible to define the principles so they are not so evidently connected? The heart of the problem is that both definitions $\mathbf{O_2}$ and $\mathbf{C_2}$ involve preservation of operation arguments. We are, therefore, led to consider this variation:

$\mathbf{O_3}$ A specification S defining the program type ADT is relatively observable iff there is a totally correct layered implementation of:

```
operation Were_Equal
    (
      alters x1: ADT
      alters x2: ADT
    ): Boolean
  ensures   Were_Equal  iff  (#x1 = #x2)
```

This definition is a bit curious because, technically in RESOLVE, a function operation may have only **preserves**-mode parameters; but a violation here seems justifiable for ease of explanation. The parallel definition for relative controllability is:

$\mathbf{C_3}$ A specification S defining the program type ADT is relatively controllable iff there is a totally correct layered implementation of:

```
operation Move
    (
      alters x1: ADT
      produces x2: ADT
    )
  ensures   x2 = #x1
```

3.4. Relationships Among the Above Definitions

Definitions $\mathbf{O_3}$ and $\mathbf{C_3}$ make the principles no stronger than with definitions $\mathbf{O_2}$ and $\mathbf{C_2}$, in the sense that any specification that is relatively observable (controllable) by $\mathbf{O_2}$ (respectively, $\mathbf{C_2}$) is equally so by $\mathbf{O_3}$ (respectively, $\mathbf{C_3}$); it is trivial to layer an implementation of Were_Equal (Move) on top of Are_Equal (respectively, Get_Replica). Furthermore, if a specification is relatively observable by definition $\mathbf{O}_3$ and relatively controllable by definition $\mathbf{C}_2$, then it is relatively observable by definition $\mathbf{O}_2$ because we can layer Are_Equal on top of Get_Replica and Were_Equal:

```
operation Are_Equal
    (
      preserves x1: ADT
      preserves x2: ADT
    ): Boolean
  local context
    variables copy1, copy2: ADT
  begin
    Get_Replica (x1, copy1)
    Get_Replica (x2, copy2)
    return Were_Equal (copy1, copy2)
  end Are_Equal
```

Also note that every RESOLVE specification is relatively controllable by definition $\mathbf{C}_3$, since every type comes with swapping. Here is a universal implementation of Move in RESOLVE:

```
operation Move
    (
      alters x1: ADT
      produces x2: ADT
    )
  begin
    x1 :=: x2
  end Move
```

In effect, a move is half a swap. This is one reason we previously suggested the guideline of testing the stronger criteria $\mathbf{O}_2$ and $\mathbf{C}_2$ [Weide 91]. For components in other languages, however, $\mathbf{C}_3$ is a non-trivial criterion. For example, consider an Ada package defining a Stack ADT as a limited private type (no assignment operator), along with operations Push, Pop, and Is_Empty having the usual meanings. This is relatively controllable by $\mathbf{C}_3$ — but not because a primitive data movement operator for Stacks is trivially assumed. Without any one of the three operations it would *not* be relatively controllable by $\mathbf{C}_3$.

The relationships among the definitions in this section are depicted in the Venn diagram of Figure 3, where we take the liberty of labeling sets of specifications with the labels of the definitions under which their member specifications qualify.

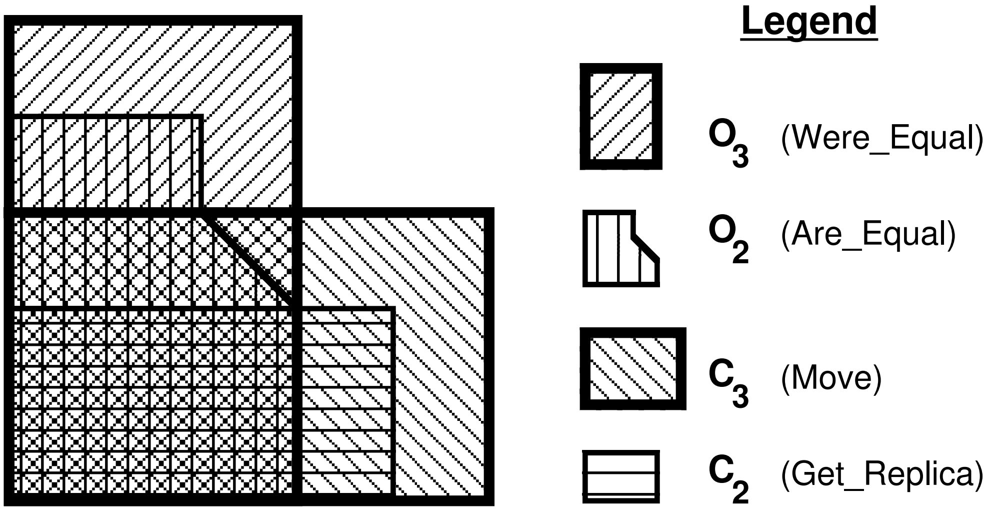

Figure 3 — Relationships Among Some Definitions

4. Difficulties with Parameterized Components

At first the above definitions seem clear and unambiguous. But suppose we try to apply those definitions to the Set_Template specification of Figure 2. It seems the specification in Figure 2 should be deemed *not* observable by $\mathbf{O_0}$ because there is no practical way to enumerate the elements of a Set, and this should be crucial in computationally distinguishing between two unequal Sets. It seems the specification should be deemed controllable by $\mathbf{C_0}$, however, because starting from an empty set it is easy to construct any finite set by repeated Inserts. Does this intuition match what the proposed definitions say? We discuss in detail only $\mathbf{O_3}$, considerations for the other definitions being similar.

4.1. Type Parameters and Modular Proofs

There is a reasonable way to interpret $\mathbf{O_3}$ that makes the Set_Template specification observable. The key features that permit this view are that $\mathbf{O_3}$ defines relative observability in terms of the existence, not the practicality, of an implementation of Were_Equal; and that there is no restriction on the assumptions an implementer of Were_Equal may make about the available operations on Items.

We start by noting that the mandated existence of “a totally correct layered implementation” of the Were_Equal operation for Set_Template means, in RESOLVE terms, the existence of a totally correct implementation of the following concept:

```
concept Set_Were_Equal_Capability

  context

    global context

      facility Standard_Boolean_Facility
      concept Set_Template

    parametric context

      type Item

      facility Set_Facility is Set_Template (Item)

  interface

    operation Were_Equal
        (
          alters s1: Set
          alters s2: Set
        ): Boolean
      ensures   Were_Equal  iff  (#s1 = #s2)

end Set_Were_Equal_Capability
```

This formulation makes clear that the implementation of Were_Equal must be layered, since an instance of Set_Template is a parameter to the concept. Moreover, it makes clear that the implementation must work for any type Item for the Set elements, since Item also is a parameter. What it does not make clear, however, is what other components and services an implementation might use and depend on.

In the absence of restrictions, presumably any such services may be assumed — a rather liberal interpretation of $\mathbf{O_3}$. But now what prevents an implementer of Were_Equal from simply assuming the existence of a (possibly thinly disguised) operation that tests equality of Sets of Items, and layering on top of that? Nothing.

So we might wish to use a less liberal interpretation of $\mathbf{O_3}$. For example, suppose we insist that an allowable implementation of Were_Equal may not use any operations with Set parameters other than those from Set_Template itself. Unfortunately, this does not solve the problem either. For example, below is a possible algorithm for Were_Equal, which is built on top of Set_Template and an "enumerator" concept for Items. In RESOLVE's modular proof system, total correctness is defined in such a way that the following code is a totally correct implementation of Were_Equal, because we *assume* there is a totally correct implementation of the enumerator interface and the total correctness of the Set_Template implementation — and because all Sets are finite. As a result we claim that Set_Template is relatively observable even by this less liberal interpretation of $\mathbf{O_3}$.

```
operation Were_Equal
        (
          alters s1: Set
          alters s2: Set
        ): Boolean
      local context
        variables x: Item
      begin
        if (Size (s1) = 0 and Size (s2) = 0)
          then return true
          else
            let x = any Item value not previously enumerated during
                    the top-level call of Were_Equal
            if Is_Member (s1, x)
              then
                if Is_Member (s2, x)
                  then
                    Remove (s1, x)
                    Remove (s2, x)
                    return Were_Equal (s1, s2)
                  else return false
                end if
              else
                if Is_Member (s2, x)
                  then return false
                  else return Were_Equal (s1, s2)
                end if
            end if
        end if
      end Were_Equal
```

This illustrates the power of a modular proof system [Ernst 94]. There might be Items for which it is impossible to implement the enumerator interface, but this does not influence the total correctness of Were_Equal. At the mathematical level, if the state space **math**[Item] is effectively enumerable then in principle there exists an implementation of the enumerator interface. But only if the specification of the actual program type Item is at least controllable, by any reasonable definition, should we expect to be able to implement the enumerator interface for it.

So perhaps we should insist that the underlying components actually should be implementable. But then should the mere possibility of instantiating Set_Were_Equal_-Capability with an Item for which the enumerator cannot be implemented be enough to render the Set_Template specification not observable? And does "possibility" here mean the library of components *actually contains* such a type, or that in principle it *might contain* such a type? Suppose, for example, that in the specification language it is simply impossible to specify a program type whose state space is not enumerable. Should this situation — which might be reasonably attributed to inexpressiveness of the specification language and not to a problem with the design of Set_Template — be the deciding factor as we attempt to apply the observability test to Set_Template?

If we use an interpretation in which the above implementation of Were_Equal is acceptable, so Set_Template is deemed relatively observable, then it is interesting to see where variants of Set_Template lie in Figure 3. In Figure 4, we have placed some of them to illustrate the limited discriminating power of the definitions. For example, Get_Replica for Sets can be layered on top of Are_Equal for Sets using only Swap and Insert: systematically generate candidate Sets by enumerating Items and inserting them into empty Sets — first one Set with one Item, then two Sets with one Item and two Sets with two Items, and so forth — stopping when the Set to be copied and the current candidate Are_Equal. There is no need for Remove, Is_Member, or Size.

It should be clear that these definitions are not really "right", in the sense that even if they do capture some sense of observability and controllability they do not rule out patently poor specifications. For example, Set_Template itself (even without Swap) is both relatively observable and relatively controllable by the strong definitions $\mathbf{O_2}$ and $\mathbf{C_2}$, despite providing no practical way to enumerate the elements of a Set. Even Set_Template without Remove is both relatively observable and relatively controllable, as is Set_Template with just Swap, Insert, and Are_Equal.

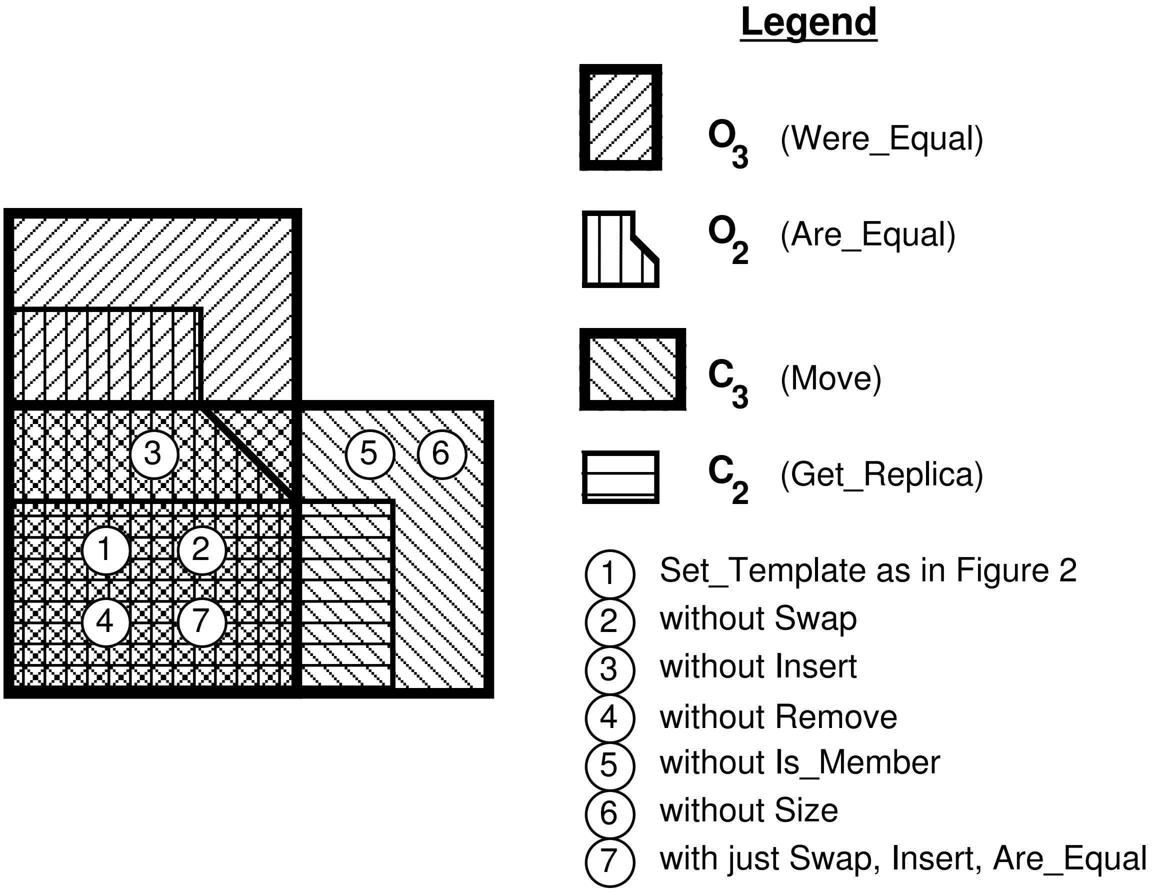

Figure 4 — Variants of Set_Template Assuming **math**[Item] is Enumerable

4.2. Refining the Definitions to Handle Parameterized Components

The difficulties in Section 4.1 are traceable to the prospect of having specifications that are parameterized by another type Item, and to the absence of restrictions on the assumptions an implementation may make about the actual Item type. Even allowing an implementation of Were_Equal to rely only on the assumption that the state space of Item is enumerable weakens the definitions so much that they are practically worthless.

Some features of RESOLVE permit us to easily clarify and strengthen the previous definitions to deal with parameterized modules, so the observability of a parameterized type is unaffected by properties of the arbitrary type by which it is parameterized. Each realization (implementation) of a concept may require additional parameters beyond those of the concept, and these appear in the realization "header" [Bucci 94]. This mechanism lets us require that the implementation of an operation Were_Equal for type Set may only count on the always-present initialization, finalization, and swapping for Items, *and* on a similarly-defined Items_Were_Equal operation. Any allowable realization of the concept exporting Were_Equal should have a realization header in which this additional operation is the only realization parameter.

This leads to a refined definition of relative observability (the others being similar):

$O_{3'}$ A specification S, parameterized by the program type Item and defining the program type ADT, is relatively observable iff there is a totally correct implementation of:

```
concept S_Were_Equal_Capability

  context

    global context

      facility Standard_Boolean_Facility
      concept S

    parametric context

      type Item

      facility S_Facility is S (Item)

  interface

    operation Were_Equal
        (
          alters x1: ADT
          alters x2: ADT
        ): Boolean
      ensures   Were_Equal  iff  (#x1 = #x2)

end S_Were_Equal_Capability
```

whose realization context makes only the following additional mention of Item:

```
realization header Allowed for S_Were_Equal_Capability

  context

    parametric context

      operation Items_Were_Equal
        (
          alters x1: Item
          alters x2: Item
        ): Boolean
      ensures   Were_Equal  iff  (#x1 = #x2)

end Allowed
```

In applying this definition to Set_Template, we find there is no way for the realization body of Set_Were_Equal_Capability to use any externally-provided operations involving Items, other than Items_Were_Equal. This rules out impractical but technically correct implementations like the one in Section 4.1.

Figure 5 is the counterpart of Figure 4, with the refined definitions. Now Set_Template is not relatively observable by $\mathbf{O}_{\mathbf{2'}}$ or by $\mathbf{O}_{\mathbf{3'}}$, nor relatively controllable by $\mathbf{C}_{\mathbf{2'}}$. However, by adding the following operation (or something similar) it becomes relatively observable and relatively controllable even by $\mathbf{O}_{\mathbf{2'}}$ and $\mathbf{C}_{\mathbf{2'}}$:

```
operation Remove_Any
    (
      alters s: Set
      produces x: Item
    )
  requires  s /= empty_set
  ensures   (x is in #s)  and  (s = #s - {x})
```

Remove_Any (s, x) removes an arbitrary element of the original s and returns it in x. Now there is a practical way to enumerate the elements of a Set, leading to obvious implementations of the required layered operations that assume no more than the ability to do with Items what the layered operation is doing to Sets.

Figure 5 shows what happens to the variants of Set_Template previously displayed in Figure 4 (circles 1-7). Two new variants help to illustrate the discrimination power of the new definitions. Set_Template with Remove_Any (circle 8) — a good design — passes both of the stronger compliance tests $\mathbf{O}_{\mathbf{2'}}$ and $\mathbf{C}_{\mathbf{2'}}$. Set_Template with Remove_Any but without Insert (circle 9) — plainly not a good design — still passes both weaker tests $\mathbf{O}_{\mathbf{3'}}$ and $\mathbf{C}_{\mathbf{3'}}$ but neither stronger one. So the definitions used for Figure 5 seem better than those used for Figure 4. But again even $\mathbf{O}_{\mathbf{2'}}$ and $\mathbf{C}_{\mathbf{2'}}$ clearly are not “right” in that they still do not rule out patently bad specifications. It is easy to circumvent their intent by attacking the symptoms and not the disease: just add Are_Equal and Get_Replica as primary operations. In fact, Set_Template with just Are_Equal and Get_Replica and no other operations whatsoever sits in precisely the same place in Figure 5 as Set_Template with Remove_Any, despite clearly not satisfying $\mathbf{C_1}$. Fixing these problems apparently requires taking a different path altogether, as we discuss in the conclusions below.

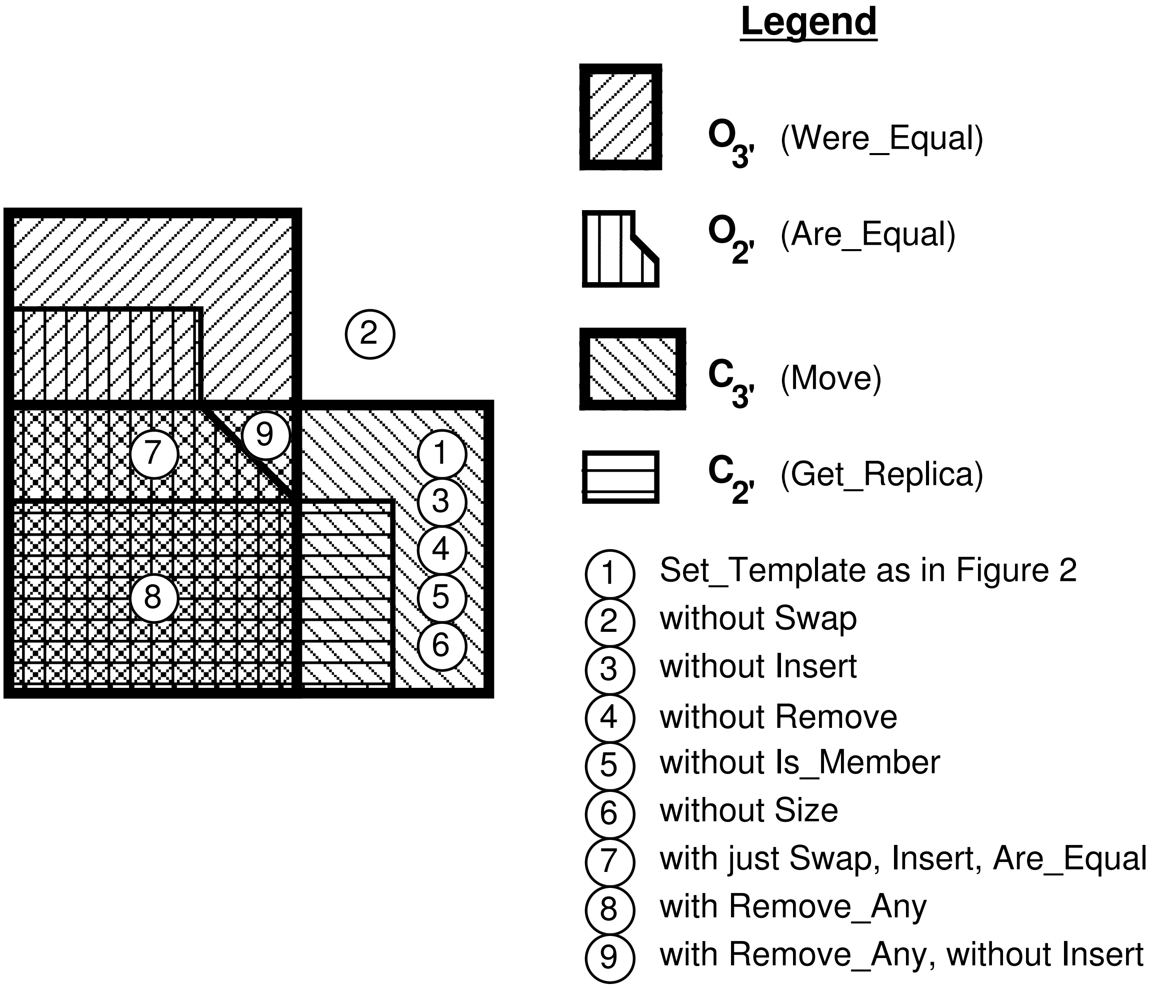

Figure 5 — Variants of Set_Template With Section 4 Definitions

5. Conclusions

A fundamental question facing the designer of a model-based specification of an ADT is the appropriateness of the chosen conceptual model. We have discussed some of the technical problems in carefully defining two principles that provide the specifier with criteria for appropriateness: Does the chosen model interact with the specified operations in a way that makes the specification observable and controllable? A negative answer on either count suggests that the specifier needs to look harder, or be prepared to justify non-compliance on the basis of other requirements. A positive answer on both counts gives a certain confidence, though among satisfactory specifications some may be “better” than others (e.g., more understandable or more flexible). However, it hardly guarantees that the specification is “good”.

We mentioned alternate paths that might be followed in formalizing observability and controllability. Here are some conclusions from preliminary exploration of other paths; these conclusions are not justified in the body of this paper.

ψ_1 When we say "computationally distinguishable" or "computationally reachable", do we mean for *some* implementation of the specified component, or for *all*?

Defining the principles using an existential quantifier over implementations is largely unexplored territory. However, there is reason to believe it might be attractive. Consider, for example, the specification of an ADT called Computational_Real modeled as a real number. The operations have relationally-defined behavior. The Add operation, for example, ensures that the result of adding two Computational_Reals is a Computational_Real whose model lies within some small interval around the sum of the models of the addends. Based on a cardinality argument, it is clear there is no way the specification can be deemed controllable if we insist that *every* implementation of it *must* support reaching every real number. However, it is not hard to imagine Computational_Real operations powerful enough to allow that every real number *might* be reachable in *some* implementation; basically, the union of the allowed intervals over all computations just has to cover the reals. The power of relationally-specified behavior is evident here, but the full implications of defining observability and controllability as suggested are not.

ψ_2 Should observability and controllability be defined in terms of relationships between two program variables ("relatively"), or in terms of a program variable and a universally quantified mathematical variable, or perhaps in some other way?

Defining both principles the second way leads to interesting phenomena and to other interesting questions involving the expressiveness of the mathematics and the relationships between those definitions and the ones in this paper. Observability basically becomes a test of whether, for every point in the state space, it is possible to tell whether a program variable Was_Equal to it. Controllability is more properly termed "constructability", using something like definition $\mathbf{C_1}$. These alternate definitions cut through diagrams like Figures 3-5 in a surprising way, since there are specifications that are observable and/or controllable by the alternate definitions but not by $\mathbf{O_{2'}}$ and/or $\mathbf{C_{2'}}$, and vice versa. So such definitions might offer distinct useful tests which should be applied in tandem with the ones described here, when evaluating a proposed specification.

6. Acknowledgment

We thank Murali Sitaraman and Stu Zweben for insightful comments on a draft of this paper, and gratefully acknowledge financial support for our research from the National Science Foundation under grant CCR-9311702, and from the Advanced Research

Projects Agency of the Department of Defense under ARPA contract number F30602-93-C-0243, monitored by the USAF Materiel Command, Rome Laboratories, ARPA order number A714.

7. References

[Bucci 94] Bucci, P., Hollingsworth, J.E., Krone, J., and Weide, B.W., "Implementing Components in RESOLVE," *Software Engineering Notes 19*, 4, October 1994, 40-52.

[Edwards 94] Edwards, S.H., Heym, W.D., Long, T.J., Sitaraman, M., and Weide, B.W., "Specifying Components in RESOLVE," *Software Engineering Notes 19*, 4, October 1994, 29-39.

[Edwards 95] Edwards, S.H., *A Formal Model of Software Subsystems*, Ph.D. dissertation, Dept. of Computer and Information Science, The Ohio State Univ., Columbus, March 1995.

[Ernst 94] Ernst, G.W., Hookway, R.J., and Ogden, W.F., "Modular Verification of Data Abstractions with Shared Realizations," *IEEE Transactions on Software Engineering 20*, 4, April 1994, 288-307.

[Goguen 78] Goguen, J.A., Thatcher, J.W., and Wagner, E.G., "An Initial Algebra Approach to the Specification, Correctness, and Implementation of Abstract Data Types," in *Current Trends in Programing Methodology 4*, R. T. Yeh, ed., Prentice-Hall, 1978, 80-149.

[Guttag 78] Guttag, J.V., Horowitz, E., and Musser, D.R., "Abstract Data Types and Software Validation," *Communications of the ACM 21*, 12, December 1978, 1048-1064.

[Guttag 93] Guttag, J.V., and Horning J.J., *Larch: Languages and Tools for Formal Specification*, Springer-Verlag, 1993.

[Jones 90] Jones, C.B., *Systematic Software Development Using VDM*, 2nd ed., Prentice-Hall, 1990.

[Liskov 75] Liskov, B.H., and Zilles, S.N., "Specification Techniques for Data Abstractions," *IEEE Transactions on Software Engineering SE-1,* 1, March 1975, 7-19.

[Liskov 86] Liskov, B., and Guttag, J., *Abstraction and Specification in Program Development*, McGraw-Hill, 1986.

[Norman 90] Norman, D.A., *The Design of Everyday Things*, Doubleday/Currency, 1990.

[Ogden 94] Ogden, W.F., Sitaraman, M., Weide, B.W., and Zweben, S.H., "The RESOLVE Framework and Discipline — A Research Synopsis," *Software Engineering Notes 19*, 4, October 1994, 23-28.

[Sitaraman 93] Sitaraman, M., Harms, D.E., and Welch, L.W., "On Specification of Reusable Software Components," *International Journal of Software Engineering and Knowledge Engineering 3*, 2, June 1993, 207-229.

[Spivey 89] Spivey, J.M., *The Z Notation: A Reference Manual*, Prentice-Hall, 1989.

[Weide 91] Weide, B.W., Ogden, W.F., and Zweben, S.H., "Reusable Software Components", in *Advances in Computers, vol. 33*, M.C. Yovits, ed., Academic Press, 1991, 1-65.

[Weide 94] Weide, B.W., Ogden, W.F., and Sitaraman, M., "Recasting Algorithms to Encourage Reuse," *IEEE Software 11*, 5, September 1994, 80-88.

[Wing 90] Wing, J.M., "A Specifier's Introduction to Formal Methods", *Computer 23*, 9, September 1990, 8-24.