1997

Foundations of object-based specification design

David March Fleming
West Virginia University

Foundations of Object-Based Specification Design

By

David March Fleming, B.S., M.S.

DISSERTATION

Submitted to
The Eberly College of Arts and Sciences
at
West Virginia University

in partial fulfillment of the requirements
for the degree of

Doctor of Philosophy
in
Computer Science

Department of Computer Science
and Electrical Engineering

Morgantown, West Virginia
1997

ABSTRACT

Foundations of Object-Based Specification Design

By David March Fleming, B.S., M.S.

To enhance applicability and encourage its use, a component or a component-based system must have a well-designed set of interface features as well as a proper explanation of these features. The dual problem of designing a suitable set of interface features in addition to properly explaining its behavior is termed the *specification design problem*. This dissertation identifies *observability*, *controllability*, and a performance-motivated *pragmatic criterion* as essential properties of desirable formal specifications for reusable object-based software components. The pragmatic criterion guides the design of component interfaces and component libraries to a suitable set of features so that they are widely applicable, both in terms of functionality and performance, yet minimal in size, whereas observability and controllability considerations lead to most suitable formal explanations of the interfaces.

This dissertation formally defines the principles of observability and controllability for object-based software specifications, including those with relational behavior. These principles, in addition to the minimality and performance considerations embodied in the pragmatic criterion, lead to the unique collection of concepts in the RESOLVE component specification library. These principles form a basis for evaluation of existing object-based software specifications, and also lead to designs of new specifications that are among the most desirable in terms of understandability and utility.

To my wife,
Monica Vint Fleming,
for all her support and patience.

And to my parents,
Jim and Shirley Fleming,
for all their love and guidance.

ACKNOWLEDGMENTS

My deepest thanks goes to my advisor, Murali Sitaraman. Without your seemingly endless interest, guidance, support, and patience with both this research and myself, finishing would have been impossible. I am also extremely grateful for the other members of my committee, John Atkins, Srinivas Kankanahalli, Timothy Long, and George Trapp. Their continual support, encouragement, and participation helped make this possible.

I also wish to thank Bill Ogden and Bruce Weide for their insightful comments and conversations. I also thank those students with whom I have worked, both past and present, for the shared interest in and many discussions about this research. Especially Brad, Narasimha, and Iwona (the “other Ph.D. students”). And to Brenda and Mickie, thanks for answering all of my irritating questions about this whole “Ph.D. process”. ☺

A special thanks to my parents for buying all of those Commodore computers and peripherals (I kept wearing them out!), and for not disturbing me come dinner-time when I was busy playing with them. I never ran out of things to learn and do, and I am sure my experience with them is why I am here today. By the way, does anyone have a 1541 disk drive? Mine is broken.

I am thankful for my wife, for her love, patience, belief, and support for me. Without that, I would never have finished. And without her, I would be single. ☺

I give thanks to our cats, Meg and Katie, and our guinea pig Fuzzy, for providing me with much-needed (and all too frequent) breaks from my work by leaving little messes for me to clean up.

To my friends, the Greens, thanks for all your support and encouragement. Sonny, one of these days I’ll beat you in tennis, but I’ll probably graduate first.

This research was funded in part by the Defense Advanced Research Projects Agency (DARPA) under contract number DAAH04-94-G-0002, monitored by the U.S. Army Research Office.

VITA

March 21, 1969	Born – Hattiesburg, Mississippi.
1992	B.S. Mathematics/Computer Science, Concord College, Athens, West Virginia.
1994	M.S. Computer Science, West Virginia University.
1993-1997	Graduate Research Assistant, West Virginia University.

Publications

D. Fleming, S. Sreerama, and M. Sitaraman, "A Practical Performance Criterion for Object Interface Design", *Journal of Object Oriented Programming*, Vol. 10, No. 4, July/August 1997, pp. 52-63.

S. Sreerama, D. Fleming, and M. Sitaraman, "Graceful Evolution of Performance in Object-Based Software", *Software Practice and Experience*, Vol. 27, No. 1, January 1997, pp. 111-122.

D. Fleming and J. Wagner, "Communicating Precise Object Interface Behavior in Complex Domains", in *Proceedings of the Eighth Annual Workshop on Software Reuse,* Columbus, OH, 1997.

M. Sitaraman, D. Fleming, J. Hopkins, and S. Sreerama, "Why (Not) Reuse (Typical) Code Components?", in *Proceedings of the Seventh Annual Workshop on Software Reuse*, St. Charles, Ill, 1995.

Fields of Study

Major Field: Computer Science
Studies in:

Software Engineering	Prof. Murali Sitaraman
	Prof. Timothy Long
Computability Theory	Prof. John Atkins
Numerical Analysis	Prof. George Trapp
Algorithmic Analysis	Prof. Srinivas Kankanahalli

TABLE OF CONTENTS

LIST OF FIGURES

Introduction I

To enhance applicability and encourage its use, a component or a component-based system must have a well-designed set of interface features as well as a proper explanation of these features. If the interface does not include suitable operations for effective manipulation of objects defined by that interface, then it might compromise functional and/or performance flexibility, thereby inhibiting its reuse. Alternatively, poor explanations of an otherwise well-conceived interface might make it impossible to understand its objects and operations, and also inhibit its use. This interconnected problem of designing an interface that provides a suitable set of features along with an appropriate formal explanation is termed the *specification design problem*, and it is the focus of this dissertation.

For a given problem, the specification design space is vast, and only a few of the designs simultaneously facilitate understanding and reasoning at the right level of abstraction, provide widely applicable functionality, and encourage performance flexibility through alternative implementations. The objective of this dissertation is to identify and describe a formal basis for evaluating alternative specifications of the same problem to narrow the design space to contain only the best ones. This objective, in the context of component-based *software* systems, is complementary to yet distinguished from much of the design work in the software engineering literature, where the principal focus is on implementations.

1.1 Fundamental Properties Underlying Good Specification Designs

To understand and motivate what might be, for any component-based device or system, potentially fundamental properties of good specifications, we consider the specification design problem for two physical devices: a water faucet and a stove burner [Norman 90]. In each case, we explain that there is something intuitively wrong about the initial designs with respect to how we use and reason about them. We are able to make explicit the problems in each case, and in the process, some of the essential properties of interface designs and explanations come into focus.

Figure 1.1 shows a design of an interface for a water faucet that is completely inflexible. In this design, the water can be turned on and off, but the lever cannot be moved left or right; i.e., the water faucet provides water of only one temperature. Clearly, this design is not widely usable or *pragmatic*.

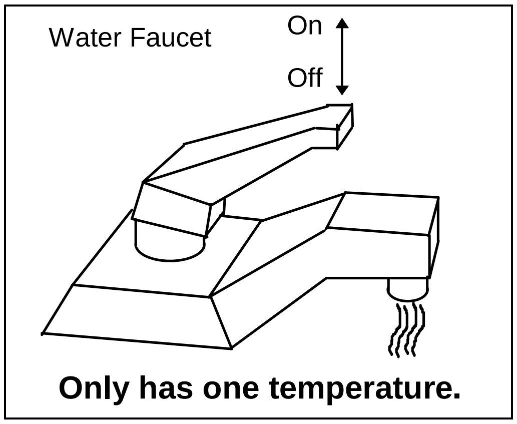


Figure 1.1 - Not Pragmatic

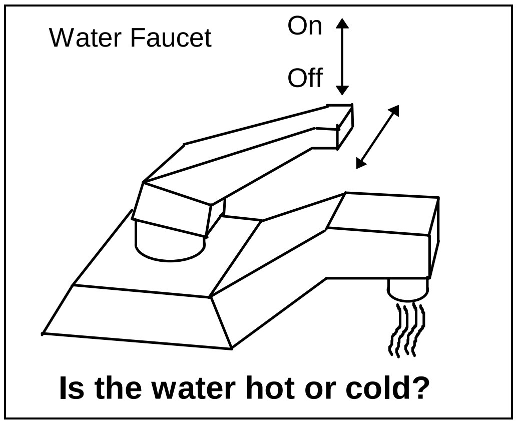


Figure 1.2 - Not Observable

While Figure 1.2 shows an interface for a water faucet in which the lever can now be moved left or right, it provides no visible relationship between the lever's position and the water's temperature. That is, this interface provides flexibility, but it does not include an explanation of how to use the lever to benefit from the flexibility. Stated in control engineering terms, this interface lacks *observability*. The lack of observability in this case forces a user to try to move the lever both ways to get water of the desired temperature (though in the real world, this problem is often avoided by the convention that "left is cold" and "right is hot").

The "normal" design in Figure 1.3 is both observable and pragmatic, because it provides means and instruction for hot and cold water. This discussion suggests that good interfaces for (software) components and systems should be both flexible and well-explained to be widely applicable and usable.

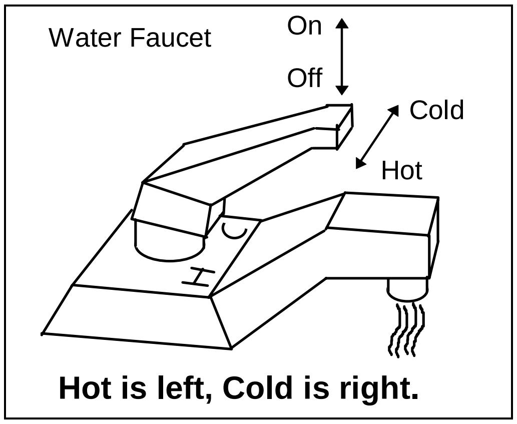


Figure 1.3 - A Good Design

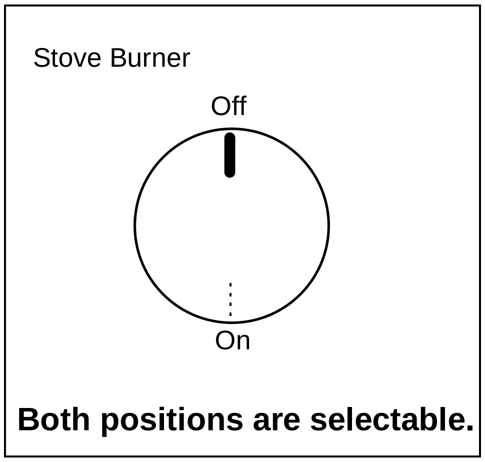


Figure 1.4 - Not Pragmatic

Figure 1.4 illustrates another example. This interface for a stove burner allows the burner to be turned on or off, but otherwise offers no finer control of the burner's temperature. This interface, again, is not widely usable or pragmatic.

Figure 1.5 contains an alternative interface for the stove burner that apparently provides finer control of the burner's temperature. However, suppose that only "off" and "medium" can actually be selected. In other words, the explanation of the interface and the provided features of the interface do not match, since there are states in the explanation that cannot be reached. Stated in control engineering terms, this interface design is not *controllable*.

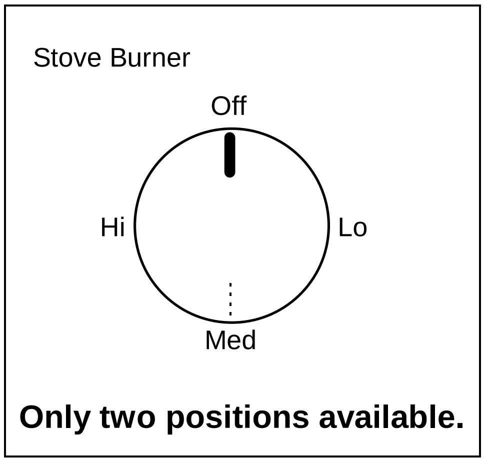


Figure 1.5 - Not Controllable

Although somewhat simplistic, the above examples show the intuitive nature of three arguably fundamental (and formalizable) properties of specification design:

observability, *controllability*, and the *pragmatic criterion*. While not obvious, similar considerations have influenced the interface design of every physical device from batteries to cars and planes.

For ease of use and practicality, a software component interface should also be observable, controllable, and satisfy a version of the pragmatic criterion concerned with functional and performance flexibility. Additionally, a library of software components for some problem domain should also satisfy the pragmatic criterion. Here, we explore the meaning of these terms for software and argue that they are fundamental for designing formal model-based specifications for object-based software components. When designing an interface for a single software component, observability and controllability ensure the development of an operation set *no smaller than is sufficient* to express computations on the entire model space, whereas the pragmatic criterion ensures the development of an operation set *no larger than is necessary* to be efficient for all intended functional variations of the component. In other words, observability and controllability approach the design of a suitable specification "from the bottom", whereas the pragmatic criterion approaches the design "from the top". Additionally, when designing a library of components for some problem domain, the pragmatic criterion ensures that the library is no larger than necessary for this purpose. Understanding and applying these terms and principles for behavioral specifications of software components and systems is the central topic of this dissertation.

1.2 Software Specification Design

For precisely explaining behaviors of software components, there are a number of formal specification notations (VDM, Larch, RESOLVE, and Z, among others). However, few of these notations include techniques, principles, and guidelines for specifiers to use in developing high-quality interface specifications [Wing 90, Sitaraman 93]. Without such guidance, even the best notations cannot compensate for poorly conceived specification designs. For example, while it can be argued that programming language notations can be used to formally describe the behavior of software, such "specifications" are often very complex and too detailed. The promise and attraction of formal specification notations for describing software components and systems comes from the ability to describe the behavior of software at the right level of abstraction for understanding and reasoning, and this is where more than just a formal notation is needed.

In addition to the design of a clear and precise explanation of behavior, the interface specification design process for a given problem involves the design of a suitable set of operations for manipulating objects provided by the specification. Unfortunately, in the formal specification community, the focus is mostly on precise notation. Likewise, and equally unfortunate, issues surrounding the clear and precise explanation of interface behavior are rarely the focus in the practicing object-oriented community. This dissertation fills this gap by providing a foundation for the formal specification design problem.

Some Desirable Properties

In one of the earlier characterizations of desirable specifications, Liskov and Guttag identify three key properties: clarity, restrictiveness, and generality [Liskov 86]. Here, clarity means that a specification should be understandable; restrictiveness means that a specification should express all that a designer intends, and nothing more; and generality means that a specification should be sufficiently abstract so as not to preclude any suitable implementation strategy. Weide et al. summarize these and other good characteristics of object-based software component specifications as listed below [Weide 91]:

- **Clarity** - a specification should be clear and understandable.
- **Restrictiveness** - a specification should state everything a designer wishes to state about the behavior that is expected of a correct implementation, and nothing more.
- **Generality** - a specification should support a variety of implementations, especially efficient ones.
- **Primitiveness** - a specification should export operations whose functionality is orthogonal and whose totality is minimal.
- **Sufficiency** - a specification should export operations that collectively offer enough functionality for a wide class of computations.
- **Potential completeness** - a specification should not export operations that can be layered using the primitive operations.
- **Low coupling** - a specification should be explained completely locally, not depending on the explanations of other components (unless it is an extension of another component).
- **High cohesion** - a specification should specify a component that cannot be further decomposed; e.g., a single data type.

While properties such as the ones above are important design goals for specifiers, it is not obvious how to formalize these properties so that they can provide a suitable basis for the evaluation of specification designs. In addition, since the specification design task needs to be performed by humans, the large number of inter-related properties makes it hard to apply them carefully and deliberately to specification designs. What is needed are more fundamental notions that will largely imply the above desirable properties, but that lend themselves to more formal definition and application. Essentially, such notions should be inherently easy to understand and be attractive for use in a variety of situations. In principle, they would apply to any interface, not necessarily to software component interfaces alone. It turns out that the notions of observability, controllability, and the pragmatic criterion can deliver on these points.

How do the three properties of observability, controllability, and the pragmatic criterion encompass or subsume the more general software engineering properties such as clarity, restrictiveness, and others mentioned above? In other words, what are the relationships in Figure 1.6? In answering these and related questions, we formalize the basic principles and validate their utility by applying them to a collection of specifications.

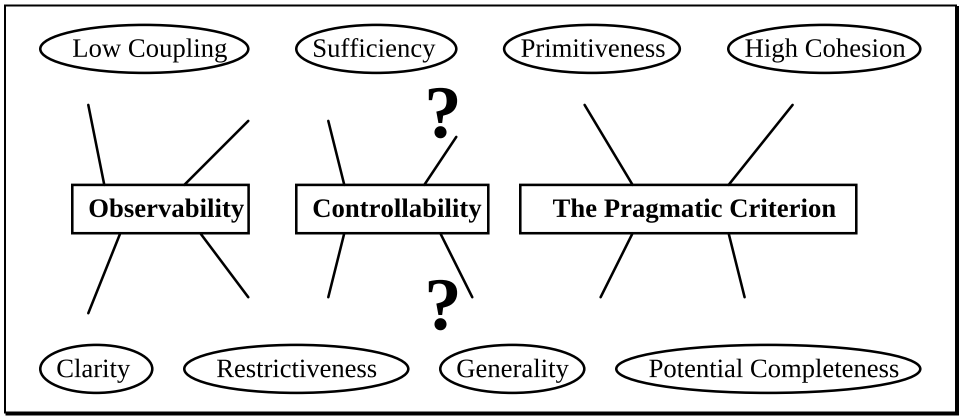


Figure 1.6 - What are the Relationships?

1.3 Exemplifying the Ideas in Software Component Specification Design

The purpose of this section is to illustrate the influences of observability, controllability, and the pragmatic criterion on formal specifications of object-based software components. Figure 1.7 below shows a common generic queue specification, or *concept*, in the RESOLVE notation [SIGSOFT 94] . There are other formal specification notations that would be equally appropriate for the purpose [Wing 90].

```
concept Bounded_Queue_Template (type Entry,
                               constant Max_Length: Integer)
      requires Max_Length > 0
      uses Standard_Integer_Facility

   type family Queue is modeled by string of Entry
      exemplar q
      initialization ensures
         |q| = 0
      constraints
         |q| <= Max_Length

   operation Enqueue (alters q: Queue, preserves x: Entry)
      requires |q| < Max_Length
      ensures  q = #q * <#x>

   operation Dequeue (alters q: Queue, produces x: Entry)
      requires |q| > 0
      ensures  #q = <x> * q

   operation Length_Of (preserves q: Queue): Integer
      ensures  Length_Of = |q|

   operation Allowed_Max_Length (): Integer
      ensures  Allowed_Max_Length = Max_Length

end Bounded_Queue_Template
```

Figure 1.7 - A Common Design of a Queue Specification

The concept in Figure 1.7 is parameterized by Entry, which is the type of entries that a Queue object contains, and Max_Length, which is the upper bound on the lengths of Queue objects. These parameters are supplied at instantiation time by a client.

The exported *type family* of Queue objects are modeled by mathematical strings of (mathematical models of) type Entry. The *initialization ensures* clause states that initially, every queue corresponds to an empty string. The interface exports Enqueue, Dequeue, Length_Of, and Allowed_Max_Length operations to manipulate queues. The mode of the parameter q in the Enqueue operation is *alters*, which indicates that q is changed as specified in the ensures clause. The *requires* clause for Enqueue states that the length of the queue must be less than Max_Length. The *ensures* clause for Enqueue states that the resulting value for the queue object q will be the incoming value of the queue (#q) concatenated with the incoming value of the entry (#x); i.e., x will be placed on the right end of the string that models the incoming q. The parameter ‘x’ is preserved;

i.e., its outgoing value will be the same as its incoming value. The *requires* clause for Dequeue states that it must not be called on an empty queue. The parameter mode for x is *produces*, which indicates that the incoming value of x is not of interest (nor can it be used) in the specification of Dequeue. The *ensures* clause states that the resulting value for the queue will be the incoming value (#q) less the leftmost element, which is returned in x. The Length_Of operation returns the number of elements in the queue. The Allowed_Max_Length operation returns the value of the generic Max_Length parameter used in instantiation.

In RESOLVE, every object-based concept is designed to include the operations Swap and Clear. The motivation for including the Swap operation is one of efficient data movement, and it is discussed in detail elsewhere [Harms 91]. It allows a client to exchange two (queue) values. The Clear operation resets a (queue) value to its initial value as specified in the *initialization ensures* clause.

Observability and Controllability Considerations

The interface design and explanation used in the specification of bounded queues in Figure 1.7 above are good in many respects[1]. The specification is clear and understandable (clarity), utilizing simple string theory to explain its actions. It abstractly states everything about a queue, and nothing more, that is expected for proper usage (restrictiveness). The exported operations are orthogonal and minimal (primitiveness). The exported operations are sufficient to allow any interesting computations involving queues (sufficiency). The concept does not export operations that can be layered on the others (potential completeness). Nor does it depend on the explanation of any other data structures (low coupling).

The specification in Figure 1.7 is also observable and controllable. It is observable since it is possible to distinguish between different abstract (string) values using the provided operations. Specifically, the operations Length_Of and Dequeue can be used to detect a difference between two values. It is controllable since it is possible to generate any abstract (string) value using the provided operations; i.e., through repeated calls to Enqueue.

[1] It does have problems with generality and high cohesion, which are addressed in the context of the pragmatic criterion in the next subsection.

The fact that the specification in Figure 1.7 is observable and controllable, and simultaneously fulfills many of the basic properties, is not coincidental. For example, if we removed the Dequeue operation, it would not be possible to distinguish between dissimilar queues of the same length and hence, the specification would become non-observable. If we instead removed the Enqueue operation, it would not be possible to generate every abstract value and hence, the specification would become non-controllable. In either case, the specification would also fail the property of *sufficiency*, as it would no longer be capable of performing many computations on queue values. In other words, it is reasonable to make the following observation:

1. There is a strong connection between *sufficiency,* and observability and controllability.
2. The set of operations provided by a specification at least in part determines whether a specification is observable and controllable.

More importantly, however, observability and controllability have more to do with the development of good formal specifications than just the selection of a suitable set of operations. For example, consider the alternative queue specification in Figure 1.8 below.

```
concept Bounded_Queue_Template (type Entry,
                               constant Max_Length: Integer)
       requires Max_Length > 0
       uses Standard_Integer_Facility

    type family Queue is modeled by (
          contents :  function from integer to Entry,
          front :     integer,
          length :    integer
        )
       exemplar q
       constraints
          0 <= q.front < Max_Length  and
          0 <= q.length <= Max_Length

       initialization ensures
          q.front = 0 and q.length = 0
```

```
   operation Enqueue (alters q: Queue, preserves x: Entry)
      requires q.length < Max_Length
      ensures  q.contents(#q.front) = #x  and
               q.front = (#q.front + 1) mod Max_Length and
               q.length = #q.length + 1   and

         for all i: integer,
            (i >= 0 and i <= #q.length-1) implies

         q.contents( (i + #q.front - #q.length + Max_Length)
            mod Max_Length
         ) =
         #q.contents((i + #q.front - #q.length + Max_Length)
            mod Max_Length
         )

   operation Dequeue (alters q: Queue, produces x: Entry)
      requires q.length > 0
      ensures
         x = #q.contents(
            (#q.front-#q.length+Max_Length) mod Max_Length
         )  and
         q.front = #q.front and q.length = #q.length-1 and

         for all i: integer,
            (i >= 1 and i <= #q.length-1) implies

         q.contents( (i + #q.front - #q.length + Max_Length)
            mod Max_Length
         ) =
         #q.contents((i + #q.front - #q.length + Max_Length)
            mod Max_Length
         )

   operation Length_Of (preserves q: Queue) : Integer
      ensures  Length_Of = q.length

   operation Allowed_Max_Length (): Integer
      ensures  Allowed_Max_Length = Max_Length

end Bounded_Queue_Template
```

Figure 1.8 - An Insufficiently Abstract Queue Specification

The specification in Figure 1.8 above has precisely the same operation set as that in Figure 1.7, and every valid implementation of the design in Figure 1.7 is also a valid implementation of the one in Figure 1.8 and vice versa. However, it is neither observable nor controllable. The generic queue specification in Figure 1.8 attempts to explain the behavior of a queue as it would be seen when viewed as a circular array implementation. Unfortunately, this is the common informal view of queues that freshmen students grasp from typical data structures texts. Stated more formally here, a queue is viewed as a mathematical tuple: a function mapping an integer (index) to an entry (it models an "array" of the *contents*), an integer representing the *front* of the queue (in the "array"), and an integer representing the *length* of the queue. The descriptions of the operations essentially mirror the workings of a circular array implementation for queues.

This specification is not observable since it is possible to have different abstract values that are not distinguishable through the provided interface. In particular, two abstract queue values with the same "contents" but different front indices are indistinguishable. In this case, for both queues calls to Length_Of or Dequeue would return indifferent results. The specification is not controllable since it is impossible to generate a subset of the abstract values. For example, while the ensures clause for Dequeue states that all the entries except the one being dequeued remain unchanged, it does *not* specify what happens to the "slot" of the entry being dequeued. Therefore, it can conceptually take on any legal abstract entry value, and this gives rise to an exponential set of values that cannot be deterministically generated through the interface.

Clearly, this specification presents problems in reasoning intuitively about the abstract behavior of a queue. Therefore, it fails the property of *clarity*. It also fails the property of *restrictiveness*, since the designer failed to meet the "…and nothing more" part of the property by allowing redundant and unreachable abstract values to enter into the specification. Based on this example, we make two more observations:

3. There is a strong connection between each of *clarity* and *restrictiveness* and each of observability and controllability.
4. A proper set of operations *and* a proper choice of mathematical modeling are both essential for designing observable and controllable specifications.

We return our attention to the generic queue specification in Figure 1.7, since it is the more suitable candidate for further consideration. It is observable and controllable, thus satisfying many basic properties as discussed. However, it does have problems with generality and high cohesion, as we reveal through the application of the pragmatic criterion in the next subsection.

Pragmatic Considerations

The pragmatic criterion, in the context of software components, is concerned with functional and performance flexibility; i.e., more general applicability. This section focuses on the question of generality in the specification of queues.

The specification in Figure 1.7 first suffers from over-specification, because it demands that the inserted entry in the Enqueue operation be preserved. The "preserves" parameter mode for x, which is simply shorthand for having "x = #x" in the ensures clause, forces all implementations of Enqueue to enqueue a *copy* of the entry. Since Bounded_Queue_Template is generic, the particular type of entry that is used in instantiation may be expensive, if not impossible, to replicate. Forcing the copy results in an execution time for Enqueue that depends on the size of the entry. More to the point, the specification in Figure 1.7 fails to satisfy the property of *generality*, since it precludes efficient implementations that do not need to make a copy of the entry during Enqueue. The solution to this problem is to change the parameter mode for Enqueue's entry parameter to *consumes* [Harms 91], which leaves the outgoing value unspecified so that an implementation is not required to copy the entry.

Alternatively, the specification in Figure 1.7 can be viewed as failing to satisfy the property of *high cohesion*. This is because, based on the above discussion, it can further be broken down into a pair of more primitive concepts. Specifically, the first concept would be similar to that in Figure 1.7, except that it *consumes* Enqueue's entry parameter (as shown in Figure 1.9), and the second concept would be an *enhancement* to the first that provides the operation "Enqueue_a_Copy (q,x)". An implementation of Enqueue_a_Copy could be layered using the Enqueue operation that consumes its entry x, by first making a copy of x and then calling the Enqueue operation using the copy of x. This is an important and general realization, since it says that the notion of copying has nothing to do with the abstract behavior of a queue, or any other object for that matter [Harms 91]. Since copying has nothing to do with the abstract behaviors of most objects, it should not be an inherent part of abstract descriptions. Also, since in general it cannot be assumed that every type of Entry is copyable, the specification in Figure 1.7 is not truly generic.

A specification such as the one in Figure 1.7 that fails either generality or high cohesion also fails the *pragmatic criterion*, which is essentially concerned with the minimality and practicality of a concept. In this case it fails the pragmatic criterion since it forces the unnecessary and expensive copy behavior in the Enqueue operation for every client.

There is arguably at least one other problem with the specification in Figure 1.7. It is not efficient for applications that need to inspect the next value in a queue before dequeueing it (e.g., to “look-ahead” in parsing a queue of tokens). This functionality can be added through an operation termed Swap_Front. The Swap_Front operation exchanges a given entry with the first one in the queue. For inspecting the first entry of a queue before dequeueing it, a client can call Swap_Front to get the entry, inspect it, and then call Swap_Front to put it back. Swap_Front can be implemented in constant time as an intrinsic operation of the interface, but not so in a layered fashion. As a layered operation, Swap_Front would have to dequeue once to get the desired entry, enqueue its “exchange” entry, then rotate it to the front of the queue; the execution time is dependent on the size of the queue. Since obtaining this functional behavior from the concept in Figure 1.7 unnecessarily constrains the performance that is otherwise available through Swap_Front, the concept does not adequately satisfy the pragmatic criterion.

```
concept Bounded_Queue_Template (type Entry,
                               constant Max_Length: Integer)
         requires Max_Length > 0
         uses Standard_Integer_Facility

      type family Queue is modeled by string of Entry
         exemplar q
         initialization ensures
            |q| = 0
         constraints
            |q| <= Max_Length

      operation Enqueue (alters q: Queue, consumes x: Entry)
         requires |q| < Max_Length
         ensures  q = #q * <#x>

      operation Dequeue (alters q: Queue, produces x: Entry)
         requires |q| > 0
         ensures  #q = <x> * q

      operation Swap_Front (alters q: Queue, alters x: Entry)
         requires |q| > 0
         ensures  there exists alpha: string of Entry
                  such that    #q = <x> * alpha and
                                q = <#x> * alpha
      operation Length_Of (preserves q: Queue): Integer
         ensures  Length_Of = |q|

      operation Allowed_Max_Length (): Integer
         ensures  Allowed_Max_Length = Max_Length

end Bounded_Queue_Template
```

Figure 1.9 - A Properly Designed Queue Specification

Figure 1.9 above shows the final generic queue specification design to be discussed. It is observable, controllable, and satisfies the pragmatic criterion. By considering the pragmatic criterion with this example, we can make at least two more observations:

5. There is a strong connection between the pragmatic criterion and each of *generality* and *high cohesion*.
6. There is a strong connection between the pragmatic criterion and each of *primitiveness* and *potential completeness*.

The identification of relationships among the properties greatly motivates the need for observable and controllable specifications that fulfill the pragmatic criterion as well as the mechanisms that can be used to develop them. Essentially, this example illustrates that

- it is *probable* that an observable and controllable specification that meets the pragmatic criterion will fulfill basic properties such as clarity, restrictiveness, generality, high cohesion, etc., and
- it is *implied* that a specification which is not observable or controllable or fails the pragmatic criterion will fail to fulfill some of the software engineering properties such as clarity, restrictiveness, generality, high cohesion, etc.

It is worth noting that, although we began this section by first considering observability and controllability followed by the pragmatic criterion, in general there is no such ordering. Different problems need repeated consideration of these issues in different orders. The design of a suitable specification demands an iterative process, and usually involves several attempts in order to develop a final specification that is observable, controllable, and satisfies the pragmatic criterion. Figure 1.10 below concludes the discussion of this example by summarizing each specification design and its (lack of) attributes.

Queue Specification	Attributes
Figure 1.7	Observable and controllable. Fails generality and high cohesion, thus failing the pragmatic criterion.
Figure 1.8	Is not observable or controllable, thus failing clarity and restrictiveness. Fails the pragmatic criterion by failing generality and high cohesion.
Figure 1.9	Observable, controllable, and satisfies the pragmatic criterion.

Figure 1.10 - Queue Specification Designs and Their Attributes

1.4 Contributions

The primary contribution of this thesis to the field of computer science is the development of practical and formal tests for good formal specification design. We accomplish the goals set forth in that:

1. We motivate and define *observability*, *controllability*, and the *pragmatic criterion* as three fundamental properties of well-designed and well-explained formal specifications of software components.
2. We define the pragmatic criterion to include both functional and performance considerations. The definition leads to good designs of both single concept interfaces as well as suitable concept libraries for a problem domain.
3. We formalize observability and controllability in a more general manner than has been done before. The properties are defined in a manner independent of the system attributes being inspected. The central difficulty in this formalization stems from the fact that non-trivial software specifications are *relational*, unlike descriptions of physical devices which are almost always functional.

4. We make explicit a large part of the RESOLVE *discipline* in defining the above properties. That is, we explicitly state, for a particular specification design methodology, the guidelines, techniques, and principles that lead to good specification designs.
5. We exemplify the results on non-trivial formal specifications, and in doing so validate a subset of the RESOLVE concept library.

These contributions allow the design of objects with which software engineers can easily reason about and use in complex software systems. This results in software systems that cost less to develop and maintain, and are of higher quality than systems developed in an ad-hoc manner.

1.5 Organization

Chapter II, titled *A Pragmatic Criterion for Component Interface Design*, discusses the *pragmatic criterion* property of good specification design in detail. In this chapter, we establish a desirable interface for an ordering concept called Prioritizer_Template. We discuss how this interface fulfills the pragmatic criterion, and how the pragmatic criterion prevents the formation of a concept library that unnecessarily contains similar and redundant ordering concepts. This interface serves as the working example for the next chapter, in which we seek to formally define the behavior of the interface.

Chapter III, titled *Observable and Controllable Software Specifications,* focuses on the proper explanation for a given interface. In the discussion, we illustrate the roles of observability and controllability in alternative descriptions of the relational behavior of Prioritizer_Template.

Chapter IV, titled *Formalizations of Observability and Controllability,* seeks to formally characterize the properties of observability and controllability for relational specifications. Among others are definitions based on *scenarios* that are exemplified on the Prioritizer_Template specification of Chapter III. Previous work on the subject is discussed, and distinctions between this work and the previous work are made.

Chapter V, titled *Validating the RESOLVE Concept Library,* gives examples of several formal specification designs from the RESOLVE library, and demonstrates the utility of the formal and practical tests developed in the previous chapters in evaluating the individual specifications, as well as the library of specifications as a whole.

Chapter VI, titled *Conclusions*, presents a summary of the research, results, and possible future research directions.

A Pragmatic Criterion for Component Interface Design

II

A common design objective in developing an object-based interface for a widely applicable software component is functional flexibility that allows the component to be used in a variety of applications. However, concepts providing functional variations that are unaccompanied by implementations providing desirable performance behaviors will remain mostly unused. In the previous chapter, we briefly introduced the *pragmatic criterion* for designing widely applicable and efficient software components. This chapter defines and exemplifies the significance of this criterion through the design of a single interface that is suitable for different classes of applications that require the ordering of a collection of entries.

While good interface designs of a reusable concept make it suitable for use in a wide-range of applications, the concept will actually be used only in situations where its usage does not compromise performance. If components developed "from scratch" provide better performance for a class of applications, these components are likely to be used instead of "reusable" concepts and their implementations. In general, there is usually no one implementation for a concept that provides desirable performance for all intended functional variations and applications. The solution to this dilemma described herein is based on the consideration of alternative implementations of reusable concepts. Reusable concepts must be designed so that they allow multiple implementations, where there is at least one implementation that provides suitable performance behavior for each intended application of the reusable concept.

Different approaches for meeting this objective have been discussed in the literature to varying degrees. One of them attempts to provide suitable performance for all intended uses by designing a concept that contains every conceivably desirable operation (see [Meyer 94], for example). Apparently, the assumption here is that there will be a most suitable implementation for this concept. Figure 2.1 below illustrates this idea.

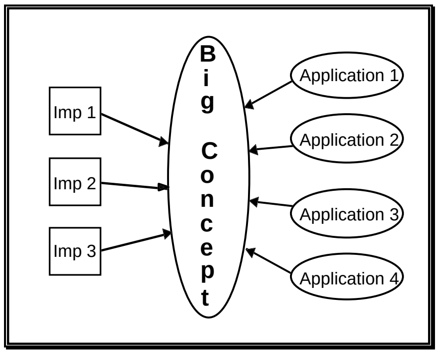


Figure 2.1 - One "Catch All" Concept

However, this approach is unacceptable. First, there is no way to envision every desirable operation for such a concept a priori. Second, this approach leads to unruly designs that require a typical client to understand a lot more than is essential in order to fully utilize the concept. Third, it is more expensive to implement such concepts. This is because choosing an overall strategy for efficiently coding the large operation set can become extremely difficult. It is even possible that the implementation strategies which would be efficient for an otherwise smaller (more appropriate) operation set will be precluded by a concept with a large operation set.

Another approach for providing suitable performance for all intended uses of a concept is to design not just one concept, but several "not-so-large" variants of some basic concept, so that together they provide suitable performance for a wide range of applications. Figure 2.2 below illustrates this idea.

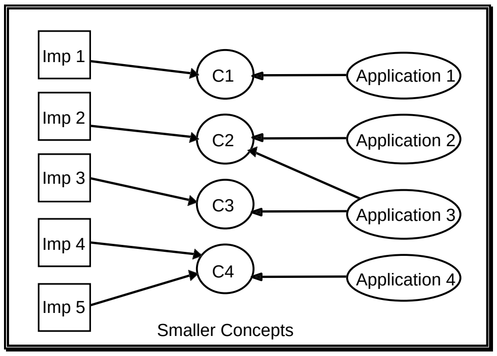


Figure 2.2 - Many Similar Concepts

However, this approach is also unacceptable. To fully understand and use the library, a client must differentiate among the many similar concepts in order to determine the one(s) that is of actual interest. Additionally, this approach leads to unnecessarily large libraries. Furthermore, the implementations for these related concepts would be largely redundant, reflecting the unnecessarily repeated effort of the implementer.

The approach we advocate concerns the design of each single concept as well as the design of a concept library for a problem domain. Essentially, when designing a single concept, the goal is that it be widely applicable, yet is no larger than it needs to be. For a collection of concepts in a particular problem domain, the goal is to design a library of core concepts that is widely applicable to the domain, yet contains no more concepts than are actually needed. In other words, the design of both a single concept as well as a concept library should be *orthogonal*:

> An object-based concept is *orthogonal* (i.e., has an orthogonal operation set) if it contains no operation such that both:
>
> - the operation's functionality can be subsumed by some combination of the others, and
> - the operation's performance (for all implementations) can always be subsumed by some combination of (implementations for) the other operations of the concept.
>
> A library of core concepts for a problem domain is *orthogonal* (i.e., has an orthogonal set of concepts) if it contains no concept such that both:
>
> - the concept's functionality can be subsumed by some combination of the other concepts, and
> - the concept's performance (for all implementations) can always be subsumed by some combination of (implementations for) the other concepts in the library.

This definition for orthogonality implies the design intent that a concept and a library should be, in some sense, minimal. A "non-minimal" concept or library would fail to be orthogonal as defined above. This definition is quite different from more traditional ones, where the focus is only on functionality. For example, based on functional consideration alone, the Bounded_Queue_Template introduced in the previous chapter is not "minimal", since the Swap_Front operation can be (functionally) layered using the other queue operations. However, based on the above definition, the concept *is* minimal since Swap_Front offers better performance as a built-in operation than if it were layered.

Even though an orthogonal concept is a minimal concept, this does not imply that it is a pragmatic concept. For example, the Bounded_Queue_Template without Swap_Front remains orthogonal, but from a pragmatic point of view the Swap_Front operation is needed for efficient usage in some applications. That is, this concept without Swap_Front introduces a *performance bottleneck*:

> A concept exhibits a performance bottleneck if it fails to permit at least one suitable implementation strategy for each intended functional usage of the concept [Fleming 97].

The goal is to design a concept that is orthogonal yet does not unnecessarily constrain its performance for some functional variations of the concept. Figure 2.3 below illustrates this idea.

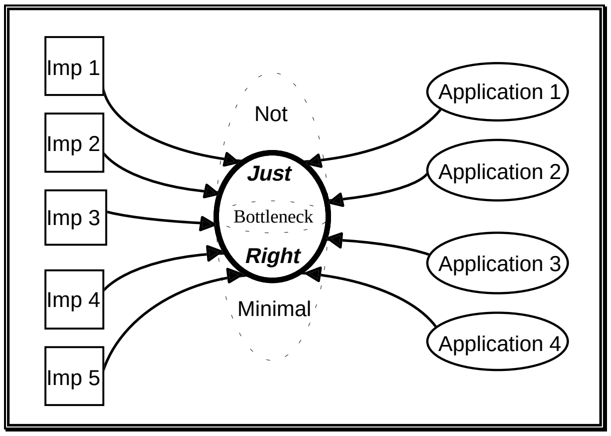


Figure 2.3 - A Minimal and Widely Applicable Concept without a Performance Bottleneck

The definition of orthogonality, coupled with the notion of preventing a performance bottleneck, succinctly states the expectations of both a single concept and a concept library for a given problem domain. Figure 2.4 below explicitly defines this statement as the *pragmatic criterion* for concept design:

- A concept should contain an orthogonal set of operations, yet should not introduce a performance bottleneck between the implementations for the concept and the applications that use the concept.
- A library of core concepts for a problem domain should contain an orthogonal set of concepts, yet should not introduce a performance bottleneck between the implementations for the concepts and the applications from the problem domain. The concepts in a library should not compromise the level of abstraction for the problem domain.

Figure 2.4 - The Pragmatic Criterion

A concept, such as the one shown in Figure 2.1, that attempts to include every desirable operation in order to make it widely applicable tends to fail the pragmatic criterion; even though it may avoid a performance bottleneck, its large operation set is rarely orthogonal. Similarly, a concept library, such as the one shown in Figure 2.2, that contains several variants of a more fundamental concept also fails the pragmatic criterion; even though the library may offer suitable performance for all applications, the large number of concepts is rarely orthogonal[2].

To illustrate the ideas, the rest of this chapter considers the interface design of a concept for ordering a collection of entries. This concept must provide (through alternative implementations) suitable performance for all the following classes of client applications:

Class I) all entries to be ordered are known a priori and all of them need to be ordered;

Class II) all entries to be ordered are known a priori, but only an arbitrary subset of the entries needs to be ordered:
a) some best k of n entries are needed;
b) some best k_1 and worst k_2 entries are needed;

Class III) entries to be ordered are not all known in advance; after some ordered entries have been obtained from a collection, additional entries may be added to the collection for subsequent ordering.

[2] Even though a library of concepts should be minimal, this minimality should not compromise the level of abstraction required for solving problems in a given domain.

The rest of this chapter is organized as follows: Section 2.1 illustrates the usefulness of the pragmatic criterion in developing a single and widely applicable concept for ordering a collection of entries. This section simultaneously illustrates how the pragmatic criterion prevents the construction of unnecessarily large concept libraries. Section 2.2 summarizes the results and ideas. Section 2.3 provides graphs confirming the expected performance benefits of the interface design through actual implementation using C++ templates.

2.1 Designing a Suitable Interface for Ordering a Collection of Entries

In this section, we consider alternative interface designs for ordering, and evaluate each design using the pragmatic criterion. In the process, we also show that some of the interface designs are subsumed by others, eventually leading to a single suitable concept for the problem.

A Procedural Interface Design

For solving “batch sorting” problems, captured in application class I, all that is needed is a generic sorting procedure that takes as its parameters a container of entries (in a specific representation) that needs to be sorted, and a comparison function for stating when two entries are ordered. There are at least two problems with a single procedural interface design. This interface fails to decouple the representation of the container from the ordering problem, thus forcing separate sorting procedures to be implemented and used for each representation for every container. The second problem is that the interface is unsuitable for applications of ordering that do not require batch sorting, such as in classes II and III. If used in those applications, the procedural interface will needlessly order all entries.

The pragmatic criterion demands that alternative, more widely applicable concepts be considered for ordering a collection of entries. In addressing this issue, the remainder of this section illustrates that object-based concept designs are fundamentally necessary for satisfying the pragmatic criterion. Moreover, the criterion sets some object-based designs for a given problem apart from others, resulting in highly reusable concepts that permit both functional and performance variability.

Object-Based Designs for Ordering a Collection of Entries

Performance flexibility is the motivation for the “recasting” technique presented in [Weide 94], whereby classical algorithms are encapsulated as objects using data abstraction principles. Recasting is a general technique in which a problem that is typically solved by a single large-effect operation (e.g., sorting) is solved using a concept that encapsulates the problem as an object-based machine. A single “recast” interface, in general, can be used in a wide range of functionally diverse applications. With a properly designed interface, the applications can get good performance through different plug-compatible implementations of the interface.

To illustrate the flexibility offered by recasting, Weide, et al., discuss an object-based interface design for sorting. An object-based interface for sorting is fundamentally superior to a procedural interface design in that it can provide *information hiding* and *data abstraction* [Weide 94]. Unlike a procedural interface, the object-based interface hides the internal structure in which the entries to be ordered are stored. The object-based interface also permits an abstract (and formal) explanation of behavior that is devoid of representation details [Weide 94]. Such an abstract explanation is not possible for a procedural interface.

An Object-Based Concept for Ordering

Figure 2.5 below shows a skeleton of an object-based interface for sorting in the RESOLVE notation. This interface is directly based on the formal specification given in [Weide 94]. Details concerning the mathematical model used as well as individual operation specifications have been deferred to Chapter III, in order to keep the current focus on the set of operations.

```
concept Sorting_Machine_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0 and
         (* ARE_ORDERED is a total pre-ordering *)

      uses  Standard_Integer_Facility,
            Standard_Boolean_Facility

   type family Sorting_Machine is modeled by …
         (details omitted to keep focus on operation set)
      exemplar m

   operation Insert (
         alters m: Sorting_Machine,
         consumes x: Entry
      )

   operation Change_To_Extraction_Phase (
         alters m: Sorting_Machine
      )

   operation Extract (
         alters m: Sorting_Machine,
         produces x: Entry
      )

   operation Is_In_Insertion_Phase (
         preserves m: Sorting_Machine
      ): Boolean

   operation Size_Of (
         preserves m: Sorting_Machine
      ): Integer

   operation Allowed_Max_Size (): Integer

end Sorting_Machine_Template
```

Figure 2.5 - A Suitable Object-Based Interface for Application Classes I, II(a)

The concept Sorting_Machine_Template in Figure 2.5 is generic. To create an instance, a client must pass the type of Entry to be ordered, a comparison operation for computing the order of two entries, and a Max_Size value that specifies the maximum number of entries a Sorting_Machine may contain[3]. It is interesting to contrast the object-based

[3] In [Weide 94], the interface also contains a Change_To_Insertion_Phase operation. It has the effect of "clearing" the object. Since every object is assumed to have a Clear operation in the current RESOLVE discipline, Change_To_Insertion_Phase is no longer needed.

ordering machine design given here with the parameterized sorting procedure design given by Stepanov and Lee [Stepanov 94].

The concept in Figure 2.5 has been designed to operate in two phases: an *insertion phase*, in which entries are inserted into a machine one at a time, and an *extraction phase*, in which entries are extracted from the machine one at a time; the entries come out in an ordered fashion. An informal explanation of the operations is given below:

- Insert (m, x): Insert Entry x into Sorting_Machine m. This operation requires that m be in the insertion phase at the time of the call.
- Change_To_Extraction_Phase (m): Prepare Sorting_Machine m for calls to the Extract operation. This operation requires that m be in the insertion phase at the time of the call.
- Extract (m, x): Extract the next ordered Entry from Sorting_Machine m, based on the client-supplied definition ARE_ORDERED, and return it in x. This operation requires that m be in the extraction phase at the time of the call.
- Is_In_Insertion_Phase (m): Determine if Sorting_Machine m is in the insertion phase.
- Size_Of (m): Return the number of entries currently contained in Sorting_Machine m.
- Allowed_Max_Size (m): Return the maximum number of entries Sorting_Machine m is allowed to contain. This operation returns the Max_Size value supplied by the client at instantiation time.

A normal usage of this component involves adding entries to be ordered through calls to Insert. The final insertion is followed by a call to Change_To_Extraction_Phase. Then, through calls to Extract, the entries can be obtained in a sorted order one at a time.

The performance advantages of object-based ordering result from the fact that the interface not only hides how the ordering is accomplished, but also *when* it is done [Weide 94]. Unlike a typical sorting procedure, which permits only batch sorting, this interface allows an implementation to distribute the computational expense of ordering over the code for the operations in any desirable fashion using any known sorting algorithm [Cormen 90, Horowitz 76]. Figure 2.6 highlights implementation strategies that provide good performance when either some or all entries need to be ordered.

Some Implementations for the Concept in Figure 2.5	Batch Sorting	Sorting during Extract	Heap Based Sorting
Suitable for Application Class:	I	II(a)	I and II(a)
Each Insertion	O(1)	O(1)	O(1)
n Insertions	O(n)	O(n)	O(n)
Change_To_Extraction_Phase	O(n log n)	O(1)	O(n)
Each Extraction	O(1)	O(n)	O(log n)
n Extractions	O(n)	O(n^2)	O(n log n)
k Extractions	O(k)	O(kn)	O(k log n)

Figure 2.6 - Some Implementations Suitable for Applications in Classes I and II(a)

Figure 2.6 outlines the performance obtainable from implementations based on batch sorting, sorting-during-extraction, and heap-based sorting. Batch sorting schemes, such as quick-sort and merge-sort, sort all the entries at once as a “batch”. Sorting during extractions, or “selection” sorting, computes the next ordered entry (in linear time) on a “per-request” basis. Heap-based sorting schemes involve dividing the computational expense of sorting into multiple steps. As illustrated in Figure 2.6, a heap can be constructed in linear time just before ordered entries are to start being obtained, and from there each next ordered entry can be obtained in log-n time on a per-request basis. As indicated in Figure 2.6, for problems where $k << n$, both sorting-during-extraction and heap-based sorting are suitable choices. In the more general case where $k \leq n$, then a heap based strategy is most suitable.

It is easy to see that the concept in Figure 2.5 is also suitable for sorting an entire collection of entries, such as needed for applications in class I, since it subsumes the procedural interface for sorting both in terms of functionality and performance. In terms of functionality, any client using a sorting procedure can instead use the concept in Figure 2.5. The client simply inserts all the entries into a Sorting_Machine object, changes to the extraction phase, and then extracts all the entries from the Sorting_Machine. While any implementation for Sorting_Machine_Template will work, to get the best performance a client can choose to implement the concept in Figure 2.5 with any batch sorting strategy, or even a heap-based strategy as shown in Figure 2.6 above.

Application of the Pragmatic Criterion to the Design

We are now ready to evaluate this design using the pragmatic criterion first by examining whether the concept in Figure 2.5 is orthogonal and then by determining if it introduces a performance bottleneck.

A concept should contain an orthogonal set of operations, …

For the concept in Figure 2.5, it is easy to see that no one operation can be implemented using any combination of the others, either functionally or otherwise. Therefore, the concept in Figure 2.5 is orthogonal.

… yet (the concept) should not introduce a performance bottleneck between the implementations for the concept and the applications that use the concept.

The concept in Figure 2.5 exhibits a performance bottleneck. For example, with respect to application class II(b), this concept provides no suitable means for obtaining the worst k_2 entries from a Sorting_Machine; a client would not be able to extract all the preceding entries efficiently. More importantly, it is unsuitable for applications is class III.

Since this is the only object-based concept in the “current library”, the orthogonality question of the library does not yet arise.

An Alternative Object-Based Concept for Ordering

Figure 2.7 below shows an object-based sorting concept that is suitable for solving problems in application class II(b) (in addition to problems in classes I and II(a)).

```
concept Sorting_Machine_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

   (same as in Figure 2.5 with the following operation)

   operation Extract_Any (
         alters m: Sorting_Machine,
         produces x: Entry
      )

end Sorting_Machine_Template
```

Figure 2.7 - A Suitable Object-Based Interface for Application Classes I, II

The above concept is the same as the previous design in Figure 2.5 with the addition of a new operation, termed Extract_Any, which is specified to return an arbitrary entry from a Sorting_Machine. This concept can be used to obtain the best k_1 and worst k_2 entries from a collection efficiently. The steps below illustrate a particular usage pattern for accomplishing this task, assuming a heap-based implementation is chosen for Sorting_Machine_Template. The complexity for each step is shown in parentheses.

1. Create an instance of Sorting_Machine_Template, say sm_a (O(1)).
2. Insert the entries to be sorted into sm_a (O(n)).
3. Change sm_a to the extraction phase (O(n)).
4. Extract and retain the best k_1 entries from sm_a ($O(k_1 \log n)$).
5. Now create another instance of Sorting_Machine_Template, say sm_b, based on the reverse ordering of sm_a for its entries (O(1)).
6. Extract the remaining entries from sm_a in an arbitrary order using the Extract_Any operation, inserting them into sm_b along the way (O(n)).
7. Change sm_b to the extraction phase (O(n)).
8. Extract and retain the best k_2 entries from sm_b (these are the worst k_2 entries with respect to the ordering of sm_a) ($O(k_2 \log n)$).

This usage pattern has time complexity bounded by $O(n + (k_1+k_2) \log n)$ if a heap-based implementation is chosen. With the previous concept in Figure 2.5, in step 6 above the Extract operation would have to be called instead of Extract_Any. If this were the case, the time complexity of step 6, and therefore the entire algorithm, would be O(n log n). Figure 2.8 below shows some suitable implementation strategies for the concept to solve problems from application classes I, II(a), and II(b) efficiently.

Some Implementations for the Concept in Figure 2.7	Batch Sorting	Sorting during Extract	Heap Based Sorting
Suitable for Application Class:	I	II(a) and II(b)	I, II(a), and II(b)
Each Insertion	O(1)	O(1)	O(1)
n Insertions	O(n)	O(n)	O(n)
Change_To_Extraction_Phase	O(n log n)	O(1)	O(n)
Extract_Any	O(1)	O(1)	O(1)
Each Extraction	O(1)	O(n)	O(log n)
n Extractions	O(n)	$O(n^2)$	O(n log n)
k Extractions	O(k)	O(kn)	O(k log n)

Figure 2.8 - Some Implementations Suitable for Classes I, II(a), and II(b)

Application of the Pragmatic Criterion to the Design

To evaluate the interface design in Figure 2.7, we first consider its orthogonality and then consider the orthogonality of the library containing this concept and the previous concept in Figure 2.5. Next, we discuss whether there is a performance bottleneck associated with either the concept in Figure 2.7 or the library.

A concept should contain an orthogonal set of operations, …

For the concept in Figure 2.7, there is the question of whether the Extract_Any operation violates the orthogonality of the interface. From a functional point of view, it may at first seem that there is no way to implement Extract_Any using the other operations, since there is no other way to obtain a “random” entry. However, note that the specification of Extract_Any would say that it simply returns “some” entry. Clearly, the Extract operation returns “some” entry, and thus the functionality of Extract_Any can be attained with a single call to Extract. The question now is whether the performance of Extract_Any is always subsumed by Extract. For applications in class II(b) that need to use both Extract and Extract_Any, the most suitable implementation strategies are “heap-based” and “ordered-extraction”. In such implementations, Extract_Any is a constant time operation whereas Extract is not. This performance argument provides the rationale for why the concept in Figure 2.7 is orthogonal.

A library of core concepts for a problem domain should contain an orthogonal set of concepts,…

The concept library should not contain the concept in Figure 2.5, but only the one in Figure 2.7 because the former is subsumed by the latter, both in terms of functionality and performance. This is easily seen, since the concept in Figure 2.7 is the same as that in Figure 2.5 with one additional operation, and therefore permits any implementation strategy that is appropriate for the interface design in Figure 2.5.

Finally, we note that the concept in Figure 2.7 constrains performance for applications in class III:

… (the concept) should not introduce a performance bottleneck between the implementations for the concept and the applications that use the concept.

The concept in Figure 2.7 is unsuitable for applications in class III, because it provides no effective means for inserting new entries after some ordered entries have already been obtained. The pragmatic criterion therefore suggests that we look further for a concept that is suitable for all intended application classes.

A More Generally-Applicable Concept for Ordering

Conceptually, the modification to the interface for interleaving insertions and extractions during ordering turns out to be straightforward: Instead of having Change_To_Extraction_Phase, we include a different operation that toggles the machine from one phase to the other, but without modifying the abstract contents of the machine. The concept in Figure 2.9 below shows this modified interface, with a new operation termed Change_Phase[4].

[4] Functionally, Change_Phase is not needed; i.e., the notion of a phase could be removed from the interface altogether, by doing (for example) the “phase change” in the first call to Extract following a call to Insert. However, this introduces a complexity in expressing the performance for Extract, since the performance for the first call to Extract following a call to Insert will be much different than for each subsequent consecutive call to Extract. The Change_Phase operation permits simpler explanations of performance.

```
concept Prioritizer_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0 and
         (* ARE_ORDERED is a total pre-ordering *)

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   type family Prioritizer is modeled by …
         (details omitted to keep focus on operation set)
      exemplar p

   operation Insert (
         alters p: Prioritizer,
         consumes x: Entry
      )

   operation Change_Phase (
         alters p: Prioritizer
      )

   operation Extract (
         alters p: Prioritizer,
         produces x: Entry
      )

   operation Extract_Any (
         alters p: Prioritizer,
         produces x: Entry
      )

   operation Is_In_Insertion_Phase (
         preserves p: Prioritizer
      ): Boolean

   operation Size_Of (
         preserves p: Prioritizer
      ): Integer

   operation Allowed_Max_Size (): Integer

end Prioritizer_Template
```

Figure 2.9 - A Generally Applicable Object-Based Interface for Ordering that is Suitable for Application Classes I, II, and III.

Notice that the interface in Figure 2.9 permits easy and efficient management of "priority queues", among other problems from application class III. In light of this added flexibility to prioritize, the concept in Figure 2.9 has been named Prioritizer_Template. It is interesting to note that in the literature, priority queues are normally considered in the formal and less formal discussions of queues [Booch 86, Jones 90]; the idea of combining priority queues with sorting into a single data abstraction never appears to have been considered. This observation underscores the important role of the pragmatic criterion in concept design.

The difference between the interfaces in Figures 2.7 and 2.9 is subtle, but significant. By designing the interface so that it can insert additional entries after some have already been extracted, it admits implementations that can efficiently interleave insertion and extraction of entries. Figure 2.10 below shows some suitable strategies for implementing the concept in Figure 2.7 to solve problems in all three application classes.

Some Implementations for the Concept in Figure 2.7	Batch Sorting	Heap Based Sorting #1	Heap Based Sorting #2
Suitable for Application Class:	I	I, II, and III	I, II, and III
Each Insertion	O(1)	O(1)	O(log n)
n Insertions	O(n)	O(n)	O(n log n)
Change_Phase	O(n log n)	O(n)	O(1)
Extract_Any	O(1)	O(1)	O(1)
Each Extraction	O(1)	O(log n)	O(log n)
n Extractions	O(n)	O(n log n)	O(n log n)
k Extractions	O(k)	O(k log n)	O(k log n)

Figure 2.10 - Some Suitable Strategies for Solving Problems in All Three Application Classes

For clients solving problems where the number of insertions and extractions is considerably more than the number of phase changes performed, the first heap based strategy is the most suitable. For the case where the number of phase changes is considerably large, the second heap-based strategy, which always maintains a heap, may be more suitable. Even better performance is possible if Fibonacci heaps are used for efficiently merging an “old” heap of entries from the first change (to extraction) phase with the “new” heap of recently inserted entries on each change (to extraction) phase thereafter.

Application of the Pragmatic Criterion to the Design

We apply the pragmatic criterion to this new design by first considering its orthogonality, and then by considering the orthogonality of the library containing this concept and the previous concept in Figure 2.7. Next, we discuss whether there is a performance bottleneck associated with either the new concept or the library.

A concept should contain an orthogonal set of operations, …

The modification of the phase-changing operation does not alter the orthogonality of the concept from the previous design, since we already explained that it is not possible to layer Change_To_Extraction_Phase using the other operations in the previous designs, and since it is not possible to layer Change_Phase using the other operations of this design. Therefore, the concept in Figure 2.9 is orthogonal.

A library of core concepts for a problem domain should contain an orthogonal set of concepts,…

A library containing a concept with Change_To_Extraction_Phase as in Figure 2.7 and a concept with the Change_Phase operation as in Figure 2.9 is not orthogonal. Obviously, the former is subsumed by the latter in terms of functionality. In terms of performance, any implementation strategy used for the previous concept can be used here as well. The only additional demand on such implementations is that they provide the additional code for switching phases without losing the contents.

Given that the interface in Figure 2.9 is most general, all that remains to be argued is whether it unnecessarily constrains performance for some application classes.

… yet (the concept) should not introduce a performance bottleneck between the implementations for the concept and the applications that use the concept.

The concept in Figure 2.9 "permits at least one suitable implementation strategy for each intended functional usage of the concept"; i.e., all three application classes. For application class I, any batch-sorting implementation can be used. For application classes II(a) and II(b), efficient heap-based implementations can be used. For application class III, heap-based implementations such as those based on Fibonacci heaps can be used.

The concept in Figure 2.9 satisfies the pragmatic criterion, since it permits efficient implementation for all three application classes outlined in the introduction. Thus, we have achieved the goal of designing a single and widely applicable concept for ordering, and illustrated the utility of the pragmatic criterion in doing so.

2.2 Discussion

For the application classes discussed in this chapter, it has turned out that a single concept with alternative implementations is sufficient, both in terms of functionality and performance. If we had outlined a sufficient number of application classes, then other questions would arise. For example, would we be able to design one concept for all the application classes (probably not)? If not, what others would there be? In particular, the distinction between stable and unstable sorting forces the need to have separate concepts for each case (as is discussed in Chapter III), thereby resulting in two orthogonal concepts in a concept library. When we have multiple such concepts, the question of a performance bottleneck for a concept library - the last part of the pragmatic criterion - will arise and need to be addressed.

Advantages of reusing components especially designed to be widely applicable have been well documented in the literature [Booch 86, Meyer 94, Weide 91, WISR 93]. However, functional flexibility and functional evolution considerations have been the dominant motivations for such object interface designs [Booch 86, Meyer 87]. Though the importance of performance as a component design issue has been occasionally noted [Harms 91, Koenig 95, Stroustrup 96], the common perception is that there has to be a trade-off between good performance and widespread applicability. This chapter has made a case that good object-based designs can provide performance benefits, thus leading to their wide applicability, rather than precluding it. We have exemplified a process for how the dual objective of functional variation and suitable performance can be achieved by careful object interface design:

1. Identify classes of applications where the new concept is intended to be used from a functionality perspective (though there will remain some unanticipated uses).
2. Design an object-based interface for the problem that is suitable for (possibly, a subset of) the desired set of applications.
3. Determine if the interface satisfies the pragmatic criterion, by considering alternative implementations.

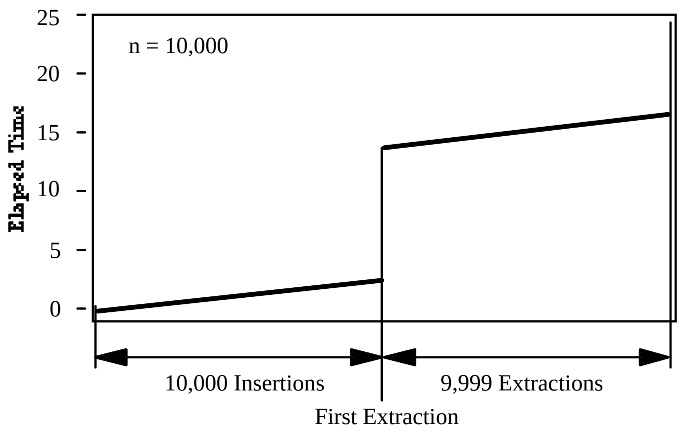
4. Refine the interface subject to the pragmatic criterion if no implementation strategy yields suitable performance for an intended class of applications.
5. Repeat steps 3 and 4 until a suitable interface has been designed; if your goal is unattainable, i.e., the intended set of applications is too broad, the steps may never terminate. This suggests re-examination of your objectives for the concept, possibly with the solution being to design more than one core concept.

In the next chapter, we discuss issues surrounding the mathematical modeling for the interface in Figure 2.9 in the light of observability and controllability considerations. In Chapter V, we apply these considerations and the pragmatic criterion to a select subset of the RESOLVE concept library, thus validating the design of these concepts.

2.3 Concrete Performance Analysis

We conclude this chapter with a discussion on concrete performance analysis that provides practical validation of the pragmatic criterion. We present graphs to illustrate the actual run-time performance characteristics obtained by different client applications using alternative plug-compatible implementations of the Prioritizer_Template in Figure 2.9. The implementations have been developed using a variant of the RESOLVE/C++ discipline [SIGSOFT 94], and are listed in the appendix. The intention of these graphs is not to illustrate any new algorithms or unexpected results, but rather to confirm through practice that which is obvious in theory. Together the graphs demonstrate quantitatively that the object-based concept in Figure 2.9 is the most widely applicable interface for sorting and prioritizing, in terms of both functionality and performance.

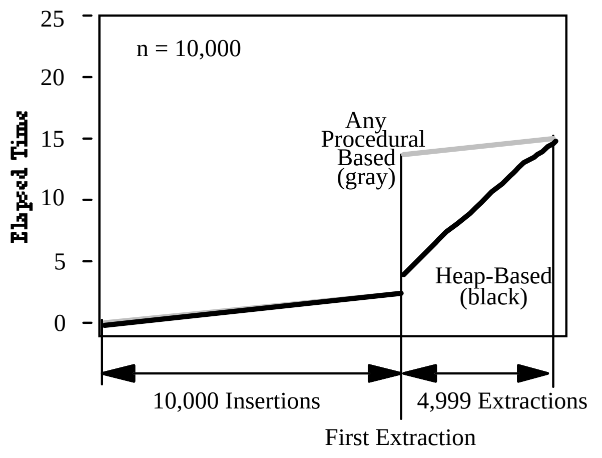
For applications in class I, Figure 2.11 shows average performance characteristics of a batch sorting-based strategy. The figure shows that the performance of the batch implementation is equivalent to a sorting procedure (in terms of performance) when using a batch sort-based implementation.



	Batch Sort Based
Each Insertion -	O(1)
First Extraction -	O(n log n)
Each Following Extraction -	O(1)

Figure 2.11 — A Performance Graph for Application Class I

For applications in class II(a), Figure 2.12 shows the performance resulting from a heap-based implementation strategy that “heapifies” on the first call to Extract, with each subsequent call to Extract rebuilding the heap using a “sift-down” procedure. As expected, the graph confirms that a heap-based strategy is a better choice than a (procedural) batch sort-based strategy for usage patterns from this class of applications.



	Batch Sort	Heap
Each Insertion -	O(1)	O(1)
First Extraction -	O(n log n)	O(n)
Each Following Extraction -	O(1)	O(log n)

Figure 2.12 — Performance Graphs for Application Class II(a)

Figure 2.13 illustrates the use of the Extract_Any operation along with the usage pattern outlined in Section 2.1 in order to handle client applications in class II(b) efficiently. The same heap-based implementation strategy used in generating the graph in the previous figure was also used to produce the data for Figure 2.13.

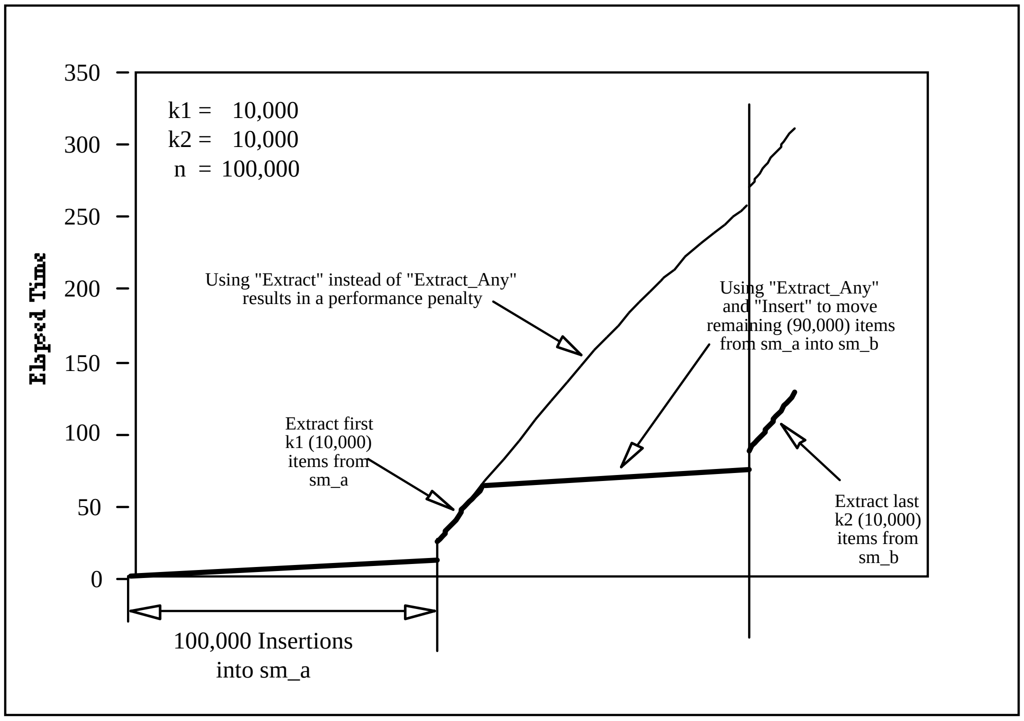


Figure 2.13 — Performance Graphs for Application Class II(b)

Figure 2.14 shows how the interleaving ability of the interface in Figure 2.9 is ideal for client applications in class III. This graph is based on a heap-based strategy in which Insert and Extract continually maintain the heap through the use of “sift-up” and “sift-down” procedures, respectively. It is interesting to notice that the process of interleaving insertions and extractions puts no strain on the implementation, as it only creates ripples in what would otherwise be a somewhat smoother curve.

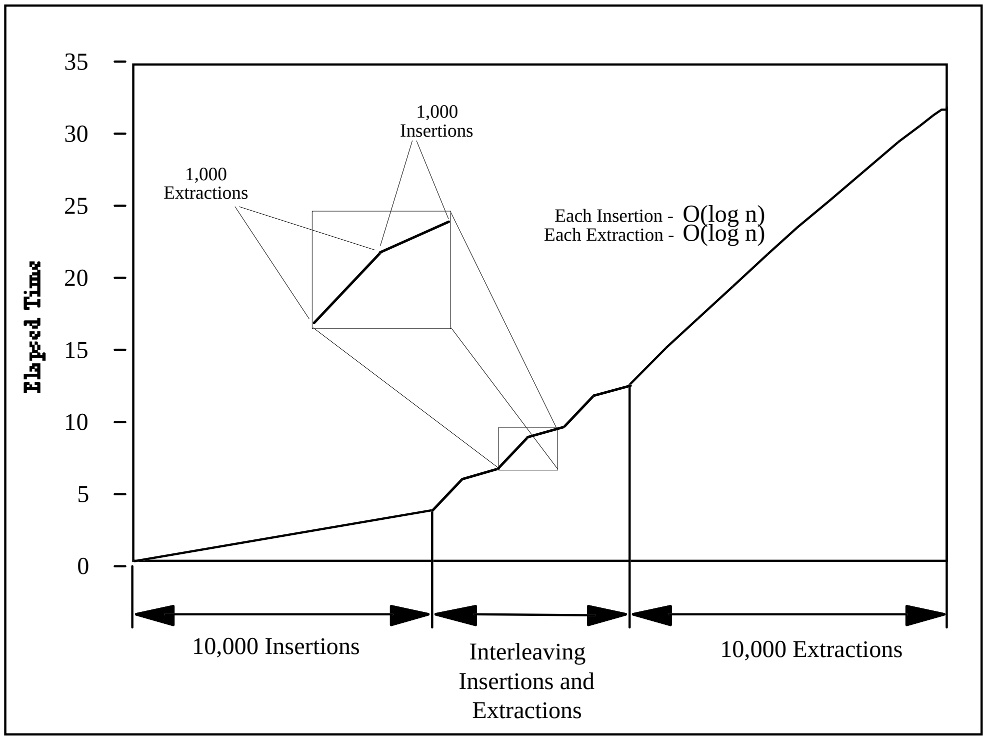


Figure 2.14 — Performance Graphs for Application Class III

Observable and Controllable Software Specifications

III

Designing model-based explanations for object-based software components that are easy to understand [Sitaraman 93], free from implementation bias [Jones 90], and flexible in terms of both functionality and performance [Weide 94, Fleming 97] is difficult. While in general there can be no way to make the process effective or trivial, research and experience has shown that these “ideal” specifications, which provide the most appropriate explanations, have some important properties in common. Arguably, two such fundamental properties are observability and controllability, and these properties are the topics of this chapter.

The complexity (and the importance) of designing observable and controllable specifications becomes apparent in designing specifications of nontrivial data abstractions, such as prioritizers and other relational data abstractions arising from recasting graph optimization problems [Sitaraman 97]. Such complex objects are likely to arise in industrial-strength software systems, and formal specification methods should be scaleable to those objects.

To illustrate the impact and significance of the properties on specification design, we consider alternative plausible explanations for the Prioritizer_Template interface of the previous chapter. We keep the discussion here at an informal level, leaving alternative formal definitions of the terms for the next chapter as they are quite intricate. Figure 3.1 below shows the intuitive working definitions which we adopt for the current discussion, based on [Weide 96]:

Observability - A model-based specification is *observable* if it is always possible to distinguish between any two values of the specified state space using the provided operations (in addition to those of imported types).

Controllability - A model-based specification is *controllable* if it is always possible to construct any given value from the specified state space using the provided operations (in addition to those of imported types).

Figure 3.1 - Intuitive Definitions of Observability and Controllability

The rest of this chapter is organized as follows: Section 3.1 considers alternative specifications for the Prioritizer_Template introduced in the previous chapter. Section 3.2 presents a discussion and summary of the results.

3.1 Alternative Formal Specifications of Prioritizer_Template

In this section, we show how observability and controllability considerations lead to the design of a precise and easily understandable formal specification for the Prioritizer_Template interface. In the process, we consider three specification designs, with varying degrees of observability and controllability.

Prioritizer_Template Specification Design #1

Figure 3.2 below contains a first formal specification of the Prioritizer_Template in the RESOLVE notation [SIGSOFT 94]. It has the same syntactic interface from Figure 2.10 in the previous chapter.

```
concept Prioritizer_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0   and
               for all x,y,z: Entry,
                  ARE_ORDERED (x,x) and
                  if ARE_ORDERED(x,y) and ARE_ORDERED(y,z) then
                     ARE_ORDERED(x,z) and
                  (ARE_ORDERED(x,y) or ARE_ORDERED(y,x))

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   definition IS_A_NEXT_ENTRY (
            s: string of Entry,
            x: Entry
         ): boolean =
            there exists alpha,beta: string of Entry such that
                  s = alpha * <x> * beta  and
            for all y: Entry,
               if ARE_ORDERED(y,x) and not ARE_ORDERED(x,y) then
                  not there exists alpha,beta: string of Entry
                     such that s = alpha * <y> * beta

   type family Prioritizer is modeled by (
            contents: string of Entry,
            insertion_phase: boolean
         )
      exemplar p
      constraints
         |p.contents| <= Max_Size
      initialization
         ensures  |p.contents| = 0 and
                  p.insertion_phase

   operation Insert (
            alters p: Prioritizer,
            consumes x: Entry
         )
      requires |p.contents| < Max_Size and
               p.insertion_phase
      ensures  p.contents = #p.contents * <#x> and
               p.insertion_phase

   operation Change_Phase (
            alters p: Prioritizer
         )
      ensures  p.contents = #p.contents and
               p.insertion_phase = not #p.insertion_phase
   operation Extract (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires |p.contents| > 0 and
               not p.insertion_phase
      ensures  IS_A_NEXT_ENTRY (#p.contents,x)  and
               not p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                   p.contents = alpha * beta and
                  #p.contents = alpha * <x> * beta

   operation Extract_Any (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires |p.contents| > 0
      ensures  p.insertion_phase = #p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                   p.contents = alpha * beta and
                  #p.contents = alpha * <x> * beta

   operation Is_In_Insertion_Phase (
            preserves p: Prioritizer
         ): Boolean
      ensures  Is_In_Insertion_Phase = p.insertion_phase

   operation Size_Of (
            preserves p: Prioritizer
         ): Integer
      ensures  Size_Of = |p.contents|

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Prioritizer_Template
```

Figure 3.2 - A Specification of Prioritizer_Template Based on Strings

Explanation of the Specification

In order to use the concept in Figure 3.2, a client must provide as parameters the type of Entry to be prioritized, a mathematical ARE_ORDERED definition for determining when two entries are ordered, and a Max_Size value for the maximum number of entries a Prioritizer object is allowed to contain. The *requires* clause below the parameter list is a concept-level requirement, and it states that the Max_Size value supplied must be greater than zero. It additionally states mathematically that the definition of ARE_ORDERED supplied by the client must be a "total pre-ordering". That is, the supplied definition should be reflexive, anti-symmetric, and transitive. Additionally, the definition must be total; i.e., “$x \leq y$ or $y \leq x$”, so that every two entries are ordered. The *uses* clause states that the concept makes use of standard Integer and Boolean facilities throughout the specification. The local *definition* IS_A_NEXT_ENTRY(s,x) is a predicate that is true when the Entry x is a next ordered entry in the string s; it is used in specifying the behavior for the Extract operation.

The *type* Prioritizer is modeled as a tuple where “contents” is a mathematical string containing the collection of entries in a Prioritizer object and “insertion_phase” is a flag denoting the current phase of a Prioritizer. The keyword *exemplar* simply provides an example Prioritizer name that is used to express the behavior of every Prioritizer object in the subsequent assertions. The *constraints* clause asserts that a Prioritizer cannot contain any more than Max_Size entries. The *initialization ensures* clause states that, upon declaration, a Prioritizer will contain no entries and will be in the insertion phase.

As noted earlier, in addition to the operations listed, every RESOLVE concept implicitly provides the operations Swap (denoted by the operator “:=:”) and Clear. The Swap operation simply exchanges the values of two (Prioritizer) objects. Since swapping can always be implemented to perform in constant time, it is chosen as the data movement operator rather than assignment (which forces copying) [Harms 91]. The Clear operation provides an efficient means for resetting a (Prioritizer) object to its initial state (as declared in the *initialization ensures* clause).

The Insert operation *requires* that its Prioritizer object (p) has room for the Entry to be inserted and that p is in the insertion phase. It *consumes* the Entry and concatenates it to the right of #p.contents; the string containing the single Entry is denoted by “< >”. It also *ensures* that the Prioritizer remains in the insertion phase. The Change_Phase operation ensures that the contents of the Prioritizer remain unchanged, but that its phase is toggled. The Extract operation requires that p is not empty and that it is not in the insertion phase. It ensures that a “smallest” value is removed and produced in x based on the definition of ARE_ORDERED. It additionally ensures that p will remain in the extraction phase. Extract_Any requires that the Prioritizer is not empty, but unlike Extract it can be called in either phase. It only ensures that an arbitrary value from the Prioritizer is removed and returned, and leaves the phase unchanged. In other words, the specifications of both Extract and Extract_Any are relational; i.e., different results may be output for the same input value. The operation Is_In_Insertion_Phase can be used to observe the current phase of a Prioritizer object and Size_Of can be used to get the number of entries.

Allowed_Max_Size is useful to remind a client of the maximum number of entries a Prioritizer object can contain. This is a module-level operation; i.e., it has no Prioritizer objects as parameters.

Evaluation of the Specification for Observability and Controllability

The specification in Figure 3.2 whereby the collection is modeled as a mathematical string is certainly reasonable and plausible . Since the problem of prioritization deals with the ordering of entries, strings seem naturally suited for modeling the contents. The specification is easy to understand and it satisfies software engineer's criteria such as minimality and comprehensiveness. And of course, it satisfies the pragmatic criterion as seen from the discussion in Chapter II. However, it is not observable, and this suggests potential problems as explained here.

To see why the specification in Figure 3.2 is not observable, consider Prioritizer_Template instantiated with an Entry type of Integer with "≤" ordering (which is a total pre-ordering), and some maximum size. Now, consider two objects of this type named p_1 and p_2 with values as follows:

p_1 = (< 1, 7, 3, 6 >, **false**)
p_2 = (< 1, 3, 7, 6 >, **false**)

The value of p_1 reveals that the Integers 1, 7, 3, and 6 were inserted in that order. Likewise, p_2 reveals that the Integers 1, 3, 7, and 6 were inserted in that order. Clearly, these two abstract values are different. However, they denote the "same observable" Prioritizer value. This is because it is not possible to use the provided operations to distinguish between these values. To see why, notice that Is_In_Insertion_Phase and Size_Of will not distinguish the values. Since Extract_Any returns arbitrary values of the objects, it is not a useful discriminator either. The only other operation of potential use in observing any difference is Extract, but it will never reveal a difference since it will always return 1, 3, 6, and 7 in that order for both p_1 and p_2. Hence, the specification in Figure 3.2 is not observable since it is not "possible to distinguish between (these) two values using the provided operations". In general, the concept in Figure 3.2 is not observable because any two Prioritizers whose "contents" strings are permutations of each other cannot be distinguished using the operations provided by the specification.

The impact of the observability problem in the present case becomes obvious in writing client programs that need to preserve a Prioritizer or test equality of two Prioritizers. That is, it is not possible to write typical implementations for the following kinds of specifications using Prioritizers, where Pre_P and Post_P are assertions involving p_1 and other parameters.

```
operation P ( preserves p1: Prioritizer,…)
   requires Pre_P
   ensures  Post_P
```

This is because the only way to manipulate a Prioritizer p1 is to dismantle it by using Extract and Extract_Any. Once dismantled, however, it cannot be restored to its original abstract value, but only to a permutation of the initial abstract string value. In other words, it is only possible to write implementations for specifications of the form:

```
operation P ( alters p1: Prioritizer, …)
   requires Pre_P
   ensures  Post_P and
            PERMUTATIONS (p1.contents, #p1.contents) and
            p1.insertion_phase = #p1.insertion_phase
```

More generally, it is not possible to implement any operation where the ensures clause involves the "=" operator on two Prioritizers (recall that "preserves p1" implies inclusion of "p = #p" in the ensures clause). For this reason, operations such as those below cannot be implemented.

```
operation  Are_Equal (
         preserves p1: Prioritizer
         preserves p2: Prioritizer
      ): boolean
   ensures  Are_Equal  iff  p1 = p2

operation  Were_Equal (
         consumes p1: Prioritizer
         consumes p2: Prioritizer
      ): boolean
   ensures  Were_Equal  iff  #p1 = #p2
```

The difficulty in doing such basic tasks with Prioritizers in turn inhibits the ability to write programs that are easy to construct, understand, or verify.

The observability problem with the specification in Figure 3.2 is a typical example of the situation where there are classes of computationally indistinguishable abstract values present in the specified model space. Such classes present unnecessary difficulty in reasoning about the behavior of the specification, thereby forcing a client to use alternative mental models in reasoning about its behavior along with sufficient mappings between them. In this case, a client has to constantly remember that all Prioritizers whose "contents" are permutations of one another are actually the "same" with respect to their (observable) behaviors. Clearly, this is "extra baggage" and it has nothing to do with the intended specification. The absence of observability can thus lead to higher development costs associated with the additional complexity placed upon software engineers in attempting to understand and reason about such specifications, as well as higher maintenance costs in the upkeep of software which is based on such specifications.

The specification in Figure 3.2 is controllable, however. This is because it is possible to construct any abstract string value in the state space. For example, the abstract value of p_1 can be reached by calling Insert with Integers 1, 7, 3, and 6 in that order, followed by a call to Change_Phase. In this fashion, it is straightforward to see that any Prioritizer value can be constructed.

Prioritizer_Template Specification Design #2

One approach for eliminating classes of computationally indistinguishable values and hence avoiding the non-observability problem is to constrain the model space to a smaller, more meaningful set of abstract values with appropriate re-specification of the operations as necessary. Figure 3.3 presents such an alteration of the previous specification for the Prioritizer_Template.

```
concept Prioritizer_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0   and
               for all x,y,z: Entry,
                  ARE_ORDERED (x,x) and
                  if ARE_ORDERED(x,y) and ARE_ORDERED(y,z) then
                     ARE_ORDERED(x,z) and
                  (ARE_ORDERED(x,y) or ARE_ORDERED(y,x))

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   type family Prioritizer is modeled by (
            contents: string of Entry,
            insertion_phase: boolean
         )
      exemplar p
      constraints
         |p.contents| <= Max_Size and
         for all alpha,beta: string of Entry and x,y: Entry,
            if p.contents = alpha * <x> * <y> * beta then
               ARE_ORDERED(x,y)
      initialization
         ensures  |p.contents| = 0 and
                  p.insertion_phase

   operation Insert (
            alters p: Prioritizer,
            consumes x: Entry
         )
      requires |p.contents| < Max_Size and
               p.insertion_phase
      ensures  p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                  #p.contents = alpha * beta and
                   p.contents = alpha * <#x> * beta  and
                  for all gamma: string of Entry and y: Entry,
                        if beta = <y> * gamma then
                           ARE_ORDERED (#x,y)

   operation Change_Phase (
            alters p: Prioritizer
         )
      ensures  p.contents = #p.contents and
               p.insertion_phase = not #p.insertion_phase
   operation Extract (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires |p.contents| > 0 and
               not p.insertion_phase
      ensures  #p.contents = <x> * p.contents and
               not p.insertion_phase

   operation Extract_Any (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires |p.contents| > 0
      ensures  p.insertion_phase = #p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                   p.contents = alpha * beta and
                  #p.contents = alpha * <x> * beta

   operation Is_In_Insertion_Phase (
            preserves p: Prioritizer
         ): Boolean
      ensures  Is_In_Insertion_Phase = p.insertion_phase

   operation Size_Of (
            preserves p: Prioritizer
         ): Integer
      ensures  Size_Of = |p.contents|

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Prioritizer_Template
```

Figure 3.3 - Prioritizer_Template Modeled Using Ordered Strings

The specification of Figure 3.3 is similar to the one in Figure 3.2 in that it still uses a mathematical string of entries to model the collection to be prioritized. However, this specification restricts the abstract value space by *constraining* it to contain only ordered strings of entries. It provides the same set of operations as in Figure 3.2. Other than this, the differences between the two specifications are in the ensures clauses of the Insert and Extract operations[5].

[5] Even though a specification may be doing the abstract "computation" at some point, this does not imply that a correct implementation for the specification must do the same thing. For example, even though the specification in Figure 3.3 appears to suggest an "insertion sorting" implementation, any other strategy (such as any one detailed in Chapter II) is certainly permissible as well [Weide 94, Sitaraman 97].

Evaluation of the Specification for Observability and Controllability

The specification in Figure 3.3 is observable. For two Prioritizer objects of different size, the Size_Of operation will detect the difference. For two Prioritizer objects with opposite values for their "insertion_phase" fields, the Is_In_Insertion_Phase operation will detect the difference. And for the previously problematic situation of having two objects whose "contents" are permutations of one another, the Extract operation will now detect a difference[6]. This is due to the fact that Extract is specified to remove and return the leftmost Entry from the string. Extracting and comparing the entries from two Prioritizer objects whose "contents" are permutations of each other will eventually reveal a difference between them. In particular, it is now possible to implement some of the operations where the ensures clause involves equality of two Prioritizer values, such as the Were_Equal operation in the last subsection.

The specification in Figure 3.3 is not controllable. For example, consider the Prioritizer_Template instantiated with an Entry type of "Two_Field_Record", where the first field is of type Integer and the second field is of type Character. Suppose the ordering for objects of Two_Field_Record is based on the "≤" relation on the Integer field. Next, consider two objects of this type, named p_3 and p_4, with values as follows:

p_3.contents: "(1,a) **(2,b)** (2,c) (3,d)" p_4.contents: "(1,a) (2,c) **(2,b)** (3,d)"
p_3.insertion_phase: false p_4.insertion_phase: false

The values for p_3 and p_4 above are within the specified state space; i.e., their "contents" strings are indeed ordered based on the Integer field. It is possible that successive calls to Insert will result in either value above. However, it is not *guaranteed* that either value in particular will be constructed. This is because the specification of Insert is relational, as it allows a new entry to be inserted on either side of an entry that is already in the string and is identically ordered. The specification in Figure 3.3, therefore, is not controllable since it is not "always possible to construct any given value from the specified state space using the provide operations."

[6] It should be noted that the constraining of the state space to contain only ordered "contents" strings has not completely eliminated the possibility of two Prioritizer variables having their "contents" strings being permutations of each other, as is illustrated in the following discussion of controllability.

To see a manifestation of the controllability problem, notice that it is not possible to write layered implementations of operations that need to “preserve” their parametric Prioritizers (e.g., Are_Equal as previously mentioned). This is essentially because it is not possible to create every value in the abstract space. The inability to preserve or copy Prioritizers hints at potential difficulty in using the concept.

The controllability problem with the specification in Figure 3.3 is a typical example of the situation where there are classes of non-guaranteeably constructable abstract values present in the model space as specified. As with the observability problem of the previous specification, such classes present unnecessary difficulty in reasoning about the behavior of the specification. A client must use alternative mental models when reasoning about the behavior of the specification. In this case, for example, a client has to constantly remember that all Prioritizers whose “contents” strings contain the same entries but have different left-to-right placement for identically ordered entries within each string are intuitively the “same” Prioritizer value. Clearly, this additional consideration has nothing to do with the intended specification. The absence of controllability can thus lead to higher software development and maintenance costs.

A Variation of Design #2

The controllability problem with the previous specification appears to suggest an easy solution: re-specify the Insert operation so that when an entry being inserted has the same ordering as an entry already in the string, the new entry is added to the right of the existing identically-ordered entry. Figure 3.4 below contains such a specification.

```
concept Priority_Queue_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0   and
               for all x,y,z: Entry,
                  ARE_ORDERED (x,x) and
                  if ARE_ORDERED(x,y) and ARE_ORDERED(y,z) then
                     ARE_ORDERED(x,z) and
                  (ARE_ORDERED(x,y) or ARE_ORDERED(y,x))

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   type family Priority_Queue is modeled by (
            contents: string of Entry,
            insertion_phase: boolean
         )
      exemplar p
      constraints
         |p.contents| <= Max_Size and
         for all alpha,beta: string of Entry and x,y: Entry,
            if p.contents = alpha * <x> * <y> * beta then
               ARE_ORDERED(x,y)
      initialization
         ensures  |p.contents| = 0 and
                  p.insertion_phase

   operation Insert (
            alters p: Priority_Queue,
            consumes x: Entry
         )
      requires |p.contents| < Max_Size and
               p.insertion_phase
      ensures  p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                  #p.contents = alpha * beta and
                   p.contents = alpha * <#x> * beta  and
                  for all gamma: string of Entry and y: Entry,
                        if beta = <y> * gamma then
                           ARE_ORDERED (#x,y) and
                           not ARE_ORDERED (y,#x)

   operation Change_Phase (
            alters p: Priority_Queue
         )
      ensures  p.contents = #p.contents and
               p.insertion_phase = not #p.insertion_phase
   operation Extract (
            alters p: Priority_Queue,
            produces x: Entry
         )
      requires |p.contents| > 0 and
               not p.insertion_phase
      ensures  #p.contents = <x> * p.contents and
               not p.insertion_phase

   operation Extract_Any (
            alters p: Priority_Queue,
            produces x: Entry
         )
      requires |p.contents| > 0
      ensures  p.insertion_phase = #p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                   p.contents = alpha * beta and
                  #p.contents = alpha * <x> * beta

   operation Is_In_Insertion_Phase (
            preserves p: Priority_Queue
         ): Boolean
      ensures  Is_In_Insertion_Phase = p.insertion_phase

   operation Size_Of (
            preserves p: Priority_Queue
         ): Integer
      ensures  Size_Of = |p.contents|

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Priority_Queue_Template
```

Figure 3.4 – Priority_Queue_Template, Correctly Modeled Using Ordered Strings

The specification in Figure 3.4 above is identical to that in Figure 3.3 with the exception of the specification for the Insert operation (and of course, the renaming of the type Prioritizer to Priority_Queue). The only difference in the specification of the Insert operation in Figure 3.4 is the addition of the last line of the ensures clause: "and not ARE_ORDERED (y,#x)". This forces the "left-to-right, first inserted-to-last inserted" placement of identically-ordered entries, thus providing the FIFO behavior of "same priority" entries that is needed by applications using priority queues.

This design is indeed observable and controllable, but it is a specification for a *different problem*: "stable" ordering; i.e., FIFO behavior for identically-ordered entries. This apparently simple "fix" unfortunately precludes all implementations that are not necessarily stable, such as quick-sort and heap-based ordering, and thus limits performance trade-offs. This restrictive specification is not demanded by the three classes of applications in Chapter II. What we need is a solution to the problem of specifying ordering without requiring stability. Such a solution is the topic of the next subsection.

Prioritizer_Template Specification Design #3

One alternative solution to the observability and controllability problem of design #2, without introducing additional constraints, is to add operations to the Prioritizer_Template interface that help distinguish every abstract value or generate any desired value. However, intuitively, the existing set of operations is just what is needed to perform the task of prioritizing. Adding other such operations, in addition to cluttering the interface, violates the pragmatic criterion in this case. For example, if Were_Equal were added to the interface in Figure 3.2 or some copy-enabling operations to the interface in Figure 3.3, then all implementations must keep a record of the order in which the entries are inserted; this precludes implementations such as those based on heaps. Adding operations in order to achieve the properties of observability and controllability only cures symptoms of the disease, and not the disease itself.

The fundamental problem with the first two specification designs is that they use a model that has too much structure for the problem of prioritizing. The ordering of entries within a string has no relevance to the problem being specified. Figure 3.5 shows an alternative Prioritizer_Template specification in which an unordered multi-set or “bag” of entries is used for modeling the collection of entries instead of strings.

```
concept Prioritizer_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0   and
               for all x,y,z: Entry,
                  ARE_ORDERED (x,x) and
                  if ARE_ORDERED(x,y) and ARE_ORDERED(y,z) then
                     ARE_ORDERED(x,z) and
                  (ARE_ORDERED(x,y) or ARE_ORDERED(y,x))

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   subtype INVENTORY_FUNCTION is function from Entry to integer
      exemplar f
      constraints
         for all x: Entry, f(x) >= 0

   definition INVENTORY_SIZE (
            f: INVENTORY_FUNCTION
         ): integer = sum of f(x) for all x: Entry

   definition IS_A_NEXT_ENTRY (
            f: INVENTORY_FUNCTION,
            x: Entry
         ): boolean = f(x) > 0 and for all y: Entry,
            if ARE_ORDERED(y,x) and not ARE_ORDERED(x,y)
            then f(y) = 0

   type family Prioritizer  is modeled by (
            contents: INVENTORY_FUNCTION,
            insertion_phase: boolean
         )
      exemplar  p
      constraints
         INVENTORY_SIZE(p.contents) <= Max_Size
      initialization
         ensures  INVENTORY_SIZE(p.contents) = 0 and
                  p.insertion_phase

   operation Insert (
            alters p: Prioritizer,
            consumes x: Entry
         )
      requires INVENTORY_SIZE(p.contents) < Max_Size and
               p.insertion_phase
      ensures  p.insertion_phase and
               p.contents(#x) = #p.contents(#x) + 1 and
               for all y: Entry,
                  if y /= #x
                  then p.contents(y) = #p.contents(y)
   operation Change_Phase (
            alters p: Prioritizer
         )
      ensures  p.contents = #p.contents and
               p.insertion_phase = not #p.insertion_phase

   operation Extract (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires INVENTORY_SIZE(p.contents) > 0 and
               not p.insertion_phase
      ensures  not p.insertion_phase and
               IS_A_NEXT_ENTRY(#p.contents,x) and
               p.contents(x) = #p.contents(x) - 1 and
               for all y: Entry,
                  if y /= x then p.contents(y) = #p.contents(y)

   operation Extract_Any (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires INVENTORY_SIZE(p.contents) > 0
      ensures  p.insertion_phase = #p.insertion_phase and
               p.contents(x) = #p.contents(x) - 1 and
               for all y: Entry,
                  if y /= x then p.contents(y) = #p.contents(y)

   operation Is_In_Insertion_Phase (
            preserves p: Prioritizer
         ): Boolean
      ensures  Is_In_Insertion_Phase = p.insertion_phase

   operation Size_Of (
            preserves p: Prioritizer
         ): Integer
      ensures  Size_Of = INVENTORY_SIZE(p.contents)

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Prioritizer_Template
```

Figure 3.5 - Prioritizer_Template Modeled Using Multi-Sets

In Figure 3.5, the *subtype* INVENTORY_FUNCTION serves as the model for the "contents" field of a Prioritizer object, by mapping each Entry to an integer which represents the number of occurrences of an Entry in the "bag". The *definition* INVENTORY_SIZE(f) returns the number of entries in the INVENTORY_FUNCTION f. The predicate IS_A_NEXT_ENTRY(f,x) is defined to be true if and only if x is a smallest entry in f.

In the *interface* section, the *type* Prioritizer is modeled as an ordered pair with a "contents" field and a boolean "insertion_phase" field. A Prioritizer's contents are modeled by an INVENTORY_FUNCTION. The *constraints* clause on the type Prioritizer is simple and it states that the size of a Prioritizer must be less than or equal to the Max_Size bound.

The ensures clause of Insert states that the only change to its contents is that the number of occurrences of the Entry x is increased by one, and that p remains in the insertion phase. The ensures clause of Extract states that p remains in the extraction phase, and that a next smallest entry based on the definition of ARE_ORDERED is removed and returned in x; the occurrence counts of other entries are not affected. The Extract_Any operation specifies that an arbitrary Entry x is removed and returned. The specifications for the remaining operations are straightforward.

Evaluation of the Specification for Observability and Controllability

The specification in Figure 3.5 is observable. For two Prioritizer objects of different size, the Size_Of operation will detect the difference. For two Prioritizer objects with opposite values for their "insertion_phase" fields, the Is_In_Insertion_Phase operation will detect the difference. Since there is no notion of the ordering of entries in a bag, the only way the contents of two abstract Prioritizer values can be different is if they are of different size or differ in their "contents" by at least one entry. When the contents differ, Extract or Extract_Any would reveal the difference. Hence, the specification in Figure 3.5 is observable.

The specification in Figure 3.5 is also controllable, because any abstract bag value can be constructed by inserting the constituent entries in any order. Unlike with the string model of the previous specification designs, the order of insertion is irrelevant since there is no notion of "one before the other" in a bag.

For a client using the specification in Figure 3.5, in particular it is possible to write totally correct layered implementations of operations that involve "=" in ensures clauses and operations that need to preserve Prioritizers. In other words, it provides sufficient and visible functionality. The specification in Figure 3.5 permits any of the usual sorting and prioritizing implementations, thus allowing clients to choose one that best suits their performance needs.

3.2 Discussion

Through the development of an observable and controllable formal specification for the Prioritizer_Template, we have shown that appropriate choices for mathematical models and explanations of objects is not always obvious or trivial. Indeed, in this example, there appeared to be nothing wrong, in terms of reasoning and comprehensiveness, with using a mathematical string, at least on the surface. In fact, in general, there is nothing wrong with using a mathematical string model when its inherent ordering (i.e., its extra "book-keeping") is appropriate. We have, for example, noted that the distinction between unstable and stable ordering forces the need to have separate concepts for each case. But the objective was to specify a data abstraction for a class of problems that did not care about the stability of ordering, and hence strings are unsuitable for modeling the objects. For applications that need a FIFO treatment for identically-ordered entries, a string model *is* an appropriate choice that can be specified in an observable and controllable manner.

The concepts of stable vs. unstable sorting are indeed orthogonal. While it is permissible to implement Prioritizer_Template using a stable ordering strategy, it is not permissible to implement the stable Priority_Queue_Template using an unstable ordering strategy. Furthermore, forcing a client to always use Priority_Queue_Template results in a performance bottleneck for those applications not requiring this behavior. Hence, the pragmatic criterion admits both concepts into a concept library.

In general, the problem of determining when a specification is observable and controllable can be difficult. This is particularly so with recast designs such as the Prioritizer_Template. And as indicated in the previous chapter, designing component interfaces that satisfy the pragmatic criterion is also in general a difficult problem. These difficulties motivate more formal and stringent tests for observability, controllability, and the pragmatic criterion, rather than having to rely on intuition for identifying "glitches". Developing formal characterizations for observability and controllability is the focus of the next chapter, where we also outline how the pragmatic criterion might be formalized.

Formalizations of Observability and Controllability IV

Formal specifications for object-based software components intended for reuse should be free of implementation bias, offer a high degree of functional and performance flexibility, and facilitate clear and precise reasoning. As discussed and exemplified in the previous chapters, the properties of observability, controllability, and the pragmatic criterion lead to such specifications. The objective of this chapter is to discuss formalizations for observability and controllability, in order to facilitate easier and more precise validation of candidate object-based specifications.

We begin with the following informal characterizations of observability and controllability from [Weide 96]:

Observability - A model-based specification S defining the program type ADT is *observable* if and only if every two unequal values in ADT's state space are "computationally distinguishable" using some combination of the operations of S.

Controllability - A model-based specification S defining the program type ADT is *controllable* if and only if every value in ADT's state space is "computationally reachable" using some combination of the operations of S.

Figure 4.1 - Intuitive Definitions for Observability and Controllability

Though these informal notions for observability and controllability appear to be rather straightforward and intuitive, it becomes clear that subtle differences in interpretation of the notions lead to several variations of formal characterizations. The intuitive definitions above raise at least two questions that force more specific statements of the properties. The first and central question arises from the consideration of specifications with *relational* behavior. Although we have motivated observability and controllability considerations from the use of the terms in traditional control engineering literature, relational specifications of behavior essential in software engineering complicate the meanings of the terms considerably [Desharnais 97, He 86, Leavens 91, Jones 90, Sitaraman 97]. In the above definitions in particular, does "computationally distinguishable" and "computationally reachable" mean that it is *always* possible to do so or that it is merely *possible* to do so? For example, consider the "computational reachability" question for the Prioritizer_Template in Figure 3.3 from the previous chapter that was modeled using ordered strings. In that specification, the ensures clause of the Insert operation leaves the arrangement of "equivalent" (identically-ordered) entries in a string unspecified. As noted there, while it is *possible* that certain strings will be generated with such a specification of Insert, it is not *always* true that a particular string will be generated. Alternatively, consider the reachability question for this specification in terms of its possible implementations. If this specification were implemented using a stable ordering strategy, then for that implementation it would be true that a particular string containing identically-ordered entries could *always* be achieved (when the abstract space corresponds to the implementation space isomorphically). If this specification, however, were instead implemented using an unstable sorting strategy, only the *possibility* exists for reaching some specific values. In other words, we can rephrase the first question as:

> Does "computationally distinguishable" and "computationally reachable" mean for *some* implementation(s) of the concept, or for *all* implementations?

The second question addresses the *asymmetry* between the informal definitions in Figure 4.1 above. In these definitions, observability is expressed as a relationship between two values whereas controllability is based on a single value in the abstract space of the type under consideration.

> Should the definitions be phrased in terms of relationships between two values (i.e., *relative*), or just in terms of one value (i.e., *absolute*), or perhaps something else?

As a result of these questions, Figure 4.2 below contains a "road map" depicting the different paths to consider when formalizing observability and controllability [Weide 96].

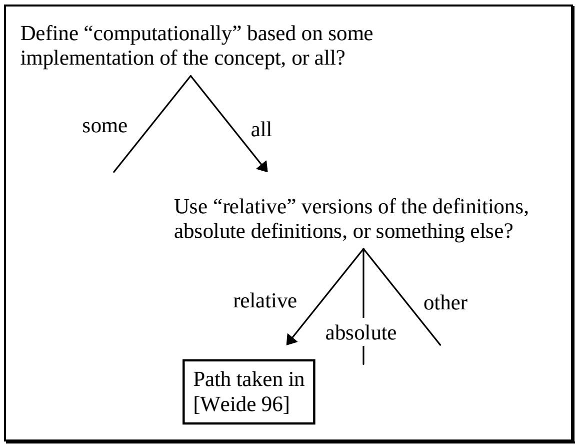

Define "computationally" based on some implementation of the concept, or all?
some
all
Use "relative" versions of the definitions, absolute definitions, or something else?
relative
absolute
other
Path taken in [Weide 96]

Figure 4.2 - Possible Interpretations for Defining Observability and Controllability

The rest of this chapter is organized as follows: Section 4.1 discusses code-based formalisms for observability and controllability. Section 4.2 introduces and formally describes scenarios for their subsequent use in defining observability and controllability. Unlike code-based definitions, the use of scenarios simplifies the fundamental notion of "total correctness" as it applies to the formalization of observability and controllability. Section 4.3 reexamines the Prioritizer_Template for the properties of observability and controllability using scenarios. Section 4.4 presents formal scenario-based definitions for observability and controllability as an alternative to the code-based definitions. Section 4.5 compares these definitions with similar notions in the literature.

4.1 Formal Code-Based Characterizations of Observability and Controllability

Observability and controllability of an object-based concept can be formally defined in terms of whether it is possible to write layered code for certain operations using calls to the basic operations provided by an object-based candidate concept (in addition to using operations available on any imported objects). The objective of this section is to explore different *code-based* formalizations of observability and controllability. This discussion as a whole leads to the discovery of interesting if not perplexing relationships among the many possible code-based definitions for observability and controllability. An earlier summary of the definitions based on our work together with Weide et al. appears in [Weide 96].

We begin this discussion by considering the case where the definitions hold for *all* implementations of the underlying concept. For observability, the informal description in Figure 4.1 suggests the following code-based definition:

A model-based specification S defining the program type ADT is *observable* if and only if there is a totally correct layered implementation of:

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

For controllability, the informal definition in Figure 4.1 suggests the following definition:

A model-based specification S defining the program type ADT is *controllable* if and only if for every constant c: ADT, there is a totally correct layered implementation of:

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

The two definitions above appear to state formally that which was stated informally in Figure 4.1. To address the asymmetry between these two definitions, we first consider a relative definition of controllability:

A model-based specification S defining the program type ADT is *relatively controllable* if and only if there is a totally correct layered implementation of:

```
operation  Get_Replica (
        preserves  x: ADT
        produces  copy_x: ADT
    )
   ensures  copy_x = x
```

Are_Equal and Get_Replica are symmetric now that they are both relative. However, the definitions are not independent of one another. This is because Are_Equal must *preserve* its arguments, and this apparently requires the ability to reconstruct or get a replica of each argument. That is, the ability to code Are_Equal hinges on the ability to code Get_Replica. Similarly, the first argument to Get_Replica must be preserved, and being able to prove this implicitly requires the ability to see if the incoming value and the outgoing value for this argument "are equal". This dependency is a result of the stipulation that the arguments to the definitions must be preserved. This observation suggests the following, more independent definitions:

A model-based specification S defining the program type ADT is *relatively observable* if and only if there is a totally correct layered implementation of:

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

A model-based specification S defining the program type ADT is *relatively controllable* if and only if there is a totally correct layered implementation of:

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

There is a variety of relationships among these sets of definitions for observability and controllability. For example, the ability to code Are_Equal in a layered fashion implies the ability to code Were_Equal. Also, the ability to layer Get_Replica implies the ability to layer Move. If it is possible to implement Were_Equal and Get_Replica, an implementation of Are_Equal can be readily constructed for any object as shown below:

```
procedure  Are_Equal (
                preserves  x1: ADT
                preserves  x2: ADT
           ) : Boolean
           copy1, copy2: ADT
        begin
           Get_Replica (x1, copy1)
           Get_Replica (x2, copy2)
           return Were_Equal (copy1, copy2)
        end Are_Equal
```

Also note that every object-based RESOLVE concept permits construction of the Move operation, because every object-based RESOLVE concept includes the Swap (":=:") operator on the provided type. Move is just "half a swap", as shown below.

```
procedure  Move (
         consumes  x1: ADT
         produces  x2: ADT
      )
   begin
      x1 :=: x2
   end Move
```

Figure 4.3 shows the relationships among the alternative relative definitions. Each region defines a set of specifications that meet the specific definitions of observability/controllability defined by that region.

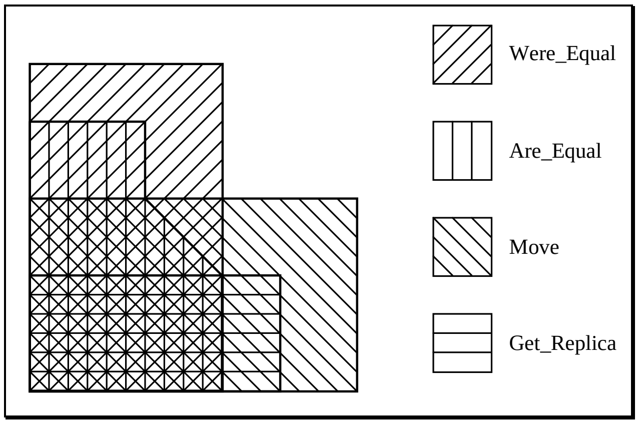


Figure 4.3 - Relationships Among the Relative Definitions

For example, the final design of the Prioritizer_Template specification in Chapter III, which is argued to be observable and controllable, happens to be placed in the most dense region of Figure 4.3 (the lower-left region). This is because it is possible to layer the operations Are_Equal and Get_Replica for Prioritizers defined in that specification. While it is not clear which of the alternative definitions are the best ones for evaluating candidate specifications, it is fair to say that a specification which fulfills all of the proposed definitions is observable and controllable (e.g., the Prioritizer_Template). This definition, unfortunately, might be too rigid for many reasonable specifications.

Before exploring other possible definitions of the terms, we note one other problem that needs to be tackled for parameterized concepts such as the Prioritizer_Template. For example, what assumptions can be made on the data types being passed in as parameters (and any other imported types)? In particular, if it is assumed that it is possible to enumerate over all the values of any imported types, then operations such as those discussed thus far can often be (inefficiently) layered on a concept even when the set of primary operations on the provided type is otherwise not sufficient for layering these operations. Hence, assuming the enumerability of any imported types makes it possible to deem patently poor specification designs as being observable and controllable. The enumerability on imported types "weakens the definitions so much that they are practically worthless" [Weide 96]. As a result, we stipulate that (the abstract data type provided by) a concept can only be deemed to be observable and controllable when it is assumed that any types provided through parameterization are also observable and controllable. For example, we say that it is possible to layer Are_Equal on the Prioritizer_Template so long as it is possible to make use of a similar Entries_Are_Equal operation for the type of Entry provided during instantiation.

The discriminating power of the relative code-based definitions becomes clear in considering variants of a concept for a generic Set abstraction. In [Weide 96], nine variants of a Set concept are discussed, where eight of the nine are deliberately lacking in quality (e.g., some variants are missing crucial Set operations), and one design is shown to be arguably a best design choice. In other words, the code-based definitions for observability and controllability admit only the one good design, and simultaneously make explicit why the poor designs fail. However, some questions still remain. For example, what about absolute versions of the definitions? And is there a way to reduce the resulting number of definitions to a more meaningful set? More importantly, is there a way to formalize the notion of "there exists a totally correct layered implementation?" Attempts to answer these questions motivate alternative formalizations of observability and controllability.

Code-Based Definitions Revisited

The objective of this subsection is to identify *absolute* versions of code-based definitions for observability and controllability, and to draw relationships between the absolute and relative definitions. The definition for absolute controllability, namely Construct_c, was among the first results above. Therefore, here we need only to define a class of absolute code-based definitions for observability. Below, we restate the previous informal definition for (relative) observability followed by a definition for absolute observability:

Relative Observability - A model-based specification S defining the program type ADT is *relatively observable* if and only if every two unequal values in ADT's state space are "computationally distinguishable" using some combination of the operations of S.

Absolute Observability - A model-based specification S defining the program type ADT is *absolutely observable* if and only if every value in ADT's state space is "computationally distinguishable" from every other value in ADT's state space using some combination of the operations of S.

The above definitions appear to be identical, thereby exemplifying the inadequacy of informal statements in separating the ideas. When formally stated, however, it is clear that the absolute notion of observability is quite different:

A model-based specification S defining the program type ADT is *absolutely observable* if and only if for every constant c: ADT, there is a totally correct layered implementation of:

```
operation  Is_Equal_To_c (
        preserves  x: ADT
     ): Boolean
   ensures  Is_Equal_To_c  iff  (x = c)
```

While this absolute definition of observability is symmetric with its controllability counterpart Construct_c, it is not independent. That is, the need to preserve the argument requires the ability to get a replica of the argument first. To make the two absolute definitions more independent, the definition of observability needs to be changed as follows:

A model-based specification S defining the program type ADT is *absolutely observable* if and only if for every constant c: ADT, there is a totally correct layered implementation of:

```
operation  Was_Equal_To_c (
        consumes  x: ADT
     ): Boolean
   ensures  Was_Equal_To_c  iff  (#x = c)
```

As in the case of the relative definitions, there are interesting relationships among absolute definitions as well. The ability to layer operations of the form Is_Equal_To_c implies the ability to layer operations of the form Was_Equal_To_c. More interesting are the intertwined relationships among the different absolute and relative definitions. For example, the ability to layer Are_Equal and operations of the form Construct_c implies the ability to layer operations of the form Is_Equal_To_c as follows:

```
operation  Is_Equal_To_c (
         preserves  x: ADT
      ) : Boolean
      copy_c: ADT
   begin
      Construct_c (copy_c)
      return Are_Equal (copy_c, x)
   end Is_Equal_To_c
```

Also, the ability to layer Get_Replica and operations of the form Was_Equal_To_c also implies the ability to layer operations of the form Is_Equal_To_c:

```
operation  Is_Equal_To_c (
         preserves  x: ADT
      ) : Boolean
      copy_x: ADT
   begin
      Get_Replica (x, copy_x)
      return Was_Equal_To_c (copy_x)
   end Is_Equal_To_c
```

Figure 4.4 below shows a summary of the implications among the various absolute and relative definitions for observability and controllability.

Are_Equal	*implies*	Were_Equal
Get_Replica	*implies*	Move
Were_Equal *and* Get_Replica	*implies*	Are_Equal
Is_Equal_To_c	*implies*	Was_Equal_To_c
Were_Equal *and* Construct_c	*implies*	Was_Equal_To_c
Are_Equal *and* Construct_c	*implies*	Is_Equal_To_c
Get_Replica *and* Was_Equal_To_c	*implies*	Is_Equal_To_c

Figure 4.4 - Relationships Among Different Code-Based Definitions

Figure 4.5 contains a depiction of the relationships among the absolute and relative definitions. Given that there can be $2^7 = 128$ possible regions, or “specification classes”, derived from the seven definitions of observability and controllability, the figure shows only those that can exist. Regions depicting relationships among the definitions that violate the implications in Figure 4.4 above cannot possibly exist. This reduces the total number of regions to at most *37 classifications* for specification designs. Figure 4.5 lists these classes of admissible specification designs.

Operation Legend

1	Are_Equal
2	Were_Equal
3	Get_Replica
4	Move
5	Is_Equal_To_c
6	Was_Equal_To_c
7	Construct_c

A "dot" means logical and.

Specification Classes

none		
2	1.2.4	2.4.5.6
4	1.2.6	2.4.6.7
6	1.4.6	2.5.6.7
7	2.5.6	3.4.5.6
1.2	2.6.7	4.5.6.7
2.4	3.4.7	1.2.4.5.6
2.6	4.5.6	1.2.5.6.7
3.4	4.6.7	2.4.5.6.7
4.6	5.6.7	3.4.5.6.7
4.7	1.2.3.4	1.2.3.4.5.6
5.6	1.2.4.6	1.2.4.5.6.7
6.7	1.2.5.6	1.2.3.4.5.6.7

Figure 4.5 - Classifying Specifications into 37 Possible Ways to Satisfy Different Definitions

For example, region ‘1’ (Are_Equal) is missing from Figure 4.5 because it is not possible for a specification to satisfy Are_Equal without satisfying anything else. In particular, Are_Equal implies Were_Equal, and this is why “1 and 2” *is* present in Figure 4.5. As another example, it is not possible for a specification to satisfy the definitions of Were_Equal and Construct_c without satisfying Was_Equal_To_c since this last operation can always be implemented using the first two. This is why the region “2 and 7” is not admissible in Figure 4.5 above, but “2 and 6 and 7” is.

A Discussion of the Alternative Code-Based Definitions

The identification of absolute and relative definitions for observability and controllability has led to a variety of relationships among them. However, this is perhaps more confusing than helpful, since the implications involving different definitions suggest that some definitions might be readily discarded. It might be true that "almost good" specifications could be better redesigned as a result of applying only a subset of the definitions. But it does not seem that a suitable basis of the definitions exists, given that there is no one definition that appears in all 37 combinations outlined in Figure 4.5.

In seeking to answer which code-based definitions are the "right" ones, we are left with one of two choices: either stipulate that a specification is observable and controllable if and only if it belongs to the class "1.2.3.4.5.6.7" in Figure 4.5, or pick one of the other 36 classes that seems most reasonable. The first choice might not be that bad, since we expect most object-based specifications should be suitable for use in layering any of the seven observability/controllability operations. It remains unclear, however, that this should be a stipulation for all practical object-based specifications.

The second choice seems more appropriate. For example, it is reasonable to think that the copying of values for certain data abstractions is undesirable (e.g., system-level or highly global structures). That is, some object-based specifications might wish to deliberately preclude the ability to layer Get_Replica. This suggests that the ability to layer operations of the form Construct_c, which in turn ensures the ability to reach every specified state, is a more appropriate condition for controllability. Similarly (and as a direct result), it is also reasonable for such specifications to deliberately preclude the ability to layer Are_Equal (in part since it might require Get_Replica), thereby suggesting that Were_Equal is a more appropriate condition for observability. More specifically, it is reasonable that specifications we intuitively would deem observable and controllable are those that facilitate the layering of Were_Equal and operations of the form Construct_c. The closest match from Figure 4.5 is the specification class "2 and 6 and 7" (Were_Equal, Was_Equal_To_c, and Construct_c), where '6' is a direct implication of "2 and 7".

It is arguable whether Were_Equal implies observability and Construct_c implies controllability, as there are perhaps equally plausible arguments advocating a different specification class from Figure 4.5. It is also arguable that attempting to define observability and controllability in a code-based fashion is not the best approach. The next section explores this last point by considering the use of *scenarios* in characterizing observability and controllability.

4.2 Scenarios

A key difficulty with the code-based definitions is their reliance on being able to define formally a “total correctness” of code that possibly involves calls to operations with relational specifications. Ogden has noted that current proof systems, including the one for RESOLVE as outlined in [Krone 88, Ernst 94, Sitaraman 97], are inadequate for the purpose [Ogden 97]. In addressing the problems and in proving the correctness of relational data abstraction implementations, Odgen has proposed the use of “scenarios”. While this research is still in progress, we apply the idea of scenarios to alternative and direct definitions of observability and controllability without having to define total correctness semantics for code involving loops over relationally specified abstract data types.

A scenario is essentially a sequence of calls to operations from a given object-based specification, and in this sense is similar to a program trace on a given piece of code. A scenario does not involve any programming constructs such as loops or conditional statements. Rather, a scenario simply involves a sequence of operation calls.

Additionally, unlike a program trace in which variables of any known type might be declared, a scenario involves only variables of the type provided by the object-based specification on which the scenario acts. That is, a scenario does not utilize variables of imported types; in a scenario, arguments of imported types are given as specific constant values rather than variable names. A key reason for the different treatment between the provided type and imported types is to allow one to investigate interesting aspects regarding the manipulation of the state space of the type provided by a specification, without having to be concerned (or overwhelmed) with similar aspects of the types imported into the specification. Another way to view this is that a scenario lets one “plug in” desired values for arguments of imported types so that attention can be focused on the effects these values have on variables of the provided type.

To exemplify the use of scenarios, Figure 4.6 below shows a simple relational specification of a concept that exports a type with a state space of only two values, appropriately called Two_Valued_Facility, and an example scenario σ on this concept. It should be noted that the syntax of the scenario σ in Figure 4.6 is not entirely precise, and that we will discuss the specifics following this example. For the concept

Two_Valued_Facility, all parameters are specified to be of alters mode because it is the most general of all specification modes. Other parameter modes, such as preserves, are treated as involving additional conjunctions in the ensures clauses.

```
concept Two_Valued_Facility

      uses Standard_Integer_Facility

   type family Two_Valued   is modeled by
                            enumeration {value1, value2}
         exemplar t
            initialization
               ensures  t=value1

   operation P(alters x: Two_Valued, alters i: Integer)
      ensures  i > #i and (x=value1 or x=value2)

   operation Q(alters x: Two_Valued)
      requires x=value1
      ensures  x=value1 or x=value2

end Two_Valued_Facility
```

$$\sigma = < x\text{: Two\_Valued},\ P(x,23) \rightarrow (x,36),\ Q(x) \rightarrow (x) >$$

Figure 4.6 - Specification of the Concept “Two_Valued_Facility” and an Example Scenario σ

The scenario σ in Figure 4.6 begins by declaring one variable x of type Two_Valued, which is initialized according to its “initialization ensures” clause; i.e., x has the value “value1”. Next, the scenario σ has a call to operation P with arguments x and integer 23 resulting in (the “→” is read “results in”) a possibly different value for x and integer 36. The scenario concludes with a call to operation Q having parameter x, resulting in a possibly different value for x.

Because of the relational behavior specified in the ensures clauses of P and Q, the scenario σ actually describes several possible paths of computation, since it does not state explicit values for the variable x. This capability proves useful in dealing with observability and controllability in a uniform fashion for handling functional and relational specifications of operations. Figure 4.7 below shows the possible paths of computation that σ characterizes:

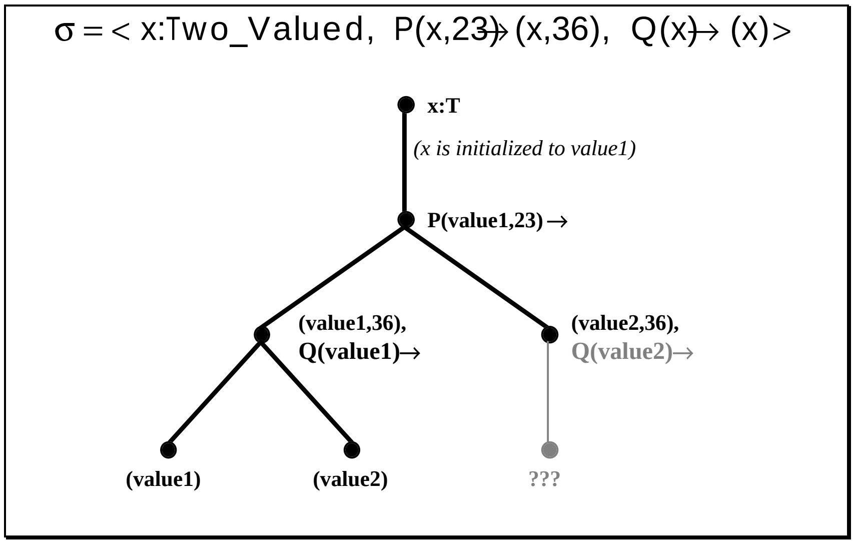


Figure 4.7 - Possible Paths of Computation Through Scenario σ for the Concept Two_Valued_Facility

The above scenario σ raises an interesting issue regarding its legitimacy with respect to the concept Two_Valued_Facility. For example, if the value of x upon calling operation Q turns out to be value2, then this violates the pre-condition for Q in Figure 4.7 above. For our purposes, we are interested only in *legitimate* scenarios, and we define these to be scenarios that describe only paths of computation which do not violate any pre- or post-conditions (such as the bold-shaded paths in Figure 4.7 above). Since σ above contains a path that violates a pre-condition, σ is not a legitimate scenario on Two_Valued_Facility objects. Given this intuitive meaning and role of scenarios, we are now ready to define the terms formally.

Syntax for Scenarios

We define a scenario to be a 4-tuple, each part of which is defined formally in subsequent pages:

```
Scenario == (
      base_ai: Abstract_Instance,
      var_decl_set: set of Variable_Declaration,
      op_call_seq: sequence of Operation_Call,
      alt_init_val_set: set of (n,v) where n ∈ Name and
                        v ∈ Constrained_Model_Space
   )
```

The attribute “base_ai” constitutes the context for a scenario, and it is the abstract instance of the concept under consideration. This abstract instance may be a non-generic concept that provides a type (e.g., Two_Valued_Facility in Figure 4.6 that uses the Standard_Integer_Facility) or an instantiated generic concept (e.g., Integer_Prioritizer, obtained by instantiating Prioritizer_Template with Integer) [Edwards 95]. The definition of scenarios, defined to act only on abstract instances, poses no difficulties in using scenarios to investigate properties of generic concepts. This is because any type T used in instantiating a generic concept becomes an imported type, and in every scenario the values of type T are provided as specific constant values, thus factoring out the particular computational expressiveness of type T. Therefore, for definitions of observability and controllability based on scenarios, what holds for an instantiated generic concept holds for the (uninstantiated) generic concept as well.

The attribute “var_decl_set” is the set of variables used in a scenario, and they must all be of the type provided by base_ai. The attribute “op_call_seq” is the sequence of operation calls in a scenario. The attribute “alt_init_val_set” is used for stating explicit initial values for certain variables, thereby overriding any values imposed on the mentioned variables by their initialization assertion. This flexibility proves useful for fully decoupling the definitions of observability and controllability. Each variable-value pair in alt_init_val_set consists of a *Name*, which is some allowable variable name, and a value from a *Constrained_Model_Space*, which is the specified mathematical model space for a type[7]. If alt_init_val_set is empty, then all variables are assumed to have values consistent with the initialization assertion of the type provided by base_ai.

Expressed in this syntax, the (illegitimate) scenario σ from Figure 4.6, for example, becomes:

$$\begin{array}{l} \sigma = (\\ \quad \text{Two\_Valued\_Facility}, \{(x\text{: Two\_Valued})\}, < P(x,23) \rightarrow (x,36), Q(x) \rightarrow (x) >, \{\} \\ ) \end{array}$$

[7] It should be noted here that we do not make explicit the particular syntax or rules governing any *italicized* words used throughout the definitions, but simply offer a brief intuitive description of each.

We continue by defining the terms used in “Scenario” above. An abstract instance (for our purposes) is defined to be:

```
Abstract_Instance == (
      type: Type_Declaration,
      op_decl_set: set of Operation_Declaration,
      uses_set: set of Abstract_Instance
   )
```

The attribute “type” is the data type exported by the abstract instance. In general, an abstract instance could actually provide multiple data types, and at times this is necessary. However, in this chapter, we restrict our attention to the more common abstract instances that provide only one type. The attribute “op_decl_set” denotes the set of operations exported by the abstract instance. It suffices to say here that each operation involves only arguments of the provided type or of any imported type. The attribute “uses_set” is the set of abstract instances used by the abstract instance.

A type declaration in an abstract instance is defined to be:

```
Type_Declaration == (
      name: Name,
      model_space: Constrained_Model_Space,
      exemplar: Name,
      init: Assertion
   )
```

The attribute “name” is the name of the type itself. The attribute “model_space” is the (possibly constrained) mathematical model used to model the type. In general, an abstract instance could contain module-level or “global” state information. A typical example would be “counters” that keep track of the number of variables of the provided type that have been declared. While it is certainly reasonable to pose observability and controllability questions for concepts containing global state information, we leave the details for future work. The attribute “exemplar”, used in the initialization assertions of an abstract instance, is needed for substituting formal parameters with actual parameters when evaluating the initialization assertion for the type. The attribute “init” is the initialization assertion for variables of the type, where “*Assertion*” denotes a valid logical assertion in RESOLVE.

The declaration of an operation (in an abstract instance) is defined to be:

```
Operation_Declaration == (
      op_name: Name,
      formal_seq: sequence of Argument,
      pre: Assertion,
      post: Assertion
   )
```

The attribute “op_name” is the name of the operation. The attribute “formal_seq” is the formal parameter sequence of the operation. The attribute “pre” is the pre-condition of the operation, and the attribute “post” is the post-condition of the operation.

We define a variable declaration to be:

```
Variable_Declaration == (
      var_name: Name,
      type_name: Name
   )
```

The attribute “var_name” is the name of the variable. The attribute “type_name” is the type of the variable.

An operation call (in a scenario) is defined as:

```
Operation_Call == (
      op_name: Name,
      in_seq: sequence of Argument,
      out_seq: sequence of Argument
   )
```

The attribute “name” is the name of the operation being called. The attribute “in_seq” is the input parameter sequence of arguments passed to the operation. The attribute “out_seq” is the output parameter sequence of arguments as a result of the operation call.

Finally, we define an argument to be:

```
Argument == (
      var_name: Name | const_value: Constrained_Model_Space,
      type_name: Name
   )
```

If the argument is a variable name, then the attribute "var_name" denotes the name of the variable. If the argument is some constant value instead, then the attribute "const_value" denotes the value of the argument. The attribute "type_name" is the type of the argument. For convenience, we assume the structural equivalence of a Variable_Declaration and an Argument. This is useful for state mapping purposes, where range values can be mapped from variable names as well as from argument names that are actually variable names.

Hypotheses for Well-Formed Scenarios

Having stated the various attributes of a scenario and defined the terms used therein, we are now ready to discuss when a given 4-tuple actually constitutes a well-formed scenario. There are three hypotheses to be met for a scenario to be well-formed. The Acceptable Variable Declaration Set Hypothesis mandates that a scenario only involves variables of the type provided by the abstract instance on which the scenario acts. The Acceptable Operation Call Sequence Hypothesis states the details concerning a valid operation call. The Acceptable Alternate Initial Value Set Hypothesis ensures that only declared variables are given explicit values and that their values are reasonable.

1) Acceptable Variable Declaration Set Hypothesis

A scenario can only have variables of the type provided by the abstract instance on which the scenario acts (i.e., base_ai). This is stated as follows:

```
definition Has_Acceptable_Variable_Declaration_Set (
         σ: Scenario
      ): boolean =
   ∀ vd ∊ σ.var_decl_set,
      vd.type_name = σ.base_ai.type.name
```

2) Acceptable Operation Call Sequence Hypothesis

There are several constraints on the kinds of things that can appear in an operation call. First of all, the operation must exist in the abstract instance on which the scenario acts. The other constraints state that the input and output argument sequences must be consistent, that arguments of the provided type must be variables, and that arguments of imported types must be constants.

```
definition Has_Acceptable_Operation_Call_Sequence (
         σ: Scenario
      ): boolean =
   ∀ op_call ∈ σ.op_call_seq,
      ∃ op_decl ∈ σ.base_ai.op_decl_set ∋
         Matching_Call_And_Declaration (op_decl, op_call) and
         Matching_Argument_Sequences (
               op_call.in_seq, op_call.out_seq,
               σ.var_decl_set, σ.base_ai
            )
```

where

```
definition Matching_Call_And_Declaration (
         od: Operation_Declaration,
         oc: Operation_Call
      ): boolean =
   od.op_name = oc.op_name and
   |od.formal_seq| = |oc.in_seq| and
   ∀ i: integer, 1 ≤ i ≤ |oc.in_seq|,
      od.formal_seq(i).type_name = oc.in_seq(i).type_name


definition Matching_Argument_Sequences (
         in_seq: sequence of Argument,
         out_seq: sequence of Argument,
         var_decl_set: set of Variable_Declaration,
         base_ai: Abstract_Instance
      ): boolean
   |in_seq| = |out_seq| and
   ∀ i: integer, 1 ≤ i ≤ |in_seq|,
      if in_seq(i).type_name = base_ai.type.name then
         ∃ vd ∈ var_decl_set ∋
            in_seq(i).var_name = vd.var_name and
            out_seq(i).var_name = vd.var_name
      else
         ∃ ai ∈ base_ai.uses_set ∋
            in_seq(i).type_name = ai.type.name and
            out_seq(i).type_name = ai.type.name and
            in_seq(i).const_value ∈ ai.type.model_space and
            out_seq(i).const_value ∈ ai.type.model_space
```

3) Acceptable Alternate Initial Value Set Hypothesis

This hypothesis ensures that the alternate initial value set for a scenario involves only variables that have been declared, and that the values for such variables are from the specified state space of the type of the variable. This is stated as follows:

```
definition Has_Acceptable_Alternate_Initial_Value_Set (
         σ: Scenario
      ): boolean =
   ∀ alt_val ∊ σ.alt_init_val_set,
      ∃ vd ∊ σ.var_decl_set and
         c ∊ σ.base_ai.type.model_space ∍
            alt_val = (vd.var_name, c)
```

The three hypotheses above are characteristic of all well-formed scenarios. The definition of Is_Well_Formed_Scenario below summarizes this notion:

```
definition Is_Well_Formed_Scenario (
         σ: Scenario
      ): boolean =
   Has_Acceptable_Variable_Declaration_Set(σ) and
   Has_Acceptable_Operation_Call_Sequence(σ) and
   Has_Acceptable_Alternate_Initial_Value_Set(σ)
```

In the next subsection, we discuss the semantic issues regarding scenarios, so that we can precisely state when a given scenario is legitimate.

State Semantics and Legitimacy of Scenarios

To define when a scenario is legitimate, we use the state spaces after each operation call in the scenario. Given that scenarios are concerned with relational data abstraction specifications in general, it is possible that the state space after each operation call has multiple variable-to-value mappings for each variable. This observation requires us to denote the state of a scenario as a *spectrum*; i.e., a set of mappings from variable names to their values as shown below. Additionally, if a pre- or post-condition does not hold for some operation call in a scenario, or if a scenario is not well-formed, then the spectrum for that scenario contains the (bottom) state $\perp$. We denote the spectrum for scenario σ as $S(\sigma)$, which has the structure:

```
      S(σ) ==  set of s: var_name → σ.base_ai.type.model_space
               where
                  ∀ vd ∈ σ.var_decl_set,
                     ∃ k ∈ σ.base_ai.type.model_space ∍
                        s(vd.var_name) = k
               union ⊥
```

We are now ready to define the semantics of S(σ) formally:

```
   S(σ) =
      if not Is_Well_Formed_Scenario (σ) then ⊥

      else if |σ.op_call_seq| = 0 then
         set of s where Is_Initial_State (σ, s)

      else
         set of s where Is_In_Spectrum (σ′, op_call, s)
         union ⊥ if Already_Has_Or_Produces_Bottom (σ′, op_call)

         and where
            σ.op_call_seq = σ′.op_call_seq • <op_call> and
            σ.base_ai = σ′.base_ai and
            σ.var_decl_set = σ′.var_decl_set and
            σ.alt_init_val_set = σ′.alt_init_val_set
```

The above characterization makes use of three sub-definitions, termed Is_Initial_State, Is_In_Spectrum, and Already_Has_Or_Produces_Bottom, which are defined below:

```
      definition Is_Initial_State (
               σ: Scenario,
               s: var_name → σ.base_ai.type.model_space
            ): boolean =
         ∀ alt_val ∈ σ.alt_init_val_set,
            ∀ vd ∈ σ.var_decl_set,
               if ∃ c ∈ σ.base_ai.type.model_space ∍
                  alt_val = (vd.var_name, c) then
                     s(vd.var_name) = c
               else
                  σ.base_ai.type.init [
                     σ.base_ai.type.exemplar → s(vd.var_name)
                  ]
```

The predicate Is_Initial_State asserts that a state s is “initial” when all variables not mentioned in the alternate initial value set map to a value satisfying the initialization clause, and when all variables that are mentioned in the alternate initial value set map to the specified value.

```
definition Is_In_Spectrum (
         σ: Scenario,
         op_call: Operation_Call,
         s: var_name → σ.base_ai.type.model_space
      ): boolean =
   ∃ s’ ∈ S(σ) and op_decl ∈ σ.base_ai.op_decl_set ∋
      Matching_Call_And_Declaration (op_decl, op_call) and
      Meets_Pre (op_decl, op_call, σ.base_ai, s’) and
      Meets_Post (op_decl, op_call, σ.base_ai, s’, s) and
      ∀ vd ∈ σ.var_decl_set,
         ∀ arg ∈ op_call.in_seq,
            if vd.var_name ≠ arg.var_name then
               s(vd.var_name) = s’(vd.var_name)
```

Is_In_Spectrum asserts that a state s is in the spectrum when: 1) there exists some state s’ from the spectrum before the current operation call such that the pre- and post-conditions hold, and 2) the mappings for all non-participating variables are not affected (i.e., they are the same between s and s’). Is_In_Spectrum makes use of two sub-definitions, Meets_Pre and Meets_Post, which are defined to be:

```
definition Meets_Pre (
         op_decl: Operation_Declaration,
         op_call: Operation_Call,
         base_ai: Abstract_Instance,
         s’: var_name → base_ai.type.model_space
      ): boolean =
   op_decl.pre [
      for i from 1 to |op_decl.formal_seq|,

      op_decl.formal_seq(i)
         →
      if op_call.in_seq(i).type_name = base_ai.type.name then
         s’(op_call.in_seq(i).var_name)
      else  op_call.in_seq(i).const_value
   ]
```

The definition for Meets_Pre evaluates the pre-condition of the operation in question by substituting the formal parameters (i.e., those appearing in the operation declaration in the abstract instance) with the actual parameters. For “actual” parameters that are variables, the values obtained by applying the state function s’ to those variables are given. For “actual” parameters that are constant values (of some imported type), those constant values are given. The expression “[**for** i **from** 1 **to** n, a(i) → b(i)]” means that, for i from 1 to n, substitute the expression a(i) with the expression b(i).

```
definition Meets_Post (
         op_decl: Operation_Declaration,
         op_call: Operation_Call,
         base_ai: Abstract_Instance,
         s’: var_name → base_ai.type.model_space
         s: var_name → base_ai.type.model_space
      ): boolean =
   op_decl.post [
      for i from 1 to |op_decl.formal_seq|,

      #op_decl.formal_seq(i)
         →
      if op_call.in_seq(i).type_name = base_ai.type.name then
         s’(op_call.in_seq(i).var_name)
      else  op_call.in_seq(i).const_value,

      op_decl.formal_seq(i)
         →
      if op_call.out_seq(i).type_name = base_ai.type.name then
         s(op_call.out_seq(i).var_name)
      else op_call.out_seq(i).const_value
   ]
```

The definition for Meets_Post evaluates the post-condition of the operation in question by substituting the formal parameters with the actual parameters. Essentially, there are two groups of formal parameters to deal with. The first group of formal parameters represents the values passed into the operation, and are designated using a “#” sign, as in “#od.formal_seq(i)” above. This is the syntax used in the RESOLVE notation. For this group of formal parameters, the substitution expression is identical to that used in the definition of Meets_Pre.

The second group of formal parameters represents the values resulting from the operation call, and are designated by the absence of the “#” sign, as in “od.formal_seq(i)” above. For this group, the parameters in the output sequence of the current operation call that are variables are substituted with their corresponding values from the state function s, whereas the parameters that are constant values are simply substituted with those constant values.

Finally, the definition of Already_Has_Or_Produces_Bottom is given as:

```
definition Already_Has_Or_Produces_Bottom (
         σ: Scenario,
         op_call: Operation_Call
      ): boolean =
   ⊥ ∈ S(σ) or
   ∃ s' ∈ S(σ) and op_decl ∈ σ.base_ai.op_decl_set ∍
      Matching_Call_And_Declaration (op_decl, op_call) and
      (
         ~Meets_Pre (op_decl, op_call, σ.base_ai, s') or
         ∀ t: var_name → σ.base_ai.type.model_space,
            ~Meets_Post (op_decl, op_call, σ.base_ai, s', t)
      )
```

Already_Has_Or_Produces_Bottom captures the effect on the spectrum of a scenario when either a pre-condition or a post-condition is violated. More precisely, if for some operation call, there is a state having values that violate the operation's pre-condition, then the resulting state in this case is ⊥. Similarly, if for some operation call, there is state such that the pre-condition holds, but for which the post-condition can never hold (due to illegitimate constant values in the scenario), then the resulting state in this case is ⊥. In either case, ⊥ becomes part of the spectrum and never leaves.

Having defined the state semantics of a scenario, we can now say that a scenario σ is legitimate if and only if ⊥ is not an element of S(σ). This is captured by the following definition:

```
definition Is_Legitimate_Scenario (
         σ: Scenario
      ): boolean =
   ⊥ ∉ S(σ)
```

4.3 Evaluation of Prioritizer_Template Specifications Using Scenarios

This section exemplifies the utility of scenarios by evaluating the Prioritizer_Template specifications from Chapter III for the properties of observability and controllability. In the process, it also lays the intuitive foundation for the formal definition of the terms using scenarios in the next section.

Using Scenarios to Show Non-Observability

Scenarios can be used to show witness to the (non-) observability of an abstract instance. Observability is essentially the ability to show a difference between two known-to-be-different values, and for scenarios such a difference can be revealed through the constant *indicator* values appearing in the output sequences of operation calls.

The basic idea of using scenarios for the purpose of observability is as follows: We start with two given different values k1 and k2 from the space of type provided by the abstract instance that is under evaluation. We then consider two *similar* scenarios σ1 and σ2 on that abstract instance which begin with the values k1 and k2, respectively, for some candidate variable of the provided type. By "similar", we mean that σ1 and σ2 have the same context, the same set of variable declarations, and call the same operations using the same input parameter sequences. In other words, the output sequences resulting from the operation calls may be different between σ1 and σ2, both in terms of the values for variables as well as the indicator values.

If both σ1 and σ2 are legitimate scenarios, and if they have different indicator values between them, then this shows the *possibility* of getting distinguishing results, which means that the abstract instance under evaluation *might* be observable. If this is the case, then all that remains is the question of whether we could instead "erase" the indicator values from σ1 and σ2 and legitimately "write down" some same sequence of indicator values in both σ1 and σ2. If so, then at this point the "replaced" versions of σ1 and σ2 are in fact identical in all respects, thus demonstrating the possibility of getting nondiscriminating results. This situation means that the abstract instance under evaluation is not observable. To the contrary, if we find that it is impossible to "erase" and "write down" some same sequence of indicator values in σ1 and σ2, then the abstract instance is deemed observable.

As an example, consider the Prioritizer_Template based on unconstrained strings in Figure 3.2. This specification is not observable, since the "erase and replacement" procedure described above is in fact possible for some different values k1 and k2. For example, consider the values k1 and k2 and scenarios σ1 and σ2 below for the abstract instance Integer_Prioritizer, which is obtained by instantiating Prioritizer_Template with Integer:

```
k1 = (<37, 43>, false)
k2 = (<43, 37>, false)

σ1 = (
   Integer_Prioritizer,{p:Integer_Prioritizer},op_call_seq1,{(p,k1)}
)
and
σ2 = (
   Integer_Prioritizer,{p:Integer_Prioritizer},op_call_seq2,{(p,k2)}
)

where
   op_call_seq1 = <
      Extract (p,3) → (p,37), Extract (p,17) → (p,43)
   >
   and
   op_call_seq2 = <
      Extract (p,3) → (p,37), Extract (p,17) → (p,43)
   >
```

As exemplified by σ1 and σ2 above, Extract called on two strings which are permutations of each other can (and always will) result in the same sequence of indicator values (which is pointed out by the **bold** constants). If Extract_Any were to be used instead of Extract, it is still possible that identical indicator values would result. Hence, the Prioritizer_Template specification based on the unconstrained string model is not observable.

Using Scenarios to Show Non-Controllability

Scenarios can be used to show witness to the (non-) controllability of an abstract instance. Controllability is essentially the ability to generate any "target" value k of the type provided by an abstract instance "from scratch" (i.e., from an initialized variable). This capability can be shown using scenarios as follows: for all target values k, if there is a scenario σ having some variable such that, for all mappings of that variable in S(σ), the variable maps to the target value k, then the abstract instance under evaluation is controllable.

As an example, consider the Prioritizer_Template based on ordered strings in Figure 3.3. This specification is not controllable, since it is not possible to demonstrate the existence of "generating" scenarios for all target values k. For example, consider an abstract instance Record_Prioritizer, which is obtained by instantiating Prioritizer_Template with Two_Field_Record. For this record, let the first field be Integer and the second be

Character. Also, assume that the records are to be prioritized based on "≤" ordering on the Integer value. Consider the following target value k and scenario σ:

```
k = (<(1,'a'), (2,'b'), (2,'c')>, true)

σ = (
   Record_Prioritizer, {p:Record_Prioritizer}, op_call_seq, {}
)

where
   op_call_seq = <
      Insert (p,(1,'a')) → (p,(0,'')),
      Insert (p,(2,'b')) → (p,(0,'')),
      Insert (p,(2,'c')) → (p,(0,''))
   >
```

The Insert operation alone is as useful as any combination of the other operations in attempting to generate a given string value. Therefore, without loss of generality, we can narrow our consideration to only those scenarios that involve calls to Insert, such as σ above.

The scenario σ could possibly result in the given target value k. However, we have seen in the previous chapter that, for an entry (record) that has the same ordering as one that is already in the Prioritizer (i.e., (2,'b') and (2,'c') above), the Insert operation could place that entry on either side of the existing one. That is, the scenario σ above could *possibly* generate the target value k, but cannot *guarantee* that it will. Indeed, it is straightforward to see (and can be formally argued using the spectrum for σ) that there is no scenario that can guarantee the generation of the intended target value k in this case. Hence, the specification of Prioritizer_Template based on ordered strings is not controllable.

Discussion of an Observable and Controllable Specification for Prioritizer_Template

The final Prioritizer_Template specification in Figure 3.5, modeled using a multi-set, was deemed observable and controllable. The goal here is to reaffirm this conclusion using scenarios.

We begin with observability. When two Prioritizers (multi-sets) are of different sizes, a pair of legitimate and similar scenarios involving a single call to Size_Of must always have different indicator values for the sizes. Likewise, for two Prioritizers in different phases, Is_In_Insertion_Phase serves the purpose. The more interesting observability question is whether it is always possible to differentiate between two same-size-same-phase Prioritizers that differ in their contents by at least one entry. Consider the two Record_Prioritizer (as previously instantiated) values k1 and k2 and scenarios σ1 and σ2:

```
k1 = ({(2,‘a’), (2,‘b’), (3,‘d’)}, false)
k2 = ({(2,‘a’), (2,‘b’), (3,‘c’)}, false)

σ1 = (
   Record_Prioritizer,{p:Record_Prioritizer},op_call_seq1,{(p,k1)}
)
and
σ2 = (
   Record_Prioritizer,{p:Record_Prioritizer},op_call_seq2,{(p,k2)}
)

where
   op_call_seq1 = <
      Extract (p,(7,‘f’)) → (p,(2,‘a’)),
      Extract (p,(2,‘h’)) → (p,(2,‘b’)),
      Extract (p,(3, ‘’)) → (p,(3,‘d’))
   >
   and
   op_call_seq2 = <
      Extract (p,(7,‘f’)) → (p,(2,‘a’)),
      Extract (p,(2,‘h’)) → (p,(2,‘b’)),
      Extract (p,(3, ‘’)) → (p,(3,‘c’))
   >
```

In this pair of legitimate and similar scenarios, the third call to Extract results in the different indicator values (3, ‘d’) and (3, ‘c’). Furthermore, for σ1 and σ2 above, it is not possible to “erase” the indicator values and legitimately “replace” them with some same sequence of indicator values. Indeed, no matter which two different Record_Prioritizer values are given, a sufficient number of calls to Extract will reveal a difference between them. For all different Record_Prioritizer values k1 and k2, there is always a pair of legitimate and similar scenarios σ1 and σ2 that will distinguish between the two values, and hence, the Prioritizer_Template specification is deemed observable.

For controllability, arguing that there is a scenario σ that will always generate a given target value k is straightforward. For a given target multi-set value k, simply construct a scenario σ involving a series of calls to Insert, one for each entry in k. For example, consider the Record_Prioritizer value k below and scenario σ that generates it:

```
k = ({(2,‘a’), (2,‘b’), (3,‘d’)}, true)

σ = (
   Record_Prioritizer, {p:Record_Prioritizer}, op_call_seq, {}
)

where
   op_call_seq = <
      Insert (p,(3,‘d’)) → (p,(0,‘’)),
      Insert (p,(2,‘a’)) → (p,(0,‘’)),
      Insert (p,(2,‘b’)) → (p,(0,‘’))
   >
```

It is easy to see (and can be formally argued using the spectrum for σ) that the variable p has (and can only have) the value k at the end of σ. Indeed, for all target values k, it is straightforward to see that there is a scenario σ involving a sufficient number of calls to Insert that always results in the value k. Hence, the Prioritizer_Template specification based on multi-sets is deemed controllable.

The manner in which we utilized scenarios to characterize observability and controllability throughout the Prioritizer_Template example suggests that more formal characterizations involving scenarios are possible. Indeed this is the case, and the topic of the next section.

4.4 Formal Scenario-Based Characterizations of Observability and Controllability

Utilizing scenarios to formally explain the notions of observability and controllability is the topic of this section. The definitions are more formal than code-based versions because they do not evade defining “total correctness” of layered implementations on relational data abstractions.

A Formal Scenario-Based Definition for Observability

Figure 4.8 below shows a formal characterization of observability based on scenarios.

```
definition Is_Observable (
         base_ai: Abstract_Instance
      ): boolean =
   ∀ k1, k2 ∈ base_ai.type.model_space,
      if k1 ≠ k2 then
         ∃ σ1: Scenario = (
            base_ai, var_decl_set, op_call_seq1, {(p, k1)}
         ) and
         σ2: Scenario = (
            base_ai, var_decl_set, op_call_seq2, {(p, k2)}
         ) ∍
            Is_Legitimate_Scenario (σ1) and
            Is_Legitimate_Scenario (σ2) and
            Similar_Scenarios (σ1, σ2) and
            Always_Different_Indicators (σ1, σ2, k1, k2)
```

Figure 4.8 - A Scenario-Based Definition for Observability

The definition for Similar_Scenarios formally captures the notion of two "similar" scenarios as used in the previous section, and is defined to be:

```
definition Similar_Scenarios (
         σ1, σ2: Scenario
      ): boolean =
   σ1.base_ai = σ2.base_ai and
   σ1.var_decl_set = σ2.var_decl_set and
   |σ1.op_call_seq| = |σ2.op_call_seq| and
   ∀ i: integer, 1 ≤ i ≤ |σ1.op_call_seq|,
      σ1.op_call_seq(i).op_name = σ2.op_call_seq(i).op_name and
      σ1.op_call_seq(i).in_seq = σ2.op_call_seq(i).in_seq
```

Intuitively stated, two scenarios are similar if they have the same context, the same set of variable declarations, and call the same operations using the same input parameter sequences. That is, they may differ only in their output constant values (i.e., indicator values) as well as their alternate initial value sets.

The definition of Always_Different_Indicators formally expresses the impossibility of performing the indicator "erase and replace" procedure discussed in the previous section, and is given as:

```
definition Always_Different_Indicators (
         σ1, σ2: Scenario,
         k1, k2 ∈ σ1.base_ai.type.model_space
      ): boolean =
   ~ ∃ γ1: Scenario = (
         base_ai, var_decl_set, op_call_seq1, {(p, k1)}
      ) and
      γ2: Scenario = (
         base_ai, var_decl_set, op_call_seq2, {(p, k2)}
      ) ∍
      Is_Legitimate_Scenario (γ1) and
      Is_Legitimate_Scenario (γ2) and
      Similar_Scenarios (σ1, γ1) and
      Similar_Scenarios (σ2, γ2) and
      ∀ i: integer, 1 ≤ i ≤ |γ1.op_call_seq|,
         γ1.op_call_seq(i).out_seq = γ2.op_call_seq(i).out_seq
```

A Formal Scenario-Based Definition for Controllability

Figure 4.9 below shows a formal characterization of controllability based on scenarios.

```
definition Is_Controllable (
         base_ai: Abstract_Instance
      ): boolean =
   ∀ k ∈ base_ai.type.model_space,
      ∃ σ: Scenario ∍
         Is_Legitimate_Scenario (σ) and
         Always_Reaches_Target_From_Scratch (σ, k)
```

Figure 4.9 - A Scenario-Based Definition for Controllability

The definition for Always_Reaches_Target_From_Scratch is given as:

```
definition Always_Reaches_Target_From_Scratch (
         σ: Scenario,
         k ∈ σ.base_ai.type.model_space
      ): boolean =
   σ.alt_init_val_set = {} and
   ∃ vd ∈ σ.var_decl_set ∍
      ∀ s ∈ S(σ),
         s(vd.var_name) = k
```

Intuitively, this definition holds when a scenario σ begins with all variables initialized and ends with at least one variable that maps to the target value k for all states in S(σ).

4.5 Related Work

While there is hardly unanimity about the precise meaning of observability, researchers with slightly dissimilar interpretations generally agree that it is an essential property of sufficiently abstract software component descriptions. Other terms used to describe the idea are “freedom from implementation bias” [Jones 80,90] and “full abstraction” [Milner 77, He 86, Nipkow 87]. A good example of a difference in interpretation, however, is that observable specifications exist for which there are provably correct implementations requiring abstraction relations [Sitaraman 97, Leavens 91], yet in the view of some researchers these same specifications have an implementation bias (are not fully abstract), since no abstraction function exists for some implementations. Jones gives the following characterization of observability [Jones 90]:

> “A model-oriented specification is based on an underlying set of states. The model is biased (with respect to a given set of operations) if there exist different elements of the set of states which cannot be distinguished by any sequence of the operations.”

He gives an example of a biased (i.e., non-observable) model-based specification for a Queue similar in spirit to the one in Figure 1.7. The specification models a Queue with a sequence of queue entries and an index into the sequence that designates the front entry of the Queue. The operations are specified such that a Queue value (unnecessarily) maintains a history of the Queue’s elements. For example, the following two different Queue model values actually designate the same “intuitive” queue of characters “bc”, where ‘b’ is the front, and this is reflected by the fact that the provided operations on Queues can never distinguish between such values:

q1: ([a,b,c], 1)   q2: ([b,c], 0)

There are at least two important ramifications that result from this exercise: First, a client trying to use such a non-observable specification as a basis for understanding the intuitive behavior of queues faces unnecessary complexity by having to continually keep in mind that values such as those above are actually the “same” queue. Second, the only provably correct implementations for this so-called specification are those that (at least) maintain a history of the queue’s entries. Clearly this is undesirable, and exposes its implementation bias.

While the utility of observable specifications is unquestioned, many researchers debate the necessity of having to determine whether a specification is observable. Jones himself, for example, after evaluating a “number of specifications” states that in his experience, “very few specifications were found to have been biased,” and that “it is therefore not envisaged that this proof obligation need normally be discharged in a formal way.” However, in our experience we have found that, by far, most initial attempts at specification result in non-observable designs, and that it certainly is not trivial to determine if a specification is observable. This is particularly so among specifications of newer and nontrivial relational data abstraction specifications with provably correct implementations that require abstraction relations (optimization problems are good examples [Sitaraman 97]).

In [Kapur 80], the notion of an “expressively complete” specification is presented. The ideas in this work are similar in spirit to the code-based definitions. There, an expressively complete specification is one that has an operation set “adequate enough to implement all computable functions on its values.” For controllability purposes, this means that it is possible to construct any value in the specified state space; however, this does not imply that this can be done efficiently. In particular, expressive completeness relies on the assumption that one can *enumerate* over all the values (in the model spaces) of the imported types. Enumeration is used to implement certain layered operations where the provided operation set is not adequate to *efficiently* do so. A typical example is a data abstraction for some collection that does not permit the efficient (or even at all the) removal of an entry, but can answer whether a particular entry is in the container when asked about that entry. Therefore, as a last resort for computing certain operations, one can enumerate over all possible values of the contained type one at a time, asking whether each such entry is contained. However, it becomes clear that such an assumption may not apply to data types in general, and therefore does not scale well when attempting to design generic data abstractions.

Kapur et al. then remove this assumption by defining the notion of an “expressively rich” specification. An expressively rich specification (for a data abstraction) “is expressively complete with an operation set that is rich enough to conveniently extract from a value, all relevant information required to reconstruct the value from scratch.” For example, a specification for any container object should have at least one operation that can conveniently remove entries from the container. In this fashion, it is possible to efficiently perform operations such as making a copy of some value, among others, without needing to enumerate over all values of the contained entry's type.

As in [Weide 96], Kapur et al. consider an example of a specification for a generic "Set" data abstraction with operations "insert(s,x)", "remove(s,x)", "is_defined(s,x)", and "size(s)". Assuming that it is possible to enumerate over values of a Set, the Set abstraction is expressively complete, but not expressively rich. For example, it may seem that making a copy of a Set of integers is not possible, since there is no way to remove values from a Set without knowing that a particular value is in the Set. However, by using "is_defined", we can check for Set membership of an integer value, one at a time, by enumerating over (the finite space of) all the integers. Each time we determine an integer to be in the Set value, we simply "insert" an integer of the same value into a new Set value. In this (inefficient) manner, we can make a copy of a Set value. Indeed, any computable function on Set values can be handled in a similar fashion. But to make the use of the Set abstraction efficient and practical, we need to make the Set abstraction expressively rich. By adding an operation that allows us to arbitrarily remove an entry from a Set value, such as "remove_any(s,x)", we can efficiently make a copy of a Set value without needing enumeration of the contained type. This makes the computation of any function on Set values possible in an efficient and convenient manner. In terms of this chapter, the Set specification with remove_any is controllable.

The use of scenarios in characterizing observability and controllability for generic data abstractions with relational specifications stands apart from similar work in the literature. To our knowledge, observability and controllability have never been formally defined in the context of relational specifications for generic data abstractions.

Validating the RESOLVE Concept Library

V

The previous chapters have focused on the development and explanation of formal and practical tests for the evaluation of model-based specifications of object-based software components. The dual purpose of this chapter is to illustrate the utility of these tests in evaluating actual formal specification designs, and to motivate the unique collection of RESOLVE object-based concepts based on [Ogden 96]. For each concept presented, we give an overview of the problem that it addresses, discuss the concept in terms of the pragmatic criterion, and discuss observability and controllability aspects of the concept using the scenario-based definitions for these properties. We discuss other relevant issues for each concept where appropriate.

We present fourteen specifications in all, ranging from concepts that capture basic data structures, such as stacks and queues, to those that capture more complex and relational behavior such as partial maps and minimum spanning forest abstractions.

5.1 Concepts Modeled Using Strings

In this section, we present five concepts whose behaviors are modeled using mathematical strings. The first three of these concepts are singly-bounded: generic queues, stacks, and priority queues. In singly-bounded concepts, each object of the type is bounded by a "max-size" constant supplied by the client at the time of instantiation. The last two concepts in this section are communal-bounded: generic lists and preemptable queues. In communal-bounded concepts, all the objects of the type from an instantiation are bounded collectively by a "total-bound" constant supplied by the client at the time of instantiation.

Bounded Queue_Template

```
concept Queue_Template (
            type Entry,
            constant Max_Length: Integer
         )
      requires Max_Length > 0

      uses  Standard_Integer_Facility

   type family Queue is modeled by string of Entry
      exemplar q
      constraints |q| <= Max_Length
      initialization
         ensures  |q| = 0

   operation Enqueue (
            alters   q: Queue,
            consumes x: Entry
         )
      requires |q| < Max_Length
      ensures  q = #q * <#x>

   operation Dequeue (
            alters   q: Queue,
            produces x: Entry
         )
      requires |q| > 0
      ensures  #q = <x> * q

   operation Swap_Front (
            alters   q: Queue,
            alters   x: Entry
         )
      requires |q| > 0
      ensures  there exists α: string of Entry such that
               q = <#x> * α and #q = <x> * α

   operation Length_Of (
            preserves   q: Queue
         ): Integer
      ensures  Length_Of = |q|

   operation Allowed_Max_Length (): Integer
      ensures  Allowed_Max_Length = Max_Length

end Queue_Template
```

Figure 5.1 - Queue_Template

Overview of the Concept

Figure 5.1 above shows a specification for a bounded generic queue abstraction. The behavior of a queue is modeled using a mathematical string of entries, with the constraint that a string cannot exceed the length specified by Max_Length. The provided operations are those typical to most queue data structures, with the exception of the Swap_Front operation.

Is it Pragmatic?

Enqueue, Dequeue, Length_Of, and Allowed_Max_Length form a functional basis on abstract queue values. The Swap_Front operation could indeed be layered using the others, and is therefore not needed functionally. As discussed in Chapter I, the Swap_Front operation is useful for “one-look-ahead” applications such as parsing and others in which there is a need to inspect the next entry before committing to its permanent removal. As a layered operation, Swap_Front takes linear time dependent on the length of the queue. However, as an intrinsic operation it can be implemented to perform in constant time. Hence, the provided set of operations is orthogonal when both functionality and performance are considered. Therefore, Queue_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

Queue_Template is observable. For any two different queue values k1 and k2, there are two legitimate and similar scenarios σ1 and σ2 such that their indicator values cannot be “erased” and legitimately “replaced” with some identical sequence of indicator values. There are two ways in which k1 and k2 could be different: they could have different lengths, or have the same length but different (arrangements of the same) entries. For the first case, σ1 and σ2 (with a representative variable that starts with the values k1 and k2, respectively) involving a single call to Length_Of must always have different indicator values (lengths) between them. For the second case, σ1 and σ2 involving |k1| successive calls to Dequeue must always have different indicator values (entries) between them.

Queue_Template is controllable. There is a scenario σ for every queue value k such that σ starts with an initialized queue variable and ends with that variable mapping to k for all states in S(σ). For example, σ could simply involve successive calls to the Enqueue operation, with the entries enqueued in the left-to-right order of the entries in k.

Other Issues

There are other possibilities for modeling the abstract behavior of a queue, and some are also observable and controllable. For example, a queue could instead be modeled as a sequence of the entries with an index pointing to the front of the queue. However, this model is not as intuitive or understandable as the string model, since it has more structure than is needed for the purpose. The specification of the operations in terms of a sequence involves significant manipulation of the indices of the entries' positions during enqueueing and dequeueing. Another alternative choice is to model the queue as a function from integers to entries, as in a "circular array" implementation. However, as discussed in Chapter I, queues modeled in this fashion are not observable or controllable.

Bounded Stack_Template

```
concept Stack_Template (
            type Entry,
            constant Max_Depth: Integer
         )
      requires Max_Depth > 0

      uses  Standard_Integer_Facility

   type family Stack is modeled by string of Entry
      exemplar s
      constraints |s| <= Max_Depth
      initialization
         ensures  |s| = 0

   operation Push (
            alters   s: Stack,
            consumes x: Entry
         )
      requires |s| < Max_Depth
      ensures  s = <#x> * #s

   operation Pop (
            alters   s: Stack,
            produces x: Entry
         )
      requires |s| > 0
      ensures  #s = <x> * s

   operation Depth_Of (
            preserves   s: Stack
         ): Integer
      ensures  Depth_Of = |s|

   operation Allowed_Max_Depth (): Integer
      ensures  Allowed_Max_Depth = Max_Depth

end Stack_Template
```

Figure 5.2 - Stack_Template

Overview of the Concept

Figure 5.2 above shows a specification for a bounded generic stack abstraction. The behavior of a stack is modeled using a mathematical string of entries, with the constraint that a string cannot exceed the length specified by Max_Depth.

Is it Pragmatic?

All of the operations provided form a functional basis on stack values. No one operation can be implemented in terms of the others, either in terms of functionality or performance. Hence, the provided operation set is orthogonal and satisfies the pragmatic criterion.

Is it Observable and Controllable?

Stack_Template is observable. For any two different stack values k1 and k2, there are two ways in which k1 and k2 could be different: they could have different depths, or have the same depth but different (arrangements of the same) entries. For the first case, two similar scenarios σ1 and σ2 involving a single call to Depth_Of must always have different indicator values (depths) between them. For the second case, σ1 and σ2 involving |k1| successive calls to Pop must always have different indicator values (entries) between them.

Stack_Template is controllable. An example scenario σ for a stack value k starts with an initialized stack variable followed by successive calls to the Push operation, with the entries pushed in the right-to-left order of the entries in k.

Other Issues

As is most often the case for string-modeled abstractions, another way in which the abstract behavior of a stack could be modeled is to use a sequence with an index to the top of the stack. However, for the same reasons discussed in the context of queues above, this model is more complex than essential for this purpose.

Bounded Priority_Queue_Template

```
concept Priority_Queue_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0   and
               for all x,y,z: Entry,
                  ARE_ORDERED (x,x) and
                  if ARE_ORDERED(x,y) and ARE_ORDERED(y,z) then
                     ARE_ORDERED(x,z) and
                  (ARE_ORDERED(x,y) or ARE_ORDERED(y,x))

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   type family Priority_Queue is modeled by (
            contents: string of Entry,
            insertion_phase: boolean
         )
      exemplar p
      constraints
         |p.contents| <= Max_Size and
         for all alpha,beta: string of Entry and x,y: Entry,
            if p.contents = alpha * <x> * <y> * beta then
               ARE_ORDERED(x,y)
      initialization
         ensures  |p.contents| = 0 and
                  p.insertion_phase

   operation Insert (
            alters p: Priority_Queue,
            consumes x: Entry
         )
      requires |p.contents| < Max_Size and
               p.insertion_phase
      ensures  p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                  #p.contents = alpha * beta and
                   p.contents = alpha * <#x> * beta  and
                  for all gamma: string of Entry and y: Entry,
                        if beta = <y> * gamma then
                           ARE_ORDERED (#x,y) and
                           not ARE_ORDERED (y,#x)

   operation Change_Phase (
            alters p: Priority_Queue
         )
      ensures  p.contents = #p.contents and
               p.insertion_phase = not #p.insertion_phase
   operation Extract (
            alters p: Priority_Queue,
            produces x: Entry
         )
      requires |p.contents| > 0 and
               not p.insertion_phase
      ensures  #p.contents = <x> * p.contents and
               not p.insertion_phase

   operation Extract_Any (
            alters p: Priority_Queue,
            produces x: Entry
         )
      requires |p.contents| > 0
      ensures  p.insertion_phase = #p.insertion_phase and
               there exists alpha,beta: string of Entry
               such that
                   p.contents = alpha * beta and
                  #p.contents = alpha * <x> * beta

   operation Is_In_Insertion_Phase (
            preserves p: Priority_Queue
         ): Boolean
      ensures  Is_In_Insertion_Phase = p.insertion_phase

   operation Size_Of (
            preserves p: Priority_Queue
         ): Integer
      ensures  Size_Of = |p.contents|

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Priority_Queue_Template
```

Figure 5.3 - Priority_Queue_Template

Overview of the Concept

Figure 5.3 above shows a specification for a bounded priority queue abstraction, and it is slightly different from the one in [Odgen 96]. The behavior of a priority queue is modeled using a mathematical string of entries, with the constraints that the entries in a string are ordered and that a string cannot exceed the length specified by Max_Size.

Is it Pragmatic?

Following the rationale in Chapter II, in the context of the Prioritizer_Template, the specification of Priority_Queue_Template also satisfies the pragmatic criterion. The only distinction between the two concepts is that the specification for Prioritizer_Template is not concerned with the notion of stability whereas Priority_Queue_Template specifies stable ordering.

Is it Observable and Controllable?

Priority_Queue_Template is observable. For any two different priority queue values k1 and k2, there are three ways in which k1 and k2 could be different: they could have different sizes, be in different phases, or have the same size and phase but different (arrangements of the same) entries. For the first case, two similar scenarios σ1 and σ2 (for k1 and k2 respectively) involving a single call to Size_Of must always have different indicator values (sizes) between them. For the second case, σ1 and σ2 involving a single call to Is_In_Insertion_Phase must always have different indicator values (phases) between them. For the third case, σ1 and σ2 involving |k1| successive calls to Extract must always have different indicator values (entries) between them.

Priority_Queue_Template is controllable. An example scenario σ for a priority queue value k starts with an initialized priority queue variable followed by successive calls to the Insert operation, with the entries inserted in the left-to-right order of the entries in k, and is ended by a call to Change_Phase if k is in the extraction phase.

Communal List_Template

```
concept List_Template (
            type Entry,
            constant Max_Total_Length: Integer
         )
      requires Max_Total_Length > 0

      uses  Standard_Integer_Facility

   type family List is modeled by (
            preceding: string of Entry,
            remaining: string of Entry
         )
      exemplar l
      initialization
         ensures  |l.preceding| = 0 and
                  |l.remaining| = 0

   definition Total_Length: integer =
      sum from i = 1 to List.Last_Specimen_Num of
         |List.Denoted_By(i).preceding| +
         |List.Denoted_By(i).remaining|
      constraints Total_Length <= Max_Total_Length

   operation Insert (
            alters   l: List,
            consumes x: Entry
         )
      requires Total_Length < Max_Total_Length
      ensures  l.preceding = #l.preceding and
               l.remaining = <#x> * #l.remaining

   operation Remove (
            alters   l: List,
            produces x: Entry
         )
      requires |l.remaining| > 0
      ensures  l.preceding = #l.preceding and
               #l.remaining = <x> * l.remaining

   operation Advance (
            alters   l: List
         )
      requires |l.remaining| > 0
      ensures  l.preceding * l.remaining =
                  #l.preceding * #l.remaining and
               |l.preceding| = |#l.preceding| + 1

   operation Move_To_Start (
            alters   l: List
         )
      ensures  |l.preceding| = 0 and
               l.remaining = #l.preceding * #l.remaining
   operation Move_To_End (
           alters   l: List
        )
      ensures  |l.remaining| = 0 and
               l.preceding = #l.preceding * #l.remaining

   operation Length_Of_Preceding (
           preserves   l: List
        ): Integer
      ensures  Length_Of_Preceding = |l.preceding|

   operation Length_Of_Remaining (
           preserves   l: List
        ): Integer
      ensures  Length_Of_Remaining = |l.remaining|

   operation Swap_Remainders (
           alters   l1: List,
           alters   l2: List
        )
      ensures  l1.preceding = #l1.preceding and
               l2.preceding = #l2.preceding and
               l1.remaining = #l2.remaining and
               l2.remaining = #l1.remaining

   operation Swap_Preceding_Entry (
           alters   l: List,
           alters   x: Entry
        )
      requires |l.preceding| > 0
      ensures  l.remaining = #l.remaining and
               there exists α: string of Entry such that
                  #l.preceding = α * <x> and
                   l.preceding = α * <#x>

   operation Available_Capacity (): Integer
      ensures  Available_Capacity =
               (Max_Total_Length - Total_Length)

   operation Allowed_Max_Total_Length (): Integer
      ensures  Allowed_Max_Total_Length =
               Max_Total_Length

end List_Template
```

Figure 5.4 - List_Template

Overview of the Concept

Figure 5.4 above shows a specification for a generic one-way list abstraction. This concept is communally-bounded, which means that all list variables from an instance of the template share the space allotted by the client-supplied value for Max_Total_Length, rather than each single list variable having its own maximum bound.

Unlike most textbooks, where lists and list operations are explained in terms of pointer structures, in List_Template a list is modeled using a pair of mathematical strings. The string labeled “preceding” contains those entries to the left of the insertion point, and the string labeled “remaining” contains those entries to the right of the insertion point. This view is analogous to a line of text in a word processor, where the characters to the left of the cursor are “preceding”, and those to the right of the cursor are “remaining”.

Is it Pragmatic?

Insert, Remove, Advance, Move_To_Start, Length_Of_Remaining, Available_Capacity, and Allowed_Max_Total_Length form a functional basis of the provided operation set. In other words, the operations Swap_Remainders, Swap_Preceding_Entry, Move_To_End, and Length_Of_Preceding can be implemented using the other operations. However, in a typical pointer-based implementation, each of these four operations can be implemented to work in constant time when provided as a primary operation of the interface.

Swap_Remainders is useful for “preserving the cursor” in instances where repositioning the cursor would otherwise take linear time dependent on the size of the “preceding” string. Many operations, such as List_Copy, can benefit from this operation. For example, consider the (recursive) code for Entry_Is_In_Remainder below:

```
procedure  Entry_Is_In_Remainder (
         l: List,
         x: Entry
      ): Boolean
   ensures
      Entry_Is_In_Remainder iff
      there exists alpha,beta: string of Entry such that
         l.remaining = alpha * <x> * beta

   variables
      temp: List;
      y: Entry;
      found: Boolean;
   begin
      found := false;
      Swap_Remainders (l, temp);
      Remove (temp, y);

      if Are_Equal (x,y) then
         found := true;
      else if Entry_Is_In_Remainder (temp, x)
         found := true;
      end if

      Insert (temp, y);
      Swap_Remainders (l, temp);
      return found;
   end
```

Without Swap_Remainders, repositioning the cursor after performing the above search would take linear time dependent on the size of the "preceding" string of the list.

Swap_Preceding_Entry is useful for "one-look-behind" applications, much like Swap_Front on queues is useful for "one-look-ahead" applications as discussed in Chapter I. For example, when searching for a given entry in a list that is maintained in a specified order, it is convenient to see whether the entry preceding the cursor is "greater" than the given entry. If not, then the list will have to be reset before beginning the search. If it isn't, then the search can begin from the current cursor position. This permits efficient searches for entries that fall in a "gap" in the ordered list. As an intrinsic operation, Swap_Preceding_Entry can always be implemented to perform in constant time, thus making it suitable for such applications.

Move_To_End could be performed by calling Advance repeatedly. However, this procedure takes linear time, arguing for the inclusion of Move_To_End as an intrinsic operation, in which case it can be implemented with constant time performance. Similarly, both Length_Of_Preceding and Length_Of_Remaining are included so that they can be implemented with constant time performance.

In terms of the pragmatic criterion, while the functionality of the four operations discussed is subsumed by the other operations, the performance is not subsumable by the others. Hence, the entire operation set is orthogonal and List_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

List_Template is observable. Any two different list values k1 and k2 could differ in their "preceding" or "remaining" lengths, or have the same lengths but different (arrangements of the same) entries. For the first case, σ1 and σ2 involving a call to Length_Of_Preceding followed by a call to Length_Of_Remaining must always have different indicator values (lengths) between them. For the second case, σ1 and σ2 involving a call to Move_To_Start followed by |k1| successive calls to Remove must always have different indicator values (entries) between them.

List_Template is also controllable. There is a scenario σ for every list value k such that σ starts with one initialized list variable followed by a sequence of calls to Insert, with the entries inserted in the right-to-left order of the entries in k, and then followed by calls to Advance until the cursor position of k is reached.

Other Issues

Sitaraman et al. discuss the drawbacks of modeling the behavior of a list as a function from indices to entries [Sitaraman 93]. Even though such a concept can be made to be observable and controllable with appropriate list constraints, it is clear that the explanations of the operations are not as intuitive or understandable.

Communal Preemptable_Queue_Template

```
concept Preemptable_Queue_Template (
            type Entry,
            constant Max_Total_Length: Integer
         )
      requires Max_Total_Length > 0

      uses  Standard_Integer_Facility

   type family Preemptable_Queue is modeled by string of Entry
      exemplar q
      initialization
         ensures  |q| = 0

   definition Total_Length: integer =
      sum from i = 1 to Preemptable_Queue.Last_Specimen_Num
         of |Preemptable_Queue.Denoted_By(i)|
      constraints Total_Length <= Max_Total_Length

   operation Enqueue (
            alters   q: Preemptable_Queue,
            consumes x: Entry
         )
      requires Total_Length < Max_Total_Length
      ensures  q = #q * <#x>

   operation Dequeue (
            alters   q: Preemptable_Queue,
            produces x: Entry
         )
      requires |q| > 0
      ensures  #q = <x> * q

   operation Inject (
            alters   q: Preemptable_Queue,
            consumes x: Entry
         )
      requires Total_Length < Max_Total_Length
      ensures  q = <#x> * #q

   operation Swap_Rear (
            alters   q: Queue,
            alters   x: Entry
         )
      requires |q| > 0
      ensures  there exists α: string of Entry such that
               q = α * <#x> and #q = α * <x>

   operation Append (
            alters   q1: Preemptable_Queue,
            alters   q2: Preemptable_Queue
         )
      ensures  q1 = #q1 * #q2 and |q2| = 0
   operation Length_Of (
            preserves   q: Preemptable_Queue
         ): Integer
      ensures  Length_Of = |q|

   operation Available_Capacity (): Integer
      ensures  Available_Capacity =
                  Max_Total_Length - Total_Length

   operation Allowed_Max_Total_Length (): Integer
      ensures  Allowed_Max_Total_Length = Max_Total_Length

end Preemptable_Queue_Template
```

Figure 5.5 - Preemptable_Queue_Template

Overview of the Concept

Figure 5.5 above shows a specification for a communal-bounded generic preemptable queue abstraction. As with Queue_Template discussed earlier, the behavior of a preemptable queue is modeled using a mathematical string of entries. However, the interface for a preemptable queue is quite different than that of an "ordinary" queue. By "preemptable", we mean that entries can be enqueued into the opposite end as well.

Is it Pragmatic?

Enqueue, Dequeue, Length_Of, Available_Capacity, and Allowed_Max_Total_Length form a functional basis on queue values. That is, the operations Inject, Swap_Rear, and Append could instead be layered using the others, and are therefore not needed functionally. As intrinsic operations in a pointer-based implementation, however, they can be implemented with constant time performance.

Inject, which subsumes the Swap_Front operation of previous designs both in terms of functionality and performance, is useful for enqueueing entries into the normally "output-only" end of a queue. Swap_Rear provides the same kind of functionality as that of Swap_Front, except that it works on the input end of the queue. Unlike in a bounded queue implemented using an array, in the communally-bounded Preemptable_Queue_Template, Append can be implemented with constant time performance if the implementation is pointer-based or if it is such that the queues share a common array. As a layered operation, Append would take a linear amount of time. Thus, Preemptable_Queue_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

Preemptable_Queue_Template is observable and controllable. This is easily seen, given that we have already shown Queue_Template to be observable and controllable, and that Preemptable_Queue_Template subsumes the functionality of Queue_Template.

Other Issues

While Preemptable_Queue_Template itself satisfies the pragmatic criterion, it is not obvious that a concept library containing both the bounded Queue_Template as well as Preemptable_Queue_Template satisfies the pragmatic criterion. Though the former is subsumed by the latter in terms of functionality and duration considerations, space considerations argue for the inclusion of both concepts. While the pragmatic criterion does not address the issue of single versus communal bounds explicitly, it is reasonable to conclude that both bounding schemes are useful depending on client needs. A "communal" Queue_Template cannot be justified in this library, because Preemptable_Queue_Template, as specified, would subsume that concept under all considerations.

5.2 Concepts Modeled Using Sets, Functions, and Bags

In this section, we present three concepts whose behaviors are modeled using mathematical sets, functions, or bags. None of these concepts is communal. Partial_Map_Template is modeled using sets, Almost_Constant_Function_Template is modeled using functions, and Prioritizer_Template is modeled using bags (as discussed in Chapter III).

Bounded Partial_Map_Template

```
concept Partial_Map_Template (
            type D_Entry,
            type R_Entry,
            constant Max_Size: Integer
         )
      requires Max_Size > 0

      uses  Standard_Integer_Facility, Standard_Boolean_Facility

   subtype PARTIAL_FUNCTION is set of (
            d: D_Entry,
            r: R_Entry
         )
      exemplar m
      constraints
         for all d: D_Entry and r1, r2: R_Entry,
            if (d,r1) is in m and (d,r2) is in m
            then (r1 = r2)

   definition DEFINED_IN (
            m: PARTIAL_FUNCTION,
            d: D_Entry
         ): boolean =
      there exists r: R_Entry such that ((d,r) is in m)

   type family Partial_Map is modeled by PARTIAL_FUNCTION
      exemplar m
      constraints |m| <= Max_Size
      initialization
         ensures  |m| = 0

   operation Define (
            alters   m: Partial_Map,
            consumes d: D_Entry,
            consumes r: R_Entry
         )
      requires not DEFINED_IN (m,d) and
               |m| < Max_Size
      ensures  m = #m union {(#d,#r)}
   operation Undefine (
            alters   m: Partial_Map,
            preserves   d: D_Entry,
            produces dcopy: D_Entry,
            produces r: R_Entry
         )
      requires DEFINED_IN (m,d)
      ensures  (d,r) is in #m and
               m = #m - {(d,r)} and
               dcopy = d

   operation Undefine_Any_One (
            alters   m: Partial_Map,
            produces d: D_Entry,
            produces r: R_Entry
         )
      requires |m| > 0
      ensures  (d,r) is in #m and m = #m - {(d,r)}

   operation Is_Defined (
            preserves   m: Partial_Map,
            preserves   d: D_Entry
         ): Boolean
      ensures  Is_Defined = DEFINED_IN (m,d)

   operation Size_Of (
            preserves   m: Partial_Map
         ): Integer
      ensures  Size_Of = |m|

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Partial_Map_Template
```

Figure 5.6 - Partial_Map_Template

Overview of the Concept

Partial_Map_Template in Figure 5.6 above is an abstraction for creating mappings from values of a domain type to values of a range type. It is useful for "search and store/retrieval" problems such as database applications. The behavior of a partial map is modeled using a mathematical set of ordered domain-range pairs, with the constraints that there can be only one pair with a given domain value and that each partial map individually is no larger than the maximum supplied bound. Partial_Map_Template is *not* one of the concepts in [Ogden 96].

Is it Pragmatic?

All of the operations provided form a functional basis on partial map values, with the possible exception of Undefine_Any_One. Undefine_Any_One has been included to permit iteration over all the mappings in a partial map object. Interestingly, it is not strictly necessary since, if the domain entry type is enumerable, then it is possible to enumerate over the domain type and then undefine any one that is defined using the operations Is_Defined and Undefine. However, this is clearly not efficient and furthermore, it cannot be assumed that an arbitrary domain entry type is enumerable. Without the ability to undefine an arbitrary mapping, for example, it is not possible to test the equality of or replicate partial maps. Undefine_Any_One, therefore, must be included for general functional completeness, and because it can be implemented in constant time as an intrinsic operation.

Is it Observable and Controllable?

Partial_Map_Template is observable. Any two different partial map values k1 and k2 may differ in their sizes or because they contain different entries. For the first case, two similar scenarios σ1 and σ2 involving a call to Size_Of must always have different indicator values (sizes) between them. For the second case, σ1 and σ2 involving |k1| successive calls to Undefine_Any_One must always have different indicator values (entries) between them. Note that it is irrelevant that two same maps may produce different values on successive calls to Undefine_Any_One.

Partial_Map_Template is also controllable. There is a scenario σ for every partial map value k such that σ starts with an initialized partial map variable, followed by a sequence of calls to Define, one for each entry in k.

Bounded Almost_Constant_Function_Template

```
concept Almost_Constant_Function_Template (
            type Index,
            type Range,
            definition ARE_ORDERED(i:Index, j:Index): boolean,
            constant Default_R_Value: Range,
            constant Max_Count: Integer
         )
      requires Max_Count> 0 and
               for all i,j,k: Entry,
                  ARE_ORDERED (i,i) and
                  if ARE_ORDERED(i,j) and ARE_ORDERED(j,k) then
                     ARE_ORDERED(i,k) and
                  (ARE_ORDERED(i,j) or ARE_ORDERED(j,i)) and
                  if ARE_ORDERED(i,j) and ARE_ORDERED(j,i) then
                     i = j

      uses  Standard_Integer_Facility, Standard_Boolean_Facility

   definition DEVIATION_COUNT (
            f: function from Index to Range
         ): integer =
      |{i:Index | f(i) ≠ Default_R_Value}|

   definition LESS_THAN (
            i: Index,
            j: Index
         ): boolean =
      ARE_ORDERED(i,j) and not ARE_ORDERED(j,i)

   type family Almost_Constant_Function is modeled by
         function from Index to Range
      exemplar f
      constraints DEVIATION_COUNT(f) ≤ Max_Count
      initialization
         ensures  for all i: Index, f(i) = Default_R_Value

   operation Give_Value (
            alters   f: Almost_Constant_Function,
            preserves   i: Index,
            consumes r: Range
         )
      requires DEVIATION_COUNT(f) < Max_Count or
               f(i) ≠ Default_R_Value or
               r = Default_R_Value
      ensures  f(i) = #r and
               for all j: Index, if i ≠ j then f(j) = #f(j)
   operation Get_Value (
            alters   f: Almost_Constant_Function,
            preserves   i: Index,
            produces r: Range
         )
      ensures  r = #f(i) and
               f(i) = Default_R_Value and
               for all j: Index, if i ≠ j then f(j) = #f(j)

   operation Next_Index (
            preserves   f: Almost_Constant_Function,
            preserves   i: Index,
            produces n: Index
         )
      requires there exists j: Index such that
                  LESS_THAN(i,j) and
                  f(j) ≠ Default_R_Value
      ensures  LESS_THAN(i,n) and
               f(n) ≠ Default_R_Value and
               for all j: Index,
                  if LESS_THAN(i,j) and LESS_THAN(j,n) then
                     f(j) = Default_R_Value

   operation Is_Last_Index (
            preserves   f: Almost_Constant_Function,
            preserves   i: Index
         ): Boolean
      ensures  Is_Last_Index =
               for all j: Index, if LESS_THAN(i,j) then
                  f(j) = Default_R_Value

   operation First_Index (
            preserves   f: Almost_Constant_Function,
            produces i: Index
         )
      requires there exists j: Index such that
                  f(j) ≠ Default_R_Value
      ensures  f(i) ≠ Default_R_Value and
               for all j: Index, if LESS_THAN(j,i) then
                  f(j) = Default_R_Value

   operation Deviation_Count_Of (
            preserves   f: Almost_Constant_Function
         ): Integer
      ensures  Deviation_Count_Of = DEVIATION_COUNT(f)

   operation Allowed_Max_Count (): Integer
      ensures  Allowed_Max_Count = Max_Count

end Almost_Constant_Function_Template
```

Figure 5.7 - Almost_Constant_Function_Template

Overview of the Concept

Almost_Constant_Function_Template in Figure 5.7 is an alternative specification to the Partial_Map_Template for essentially the same problem. This is the data abstraction for searching found in [Ogden 96]. Unlike Partial_Map_Template, which conceptualizes a map as a set of domain-range pairs, in Almost_Constant_Function_Template the provided type is modeled as a function from "index" values to the range values. To capture the notion of an "undefined" range value, the client supplies a default range value, to which all "uninteresting" index values map. Since an almost-constant-function maps to this default range value for all indices other than those that have been explicitly defined using Give_Value, the function "almost" maps to this default range value everywhere, hence the name. The design of Almost_Constant_Function_Template is also different in that it allows extraction of individual mappings in an order supplied through the ARE_ORDERED parameter.

Is it Pragmatic?

The operations Give_Value, Get_Value, Deviation_Count_Of, and Allowed_Max_Count clearly form a functional basis on almost-constant functions. The operations First_Index, Next_Index, and Is_Last_Index help iterate over the mappings in an almost-constant-function object based on the provided order. As primary operations, the index operations can be coded to work efficiently, for example, in a binary search tree-based implementation. Hence, Almost_Constant_Function_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

Almost_Constant_Function_Template is observable. Any two different almost-constant-function values k1 and k2 could differ in their deviation counts, or contain different mappings. For the first case, σ1 and σ2 involving a call to Deviation_Count_Of must always have different indicator values (counts) between them. For the second case, σ1 and σ2 involving a call to First_Index, followed by a call to Get_Value, followed by Deviation_Count_Of(k1)-1 calls to pairs of Next_Index - Get_Value operations, must always have different indicator values (entries) between them.

Almost_Constant_Function_Template is also controllable. For example, for an almost-constant-function value k, σ could involve successive calls to Give_Value, one for each "defined" index value of k.

Other Issues

Almost_Constant_Function_Template and Prioritizer_Template in the library of concepts in [Odgen 96] have some overlapping functionality in their abilities to produce entries in order. However, neither is subsumed by the other both in terms of functionality and performance. Whereas the "searching" functionality is not available for Prioritizers, the ability to retrieve specified entries is not demanded of implementations of Prioritizer_Template. If Partial_Map_Template is also considered, then the degree of orthogonality of a library containing all three concepts is not clear or obvious. The pragmatic criterion that has raised this issue also suggests exploration of a more general concept that has more combined functionality as well as scaling down the functionality of one of the existing ones.

Bounded Prioritizer_Template

```
concept Prioritizer_Template (
            type Entry,
            definition ARE_ORDERED (x: Entry, y: Entry): boolean,
            constant Max_Size: Integer
         )

      requires Max_Size > 0   and
               for all x,y,z: Entry,
                  ARE_ORDERED (x,x) and
                  if ARE_ORDERED(x,y) and ARE_ORDERED(y,z) then
                     ARE_ORDERED(x,z) and
                  (ARE_ORDERED(x,y) or ARE_ORDERED(y,x))

      uses Standard_Integer_Facility, Standard_Boolean_Facility

   subtype INVENTORY_FUNCTION is function from Entry to integer
      exemplar f
      constraints
         for all x: Entry, f(x) >= 0

   definition INVENTORY_SIZE (
            f: INVENTORY_FUNCTION
         ): integer = sum of f(x) for all x: Entry

   definition IS_A_NEXT_ENTRY (
            f: INVENTORY_FUNCTION,
            x: Entry
         ): boolean = f(x) > 0 and for all y: Entry,
            if ARE_ORDERED(y,x) and not ARE_ORDERED(x,y)
            then f(y) = 0

   type family Prioritizer is modeled by (
            contents: INVENTORY_FUNCTION,
            insertion_phase: boolean
         )
      exemplar  p
      constraints
         INVENTORY_SIZE(p.contents) <= Max_Size
      initialization
         ensures  INVENTORY_SIZE(p.contents) = 0 and
                  p.insertion_phase

   operation Insert (
            alters p: Prioritizer,
            consumes x: Entry
         )
      requires INVENTORY_SIZE(p.contents) < Max_Size and
               p.insertion_phase
      ensures  p.insertion_phase and
               p.contents(#x) = #p.contents(#x) + 1 and
               for all y: Entry,
                  if y /= #x
                  then p.contents(y) = #p.contents(y)
   operation Change_Phase (
            alters p: Prioritizer
         )
      ensures  p.contents = #p.contents and
               p.insertion_phase = not #p.insertion_phase

   operation Extract (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires INVENTORY_SIZE(p.contents) > 0 and
               not p.insertion_phase
      ensures  not p.insertion_phase and
               IS_A_NEXT_ENTRY(#p.contents,x) and
               p.contents(x) = #p.contents(x) - 1 and
               for all y: Entry,
                  if y /= x then p.contents(y) = #p.contents(y)

   operation Extract_Any (
            alters p: Prioritizer,
            produces x: Entry
         )
      requires INVENTORY_SIZE(p.contents) > 0
      ensures  p.insertion_phase = #p.insertion_phase and
               p.contents(x) = #p.contents(x) - 1 and
               for all y: Entry,
                  if y /= x then p.contents(y) = #p.contents(y)

   operation Is_In_Insertion_Phase (
            preserves p: Prioritizer
         ): Boolean
      ensures  Is_In_Insertion_Phase = p.insertion_phase

   operation Size_Of (
            preserves p: Prioritizer
         ): Integer
      ensures  Size_Of = INVENTORY_SIZE(p.contents)

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Prioritizer_Template
```

Figure 5.8 - Prioritizer_Template

Is it Pragmatic?

As discussed in Chapter II, Prioritizer_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

Prioritizer_Template is observable and controllable, as discussed informally in Chapter III and formally using scenarios in Chapter IV.

5.3 Other Concepts

Bounded Coalescable_Equivalence_Relation_Template

```
concept Coalescable_Equivalence_Relation_Template(
         constant Range_Size: Integer
         )
      requires Range_Size > 0

      uses Standard_Boolean_Facility, Standard_Integer_Facility

   subtype RANGE is integer
      exemplar n
      constraints 1 ≤ n ≤ Range_Size

   type family Equivalence_Relation is modeled by function from (
            x : RANGE,
            y : RANGE
         ) to boolean
      exemplar e
      constraints
         for all x: RANGE, e(x,x) and
         for all x, y: RANGE, if e(x,y) then e(y,x) and
         for all x, y, z: RANGE,
            if e(x,y) and e(y,z) then e(x,z)
      initialization
         ensures
            for all x, y: RANGE, if e(x,y) then x = y

   operation Are_Equivalent(
            preserves   e: Equivalence_Relation,
            preserves   x: RANGE,
            preserves   y: RANGE
         ): Boolean
      ensures  Are_Equivalent iff e(x,y)

   operation Make_Equivalent(
            alters   e: Equivalence_Relation,
            preserves   x: RANGE,
            preserves   y: RANGE
         ): Boolean
      ensures  for all u, v: RANGE,
                  e(u,v) = (
                     #e(u,v) or
                     #e(u,x) and #e(v,y) or
                     #e(u,y) and #e(v,x)
                  )

   operation Range_Of (): Integer
      ensures  Range_Of = Range_Size

end Coalescable_Equivalence_Relation_Template
```

Figure 5.9 - Coalescable_Equivalence_Relation_Template

Overview of the Concept

Figure 5.9 above shows a specification for an equivalence relation abstraction. The behavior is modeled using a function that maps pairs of (a finite subset of) integers to a boolean value that is true only for pairs that have been made “equivalent”. Initially, no two integers are equivalent.

Is it Pragmatic?

The operations Are_Equivalent, Make_Equivalent, and Range_Of are functionally orthogonal. Hence Coalescable_Equivalence_Relation_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

Coalescable_Equivalence_Relation_Template is observable. The approach for demonstrating the existence of two similar scenarios σ1 and σ2 for distinguishing two equivalence relations k1 and k2 is unlike any discussed thus far, since Coalescable_Equivalence_Relation_Template does not involve any types as generic parameters. In this case, σ1 and σ2 involve a sequence of calls to Are_Equivalent, one call corresponding to each pair of the finite set of integers of the subtype RANGE. Where k1 and k2 have different mappings, σ1 and σ2 must always produce different indicator values (booleans) between them.

Coalescable_Equivalence_Relation_Template is also controllable. There is a scenario σ for every function k that involves successive calls to Make_Equivalent, one for each pair of integers that are equivalent in k.

Bounded Spanning_Forest_Machine_Template

```
concept Spanning_Forest_Machine_Template (
            constant Max_Vertex: Integer,
            constant Max_Edges: Integer
         )
      requires Max_Vertex > 0 and Max_Edges > 0

      uses  Standard_Integer_Facility, Standard_Boolean_Facility

   subtype EDGE is (
            v1: integer,
            v2: integer,
            w: integer
         )
      exemplar e
      constraints 1 <= e.v1 <= Max_Vertex and
                  1 <= e.v2 <= Max_Vertex and
                  e.w > 0

   subtype GRAPH is finite set of EDGE

   definition IS_MSF (msf: GRAPH, g: GRAPH): boolean =
      (* true iff msf is an MSF of g *)

   definition SHARE_AN_MSF (g1: GRAPH, g2: GRAPH): boolean =
      there exists msf: GRAPH such that
         IS_MSF (msf, g1) and IS_MSF (msf, g2)

   type family Spanning_Forest_Machine is modeled by (
            edges: GRAPH,
            insertion_phase: boolean
         )
      exemplar m
      initialization
         ensures  |m.edges| = 0 and m.insertion_phase

   operation Insert (
            alters   m: Spanning_Forest_Machine,
            consumes v1: Integer,
            consumes v2: Integer,
            consumes w: Integer
         )
      requires |m.edges| < Max_Edges and
               m.insertion_phase and
               1 <= v1 <= Max_Vertex and
               1 <= v2 <= Max_Vertex and
               w > 0
      ensures  SHARE_AN_MSF (
                     m.edges,
                     #m.edges union { (#v1, #v2, #w) }
                  ) and
               m.insertion_phase
   operation Change_To_Extraction_Phase (
            alters   m: Spanning_Forest_Machine
         )
      requires m.insertion_phase
      ensures  IS_MSF (m.edges, #m.edges) and
               not m.insertion_phase

   operation Extract (
            alters   m: Spanning_Forest_Machine,
            produces v1: Integer,
            produces v2: Integer,
            produces w: Integer
         )
      requires |m.edges| > 0
      ensures  (v1, v2, w) is in #m.edges and
               m.edges = #m.edges - {(v1, v2, w)} and
               m.insertion_phase = #m.insertion_phase

   operation Size_Of (
            preserves   m: Spanning_Forest_Machine
         ): Integer
      ensures  Size = |m.edges|

   operation Is_In_Insertion_Phase (
            preserves   m: Spanning_Forest_Machine
         ): Boolean
      ensures  Is_In_insertion_Phase = m.insertion_phase

   operation Allowed_Max_Vertex (): Integer
      ensures  Allowed_Max_Vertex = Max_Vertex

   operation Allowed_Max_Edges (): Integer
      ensures  Allowed_Max_Edges = Max_Edges

end Spanning_Forest_Machine_Template
```

Figure 5.10 - Spanning_Forest_Machine_Template

Overview of the Concept

Spanning_Forest_Machine_Template in Figure 5.10 above is a specification for an abstraction of a machine that computes a minimum spanning forest of a graph, reproduced from [Sitaraman 96]. It is modeled using a tuple consisting of a set of the inserted graph edges, and a flag indicating whether a spanning forest machine is in the insertion phase or the extraction phase. Edges are inserted into the machine one at a time, a change to the extraction phase is done, and then the edges are extracted from the machine, one at a time. The edges that are returned constitute a minimum spanning forest of the original graph. The performance advantages for this recast abstraction of the minimum spanning forest problem have been discussed in the literature [Weide 94].

Is it Pragmatic?

The operations Insert, Change_To_Extraction_Phase, Extract, Size_Of, Is_In_Insertion_Phase, Allowed_Max_Vertex, and Allowed_Max_Edges form an orthogonal set of operations on spanning forest machine values. Hence, Spanning_Forest_Machine_Template satisfies the pragmatic criterion.

Is it Observable and Controllable?

Spanning_Forest_Machine_Template is observable. For any two different spanning forest machine values k1 and k2, there are three ways they could be different: they could have different sizes, have the same sizes but be in different phases, or have the same size and phase but contain different entries. For the first case, σ1 and σ2 involving a call to Size_Of must always have different indicator values (sizes) between them. For the second case, σ1 and σ2 involving a call to Is_In_Insertion_Phase must always have different indicator values (phases). For the third case, σ1 and σ2 involving |k1| successive calls to Extract must always have different indicator values (edges) between them.

However, Spanning_Forest_Machine_Template is *not* controllable, since there does not exist a scenario σ for some spanning forest machine value k such that σ starts with an initialized spanning forest machine variable and ends with that variable mapping to k for all states in S(σ). This is because the Insert operation is specified so that it might not keep all the edges that are inserted.

Other Issues

This specification is an excellent example of an abstraction that, while failing one characterization for controllability, satisfies some others. In the discussion on code-based definitions for controllability in the previous chapter, we illustrated the possibility of formalization along the lines of “for some implementations” of the concept. That is, a specification S is controllable if it is *possible* that every value can be reached. If we were to assume this notion of controllability, it would be possible to rework the scenario-based definition for controllability to accommodate this assumption by replacing the predicate Always_Reaches_Target_From_Scratch with Might_Reach_Target_From_Scratch, where the requirement here is that S(σ) contains at least one state where some spanning forest machine variable maps to the target value k. By such a definition, Spanning_Forest_Machine_Template would be deemed controllable.

This example does not necessarily say that one definition is better than the other, since it is arguable whether the specification of Spanning_Forest_Machine_Template in Figure 5.10 is the most desirable. The specification in Figure 5.10 is the result of performance considerations on specification design [Sitaraman 96], as it permits implementation strategies ranging from those that store all the edges inserted to those that only keep edges constituting an MSF of the input graph at all times. A potential drawback here is the behavior of the Size_Of operation, which during the insertion phase has potentially erratic behavior depending on what a specific implementation of the concept is doing with the inserted edges. Sitaraman considers other specification designs, such as those that keep all inserted edges and those that only keep MSF edges. While each of these designs is not as general as that in Figure 5.10, they are observable and controllable with respect to the scenario-based definitions in Chapter IV, and their behavior with respect to the Size_Of operation is predictable as well.

5.4 “Built-In” Concepts

In this section, we present four concepts that are typically built into programming languages. These four concepts are for booleans, integers, records, and arrays. The RESOLVE implementation language also contains “syntactic sugar” for calling operations on these basic types, such as allowing the use of “a+b” instead of a call to “Add(a,b)”. However, in RESOLVE all types - including built-in types - are defined and used through concepts.

Boolean_Facility

```
concept Boolean_Facility

      uses  Two_Valued_Boolean_Algebra_Theory

   type family Boolean is modeled by boolean
      exemplar b
      initialization
         ensures  b = true

   operation And (
            preserves   a: Boolean,
            preserves   b: Boolean
         ): Boolean
      ensures  And = (a and b)

   operation Not (
            preserves   b: Boolean
         ): Boolean
      ensures  Not = (not b)

end Boolean_Facility
```

Figure 5.11 - Boolean_Facility

Overview of the Concept

Figure 5.11 above shows a specification for a boolean abstraction. The behavior of a Boolean object is modeled using the logical boolean values true and false. The use of the term “Standard_Boolean_Facility” as used throughout this dissertation refers to some standard implementation for this concept with suitable enhancements such as “or”.

Is it Pragmatic?

The provided operations And and Not form a functional basis on Boolean values, since all other logical operators can be easily and efficiently layered using these two operations.

Is it Observable and Controllable?

Boolean_Facility is observable. While in terms of code it may seem trivial that the operation And can easily be used to reveal a difference between k1 and k2, when applying the scenario-based definition for observability there can be no "communication" between σ1 and σ2. That is, σ1 and σ2 cannot simply involve a single call to "And (a,b)", where a=k1 and b=k2, because σ1 involves only k1 and σ2 involves only k2. However, one approach in this situation is to compare k1 or k2 with an initial boolean variable (which has a value of true) using the And operation. For example, consider the values k1 and k2 and the scenarios σ1 and σ2 below:

```
k1 = true
k2 = false

σ1 = (
   Boolean_Facility, {p:Boolean, b:Boolean},
   < And (p,b,true) → (p,b,true) >,{(p,k1)}
)
and
σ2 = (
   Boolean_Facility, {p:Boolean, b:Boolean},
   < And (p,b,true) → (p,b,false) >, {(p,k2)}
)
```

Boolean_Facility is also controllable. For example, if k = true, σ simply involves no operation calls; if k = false, σ simply involves a call to Not.

Bounded Integer_Facility

```
concept Integer_Facility

      uses  Standard_Boolean_Facility, Integer_Theory

   constant Min_Int: integer
      constraints Min_Int <= 0

   constant Max_Int: integer
      constraints Max_Int > Min_Int

   type family Integer is modeled by integer
      exemplar i
      constraints Min_Int <= i <= Max_Int
      initialization
         ensures  i = 0

   operation Add (
            preserves   i: Integer,
            preserves   j: Integer,
            produces sum: Integer
         )
      requires Min_Int <= i+j <= Max_Int
      ensures  sum = i+j

   operation Multiply (
            preserves   i: Integer,
            preserves   j: Integer,
            produces product: Integer
         )
      requires Min_Int <= i*j <= Max_Int
      ensures  product = i*j

   operation Subtract (
            preserves   i: Integer,
            preserves   j: Integer,
            produces diff: Integer
         )
      requires Min_Int <= i-j <= Max_Int
      ensures  diff = i-j
   operation Int_Divide (
            preserves   i: Integer,
            preserves   j: Integer,
            produces quot: Integer
         )
      requires if j <= 0
               then j * (Max_Int + 1) < i < j* (Min_Int - 1)
      ensures  |j*quot| <= |i| and |i-j*quot| < |j|

   operation Less_Equal (
            preserves   i: Integer,
            preserves   j: Integer
         ): Boolean
      ensures  Less_Equal = (i <= j)

   operation Min_Allowed (): Integer
      ensures  Min_Allowed = Min_Int

   operation Max_Allowed (): Integer
      ensures  Max_Allowed = Max_Int

end Integer_Facility
```

Figure 5.12 - Integer_Facility

Overview of the Concept

Figure 5.12 above shows a specification for a bounded integer abstraction. The behavior of an Integer is modeled using mathematical integers, constrained within minimum and maximum bounds. The bounding values of Min_Int and Max_Int are not parameters to the concept in the typical sense, but are rather implementation-dependent values. Such values are implicitly passed from an implementation into the specification, rather than from the specification to an implementation, as is the case with parameterization. The use of the term "Standard_Integer_Facility" as used throughout this dissertation in many specifications refers to some standard implementation for this concept along with some common enhancements for easier usage, such as "equal".

Is it Pragmatic?

The provided operations Add, Subtract, Multiply, Int_Divide, Less_Equal, Min_Allowed, and Max_Allowed form a functional basis on Integer values. Operations such as Equal, Greater_Equal, etc., can be layered using these four operations without suffering a performance penalty.

Is it Observable and Controllable?

Integer_Facility is observable. While in terms of code it may seem trivial that the operation Less_Equal can easily be used to reveal a difference between any two Integer values k1 and k2, when applying the scenario-based definition for observability there can be no "communication" between σ1 and σ2 that correspond to initial values k1 and k2, respectively. This is because σ1 and σ2 cannot simply involve a single call to "Less_Equal (a,b)", where a=k1 and b=k2, because σ1 involves only k1 and σ2 involves only k2. However, one approach in this situation is to incrementally *construct* the value of k1 in both σ1 and σ2. Then, σ1 and σ2 can involve a call to Less_Equal (p, same_as_k1), where p=k1 for σ1 and p=k2 for σ2 and where "same_as_k1" is the constructed value of k1. The resulting indicator values (booleans) for σ1 and σ2 must be different, since k1 ≠ k2. For example, consider the values k1 and k2 and the scenarios σ1 and σ2 below[8]:

```
k1 = 1
k2 = 2

σ1 = (
   Integer_Facility,
   {p:Integer, max1:Integer, max2:Integer, one: Integer},
   op_call_seq1, {(p,k1)}
)
and
σ2 = (
   Integer_Facility,
   {p:Integer, max1:Integer, max2:Integer, one: Integer},
   op_call_seq2, {(p,k2)}
)

where
   op_call_seq1 = <
      Max_Allowed (max1) → (max1),
      Max_Allowed (max2) → (max2),
      Int_Divide (max1, max2, one) → (max1, max2, one),
      Less_Equal (p, one, false) → (p, one, true)
   >
   and
   op_call_seq2 = <
      Max_Allowed (max1) → (max1),
      Max_Allowed (max2) → (max2),
      Int_Divide (max1, max2, one) → (max1, max2, one),
      Less_Equal (p, one, false) → (p, one, false)
   >
```

[8] Note that in a scenario, we alter the signatures of operations that return a result through their name (i.e., functions such as Less_Equal) by adding an extra produces parameter to their parameter lists.

In the more general case where k1 is not 1, we could incrementally add or subtract “one” until we reach the value of k1 in both σ1 and σ2, and then make the comparison using Less_Equal.

Integer_Facility is also controllable. There is a scenario σ for every Integer value k such that σ starts with an initialized Integer variable and makes a sequence of calls to Add (or Subtract, if k is negative) until the value of k is reached.

(Two Field) Record_Template

```
concept Record_Template (
            type Entry_1,
            type Entry_2
         )

   type family Record is modeled by (
            field1 : Entry_1,
            field2 : Entry_2
         )
      exemplar r
      initialization
         ensures  r.field1 = Entry_1.initial_value and
                  r.field2 = Entry_2.initial_value

   operation Swap_Field_1 (
            alters   r: Record,
            alters   x: Entry_1
         )
      ensures  r.field1 = #x and x = #r.field1 and
               r.field2 = #r.field2

   operation Swap_Field_2 (
            alters   r: Record,
            alters   x: Entry_2
         )
      ensures  r.field2 = #x and x = #r.field2 and
               r.field1 = #r.field1

end Record_Template
```

Figure 5.13 - (Two Field) Record_Template

Overview of the Concept

The Record_Template in Figure 5.13 is a specification for a generic two-field record abstraction. It actually serves as an example for a collection of specifications of n-field records in general, since variable-length generic parameter lists are not syntactically supported in the RESOLVE notation nor in most other languages.

The operations Swap_Field_1 and Swap_Field_2 form an orthogonal operation set on two-field record values. Hence, Record_Template satisfies the pragmatic criterion. Also, it is easy to show that Record_Template is observable and controllable, using calls to Swap_Field_1 and Swap_Field_2.

Bounded Array_Template

```
concept Array_Template (
            type Entry,
            constant Max_Size: Integer
         )
      requires Max_Size > 0

      uses  Standard_Integer_Facility

   type family Array is modeled by function from integer to Entry
      exemplar a
      constraints
         for all i: integer,
            if i < 1 or i > Max_Size then a(i) = Entry.Base_Point
      initialization
         ensures  for all i: integer,
            if i >= 1 and i <= Max_Size then
               a(i) = Entry.Initial_Value

   operation Swap_Entry (
            alters   a: Array,
            preserves   i: Integer,
            alters   x: Entry
         )
      requires i >= 1 and i <= Max_Size
      ensures  a(i) = #x and x = #a(i) and
               for all j: integer, if (j != i) then a(j) = #a(j)

   operation Allowed_Max_Size (): Integer
      ensures  Allowed_Max_Size = Max_Size

end Array_Template
```

Figure 5.14 - Array_Template

Overview of the Concept

Figure 5.14 above shows a specification for a bounded array abstraction. The behavior of an array is modeled using a function mapping integers to entries, with the constraint that any domain value i not with the range [1,Max_Size] maps to a pre-defined **Base_Point**.

Array_Template only has one operation for array manipulation: Swap_Entry. This single operation serves both purposes of getting an entry from a given index and/or setting an entry of a given index to a particular value. Clearly, this interface is orthogonal and introduces no performance bottlenecks. Also, it is easy to see that Array_Template is both observable and controllable.

Conclusions VI

To enhance applicability and encourage its use, a component or a component-based system must have a well-designed set of interface features as well as a proper explanation of these features. If the interface does not include suitable operations for effective manipulation of objects defined by that interface, then it might compromise functional and/or performance flexibility, thereby inhibiting its reuse. Alternatively, poor explanations of an otherwise well-conceived interface might make it impossible to understand its objects and operations, and also inhibit its use. This interconnected problem of designing an interface that provides a suitable set of features along with an appropriate formal explanation is termed the *specification design problem*.

Unfortunately, both aspects of the specification design problem are rarely addressed simultaneously in the software engineering community. For example, in the formal specification community, the focus is mostly on precise notation, whereas issues surrounding the clear and precise explanation of interface behavior are rarely the focus in the practicing object-oriented community. This dissertation fills this gap by providing a foundation for the formal specification design problem. Specifically, the properties of *observability*, *controllability*, and a particular version of the *pragmatic criterion* are shown to comprise this foundation.

The performance-motivated pragmatic criterion guides the design of component interfaces and component libraries so that they are widely applicable in terms of both functionality and performance. The pragmatic criterion accomplishes this task by simultaneously demanding the inclusion of features that augment the functionality and performance of a component or library and by disallowing features that are redundant along either of these lines.

To complement the pragmatic criterion, the principles of observability and controllability direct formal explanations of software components to be precise and understandable. The principles guide this task by requiring that the explanations of software components thoroughly and minimally describe the intended problem. In this dissertation, we have formally characterized observability and controllability using both code-based definitions and scenario-based definitions. The scenario-based definitions are especially precise because they are self-contained, unlike the code-based definitions which rely on suitable definitions of “total correctness” of code involving relational data abstractions. The difficulty in formalization stems from the need to specify relational behavior in software engineering.

In defining fundamental properties for assisting with the specification design problem, we are also making explicit a large part of the RESOLVE discipline for designing specifications of object-based software components. In particular, we have illustrated how these concerns have led to the unique collection of RESOLVE component specifications.

Future Research Directions

There are several interesting possibilities for future research involving the discovery and description of desirable attributes of formal specification designs. We discuss some of these possibilities here.

Performance-Based Observability and Controllability Definitions. We have stated that it might be possible to formalize the pragmatic criterion. One approach for such formalization is to view the problem as defining observability and controllability of *non-functional* aspects. The model spaces of concepts, for example, could be augmented with information for capturing precise execution times and space utilization of the provided operations. To justify the inclusion of functionally non-orthogonal operations, the pragmatic criterion may demand the existence of implementations that will show a distinction in performance for those operations. The use of observability and controllability principles along these lines is also useful to show the need for intermediate abstract models for performance.

Specifications with Global State and Multiple Data Types. In this dissertation, we restricted our attention to specifications that provided only one data type and where each object of the provided type contained an entire “copy” of the specified state. Given that there are situations in which components providing multiple data types and/or having global state information are necessary, it is required to identify characterizations of observability, controllability, and the pragmatic criterion for such specifications.

Automating Validation of Specifications. It remains a question whether it is possible to show the existence of the properties of observability, controllability, and the pragmatic criterion for a given specification, at least in part mechanically. It seems clear that the process is not effective in general. However, it may be possible, for example, to construct a database of particular good and bad specification designs for specific modeling strategies. Then an automation tool could use such a database to identify common problems and suggest routine fixes in the specification design process.

Design of New Specifications. The recasting strategy has led to a collection of new object-based concepts in RESOLVE such as Spanning_Forest_Machine_Template, Cheapest_Path_Template, and Rank_Ordering_Template, among others [Ogden 97]. The design of new specifications can be proudly justified using the principles proclaimed in this dissertation. In addition, the overall collection of concepts in a RESOLVE library can be argued to be robust and minimal using a formalization of the pragmatic criterion.

Sample Component Implementations for Prioritizer_Template A

This appendix presents some implementations based on arrays (for simplicity) for the bag-modeled Prioritizer_Template concept. These implementations were used to generate the timing data from which the performance graphs of Chapter II were produced.

The first implementation, called “Prioritizer_Template_1”, is based on quick-sort. The entire sorting takes place during a call to “Change_Phase”, in which the entire collection is reverse-ordered from the first index in the array towards the last index. Each insertion places the entry in the next available slot in the array. Each extraction gets the next ordered entry in the array, which begins with the highest index containing an entry (since the array is reverse-ordered). This strategy provides Insert and Extract with constant time performance. Change_Phase has an *average* time bounded by O (n log n) when changing from the insertion phase to the extraction phase, and has constant time performance when changing the other direction.

For the heap-based implementation shown next, termed “Prioritizer_Template_2”, the ordering process is distributed among two of the operations. Each insertion places the entry in the next available slot in the array. The Change_Phase operation uses a “heapify” operation to construct a heap from all the entries in the array. Doing so takes time O (n). During each call to Extract, the root of the heap (the first array value) is returned. To re-construct the heap, the last value in the array is moved to the top and “sifted down” to its appropriate position in the heap.

The third and final implementation, called “Prioritizer_Template_3”, is also heap-based and makes use of two operations: “sift-up” and “sift-down”. During each call to Insert, the entry is placed adjacent to the last entry in the array, then “sifted up” to its appropriate position in the heap. The Extract operation is the same as in the first heap-based implementation described. This strategy provides Insert and Extract each with performance bounded by O (log n), and Change_Phase with constant time performance.

A.1 A Quick Sort-Based Implementation Suitable for Application Class I

Implementation Header

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
class Prioritizer_Template_1: public
Prioritizer_Template <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >
{
      public:
            Prioritizer_Template_1 ();
            virtual ~Prioritizer_Template_1 ();
            virtual void operator &= (Prioritizer_Template_1& rhs);

            virtual void Insert (Entry& x);
            virtual void Change_Phase ();
            virtual void Extract (Entry& x);
            virtual void Extract_Any (Entry& x);
            virtual Boolean Is_In_Insertion_Phase ();
            virtual Integer Size_Of ();
            virtual Allowed_Max_Size ();

      private:
            /* Implicit assignment and copy constructor are prohibited
*/
            Prioritizer_Template_1 (const Prioritizer_Template_1& m);
            Prioritizer_Template_1& operator = (
                  const Prioritizer_Template_1& rhs
            );

            Boolean filling;
            Integer size;
            Entry* ele;

            void QuickSort (Integer a, Integer b);
            Integer Partition (Integer a, Integer b);
};
```

Implementation

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Prioritizer_Template_1 ()
{
      ele = new Entry [Max_Size+1];
      filling = true;
      size = 0;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
~Prioritizer_Template_1 ()
{
      delete [] ele;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
operator &= (
            Prioritizer_Template_1 <
                  Entry,
                  Entry_Compare_Capability,
                  Max_Size
            >& rhs
      )
{
      Entry *tele;

      tele = ele;
      ele = rhs.ele;
      rhs.ele = tele;

      rhs.filling &= filling;
      rhs.size &= size;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Partition (Integer left, Integer right)
{
      Integer up, down;

      up = left+1;
      down = right;

      while (true) {
            while (
                  Entry_Compare_Capability::Compare (
                        ele[left],
                        ele[up]
                  ) &&
                  (up < right)
            )
                  up++;

            while (
                  !Entry_Compare_Capability::Compare (
                        ele[left],
                        ele[down]
                  )
            )
                  down--;

            if (up < down)
                  ele[up] &= ele[down];
            else {
                  ele[left] &= ele[down];
                  return down;
            }
      }
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
QuickSort (Integer start, Integer end)
{
      Integer split;

      if (start < end) {
            split = Partition (start, end);
            QuickSort (start, split - 1);
            QuickSort (split + 1, end);
      }
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Insert (Entry& x)
{
      size++;
      ele[size] &= x;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Change_Phase ()
{
      Integer a;

      if (filling) {
            filling = false;
            QuickSort (1, size);
      }
      else
            filling = true;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Extract (Entry& x)
{
      Entry temp;

      x &= temp;
      ele [size] &= x;
      size--;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Extract_Any (Entry& x)
{
      Entry temp;

      x &= temp;
      ele [size] &= x;
      size--;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Size_Of ()
{
      return size;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Boolean Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Is_In_Insertion_Phase ()
{
      return filling;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_1 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Allowed_Max_Size ()
{
      return Max_Size;
};
```

A.2 A Heap-Based Implementation Suitable for Application Class II

Implementation Header

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
class Prioritizer_Template_2: public
Prioritizer_Template <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >
{
      public:
            Prioritizer_Template_2 ();
            virtual ~Prioritizer_Template_2 ();
            virtual void operator &= (Prioritizer_Template_2& rhs);

            virtual void Insert (Entry& x);
            virtual void Change_Phase ();
            virtual void Extract (Entry& x);
            virtual void Extract_Any (Entry& x);
            virtual Boolean Is_In_Insertion_Phase ();
            virtual Integer Size_Of ();
            virtual Allowed_Max_Size ();

      private:
            /* Implicit assignment and copy constructor are prohibited
*/
            Prioritizer_Template_2 (const Prioritizer_Template_2& m);
            Prioritizer_Template_2& operator = (
                  const Prioritizer_Template_2& rhs
            );

            Boolean filling;
            Integer size;
            Entry* ele;

            void sift_down (Integer a, Integer b);
            void heapify (Integer a);
};
```

Implementation

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Prioritizer_Template_2 ()
{
      ele = new Entry [Max_Size+1];
      filling = true;
      size = 0;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
~Prioritizer_Template_2 ()
{
      delete [] ele;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
operator &= (
            Prioritizer_Template_2 <
                  Entry,
                  Entry_Compare_Capability,
                  Max_Size
            >& rhs
      )
{
      Entry *tele;

      tele = ele;
      ele = rhs.ele;
      rhs.ele = tele;

      rhs.filling &= filling;
      rhs.size &= size;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
sift_down (Integer n, Integer k)
{
      Integer left = 2*k,
                  right = 2*k + 1,
                  smallest = k;

      if (left <= n) {
            if (!Entry_Compare_Capability::Compare (
                        ele[smallest],
                        ele[left]
                  )
            )
                  smallest = left;

            if (right <= n)
                  if (!Entry_Compare_Capability::Compare (
                              ele[smallest],
                              ele[right]
                        )
                  )
                        smallest = right;
      }

      if (smallest != k) {
            ele[smallest] &= ele[k];
            sift_down (n, smallest);
      }
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
heapify (Integer n)
{
      Integer i = 0;

      for (i = n/2; i >= 1; i--)
            sift_down (n, i);
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Insert (Entry& x)
{
      size++;
      ele [size] &= x;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Change_Phase ()
{
      if (filling) {
            filling = false;
            heapify (size);
      }
      else {
            filling = true;
      }
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Extract (Entry& x)
{
      Entry temp;

      x &= temp;
      ele[1] &= x;
      ele[1] &= ele[size];
      size--;
      sift_down (size, 1);
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Extract_Any (Entry& x)
{
      Entry temp;

      x &= temp;
      ele[size] &= x;
      size--;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Size_Of ()
{
      return size;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Boolean Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Is_In_Insertion_Phase ()
{
      return filling;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_2 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Allowed_Max_Size ()
{
      return Max_Size;
};
```

A.3 A Heap-Based Implementation Suitable for Application Class III

Implementation Header

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
class Prioritizer_Template_3: public
Prioritizer_Template <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >
{
      public:
            Prioritizer_Template_3 ();
            virtual ~Prioritizer_Template_3 ();
            virtual void operator &= (Prioritizer_Template_3& rhs);

            virtual void Insert (Entry& x);
            virtual void Change_Phase ();
            virtual void Extract (Entry& x);
            virtual void Extract_Any (Entry& x);

            virtual Boolean Is_In_Insertion_Phase ();
            virtual Integer Size_Of ();
            virtual Allowed_Max_Size ();

      private:
            /* Implicit assignment and copy constructor are prohibited
*/

            Prioritizer_Template_3& operator = (
                  const Prioritizer_Template_3& rhs
            );
            Prioritizer_Template_3 (const Prioritizer_Template_3& m);

            Boolean filling;
            Integer size;
            Entry* ele;

            void sift_up ();

            void sift_down (
                  Integer n,
                  Integer k
            );
};
```

Implementation

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Prioritizer_Template_3 ()
{
      ele = new Entry [Max_Size+1];
      filling = TRUE;
      size = 0;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
~Prioritizer_Template_3 ()
{
      delete [] ele;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
operator &= (
            Prioritizer_Template_3 <
                  Entry,
                  Entry_Compare_Capability,
                  Max_Size
            >& rhs
      )
{
      Entry *tele;

      tele = ele;
      ele = rhs.ele;
      rhs.ele = tele;
      rhs.filling &= filling;
      rhs.size &= size;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
sift_up ()
{
      Integer n = size;
      Integer parent = n/2;

      while (parent > 0) {
            if (Entry_Compare_Capability::Compare (
                        ele[parent],
                        ele[n]
                  )
            ) return;

            ele[parent] &= ele[n];
            n = parent;
            parent = n/2;
      }
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
sift_down (
                  Integer n,
                  Integer k
            )
{
      Integer left = 2*k,
                  right = 2*k + 1,
                  smallest = k;

      if (left <= n) {
            if (!Entry_Compare_Capability::Compare (
                        ele[smallest],
                        ele[left]
                  )
            )
                  smallest = left;

            if (right <= n)
                  if (!Entry_Compare_Capability::Compare (
                        ele[smallest],
                        ele[right]
                        )
                  )
                        smallest = right;
      }

      if (smallest != k) {
            ele[smallest] &= ele[k];
            sift_down (n, smallest);
      }
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Insert (Entry& x)
{
      size++;
      ele [size] &= x;
      sift_up ();
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Change_Phase ()
{
      filling = !filling;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Extract (Entry& x)
{
      Entry temp;
      x &= temp;

      ele[1] &= x;
      ele[1] &= ele[size];
      size--;
      sift_down (size, 1);
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
void Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Extract_Any (Entry& x)
{
      Entry temp;

      x &= temp;
      ele[size] &= x;
      size--;
};
```

```
template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Size_Of ()
{
      return size;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Boolean Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Is_In_Insertion_Phase ()
{
      return filling;
};


template <
      class Entry,
      class Entry_Compare_Capability,
      int Max_Size
>
Integer Prioritizer_Template_3 <
            Entry,
            Entry_Compare_Capability,
            Max_Size
      >::
Allowed_Max_Size ()
{
      return Max_Size;
};
```

Bibliography

[Booch 86] Booch, G., “Object-Oriented Software Development,” *IEEE Transactions on Software Engineering 12,* no. 2, February 1986, pp. 211-221.

[Cormen 90] Cormen, T.H., Leiserson, C.E., and Rivest, R.L., *Introduction to Algorithms*, MIT Press, Cambridge, Mass., 1990.

[Desharnais 97] Desharnais, J., Frappier, M., Khédri, R., Mili, A., “Integration of Sequential Scenarios”, *Proc. Sixth European Software Engineering Conference ESEC 97*, 1997, pp. 310-326.

[Edwards 95] Edwards, S.H., *A Formal Model of Software Subsystems*, Ph.D. dissertation, Dept. of Computer and Information Science, The Ohio State University, 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*, Vol. 20, No. 4, April 1994, pp. 288-307.

[Fleming 97] Fleming D., Sitaraman M., and Sreerama S., “A Practical Performance Criterion for Object Interface Design”, *Journal of Object-Oriented Programming*, Vol. 10, No. 4, July/August 1997, pp. 52-63.

[Harms 91] Harms, D.E., and Weide, B.W., “Copying and Swapping: Influences on the Design of Reusable Software Components,” *IEEE Transactions on Software Engineering*, Vol. 17, No. 5, May 1991, pp. 424-435.

[He 86] He, J., Hoare, C.A.R, and Sanders, J.W., "Data Refinement Refined", in: B. Robinet and R. Wilhelm, eds., *Proceedings European Symposium on Programming*, Lecture Notes in Computer Science 213, Springer, Berlin, 1986, pp. 187-196.

[Horowitz 76] Horowitz, E. and Sahni, S., *Fundamentals of Data Structures*, Computer Science Press, Rockville, MD., 1976.

[Jones 80] Jones, C.B., *Software Development: A Rigorous Approach*, Prentice-Hall International Series in Computer Science, Prentice-Hall, London, 1980.

[Jones 90] Jones, C. B., *Systematic Software Development Using VDM*, Prentice-Hall, Englewood Cliffs, NJ, 1990.

[Kapur 80] Kapur, D., and Mandayam, S., "Expressiveness of the Operation Set of a Data Abstraction," in *Conference Record 7th Annual Symposium on Principles of Programming Languages*, ACM, 1980, pp. 139-153.

[Koenig 95] Koenig, A. and Stroustrup, B., "A Foundation for Native C++ Styles," *Software Practice and Experience*, December 1995.

[Krone 88] Krone, J., *The Role of Verification in Software Reusability*, Ph.D. dissertation, Dept. of Computer and Information Science, The Ohio State University, 1988.

[Leavens 91] Leavens, G., "Modular Specification and Verification of Object-Oriented Programs", *IEEE Software*, Vol. 8, No. 4, July 1991, pp. 72-80.

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

[Meyer 87]	Meyer, B., “Reusability: the Case for Object-Oriented Design,” *IEEE Software 4*, no. 2, March 1987, pp. 50-64.
[Meyer 94]	Meyer, B., *Reusable Software: The Base Object-Oriented Component Libraries*, Prentice-Hall International, 1994.
[Milner 77]	Milner, R., “Fully Abstract Models of Typed λ-Calculi”, *Theoretical Computer Science 4*, 1977.
[Nipkow 87]	Nipkow, T., “Are Homomorphisms Sufficient for Behavioral Implementations of Deterministic and Nondeterministic Data Types?”, *Lecture Notes in Computer Science 247*, F.J. Brandenburg, eds. G. Vidal-Naquet and M. Wirsing, Springer-Verlag, 1987, pp. 260-271.
[Norman 90]	Norman, D.A., *The Design of Everyday Things,* Doubleday/Currency, 1990.
[Odgen 96]	Odgen, W.F., *The Proper Conceptualization of Data Structures,* Class Notes, Department of Computer and Information Sciences, The Ohio State University, 1996.
[Odgen 97]	Odgen, W.F., Department of Computer and Information Sciences, The Ohio State University, Personal Communication, 1997.
[SIGSOFT 94]	Special Feature: Component-Based Software Using RESOLVE, *ACM SIGSOFT Software Engineering Notes,* Vol. 19, No. 4, eds. M. Sitaraman and B.W. Weide, October 1994, pp. 21-67.
[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, pp. 207-229.

[Sitaraman 96] Sitaraman, M., "Impact of Performance Considerations on Formal Specification Design," *Formal Aspects of Computing*, Springer-Verlag International, Vol. 8, No. 6, January 1997, pp. 716-736.

[Sitaraman 97] Sitaraman, M., Ogden, W.F., and Weide, B.W., "On the Practical Need for Abstraction Relations to Verify Abstract Data Type Representations", *IEEE Transactions on Software Engineering*, Vol. 23, No. 3, March 1997, pp. 157-170.

[Stepanov 94] Stepanov, A. and Lee, M., *The Standard Template Library*, ISO Programming Language C++ Project, Doc. No. X3J16/94-0095, WG21/N0482.

[Stroustrup 96] Stroustrup, B., "Language-Technical Aspects of Reuse, " *Proceedings of the Fourth International Conference on Software Reuse*, Ed. M. Sitaraman, IEEE Computer Society Press, April 1996.

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

[Weide 94] Weide, B.W., Ogden, W.F., and Sitaraman, M., "Recasting Algorithms to Encourage Reuse," *IEEE Software*, Vol. 11, No. 5, September 1994, pp. 80-88.

[Weide 96] Weide, B.W., Edwards, S.H., Heym, W.D., Long, T.J., and Ogden, W.F., "Characterizing Observability and Controllability of Software Components", *Proc. Fourth Int. Conf. on Software Reuse*, M. Sitaraman, ed., IEEE, April 1996.

[Wing 90] Wing, J.M., "A Specifier's Introduction to Formal Methods," *IEEE Computer*, Vol. 23, No. 9, 1990, pp. 8-24.

[WISR 93] *Proceedings of the Sixth Annual National Workshop on Software Reuse*, Owego, NY, November 1993.