Specifying Weak Sets

Jeannette M. Wing and David C. Steere

wing@cs.cmu.edu dcs@cs.cmu.edu
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213 USA

Abstract

We present formal specifications of a new abstraction, weak sets*, which can be used to alleviate high latencies when retrieving data from a wide-area information system like the World Wide Web. In the presence of failures, concurrency, and distribution, clients performing queries may observe behavior that is inconsistent with the stringent semantic requirements of mathematical sets. For example, an element retrieved and returned to the client may be subsequently deleted before the query terminates. We chose to specify formally the behavior of weak sets because we wanted to understand the varying degrees of inconsistency clients might be willing to tolerate and to understand the tradeoff between providing strong consistency guarantees and implementing weak sets efficiently. Our specification assertion language uses a novel construct that lets us model* reachability *explicitly; with it, we can distinguish between the existence of an object and its accessibility. The specifications were instrumental in understanding the design space, and we are currently implementing the most permissive of the specifications in several types of Unix systems.*

1 Motivation for weak sets

Suppose you are browsing the World Wide Web (WWW) and want to display the `.face` files of all people listed on Carnegie Mellon's home page. Or, suppose through the on-line library information system (LIS) you want to get a list of papers by a particular author. Or, suppose you are a tourist in Pittsburgh and want to look at the on-line menus of all Chinese restaurants before choosing where to eat dinner.

Each of these kinds of queries returns a set of objects (`.face` files, card catalog entries, menus). What properties should we expect these sets to have? We claim that some standard properties of mathematical sets are desired but others are not. We expect that:

- Membership of an element is determined at some time between starting the query and finishing the query. Membership may not necessarily hold before the query, continuously throughout the run of the query, or even after the query completes. For example, if the LIS database is not up-to-date, we would not be surprised if an author's most recent paper is not listed; we would not go hungry if our restaurant search missed some (but not all) Chinese restaurants in Pittsburgh.
- Order among elements does not matter. Hence retrieval of elements can be optimized.
- There are no duplicates. (Though we probably would not be overly annoyed if there were.)
- Elements in the set change infrequently. A restaurant's menu may change weekly or seasonally; a `.face` file, annually; an LIS entry, never.

In short, weak sets are unordered, dynamically changing collections of elements with no duplicates.

Because of the nature of the information repositories over which we run these queries, we would not expect concurrent reads and writes on the repository to be serializable. In particular, user A may be updating the information repository concurrently with user B who is reading from it. User B may see partial writes of A. This non-serializable behavior implies that:

- Two people running the same query at the same time may obtain different sets of elements.
- Running the same query twice in a row may return different sets of elements.

Thus these sets provide weaker guarantees to the user than traditional set semantics or traditional distributed databases. However, for the kinds of wide area systems we consider, clients do not expect strong consistency properties, and implementations that provide stronger guarantees may prove inefficient. The key difference, of course, is that unlike transaction-oriented databases (e.g., a bank's set of accounts), there is no global consistency requirement that must be upheld across a set of information repositories in the WWW. This paper explores the design space for variations on the semantics of *weak sets*.

1.1 Context for this work

Our original motivation for investigating the semantics of weak sets arose in the context of distributed file systems. Our target environment is a wide-area file system on a network of (possibly mobile) workstations. Failures are assumed to be common, e.g., disconnecting a mobile client from the network while traveling is an induced failure, and consistency of data may be sacrificed to gain high performance and high availability. In a distributed file system, files and subdirectories in the same directory may reside on nodes different from each other and/or from the directory itself.

To reduce the high latency of accessing a group of objects in a distributed file system, one of us (DCS) as part of a Ph.D. thesis is adding a weak set abstraction called *dynamic sets* to the Unix Application Programmer's Interface. A dynamic set [1] is a non-volatile, non-distributed collection of objects, similar in concept to an open file in Unix. A dynamic set's use informs a system's lower layers of a user's future activity, allowing the system to prefetch objects to hide the delays of remote access.

For example, consider a simple Unix-like command such as "`grep foo *.c`". When executed, the shell expands the "`*.c`" to produce a list of files which are passed as arguments to `grep`. This expansion forces an ordering on the processing of the files and destroys the grouping information known to the user. In a distributed file system, the performance implications of such an ordering may be significant; in the worst case, because of failures some files may no longer be accessible and so excessive delays due to timeouts are possible. By using dynamic sets, we gain two advantages: (1) We can return information to the user more quickly by yielding accessible or local information first; and (2) we can execute such file system operations more efficiently by fetching files in parallel, fetching "closer" files first, and fetching all accessible files despite network failures. The resulting behavior observed by the user is akin to that of a set, where ordering of the items does not matter. Also, by supporting a set-like abstraction, we can support database-like queries, e.g., finding all files that satisfy a given predicate.

1.2 Contributions of paper

To better understand the semantics of dynamic sets, in particular what guarantees the implementation must provide to its users, we decided to more formally specify the properties of weak sets. In so doing, we realized that there is a wide range of reasonable semantics, resulting in our variations of weak sets. This paper presents some of the points in this range. The weakest of the behaviors corresponds exactly to the semantics of dynamic sets that we are implementing.

In our first attempt at writing formal specifications of weak sets we ran up against two limitations of current formal methods. First, we need to deal more explicitly with the failure case due to the distributed nature of our context. In particular, we need to distinguish between the existence of an object, say an element of a set, and its accessibility; an element may satisfy a query but we may not be able to reach it because of a failure. Second, membership for weak sets is determined by invoking an iterator, which incrementally retrieves elements that satisfy a given query. Little work has addressed the formal specification of iterators (we discuss related work in Section 4); none that we are aware of is suitable for a concurrent or distributed environment.

In summary the two main contributions of this paper are:

- A design space for the semantics of weak sets in a distributed environment. We present in Section 3 a set of dimensions for our design space and describe four of the points most relevant to implementors of real systems.
- A novel specification construct needed to capture the inherent distributed nature of the application. In a distributed system where node and network failures are possible, knowing about the existence of an object does not imply it can be accessed. We introduce a **reachable** function to our assertion language to help make this distinction.

Secondary contributions of this paper are (1) a way of specifying iterators in the presence of concurrency and distribution and (2) a more precise semantics for dynamic sets, a new distributed file system abstraction.

Both the notion of weak sets and our specification technique can be applied to other contexts. A file system is a special kind of persistent object repository where files are objects and directories are collections. A distributed file system is a special kind of a wide-area information system, for which clients expect continuous operation despite faults and transmission delays. So, though originally motivated to support distributed file systems, weak sets are more generally abstractions useful for both persistent object repositories, e.g., Cricket [2], EOS [3], Gemstone [4], and Thor [5] (see [6] for others), and wide-area information systems and their applications, e.g., the World Wide Web (WWW) [7], WAIS [8], and Gopher[9]. Using an iterator-like operation (or performing iterator-like activity) to perform search and retrieval is common in these systems.

1.3 Roadmap

The rest of the paper is structured as follows: Section 2 introduces our specification notation through the example of a specification of an immutable set, which includes the *elements* iterator; it explains special specification constructs used to accommodate concurrency and distribution in our model of computation. Section 3 presents the dimensions of our design space and four different points that would represent a reasonable semantics for weak sets in a distributed environment. It presents, in particular, the different specifications of the *elements* iterator for a set abstraction. The specifications themselves are fairly intuitive, so those readers either desiring only a cursory understanding of the design space or familiar with the Larch specification method may choose to skip Section 2. We close with a discussion of related work and a summary of our contributions.

2 Model of computation and specifications

A *computation*, i.e., program execution, is a sequence of alternating states and (atomic) transitions starting in some initial state, σ_0:

$$\sigma_0 \;\; \Delta_1 \;\; \sigma_1 \;\; \ldots \;\; \sigma_{n-1} \;\; \Delta_n \;\; \sigma_n$$

Each transition, Δ_i, of a computation sequence is a partial function on states. A *history* is the subsequence of states of a computation. States can change over time through the *invocation* of a procedure or iterator; each invocation is atomic. Like a procedure an iterator is *called*; but unlike set = **type** create, add, remove, size, elements

constraint $s_i = s_j$ *% set is immutable*

create = **proc** () **returns** (t: set)
ensures $t_{post} = \{\} \ \wedge \ \mathbf{new}(t)$

add = **proc** (s: set, e: elem) **returns** (t: set)
ensures $t_{post} = s_{pre} \cup \{e\} \ \wedge \ \mathbf{new}(t)$

remove = **proc** (e: elem, s: set) **returns** (t: set)
ensures $t_{post} = s_{pre} - \{e\} \ \wedge \ \mathbf{new}(t)$

size = **proc** (s: set) **returns** (i: int)
ensures $i_{post} = |s_{pre}|$

elements = **iter** (s: set) **yields** (e: elem)
remembers $yielded : set$ **initially** $\{\}$
ensures if $yielded_{pre} \subset s_{first}$
then *% still more to yield*
$yielded_{post} - yielded_{pre} = \{e\}$
$\wedge \ yielded_{post} \subseteq s_{first}$
$\wedge \ e \in s_{first} - yielded_{pre}$
$\wedge$ **suspends**
else *% no more to yield*
returns

Figure 1: A specification of an immutable set (ignoring failures)

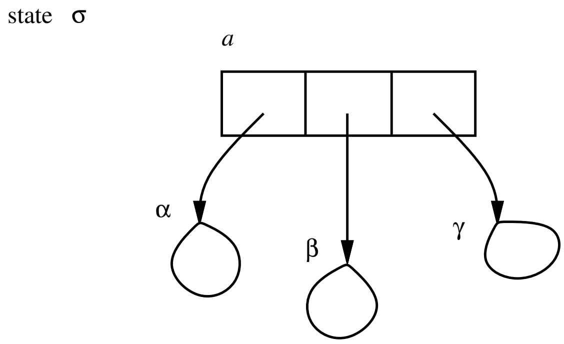

Figure 2: An array object that contains three objects

a procedure, it may *suspend* its state and later be *resumed* (invoked again), continuing from its suspended state. We consider the first call to an iterator as well as each resumption as an invocation of the iterator. Eventually, like a procedure, an iterator may terminate, returning normally or exceptionally.

Specifications of an object's operations (procedures and iterators) determine the legal state transitions in a computation. We adopt the Larch style of specifying procedures, iterators, and types [10, 11]. Figure 1 gives a type specification for an immutable set, s, that exports the *create*, *add*, *remove*, and *size* procedures and the *elements* iterator. We now explain the specification language in more detail.

2.1 Specification assertion language

We use the Larch Shared Language (LSL) [10] as an assertion language with which to write the pre- and post-conditions of the specifications of procedures and iterators. LSL is also used to specify a type's value space for objects. We omit the details of LSL here since in our examples we use standard set notation for the functions on sets, e.g., $\cup$ for set union and $-$ for set difference. The salient features, which have been introduced elsewhere (e.g., [10]), of the assertion language are as follows:

- We distinguish between an object and its value. An unsubscripted identifier, e.g., *x*, always denotes an object, and a subscripted identifier, e.g., x_σ, denotes its value in a particular state, σ. We also need to model objects that are collections of other objects. In order to treat a contained object as part of the value of the containing object, we treat objects as special kinds of values. In Figure 2, we depict an array object, a, in state σ where a contains the objects $\{\alpha, \beta, \gamma\}$; $a_\sigma[2] = \beta$.

- For example, in the specification of a procedure, *P*, we use the subscripts *pre* and *post* to distinguish between the value of an object in the state in which *P* is called (the "pre-state") and its value in the state in which *P* returns (the "post-state"). For an iterator, *I*, *pre* and *post* distinguish between the pre- and post-states for each invocation (i.e., the initial call and subsequent resumptions); we additionally use the subscript *first* to denote the state in which the iterator is first called and *last* for the state when the iterator terminates.[1]

- We assume a special object in the state called *terminates* whose value ranges over normal and exceptional termination conditions. We write **returns** to stand for the assertion that the operation terminates normally. We write **suspends** in iterator specifications to stand for the assertion that the iterator has yielded control back to the caller normally (but the iterator has not yet terminated).

- The assertion **new**(x) says that x is an object in the domain of the post-state that was not in the domain of the pre-state.

In the specification of *create* in Figure 1, t_{post} stands for the value of the newly created object t returned as a result of invoking *create*. The value is the empty set, $\{\}$.

We need to add two new constructs to our assertion language to deal specifically with the distributed nature of our application. We assume a model of a distributed system that is a set of connected nodes, not necessarily strongly connected. Processes (e.g., clients and servers) communicate via remote procedure calls. Thus the execution of an operation by a client at one node might actually involve a

[1] For the first invocation of an iterator, the "first-state" and first "pre-state" are the same; similarly, for the last invocation the "last-state" and last "post-state" are the same.

remote call to the operation exported by a server at a different node. Nodes may crash and communication links may fail. These failures may lead to network partitions, which implies that a process at one node may not be able to access objects residing at a node in a different partition. We assume we can detect failures, e.g., those signaled from the lower network and transport layers of the communication substrate. We write **fails** to stand for the assertion that an operation terminates with a special "failure" exception, denoting any kind of failure, e.g., a timeout, node crash, or link down, due to the distributed nature of the system.

The possibility of a network partition means that a caller may not be able to access a remote object. The caller will be able to detect this situation because the "failure" exception will be signaled. The unfortunate situation is when an accessible object (like a collection) contains ("points to") other objects, but one or more of the contained objects may not be accessible. For a collection object, x, we will assume a function **reachable**(x_σ) which determines the set of objects contained in x that are accessible in state σ. For example, in Figure 2, **reachable**$(a_\sigma) = \{\alpha, \beta, \gamma\}$. If a is on node N and α, β, and γ are on nodes A, B, and C, respectively, and there is a partition between N and C in state ρ then **reachable**$(a_\rho) = \{\alpha, \beta\}$.

2.2 Specifying procedures, iterators, and types

In the specification of a procedure, *P*, the predicate in the **requires** clause is *P*'s pre-condition. An omitted **requires** clause stands for the trivial predicate "true." The **modifies** clause is shorthand for a predicate that asserts that all objects *not* listed do not change in value; hence the value of an object listed explicitly in a **modifies** clause is allowed to change (but does not have to) as a result of calling *P*. An omitted **modifies** clause means no object may be mutated. The conjunction of the predicate denoted by the **modifies** clause and the predicate in the **ensures** clause is the specification of *P*'s post-condition. In Figure 1, all the operations have the trivial pre-condition; none modify their arguments.

The specification of an iterator, *I*, is similar to that for a procedure except the interpretation is slightly different since its behavior is slightly different. The pre-condition must hold each time the iterator is invoked. The post-condition holds each time the iterator yields and/or terminates. In the *elements* iterator in Figure 1, each time the iterator is invoked an element not already yielded is returned to its caller; this process continues until all elements in the original set (s_{first}) have been yielded.

For convenience, we use *history* objects (like *history variables* [12]) in the specification of iterators to help keep track of history information. These are local to the iterator and not accessible to its clients.[2] The clause

remembers $x : T$ **initially** *init*

introduces the history object x of type T with an initial value *init*. We require that for any history object, x, for an iterator I, that in any state, *first*, in which I is first called, $x_{first} = init$. In the specification of *elements* in Figure 1, the *yielded* set is a history object that keeps track of the elements already yielded; it starts out empty.

A type specification is a set of specifications of the procedures and iterators exported by the type. In the presence of concurrent processes, if two or more processes can access a shared, mutable object, then the effect of one may violate a property assumed by another. To capture what properties all processes must agree to uphold, we use a **constraint** clause in the specification for type T. The predicate we write in this clause states a *history property* of all computations involving any object of type T. (We borrow this idea from Liskov and Wing's technique for specifying subtypes [13].) A *history property* must hold for all successive pairs of states in a computation, and thus we formulate them as predicates over pairs of states. More formally, the **constraint** clause that appears in the specification of type T, **constraint** $P(x_i, x_j)$, stands for the predicate, for all computations, $\sigma_0\ \Delta_1\ \sigma_1\ \ldots\ \sigma_{n-1}\ \Delta_n\ \sigma_n$,

$$\forall\ x : T\ \ \forall\ 1 \le i < n, 1 < j \le n\ .\\ i < j \Rightarrow P(x_{\sigma_i}, x_{\sigma_j}).$$

Notice that we do not require that σ_j be an immediate successor of σ_i in the computation.

We capture the immutability property of sets in Figure 1 in the **constraint** clause, where s_i and s_j correspond to x_{σ_i} and x_{σ_j} in the predicate above. It requires that a set does not change in value, even in between invocations.

3 Exploring the design space

Sets are unordered collections of elements with no duplicates. In our context, sets can consist of objects, or of higher level groupings such as directories or indexes. A set is defined by the members it contains; hence membership of an element in a set is the key determining factor of what the set's value is. The operation in our set interface that defines membership is the *elements* iterator, so this section presents only the specification of this iterator; the differences in its specifications determine the differences in the semantics of weak sets.

The cases we need to consider are as follows:

- The collection over which we are iterating may or may not be mutated.
  - If the collection can mutate, then the cases are whether it can only grow, only shrink, or both grow and shrink.
- The items in the collection may or may not be mutated.
- If mutations are allowed, then either the iterator or some other process (like the caller) may cause them.
- In a distributed environment where we need to accommodate failures, we need to determine what the iterator's behavior should be if it cannot access all objects in the collection.

This last dimension is worth further elaboration. Communication failures have two effects. First, they could prevent an object that is known to be a member of the collection from being accessed. In the specifications that

[2] For brevity in our specifications, we omit them from the **modifies** clause; their values do not change unless otherwise explicitly specified.

Mutability	Failures	Implications
Immutable	Not allowed	$+$ Simplest, but strongest guarantees $-$ Impractical
Immutable	Allowed, but set unaffected	$+$ High data integrity $-$ Requires locking
Grow/Shrink	Loss of mutations	$+$ Clean membership semantics $-$ Requires atomic snapshot
Grow-only	Pessimistic	$+$ Low overhead $-$ Restricts availability of data
Grow/Shrink	Optimistic	$+$ Best availability of data $-$ Weakest consistency semantics

This table presents the five design points specified in this paper. The top entry's specification is in Figure 1; the others are discussed in Sections 3.1 through 3.4. Our implementation of dynamic sets is based on the semantics with the highest availability, discussed in Section 3.4.

Table 1: Summary of the design space of weak sets.

follow, we use the **reachable** function to help us determine the accessibility of objects in a given state. Second, they could prevent an iterator from seeing mutations to the collection. This second effect is subtle since it implies that the collection object itself may be distributed; logically there is a single object, but physically different parts of it may be scattered across many nodes, or the single "logical" object may be represented by a set of replicas. Whenever there is such distributed state, there is always the possibility of inconsistent data. One node may have more up-to-date information than another; cached data may be stale.

There are two main ways of handling the effects of communication failures[14]: pessimistically and optimistically. The pessimistic approach assumes that if a failure occurs, an update might have been missed (e.g., in the absence of a quorum), and any data available might be stale. Thus, it would be most appropriate to return a failure. The optimistic approach assumes the reverse is true, and allows access to the data even though it may be stale. The appropriate choice depends on the number of failures, and the tradeoff between high availability and consistency of the data. Thus, in our design space, along the dimension of dealing with failures, we will consider two cases: pessimistic, where if a failure is detected then the iterator should immediately terminate, and optimistic, where the iterator tries to make progress with the expectation that in a later invocation inaccessible objects will become accessible again (because the failure has been repaired by that time).

Two dimensions we will not discuss are who is responsible for mutations and what may be mutated. For a concurrent or distributed system it is reasonable to assume that the iterator does not mutate the set (it might keep a cached version, which is a way to implement a history object), but that any other process might. And, to keep things simple, we will assume that items in the set do not change; we could model this by the deletion of an old item from the set followed by the addition of a new item.

The remainder of this section describes four points in the design space. Table 1 summarizes the salient characteristics. A choice of one point over another has serious implications for the implementor. The more restrictive the specification, the harder it is to implement efficiently in a distributed system. For instance, preventing mutation requires distributed locking; allowing only growth requires

constraint $s_i = s_j$

elements = **iter** (s: set) **yields** (e: elem) **signals** (failure)
  remembers $yielded : set$ **initially** $\{\}$
  ensures **if** $yielded_{pre} \subset \textbf{reachable}(s_{first})$
    then $yielded_{post} - yielded_{pre} = \{e\}$
      $\wedge\ yielded_{post} \subseteq s_{first}$
      $\wedge\ e \in \textbf{reachable}(s_{first})$
      $\wedge$ **suspends**
    else if $(yielded_{pre} = \textbf{reachable}(s_{first})$
      $\wedge yielded_{pre} \subset s_{first})$
      then *% No more reachable elements*
        fails
      else % $yielded_{pre} = s_{first}$
        returns

Figure 3: Immutable set with failures

the ability either to prevent certain mutations or to cache the entire set. Although this functionality may be mandatory for some high-integrity systems (e.g., a bank's distributed database), it may too constraining for low-integrity systems, especially loosely-coupled ones (e.g., WWW).

3.1 Immutable set with failures

The specification in Figure 3 describes an iterator for an immutable set. As before, the **constraint** clause asserts the immutability property. The *yielded* set starts out empty and at each invocation it grows by one element unless a failure is detected. The **ensures** clause handles three cases: In the normal case of suspending the iterator, if there are still elements to yield (the set of elements already yielded is a strict subset of the set of reachable elements of the original set, s), then *yielded* grows by an element of s that is not already in *yielded*. A failure occurs if everything reachable has been yielded and the reachable set of elements is a strict subset of the original set. Finally, if all elements of the original set have been yielded, we can terminate the iterator.

Because the set is immutable, we can use the value of s in any state between the first-state and last-state. Our use

constraint $true$

elements = **iter** (s: set) **yields** (e: elem) **signals** (failure)
  remembers $yielded : set$ $\mathbf{initially}$ $\{\}$
  ensures if $yielded_{pre} \subset \mathbf{reachable}(s_{first})$
    then $yielded_{post} - yielded_{pre} = \{e\}$
      $\wedge\ yielded_{post} \subseteq s_{first}$
      $\wedge\ e \in \mathbf{reachable}(s_{first})$
      $\wedge\ \mathbf{suspends}$
    else if $(yielded_{pre} = \mathbf{reachable}(s_{first})$
      $\wedge yielded_{pre} \subset s_{first})$
      then % *No more reachable elements*
        fails
      else % $yielded_{pre} = s_{first}$
        returns

Figure 4: Mutable set, loss of some mutations

constraint $s_i \subseteq s_j$

elements= **iter**(s: set) **yields** (e: elem) **signals** (failure)
  remembers $yielded : set$ $\mathbf{initially}$ $\{\}$
    ensures if $yielded_{pre} \subset \mathbf{reachable}(s_{pre})$
      then $yielded_{post} - yielded_{pre} = \{e\}$
        $\wedge\ yielded_{post} \subseteq s_{pre}$
        $\wedge\ e \in \mathbf{reachable}(s_{pre})$
        $\wedge\ \mathbf{suspends}$
      else if $yielded_{pre} = s_{pre}$
        then
          $\mathbf{returns}$
        else
          $\mathbf{fails}$

Figure 5: Growing-only set, pessimistic failure handling

here of s_{first} allows us to make a sharp distinction between this specification and the one in the next section.

A less stringent specification would allow mutations to occur to the set when no one is iterating over it, but prohibit mutations during iteration. We could relax the constraint to be:

$$\textbf{constraint}\ \forall\ i < k < j\ .\ (\ terminates_i \neq \text{suspend} \wedge terminates_j \neq \text{suspend} \wedge terminates_k = \text{suspend}\ ) \Rightarrow (s_i = s_k = s_j)$$

which captures the property that between the first-state and last-state of the iterator, the set does not change. Thus mutations may occur between different uses of the iterator, but not between invocations of any one use.

Choosing this behavior, or even the less stringent one, has serious performance implications since typical implementations would use locks to synchronize access to the set and its elements. Iterating over a large, geographically dispersed set of objects is time consuming, especially if a human is responsible for flow control. The use of mobile (and possibly) disconnected computers may extend the period a lock is held indefinitely, thereby making it unacceptable to place such tight restrictions on the system. However, in a environment in which mutations and failures are rare, and the desire for data integrity is high, this behavior is an appropriate choice.

3.2 Mutable set with loss of mutations

The only visual difference between the specification in Figure 4 and the previous one in Figure 3 is the change in the **constraint** clause. Here, the predicate is "true"; the set may change arbitrarily over time.

The semantic difference is much greater, however. The iterator will yield only those elements of s as it appears the first time the iterator is called. Since in between subsequent invocations the set may change, the iterator may miss elements added to s after the first invocation and/or have yielded elements that have been removed. Thus, it "loses" mutations between the first-state and last-state. If clients were concerned about these possible losses, after the iterator terminates (**returns**), they could run the iterator again and hope to catch discrepancies. The failure case is handled as in the previous specification, based on the value of s in the first-state.

The implementation implications are not nearly as severe as in the immutable set case. This specification relaxes the need for locking since mutations are allowed to s after the initial invocation of the iterator. However, it still assumes that the set can be obtained in one atomic action (to get a snapshot of s in the first-state), and distributed atomic actions are extremely expensive in practice. Thus, this model is appropriate in environments in which failure is rare, and the consistency of the set is important. Note that for neither this nor the previous specification did we need to worry about failures masking mutations to s.

3.3 Growing-only set, pessimistic

The specification in Figure 5 allows the set only to grow and takes a pessimistic approach to consistency in the presence of failures.

The **constraint** clause asserts that the set may only grow. Unlike in the previous two specifications, each invocation uses the current state of s, i.e., the pre-state, not first-state. If there are still elements to yield based on the remembered set and the current state of the set, then we choose a reachable one and yield it. If there are no more elements to yield, we terminate. Otherwise, because we cannot reach an element that we know is in the set, we fail. Alternatively, one could easily specify the iterator to use a quorum or token-based scheme by changing the last line.

Notice that since the set may grow faster than the iterator yields elements from it, an iterator satisfying this specification may never terminate. Though the iterator could yield elements ad infinitum, in practice this behavior will not occur if objects are consumed more rapidly than they are produced.

Just as for the specification for the immutable set with failures (Figure 3), we could modify the **constraint** clause to permit arbitrary mutations between different runs of the iterator and growth only between invocations of any one

constraint $true$

elements= **iter**(s: set) **yields** (e: elem)
 remembers $yielded : set$ **initially** $\{\}$
 ensures **if** $\exists e \in s_{pre} \,.\, e \notin yielded_{pre}$
 then $yielded_{post} - yielded_{pre} = \{e\}$
 $\wedge\ e \in \mathbf{reachable}(s_{pre})$
 $\wedge$ **suspends**
 else returns

Figure 6: Growing and shrinking set, optimistic failure handling

run. Implementing this less stringent behavior is not difficult. To ensure that sets only grow during the iterator's use of the set, we can prevent objects from being deleted until the iterator terminates. Alternatively, we can create copies of any deleted objects and then garbage collect these "ghost" copies upon termination.

3.4 Growing and shrinking set, optimistic

The behavior of *elements* captured in our last specification (Figure 6) is the weakest of the four presented in this paper. There are no restrictions on mutation, there is only a weak guarantee about what is yielded, and it takes an optimistic approach to consistency.

As in the specification for the mutable set with losses (Figure 4), we allow the set to grow and shrink in between invocations. Here, however, we will not miss any additions since the yielded element is based on the current state of the set, not the state at the first invocation. However, we may still miss deletions, which means we may yield elements that are subsequently deleted.

We might specify that the yielded set will be a subset of the value of the set at some state between the first-state and the last-state, i.e., $yielded_{last} \subseteq s_i$ for some i between *first* and *last*. However, in the presence of deletions this may not be the case. An alternative would be to specify that $yielded_{last} \supseteq s_i$ for some i between *first* and *last*. However, this is not strong enough because this allows $yielded_{last}$ to have elements that were never in s_i. The specification we give requires that any element yielded must actually be in the set, for some state of the set between the first-state and last-state.

Orthogonal to handling mutations is handling failures. This specification takes an optimistic approach since it may never return if a failure is detected. The post-condition captures this blocking behavior by testing for the existence of some element of s_{pre} ($\exists e \in s_{pre} \ldots$) not yet yielded, but yielding, of course, only a reachable element ($e \in \mathbf{reachable}(s_{pre})$). We would not block if the test were for the existence of some element in the reachable set of s_{pre} (i.e., $\exists e \in \mathbf{reachable}(s_{pre}) \ldots$).

The guarantee on what is yielded may seem too weak to be usable, but it is entirely appropriate for the kinds of systems that we expect to be common in the future: loose collections of reference objects (e.g., encyclopedias or papers in archival journals) that are stored across many organizations. As mentioned in the introduction, several examples of such systems currently exist, and many more will be built. In these systems, performance and availability are key concerns, and since reference objects rarely or never change, inconsistent or stale data will rarely be seen. For these pragmatic reasons, we have chosen these semantics for our implementation of dynamic sets.

4 Related work

Our specification was greatly influenced by Garcia-Molina and Wiederhold's [15] taxonomy of queries. They use two dimensions for classification, and ignore communication failures. *Consistency* is the degree to which application constraints on data can be satisfied while *currency* is concerned with the version of the data returned by the query. In our terminology, set membership corresponds to consistency and mutability to currency. The specification in Figure 3 corresponds to a *strong consistency* (serializable), σ_{first}-*vintage* query; the one in Figure 4, to *weak consistency*, σ_{first}-*vintage*, and the last two specifications are both *no consistency,* σ_{first}-*bound*. However, as discussed in the introduction, the weak consistency semantics in Figures 5 and 6 may be most appropriate for modern wide area systems.

Tangential to the specification method we present here is work related to specifying iterators, which is the heart of our specifications. Wing's thesis [11] presents a formal approach to specifying iterators for CLU. We borrowed the main ideas needed: the distinction between the first and subsequent invocations, the distinction between suspension and termination, and the utility of history objects. In a similar Larch two-tiered style for specifying iterators for C [10], the specification of the control abstraction is broken into the specification of three C functions: one for initializing an iterator, one for getting the next element, and one for terminating the iterator; the equivalent of our history objects (specification variables) are defined in the underlying assertion language, LSL. Finally, Reynolds describes the semantics of iterators in terms of higher-order procedures in the context of the sequential programming language Algol-W [16]. None of these pieces of work address issues like failure, concurrency, and distribution because their context is a sequential programming language.

Also, in all three pieces of work, the researchers advise against allowing mutation to the collection object. The rationale given for Larch/C Interface Language is that such behavior would be inefficient to implement. Ironically, in distributed systems just the opposite is true. The implicit motivation given for Algol-W is that the behavior would be hard to reason about. The wide variation of possible behaviors within our design space justifies that reasoning about iterators can indeed be tricky.

5 Summary and status

The main contribution of this paper is a presentation of formal specifications of some of the more relevant design points for a new abstraction, weak sets. These sets must operate in a very general context, where concurrency and failures in a distributed system cannot be ignored. The power of our specification technique is in its ability to capture concisely and accurately subtleties in the different semantics presented in Section 3.

The process of specifying the semantics of weak sets had several benefits beyond the specifications themselves. First, the act of formally specifying the semantics allowed us to clarify and identify the salient aspects of the design space. Second, the process of mapping out the design space provided us with a better understanding of the implications of the design tradeoffs. In particular, by thinking about the way in which a set could be modified we realized it was impossible to offer strong consistency without modifying the protocols of the underlying system. Third, we are confident that our specifications capture the appropriate semantics for systems like the WWW.

We are currently implementing the weakest design, presented in Section 3.4, on a variety of Unix-like systems (Mach 2.6, OSF1, Linux). Our decision to choose this particular alternative was based on the desire to maximize the usability of the system while preserving good performance and ease of implementation. We expect users will want some minimal guarantee about the relationship between what history information they have accumulated about a set and its observable (and reachable) current state. At the same time anything stronger is unnecessary; users are usually willing to tolerate some inconsistency for a gain in performance. We hope to prove the performance benefits resulting from the use of a weak consistency semantics by evaluation of our system when it is complete.

Acknowledgments

We thank David Garlan and Steve King for their comments on an earlier version of this paper. We are grateful to the referees for their helpful comments.

Wing is supported in part by the Avionics Lab, Wright Research and Development Center, Aeronautical Systems Division (AFSC), U. S. Air Force, Wright-Patterson AFB, OH 45433-6543 under Contract F33615-90-C-1465, ARPA Order No. 7597. Steere is supported in part by the Advanced Research Projects Agency (Hanscom Air Force Base under Contract F19628-93-C-0193, ARPA Order No. A700), IBM Corporation, Digital Equipment Corporation, Intel Corporation, and Bellcore.

References

[1] D. Steere and M. Satyanarayanan, "A case for dynamic sets in operating systems," Tech. Rep. CMU-CS-94-216, School of Computer Science, Carnegie Mellon University, November 1994.

[2] E. Shekita and M. Zwilling, "Cricket: A mapped, persistent object store," in *Proc. of the 4th International Workshop on Persistent Object Systems*, (Martha's Vineyard, MA), 1990.

[3] O. Gruber, L. Amsaleg, L. Daynès, and P. Valduriez, "Eos, an environment for object-based systems," Tech. Rep. 1499, Institut National de Recherche en Informatique et en Automatique, 1991.

[4] D. Maier and J. Stein, "Development and implementation of an object-oriented DBMS," in *Readings in Object-Oriented Database Systems* (S. Zdonik and D. Maier, eds.), pp. 167–185, Morgan Kaufmann, 1990.

[5] B. Liskov, "Preliminary design of the Thor object-oriented database system," in *Proc. of the Software Technology Conference*, DARPA, April 1992.

[6] A. Albano and R. Morrison, eds., *Persistent Object Systems*. Workshops in Computing, London: Springer-Verlag, 1992. Proc. of the 5th Int'l Workshop on Persistent Object Systems, San Miniato, Italy.

[7] T. Berners-Lee, R. Cailliau, J. F. Groff, and B. Pollerman, "World wide web: The information universe," *Electronic Networking: Research, Applications, and Policy*, vol. 1, Spring 1992.

[8] B. Kahle and A. Medlar, "An information system for corporate users: Wide area information servers," *ConneXions – The Interoperability Report*, vol. 5, Nov 1991.

[9] M. McCahill, "The internet gopher: A distributed server information system," *ConneXions – The Interoperability Report*, vol. 6, July 1992.

[10] J. Horning, J. Guttag, S. Garland, K. Jones, A. Modet, and J. Wing, *Larch : Languages and Tools for Formal Specification*. New York: Springer-Verlag, 1993.

[11] J. Wing, *A Two-tiered Approach to Specifying Programs*. PhD thesis, MIT, Lab. for Comp. Sci., 1983.

[12] S. Owicki, *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, Cornell University, 1975.

[13] B. Liskov and J. Wing, "Specifications and their use in defining subtypes," in *Proc. of OOPSLA '93*, pp. 16–28, Sept. 1993.

[14] S. Davidson, H. Garcia-Molina, and D. Skeen, "Consistency in partitioned networks," *ACM Computing Surveys*, vol. 17, September 1985.

[15] H. Garcia-Molina and G. Wiederhold, "Read-only transactions in a distributed database," *ACM Trans. Database Systems*, vol. 7, June 1982.

[16] J. Reynolds, *The Craft of Programming*. Prentice-Hall International series in computer science, Englewood Cliffs, N.J.: Prentice/Hall International, 1981.