Specifying Recoverable Objects

Jeannette M. Wing
Computer Science Department
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper describes the results of an exercise in writing formal specifications. The specifications capture the system-critical *recoverability* property of data objects that are accessed by fault-tolerant distributed programs. Recoverability is a ''non-functional'' property requiring that an object's state survives hardware failures.

This exercise supports the claim that applying a rigorous specification method can greatly enhance one's understanding of software's complex behavior. The specifications enabled us to articulate precisely questions about an unstated assumption in the underlying operating system, incompleteness in the implementation of recoverable objects, implementation bias in the language design, and even incompleteness in the specifications themselves.

Biographical Sketch

Jeannette M. Wing is an Assistant Professor of Computer Science at Carnegie Mellon University. She received her S.B. and S.M. in Electrical Engineering and Computer Science in 1979, and her Ph.D. degree in Computer Science in 1983, all from the Massachusetts Institute of Technology. Her research interests include formal specifications, language design, concurrent and distributed systems, and visual languages. She was a key designer of the Larch family of specification languages and is internationally recognized in the field of formal specifications. Her current research activity includes the design and implementation of Avalon/C++, a programming language for fault-tolerant distributed computing.

Specifying Recoverable Objects

Jeannette M. Wing[1]

Department of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

1. Introduction

Formal specification languages have matured to the point where industry is receptive to using them and researchers are building tools to support their use. People use these languages for specifying the input-output behavior, i.e., *functionality*, of programs, but have largely ignored specifying a program's "non-functional" properties. For example, the functionality of a program that sorts an array of integers might be informally specified as follows: given an input array A of integers, an array B of integers is returned such that B's integers are the same as A's, and B's are arranged in ascending order. Nothing is said about the performance of the program like whether the algorithm for sorting should be $O(n)$ or $O(n^2)$. Performance is one example of a non-functional property.

In this paper, I will demonstrate the applicability of formal specifications to the non-functional property, *recoverability*. Recoverability requires that an object's state survives hardware failures. The correct behavior of these objects is fundamental to the correctness of the programs that create, access, and modify them. Sections 1.1 and 1.2 describe in more detail the context in which recoverable objects are used: fault-tolerant distributed systems. Section 2 describes how they are implemented at both the operating-system and programming-language levels.

The work described here is both theoretical and experimental in nature since the application of a formal (theoretical) specification language can itself be viewed as an experiment. Section 3 describes this specification exercise. The results of writing out specifications formally, summarized in Section 4, are extremely gratifying: they provide evidence that an existing specification language and method is suitable for describing a new class of objects; they validate the correctness of the design and implementation of a key part of an ongoing software development project; and not surprisingly, they demonstrate that the process of writing formal specifications greatly clarifies one's understanding of complex behavior.

1.1. Abstract Context: Fault-tolerant Distributed Systems

A distributed system runs on a set of nodes that communicate over a network. Since nodes may crash and communications may fail, such a system must tolerate faults; processing must continue despite failures. For example, an airline reservations system must continue servicing travel agents and their customers even if an airline's database is temporarily inaccessible; an automatic teller machine must continue dispensing cash even if the link between the ATM and the customer's bank account is down.

A widely-accepted technique for preserving data consistency and providing data availability in the

[1] This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), ARPA Order No. 4864 (Amendment 20), under contract F33615-87-C-1499 monitored by the Avionics Laboratory, Air Force Wright Aeronautical Laboratories, Wright-Patterson AFB and in part by the National Science Foundation under grant CCR-8620027.

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the US Government.

presence of failures and concurrency is to organize computations as sequential processes called transactions. A *transaction* is a sequence of operations performed on data objects in the system. For example, a transaction that transfers $25 from a savings account, *S*, to a checking account, *C*, might be performed as the following sequence of three operations on *S* and *C* (both initially containing $100):

$$\{S = \$100 \wedge C = \$100\}$$

Read(*S*)
Debit(*S*, $25)
Credit(*C*, $25)

$$\{S = \$75 \wedge C = \$125\}$$

In contrast to standard sequential processes, transactions must be serializable, total, and recoverable. *Serializability* means that the effects of concurrent transactions must be the same as if the transactions executed in some serial order. In the above example, if two transactions, T1 and T2, were simultaneously transferring $25 from S to C, the net effect to the accounts should be that S = $50 and C = $150 (that is, as if T1 occurred before T2 or vice versa). *Totality* means that a transaction either succeeds completely and *commits,* or *aborts* and has no effect. For example, if the transfer transaction aborts after the Debit but before the Credit, the savings account should each be reset to $100 (its balance before the transfer began). *Recoverable* means that the effects of committed transactions survive failures. If the above transfer transaction commits, and a later transaction that modifies S or C aborts, it should be possible to "roll back" the state of the system to the previous committed state where S = $75 and C = $125.

It can be guaranteed that the integrity of the entire system is maintained if each object accessed within transactions is *atomic*. That is, each object is an instance of an abstract data type with the additional requirement that it must ensure the serializability, totality, and recoverability of all the transactions that use its operations. For example, as long as the bank account's Read, Debit, and Credit operations are implemented "correctly," then any set of transactions that access the account will be serializable, total, and recoverable. The advantage of constructing a system by focusing on individual objects instead of on a set of concurrent transactions is modularity: one need only ensure that each object is atomic to ensure the more global atomicity property of the entire system.

Informally, a *recoverable* object is an object whose state can be restored to a previously "checkpointed" state if a node crash occurs. After a crash, a recoverable object is restored to a state that reflects only completed operations; the effects of operations in progress at the time of the crash are never observed. The restored state of a recoverable object must moreover reflect all operations performed by transactions that committed before the crash. Note that since a recoverable object's state may also reflect completed operations of aborted transactions, e.g., those active transactions that are automatically aborted at the time of the crash, recoverability is a weaker consistency property than totality. In the above bank account example, suppose immediately after the Debit a checkpoint of the system is made, and then while the Credit is in progress a crash occurs. The recoverable state of the system would be where S = $75 and C = $100.

The non-functional property of objects this paper focuses on is the recoverability aspect of atomic objects.

1.2. Concrete Context

The Avalon Project, co-managed by the author and Maurice Herlihy at Carnegie Mellon University, provides a concrete context for this work. We are implementing language extensions to C++ [10] to support application programming of fault-tolerant distributed systems. We rely on the Camelot System [9], also being developed at CMU, to handle operating-systems level details of transaction management, inter-node communication, commit protocols, and automatic crash recovery.

The formal specification language used in Section 3's specifications is Larch [7], though others such as Gypsy [5], VDM [2], Z [1], and OBJ [4], might also be suitable. The advantage gained in using Larch is the explicit separation between specifying state-dependent behavior (for example, side effects and resource allocation) and state-independent behavior (for example, the last-in-first-out property of stacks).

Where appropriate the details of Avalon, Camelot, and Larch are given, but the implications of the results of the specification exercise are independent of these projects.

A chart of the various people involved, the level (language or system) at which they are involved, and kinds of questions they address is shown below:

Person(s)	Language/System	Questions Addressed
Specifier	Larch	What is a recoverable object? What are the effects of its operations?
Language implementor	Avalon	How is a recoverable object represented in memory? How are its operations implemented?
Operating system builders	Camelot	How is memory managed? What protocol is used to recover memory after crashes?

2. Recoverable Objects

In order to appreciate the issues that arose in the specification of recoverable objects, in particular with respect to their implementation, it helps to understand their physical storage implementation (Camelot) and their programmer interface (Avalon).

2.1. The Operating System's View

Conceptually, there are two kinds of storage for objects: *local* storage whose contents are lost upon crashes, and *stable* storage whose contents survive crashes with high probability. (Stable storage may be implemented using redundant hardware [8] or replication [3].) Recoverable objects are allocated in local storage, but their state is written to stable storage so that recovery from crashes can be performed. If every recoverable object is *logged* to stable storage after modifying operations are performed on it in local storage, then its state may be recovered after a crash by ''replaying'' the log. Replaying the log is a sufficient method for restoring a object's state.

However, recovering the state of an object entirely from the log is a time-consuming operation. Camelot speeds up crash recovery by dividing local storage into two classes, volatile storage and non-volatile storage, and by distinguishing between two crash modes, node failures and media failures. In a *media failure*, both volatile and non-volatile storage are destroyed, while in a *node failure*, only volatile storage is lost. In practice, node failures are far more common than media failures. To optimize recovery from node failures, a protocol known as *write-ahead logging* [6] is used. An object is modified in the following steps:

1. The page(s) containing the object are *pinned* in volatile storage; they cannot be returned to non-volatile storage until they are *unpinned.*
2. Modifications are made to the object in volatile memory.
3. The modifications are logged on stable storage.
4. The page(s) are unpinned.

The first step of the protocol ensures that the pages containing the object are not written to non-volatile storage while a modifying operation is in progress. This protocol ensures that a recoverable object can be restored to a consistent state quickly and efficiently. Upon crash recovery, the status of each transaction is determined, and by comparing what is in non-volatile storage to what is in stable storage, one can "redo" the effects of committed transactions and "undo" the effects of aborted ones. (For more details, see [9]). Notice that modifications must still be logged to stable storage to protect against the occurrence of a media failure.

2.2. The Programmer's View

The programmer's interface to a recoverable object is through the Avalon class header shown in Figure 2-1.

```
class recoverable {
 public:

  void pin(int size);       // Pins object in physical memory.
  void unpin(int size);     // Unpins and logs object to stable storage.
}
```

Figure 2-1: Recoverable Class Definition

Informally, the *pin* operation causes the pages containing the object to be pinned, as required by the write-ahead logging protocol, while *unpin* logs the modifications to the object and unpins its pages. A recoverable object must be pinned before it is modified, and unpinned afterwards. For example if *x* is a recoverable object, a typical use of the pin and unpin operations within a transaction would be:

```
start {  // begin transaction
        ...
     x.pin();
         // modify x here
     x.unpin()
        ...
};        // end transaction
```

After a crash, a recoverable object will be restored to a previous state in which it was not pinned. Transactions can make nested *pin* calls; if so, then the changes made within inner *pin/unpin* pairs do not become permanent, i.e., written to stable storage, until the outermost *unpin* is executed.[2] Classes derived from *class recoverable* inherit *pin* and *unpin* operations, which can be used to ensure recoverability for

[2] Calls to *pin* and *unpin* must balance much like left and right parentheses should.

objects of the derived class.

The purpose of the specification exercise was to specify formally the effects of the *pin* and *unpin* operations, and hence the properties recoverable objects preserve.

3. Specifications

This section presents a sequence of three specifications as the different versions evolved during the specification process. The first version is more general than the second, but unimplementable. The third version removes an implementation bias that appears in both the first and second.

3.1. Version 1

The first version (see Figure 3-1) captures the following two properties of recoverable objects:

1. Each transaction can pin and unpin the same object multiple times.
2. Only at the last unpin does the object's value get written to stable storage.

First, we walk through the specification step-by-step. The following Larch *interface* specifications specify, using pre-and post-conditions, the effects of the *pin* and *unpin* operations:

```
pin = oper (x: recoverable)
  post x' = pn(x, self)

unpin = oper (x: recoverable)
  pre  pinned(x)
  post x' = un(x, self)
```

Self denotes the transaction (intuitively, the thread of control) that calls the operation. The precondition for *pin* is identically equal to true, meaning that a transaction can call *pin* in any state. The precondition for *unpin* requires that an object cannot be unpinned unless it is already pinned, given by the predicate of the same name.

The postconditions of the *pin* and *unpin* operations state what the changed value of a recoverable object is: *x* stands for the object's value in the initial state (upon invocation) and *x'* stands for its value in the final state (upon return). The postconditions make use of two auxiliary functions, specified in the Larch RecObj *trait*. *Pn* and *un* have the following signatures:

```
pn: R, Tid → R
un: R, Tid → R
```

where *R* and *Tid* are *sort* identifiers. *Pn* and *un* each take a recoverable object's value and a transaction identifier and return a (new) value for a recoverable object.

In any given state, a recoverable object's value is determined by the states of the transactions that have pinned it and the actual value of the object in memory. Thus, it is useful to "model" the value of a recoverable object as a pair of a Table and Memory.

```
Pair (R for T, Table for T1, Memory for T2)
```

where the **for** clauses rename sort identifiers (*T*, *T1*, and *T2*) that appear in one specification (*Pair*) used in another (*RecObj*).

The table component, indexed by transaction identifiers, keeps track of the number of times each transaction pins and unpins an object. The memory component keeps track of the actual value (with sort

```
class recoverable: interfaces
     based on R from RecObj

pin = oper (x: recoverable)
  post  x' = pn(x, self)

unpin = oper (x: recoverable)
  pre  pinned(x)        // cannot unpin something that's not already pinned
  post  x' = un(x, self)


RecObj: trait
  includes
     Pair (R for T, Table for T1, Memory for T2)
     TableSpec (Table for T, Tid for Index, Card for Val)
     Triple (Memory for T, M for T1, M for T2, M for T3, v for .first, n for .second, s for .third)
  introduces
     pn: R, Tid → R
     un: R, Tid → R
     pinned: R → Bool
  asserts for all (m: Memory, tb: Table, t: Tid, r: R)
     pn(<tb, m>, t) =
        if t ∈ tb                                  // already pinned?
        then <change(tb, t, eval(tb, t)+1), m>     // increment count
        else <add(tb, t, 1), m>                    // initialize it
     un(<tb, m>, t) =
        if eval(tb, t) = 1                         // if last unpin
        then <remove(tb, t), <m.v, m.v, m.v>>      // write to stable storage
        else <change(tb, t, eval(tb, t)-1), m>     // or just decrement count
     pinned(<tb, m>) = ~isEmpty(tb)
```

Figure 3-1: Specification of Class Recoverable: Version 1

M) of the object, as stored in each of the three levels of storage: volatile (*v*), non-volatile (*n*), and stable (*s*).

```
TableSpec (Table for T, Tid for Index, Card for Val)
Triple (Memory for T, M for T1, M for T2, M for T3, v for .first, n for .second, s for .third)
```

The meaning of *pn* is given by the following equation:

```
pn(<tb, m>, t) =
  if t ∈ tb
  then <change(tb, t, eval(tb, t)+1), m>
  else <add(tb, t, 1), m>
```

If the object (the pair <*tb, m*>) is already pinned by the given transaction (*t*), then *t*'s count is incremented in the table; otherwise a new entry is added to the table where the count is initialized to 1.

The meaning of *un* is as follows:

```
un(<tb, m>, t) =
  if eval(tb, t) = 1
  then <remove(tb, t), <m.v, m.v, m.v>>
  else <change(tb, t, eval(tb, t)-1), m>
```

Upon unpinning an object, for a given transaction (*t*), if its count of pins is down to 1, the object's value in volatile storage should be written to non-volatile and stable storage; otherwise, the count should merely be decremented by 1 and no change should be made to memory.

Putting all these pieces together results in the full specification shown in Figure 3-1. The Appendix contains the *Pair*, *TableSpec*, and *Triple* specifications.

3.2. Version 2

The specification of the previous section was shown to the implementor of *class recoverable* (Figure 2-1) in order to verify that indeed the implementation satisfies the specification. The implementor immediately noticed what he thought was an error in his implementation: The specification permits different transactions to pin the same object at the same time, whereas the implementation does not. The implementor proceeded to change his implementation to satisfy the specification, but then realized that the specified semantics was unimplementable! The underlying operating system (Camelot) forbids more than one transaction to pin an object (as represented as pages in volatile memory) at once. It assumes that any transaction pinning an object will modify that object and thus would want to prevent any other transaction from simultaneously accessing that object. A pinned object is a write-locked one as well. Thus, it was impossible to implement the less restrictive, but desired, semantics of *pin*; in short, the specification was "correct," but unimplementable.

The revised specification (Figure 3-2), which is more restrictive but implementable, captures this third property of recoverable objects:

3. Only one transaction can pin an object at once.

This specification is simpler to understand than the previous one because there is less information to keep track of. In essence, the table of transaction identifiers and their corresponding pin counts reduces to a single transaction and its count.

The specifications for *pin* and *unpin* change slightly:

```
class recoverable: interfaces
      based on R from RecObj

pin = oper (x: recoverable) signals (already_claimed)
  post  x' = pn(x, self) ∧
         x.trans ≠ self ⟹ signal already_claimed

unpin = oper (x: recoverable)
  pre   pinned(x) ∧ x.trans = self
      post  x' = un(x, self)


RecObj: trait
  includes
      Triple (R for T, Tid for T1, Memory for T2, Card for T3, trans for .first, count for .third)
      Triple (Memory for T, M for T1, M for T2, M for T3)
  introduces
      pn: R, Tid → R
      un: R, Tid → R
      pinned: R → Bool
  asserts for all (m: Memory, m1, m2, m3: M, c: Card, t1, t2: Tid)
      pn(<t1, m, c>, t2) =
          if c > 0                                // is already pinned?
          then if t1 = t2                         // by same transaction
                  then <t1, m, c+1>               // increment count
                  else <t1, m, c>                 // otherwise, leave unchanged
          else <t2, m, 1>                         // initialize it
      un(<t1, <m1, m2, m3>, c>, t2) =
          if t1 = t2                              // don't have to check if pinned already
          then if c = 1                           // if last unpin
                  then <t1, <m1, m1, m1>, 0>      // write to stable storage
                  else <t1, <m1, m2, m3>, c-1>    // or just decrement count
          else <t1, <m1, m2, m3>, c>              // no change
      pinned(r) = r.count > 0
```

Figure 3-2: Specification of Class Recoverable: Version 2

```
pin = oper (x: recoverable) signals (already_claimed)
  post x' = pn(x, self) ∧
       x.trans ≠ self ⇒ signal already_claimed

unpin = oper (x: recoverable)
  pre pinned(x) ∧ x.trans = self
  post x' = un(x, self)
```

Pin might terminate with an error condition signaled to the invoker to indicate that the object to be pinned is already pinned by some other transaction. *Unpin* requires not only that its argument is already pinned, but that it is pinned by the calling transaction.

Since concurrent pins by different transactions are not allowed, it is unnecessary to keep track of a table of pin counts per transaction. It suffices to associate with a recoverable object, a single transaction identifier, its value in memory, and a pin count:

```
Triple (R for T, Tid for T1, Memory for T2, Card for T3, trans for .first, count for .third)
```

Assume initially that each recoverable object, *x*, is unpinned, i.e., *x*.count = 0.

The auxiliary functions, *pn* and *un*, change accordingly:

```
pn(<t1, m, c>, t2) =
  if c > 0
  then if t1 = t2
         then <t1, m, c+1>
         else <t1, m, c>
  else <t2, m, 1>
```

If the count (*c*) is non-zero, then the object must be pinned. If the object is pinned by a transaction (*t1*) that is the same as the transaction (*t2*) attempting to pin the already pinned object, then the count is incremented; otherwise, the object is left unchanged. If the object is not already pinned, then its value is initialized with the pinning transaction's identifier and a count of 1.

```
un(<t1, <m1, m2, m3>, c>, t2) =
  if t1 = t2
  then if c = 1
         then <t1, <m1, m1, m1>, 0>
         else <t1, <m1, m2, m3>, c-1>
  else <t1, <m1, m2, m3>, c>
```

Unlike for *pn*, it is unnecessary for *un* to check if the object is already pinned since the precondition of *unpin* checks for this case. So *un* first checks to see if the transaction (*t1*) that currently has the object pinned is the same as the unpinning transaction (*t2*). If so, then if there is only one outstanding call to pin (*c = 1*), the value of the object in volatile storage is written to non-volatile and stable storage; otherwise, the count is decremented. If the unpinning transaction is different from the pinning one, then no change is made.

An Aside for Larch Readers

A typical use of *class recoverable* is to define a derived class for a recoverable type of object, say *class rec_foo*. If *foo* is the sort identifier associated with values of objects of type *rec_foo*, then the identifier M that appears in the *RecObj* specification would be renamed with *foo*. That is, the header for the Larch interface specification for a rec_foo class would look like:

```
class rec_foo: interfaces
   based on R from RecObj (foo for M)

   // ... specifications of operations for rec_foo objects ...
```

3.3. Version 3

Notice that nowhere in the previous specification is the distinction between non-volatile and stable storage used. For example, when an object is finally unpinned, its value is written out to both non-volatile and stable storage:

```
un(<t1, <m1, m2, m3>, c>, t2) =
     ...
        then <t1, <m1, m1, m1>, 0>
     ...
```

The second two components of Memory are treated identically. In unpinning an object, it is necessary that stable storage be updated using volatile storage's value, but writing out to non-volatile storage is strictly not necessary.

This observation reveals an implementation bias in the specification. The underlying operating system implements memory as a three-level storage hierarchy, and uses the write-ahead logging protocol to exploit the distinction between volatile and non-volatile storage for crash recovery. Recall, however, that conceptually a recoverable object has only two possible "values": that in volatile storage and that in stable storage. It suffices to consider only a two-level storage hierarchy with just volatile and stable storage. The change to the previous specification is trivial since Memory simply becomes a pair:

```
Pair (Memory for T, M for T1, M for T2, v for .first, s for .second)
```

and *un* changes accordingly:

```
un(<t1, <m1, m2>, c>, t2) =
     ...
        then <t1, <m1, m1>, 0>
        else <t1, <m1, m2>, c-1>
     else <t1, <m1, m2>, c>
     ...
```

4. Observations

The different versions of the specification made it possible to articulate precisely questions about the semantics of recoverable objects as well as questions about the implementation. The feedback between the specifier and implementor and between the specifier and language designers helped everyone gain insight about the implementability of the desired semantics, incompleteness in the current implementation, implementation bias in the language design, and even incompleteness in the specifications as presented.

4.1. Unstated Assumption

The major observation as a result of this specification exercise is that the specification helped identify an unstated and critical assumption in the underlying operating system that was reflected in the implementation. The implementation precluded the possibility of concurrent pins by different transactions. The underlying system forbids this situation because it assumes that any transaction that pins an object intends to modify it.

This assumption reflects a key point at the operating-system level where recovery and synchronization of objects are inseparable. Without concurrency, one can give a meaning to recoverability; without recovery, one can give a meaning to the correct synchronization of processes. But to support both, there are points when one must consider both recovery and synchronization together. Here is exactly one of those points. Synchronization of concurrent, modifying transactions is built into the meaning of recoverability of objects. This point was not well understood by either the language designers or the language implementors because the assumption was never stated by the underlying operating system builders. Only through this specification exercise and subsequent discussion between the language implementors and system builders was this point clarified.

4.2. Incompleteness in the Implementation

When presented with the specification of the *unpin* operation (any version), the implementor was asked whether the precondition on *unpin* (requiring that the object be pinned and that the check is with respect to the calling transaction) could be removed. That is, should the responsibility of checking the stated precondition be on the caller of *unpin* or the implementor? Currently, the responsibility lies with the caller; however, it could easily be checked at runtime as part of the implementation. If the object is not pinned or pinned by some other transaction, an appropriate error message could be signaled to the caller, much like the error condition signaled in the *pin* opeation. The implementor was alerted to this assymmetry in handling error conditions only when the formal specification was presented to him.

4.3. Implementation Bias in the Language Design

The specification also revealed a subtle point of misunderstanding between the language designers and language implementor. *Class recoverable* is actually implemented to provide a stronger property, operation-consistency, than just recoverability. *Operation-consistency* requires that an object be restored to some consistent state that reflects all operations of committed transactions plus some prefix of the sequence of operations performed on the object by transactions active at the time of a crash. Since the implementation supported this stronger property and since the designers never carefully defined (that is, specified) recoverability, the meaning of recoverability was confused with the implementation of recoverability; thus, until this specification exercise was performed, the language designers believed that operation-consistency was inherent to recoverability.

The nondeterminism inherent in this stronger definition would force the specification to keep track of a set of possible values (each representing a prefix of operations of uncommitted transactions) in stable storage (the third component of Memory in Figure 3-2) rather than a single value. When the object's state is restored upon recovery, any one of the values in this set would correctly represent a previous operation-consistent state. One would additionally need to ensure that the restored states of all objects reflect the same prefix of operations of all uncommitted transactions. For example, if transaction *T* were active at the time of the crash and states of objects *x* and *y* are restored, if some prefix of *T*'s operations is reflected by the *x*'s restored state, then the same prefix must also be reflected in *y*'s. Note that specifying this property cannot be done locally, i.e., per object; it is inherently a global property that involves the states of all objects in the system. One would specify a system-wide operation, *recover*, which would refer to the recovered, operation-consistent states of all the system's objects.

4.4. Incompleteness in the Specification

As is, the specification for *class recoverable* is not complete: initialization of a recoverable object is unspecified. Informally, a recoverable object is initially some block of memory with no associated transaction identifier (and of course no pin count) and no initial value. No transaction identifier is associated with a recoverable object until it is first pinned. Allocation of memory should be specified in the postcondition for a separate *create* operation:

```
create = oper () returns (x: recoverable)
  post new x
```

where ''**new** x'' is a special Larch assertion stating that *x* denotes some previously free block of memory. Also, either *pin*'s precondition should require that its argument has been previously allocated (making it the responsibility for the caller to check), or the auxiliary function *pn* should be modified accordingly (making it the responsibility of the implementor to check).

5. Concluding Remarks

In some sense the details of the problems discussed in the previous sections are less interesting than the insights gained from undertaking the process of rigorously specifying recoverability. This process enabled us to clarify fuzzy notions about recoverable objects; and to state precisely problems revealed in the specification, design, and implementation and to resolve their discrepancies.

Since this specification exercise was performed in the context of an ongoing large software development project, it was especially rewarding to identify points of confusion between desired and implementable semantics, to discover incompleteness in the implementation, and to separate out implementation biases from the design. A language like Avalon has more complex semantics than a standard sequential programming language; knowing early on that a fundamental part of its semantics is implemented correctly is a tremendous reassurance to us and future Avalon programmers. As language implementors, we promise to provide certain properties of the built-in classes like *class recoverable* so that when people use Avalon they need not worry that some error they find in their code might in fact be an error in ours. In particular, recoverability is a nontrivial, system-critical property of objects. The rest of the Avalon language class hierarchy derives from *class recoverable*, both in defining other built-in classes like *class atomic*, and in defining user-defined classes like recoverable strings or atomic queues. It is still impractical and unreasonable to specify formally large software systems completely, but the benefits of tackling smaller, system-critical pieces are large.

Finally, as mentioned in the introduction, we are able to demonstrate that formal specification techniques can be extended naturally to specify non-functional properties like recoverability. We intend to continue this specification exercise for the other built-in and user-defined classes of Avalon, in particular those that support other aspects of the *atomicity* property of objects.

Acknowledgments

I thank Maurice Herlihy for helping me better understand recoverability, David Detlefs for actually implementing recoverable objects, and Mathew Brozowski for his interest in specifying their behavior.

I. Other Specifications Used

```
TableSpec: trait
  introduces
     new: → Table
     add: Table, Index, Val → Table
     _∈_: Index, Table → Bool
     remove: Table, Index → Table
     eval: Table, Index → Val
     change: Table, Index, Val → Table
     isEmpty: Table → Bool
  asserts
     Table generated by (new, add)
     Table partitioned by (_∈_ , eval)
     for all (t: Table, ind, ind1, Index, val, val1: Val)
        ind ∈ new = false
        ind ∈ add(t, ind1, val) = (ind = ind1) | ind ∈ t
        eval(add(t, ind, val), ind1) = if ind = ind1 then eval else eval(t, ind1)
        remove(add(t, ind, val), ind1) =  if ind = ind1
                     then t
                     else add(remove(t, ind1), ind, val)
        change(add(t, ind, val), ind1, val1) = if ind = ind1
                        then add(t, ind, val1)
                     else add(change(t, ind1, val1), ind, val)
        isEmpty(new) = true
        isEmpty(add(t, ind, val)) = false
  implies converts (_∈_, remove, eval, change)
     exempting eval(new, ind), remove(new, ind), change(new, ind, val)

Triple: trait
  introduces
  <_, _, _>: T1, T2, T3 → T
  _.first: T → T1
  _.second: T → T2
  _.third: T → T3
  asserts
     T generated by (<_, _, _>)
     T partitioned by (first, .second, .third)
     for all (a: T1, b: T2, c: T3)
        <a, b, c>.first = a
        <a, b, c>.second = b
        <a, b, c>.third = c

Pair: trait
  introduces
  <_, _>: T1, T2 → T
  _.first: T → T1
  _.second: T → T2
  asserts
     T generated by (<_, _>)
     T partitioned by (.first, .second)
     for all (a: T1, b: T2)
        <a, b>.first = a
        <a, b>.second = b
```

References

[1] J.R. Abrial.
The Specification Language Z: Syntax and Semantics.
Technical Report, Programming Research Group, Oxford University, 1980.

[2] D. Bjorner and C.G. Jones (Eds.).
Lecture Notes in Computer Science. Volume 61: *The Vienna Development Method: the Meta-language.*
Springer-Verlag, Berlin-Heidelberg-New York, 1978.

[3] D. S. Daniels.
Distributed Logging for Transaction Processing.
In *Proceedings of the 1987 ACM Sigmod International Conference on Management of Data.* Association for Computing Machinery, San Francisco, CA, May, 1987.

[4] J.A. Goguen and J.J. Tardo.
An Introduction to OBJ: A Language for Writing and Testing Formal Algebraic Program Specifications.
In *Proceedings of the Conference on Specifications of Reliable Software*, pages 170-189. Boston, MA, 1979.

[5] D.I. Good, R.M. Cohen, C.G. Hoch, L.W. Hunter, and D.F. Hare.
Report on the Language Gypsy, Version 2.0.
Technical Report ICSCA-CMP-10, Certifiable Minicomputer Project, The University of Texas at Austin, September, 1978.

[6] J. Gray.
Notes on Database Operationg Systems.
In *Lecture Notes in Computer Science.* Volume 60: *Operating Systems: an Advanced Course.* Springer-Verlag, Berlin, 1978.

[7] J.V. Guttag, J.J. Horning, and J.M. Wing.
The Larch Family of Specification Languages.
IEEE Software 2(5):24-36, September, 1985.

[8] B. Lampson.
Atomic transactions.
Lecture Notes in Computer Science 105. Distributed Systems: Architecture and Implementation.
Springer-Verlag, Berlin, 1981, pages 246-265.

[9] A. Spector, J. Bloch, D. Daniels, R. Draves, D. Duchamp, J. Eppinger, S. Menees, D. Thompson.
The Camelot Project.
Database Engineering 9(4), December, 1986.

[10] B. Stroustrup.
The C++ Programming Language.
Addison-Wesley, Reading, Massachusetts, 1986.