53

IO Streams: Abstract Types, Real Programs

Mark R. Brown and Greg Nelson

November 15, 1989

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301

Systems Research Center

DEC's business and technology objectives require a strong research program. The Systems Research Center (SRC) and three other research laboratories are committed to filling that need.

SRC began recruiting its first research scientists in 1984 — their charter, to advance the state of knowledge in all aspects of computer systems research. Our current work includes exploring high-performance personal computing, distributed computing, programming environments, system modelling techniques, specification technology, and tightly-coupled multiprocessors.

Our approach to both hardware and software research is to create and use real systems so that we can investigate their properties fully. Complex systems cannot be evaluated solely in the abstract. Based on this belief, our strategy is to demonstrate the technical and practical feasibility of our ideas by building prototypes and using them as daily tools. The experience we gain is useful in the short term in enabling us to refine our designs, and invaluable in the long term in helping us to advance the state of knowledge about those systems. Most of the major advances in information systems have come through this strategy, including time-sharing, the ArpaNet, and distributed personal computing.

SRC also performs work of a more mathematical flavor which complements our systems research. Some of this work is in established fields of theoretical computer science, such as the analysis of algorithms, computational geometry, and logics of programming. The rest of this work explores new ground motivated by problems that arise in our systems research.

DEC has a strong commitment to communicating the results and experience gained through pursuing these activities. The Company values the improved understanding that comes with exposing and testing our ideas within the research community. SRC will therefore report results in conferences, in professional journals, and in our research report series. We will seek users for our prototype systems among those with whom we have common research interests, and we will encourage collaboration with university researchers.

Robert W. Taylor, Director

IO Streams: Abstract Types, Real Programs

Mark R. Brown and Greg Nelson

November 15, 1989

Authors' abstract

The paper proposes standard Modula-3 interfaces for text input and output. It also describes an implementation of the interfaces, focusing on two novel features of Modula-3: the partially opaque type and the explicit isolation of unsafe code.

Mark R. Brown and Greg Nelson

Capsule review

In this report the authors make good use of the example of a text IO package, written in Modula-3, to achieve a number of different aims. They state clearly at the start what those aims are and then proceed to accomplish them.

The package is presented in complete detail, starting from an abstract interface, and eventually reaching an efficient machine-dependent implementation. This specifies the package completely in a manner that the authors hope will be used as a standard for a text IO package.

The example is used to illustrate how the Modula-3 construct of a *partially opaque type* can be used to finely control the amount of information-hiding within a module. Good use of this feature means that each layer of interface reveals only the amount of information actually required at that level, thereby preserving security and aiding comprehension.

One of the original features of Modula-3 is the notion of a module presenting a *safe interface*; that is, an interface that is guaranteed not to produce an unchecked runtime error. Since the IO Streams require unsafe components at the lower levels to achieve efficiency, the authors use the example to illustrate how a safe interface can be constructed from an unsafe one, showing how the safe boundary is clearly delineated.

The authors accomplish the difficult task of presenting a complex example in a great deal of detail, without obscuring the issues they wish to illustrate. This report could be recommended to a number of different classes of reader, ranging from those who wish to implement an IO package, through those who wish to see Modula-3 in use, to those who have to document a package for use by others.

Kevin D. Jones

Contents

Where the stream runneth smoothest, the water is deepest.
—John Lyly

1 Introduction

Our first goal is to define Modula-3 interfaces for text input and output. The interfaces define two types of objects, *readers* and *writers*, collectively called *streams*. Streams come in a potentially unlimited number of classes, such as streams to and from terminals, disk files, etc. We hope these interfaces will become standards.

Our second goal is to illustrate the *partially opaque type*, a Modula-3 feature that allows flexible data abstraction. A quick survey of the literature will show that there are hundreds of language features to support abstract data types, but only one example—the stack. To give a realistic example of the partially opaque type in action, we will describe the Modula-3 streams package in detail, from top to bottom.

Our final goal is to illustrate the explicit isolation of unsafe code. Reading and writing characters must be fast, and on some systems this will require unsafe, machine-dependent code. The program described in this paper contains two modules that can be reprogrammed in a machine-dependent way. (Of course, reprogramming them does not affect the abstract properties of streams.) We present versions of the modules that are suitable for byte-addressable machines. They use pointer arithmetic, and are therefore unsafe.

As a general rule, the upper layers of a system are safer than the lower layers. In Modula-3, where safety has a precise technical meaning, the transition between the safe and the unsafe is not gradual: it occurs where an unsafe module exports a safe interface. Programming this layer is very error-prone; the streams package provides a realistic example of the dangers.

We will view streams at three levels of detail. At the highest level, the client interfaces `Rd` and `Wr` define streams as abstract types. In this view the types are completely opaque. At the intermediate level, the class interfaces `RdClass` and `WrClass` reveal the buffer structure that is needed to implement new classes of streams. Here the types are partially opaque. At the lowest level, the modules `RdRep` and `WrRep` reveal the complete representation, and contain the potentially machine-dependent code.

The client and class interfaces are safe; the low-level modules are unsafe. There are also two interfaces, `UnsafeWr` and `UnsafeRd`, which reveal the semaphores that make operations on readers and writers atomic.

Perhaps the first object-oriented I/O package was part of the Simula system [1]. The first to use class-independent buffering seems to be the I/O system for the OS6 described by J. E. Stoy and C. Strachey in 1972 [5]. The package described in this paper is closely based on the Modula-2+ streams package used in the Topaz System at Digital's Systems Research Center.

The program in this paper is written in revised Modula-3 [2].

2 The Wr interface

A `Wr.T` (or "writer") is a character output stream. The basic operation on a writer is `PutChar`, which extends a writer's character sequence by one character. Some writers (called "seekable writers") also allow overwriting in the middle of the sequence. For example, writers to random access files are seekable, but writers to terminals and sequential files are not.

Writers can be (and usually are) buffered. This means that operations on the writer don't immediately affect the underlying target of the writer, but are saved up and performed later. For example, a writer to a disk file is not likely to update the disk after each character.

Abstractly, a writer `wr` consists of:

`len(wr)`	a non-negative integer
`c(wr)`	a character sequence of length `len(wr)`
`cur(wr)`	an integer in the range `[0..len(wr)]`
`target(wr)`	a character sequence
`closed(wr)`	a boolean
`seekable(wr)`	a boolean
`buffered(wr)`	a boolean

These values are generally not directly represented in the data fields of a writer object, but in principle they determine the state of the writer.

The sequence `c(wr)` is zero-based: `c(wr)[i]` is valid for `i` from `0` through `len(wr)-1`. The value of `cur(wr)` is the index of the character in `c(wr)` that will be replaced or appended by the next call to `PutChar`. If `wr` is not seekable, then `cur(wr)` is always equal to `len(wr)`, since in this case all writing happens at the end.

The difference between `c(wr)` and `target(wr)` reflects the buffering: if `wr` is not buffered, then `target(wr)` is updated to equal `c(wr)` after every operation; if `wr` is buffered, then updates to `target(wr)` can be delayed. For example, in a writer to a file, `target(wr)` is the actual sequence of characters on the disk; in a writer to a terminal, `target(wr)` is the sequence of characters that have actually been transmitted (this sequence may not exist in any data structure, but it still exists in the eye of God).

Every writer is a monitor; that is, it contains an internal lock that is acquired and held for each operation in this interface, so that concurrent operations will appear atomic. For faster, unmonitored access, see the `UnsafeWr` interface (Section 7).

Since there are many classes of writers, there are many ways that a writer can break—for example, the network can go down, the disk can fill up, etc. All problems of this sort are reported by raising the exception `Failure`. Each writer class should specify what failures it can raise and how they are encoded in the argument to `Wr.Failure` (which has type `REFANY`).

Illegal operations (for example, writing to a closed writer) raise the exception `Error`.

Many operations on a writer can wait indefinitely. For example, `PutChar` can wait if the user has suspended output to his terminal. These waits can be alertable, so each procedure that might wait includes `Thread.Alerted` in its raises clause.

The rest of this section is a listing of the `Wr` interface, together with comments specifying the effect of each procedure. It is convenient to define the action `PutC(wr, ch)`, which outputs the character `ch` to the writer `wr`:

```
PutC(wr, ch) =
  IF closed(wr) THEN RAISE Error(Code.Closed) END;
  IF cur(wr) = len(wr) THEN
    "Extend c(wr) by one character, incrementing len(wr)"
  END;
  c(wr)[cur(wr)] := ch;
  INC(cur(wr));
  "Possibly Flush wr"
```

where "`Possibly Flush wr`" specifies a non-deterministic choice between assigning `target(wr) := c(wr)` and doing nothing. `PutC` is only used in specifying the interface; it is not a real procedure.

```
INTERFACE Wr;
FROM Thread IMPORT Alerted;

TYPE
  T <: ROOT;
  Code = {Closed, Unseekable};
EXCEPTION Failure(REFANY); Error(Code);

PROCEDURE PutChar(wr: T; ch: CHAR)
  RAISES {Failure, Alerted, Error};
```

Output `ch` to `wr`. More precisely, this is equivalent to:

```
PutC(wr, ch); IF NOT buffered(wr) THEN Flush(wr) END
```

```
PROCEDURE PutText(wr: T; t: TEXT)
  RAISES {Failure, Alerted, Error};
```

Output `t` to `wr`. More precisely, this is equivalent to:

```
FOR i := 0 TO Text.Length(t) - 1 DO
  PutC(wr, Text.GetChar(t, i))
END;
IF NOT buffered(wr) THEN Flush(wr) END
```

except that, like all operations in this interface, it is atomic with respect to other operations in the interface. (It would be wrong to write `PutChar` instead of `PutC`, since `PutChar` always flushes if the writer is unbuffered.)

```
PROCEDURE PutString(wr: T; a: ARRAY OF CHAR)
  RAISES {Failure, Alerted, Error};
```

Output `a` to `wr`. More precisely, this is equivalent to:

```
FOR i := FIRST(a) TO LAST(a) DO PutC(wr, a[i]) END;
IF NOT buffered(wr) THEN Flush(wr) END
```

except that it is atomic.

```
PROCEDURE Seek(wr: T; n: CARDINAL)
  RAISES {Failure, Alerted, Error};
```

Set the current position of `wr` to `n`. This is a no-op if `wr` is closed. More precisely, this is equivalent to:

```
IF NOT seekable(wr) THEN RAISE Error(Code.Unseekable) END;
cur(wr) := MIN(n, len(wr));
"Possibly Flush wr"
```

```
PROCEDURE Flush(wr: T) RAISES {Failure, Alerted, Error};
```

Perform all buffered operations. That is, set `target(wr) := c(wr)`. This is a no-op if `wr` is closed.

```
PROCEDURE Close(wr: T) RAISES {Failure, Alerted, Error};
```

Flush `wr`, release any resources associated with `wr`, and set `closed(wr) := true`. The documentation for a procedure that creates a writer should specify what resources are released when the writer is closed. This leaves `closed(wr)` equal to `TRUE` even if it raises an exception, and is a no-op if `wr` is closed.

```
PROCEDURE Length(wr: T): CARDINAL
  RAISES {Failure, Alerted, Error};

PROCEDURE GetIndex(wr: T): CARDINAL RAISES {};

PROCEDURE Seekable(wr: T): BOOLEAN RAISES {};

PROCEDURE Closed(wr: T): BOOLEAN RAISES {};

PROCEDURE Buffered(wr: T): BOOLEAN RAISES {};
```

These procedures return `len(wr)`, `cur(wr)`, `seekable(wr)`, `closed(wr)`, and `buffered(wr)`, respectively.

```
END Wr.
```

3 The Rd interface

An `Rd.T` (or "reader") is a character input stream. The basic operation on a reader is `GetChar`, which returns the source character at the "current position" and advances the current position by one. Some readers are "seekable", which means that they also allow setting the current position anywhere in the source. For example, readers from random access files are seekable; readers from terminals and sequential files are not.

Some readers are "intermittent", which means that the source of the reader trickles in rather than being available to the implementation all at once. For example, the input stream from an interactive terminal is intermittent. An intermittent reader is never seekable.

Abstractly, a reader `rd` consists of

`len(rd)`	the number of source characters
`src(rd)`	a sequence of length `len(rd)+1`
`cur(rd)`	an integer in the range `[0..len(rd)]`
`avail(rd)`	an integer in the range `[cur(rd)..len(rd)+1]`
`closed(rd)`	a boolean
`seekable(rd)`	a boolean
`intermittent(rd)`	a boolean

These values are not necessarily directly represented in the data fields of a reader object, but conceptually they determine the state of the reader. In particular, for an intermittent reader, `len(rd)` may be unknown to the implementation, but it still exists in the aforementioned God's-eye view.

The sequence `src(rd)` is zero-based: `src(rd)[i]` is valid for `i` from `0` to `len(rd)`. The first `len(rd)` elements of `src` are the characters that are the source of the reader. The final element is a special value `eof` used to represent end-of-file. The value `eof` is not a character.

The value of `cur(rd)` is the index in `src(rd)` of the next character to be returned by `GetChar`, unless `cur(rd) = len(rd)`, in which case a call to `GetChar` will raise the exception `EndOfFile`.

The value of `avail(rd)` is important for intermittent readers: the elements whose indexes in `src(rd)` are in the range `[cur(rd)..avail(rd)-1]` are available to the implementation and can be read by clients without blocking. If the client tries to read further, the implementation will block waiting for the other characters. If `rd` is not intermittent, then `avail(rd)` is equal to `len(rd)+1`. If `rd` is intermittent, then `avail(rd)` can increase asynchronously, although the procedures in this interface are atomic with respect to such increases.

The definitions above encompass readers with infinite sources. If `rd` is such a reader, then `len(rd)` and `len(rd)+1` are both infinity, and there is no final `eof` value.

Every reader is a monitor; that is, it contains an internal lock that is acquired and held for each operation in this interface, so that concurrent operations will appear atomic. For faster, unmonitored access, see the `UnsafeRd` interface (Section 7).

Since there are many classes of readers, there are many ways that a reader can break—for example, the connection to a terminal can be broken, the disk can signal a read error, etc. All problems of this sort are reported by raising the exception `Failure`. Each reader class should specify what failures it can raise and how they are encoded in the argument to `Failure` (which has type `REFANY`).

Illegal operations raise the exception `Error`.

Many operations on a reader can wait indefinitely. For example, `GetChar` can wait if the user is not typing. In general these waits are alertable, so each procedure that might wait includes `Thread.Alerted` in its `RAISES` clause.

The remainder of this section is a listing of the `Rd` interface, together with comments specifying the effect of each procedure.

```
INTERFACE Rd;
FROM Thread IMPORT Alerted;

TYPE
  T <: ROOT;
  Code =
    {Closed, Unseekable, Intermittent, CantUnget};
EXCEPTION EndOfFile; Failure(REFANY); Error(Code);

PROCEDURE GetChar(rd: T): CHAR
  RAISES {EndOfFile, Failure, Alerted, Error};
```

Return the next character from `rd`. More precisely, this is equivalent to the following, in which `res` is a local variable of type `CHAR`:

```
IF closed(rd) THEN RAISE Error(Code.Closed) END;
Block until avail(rd) > cur(rd);
IF cur(rd) = len(rd) THEN
  RAISE EndOfFile
ELSE
  res := src(rd)[cur(rd)]; INC(cur(rd)); RETURN res
END
```

```
PROCEDURE EOF(rd: T): BOOLEAN RAISES {Failure, Alerted, Error};
```

Return `TRUE` iff `rd` is at end-of-file. More precisely, this is equivalent to:

```
IF closed(rd) THEN RAISE Error(Code.Closed) END;
Block until avail(rd) > cur(rd);
RETURN cur(rd) = len(rd)
```

Notice that on an intermittent reader, `EOF` can block. For example, if there are no characters buffered in a terminal reader, `EOF` must wait to see if the user types the end-of-file escape. If you are using `EOF` in an interactive input loop, the right sequence of operations is:

1. prompt the user
2. call `EOF`, which probably waits on user input
3. presuming that `EOF` returned `FALSE`, read the user's input

```
PROCEDURE UnGetChar(rd: T) RAISES {Error}
```

"Pushes back" the last character read from `rd`, so that the next call to `GetChar` will read it again. More precisely, this is equivalent to the following

```
IF closed(rd) THEN RAISE Error(Code.Closed) END;
IF cur(rd) > 0 THEN DEC(cur(rd)) END
```

except there is a special rule: `UnGetChar(rd)` is guaranteed to work only if `GetChar(rd)` was the last operation on `rd`. Thus `UnGetChar` cannot be called twice in a row, or after `Seek` or `EOF`. If this rule is violated, the implementation is allowed (but not required) to raise `Error(CantUnget)`.

```
PROCEDURE CharsReady(rd: T): CARDINAL RAISES {Failure}
```

Return some number of characters that can be read without indefinite waiting. The "end of file marker" counts as one character for this purpose, so `CharsReady` will return `1`, not `0`, if `EOF(rd)` is true. More precisely, this is equivalent to the following:

```
IF closed(rd) THEN RAISE Error(Code.Closed) END;
IF avail(rd) = cur(rd) THEN
  RETURN 0
ELSE
  RETURN some number in the range [1 .. avail(rd) - cur(rd)]
END;
```

Warning: `CharsReady` can return a result less than `avail(rd) - cur(rd)`; also, more characters might trickle in just as `CharsReady` returns. So the code to flush buffered input without blocking requires a loop:

```
LOOP
  n := Rd.CharsReady(rd);
  IF n = 0 THEN EXIT END;
  FOR i := 1 TO n DO EVAL Rd.GetChar(rd) END
END;
```

```
PROCEDURE GetSub(
    rd: T;
    VAR (*out*) str: ARRAY OF CHAR)
    : CARDINAL
  RAISES {Failure, Alerted, Error};
```

Read from `rd` into `str` until `rd` is exhausted or `str` is filled. More precisely, this is equivalent to the following, in which `i` is a local variable:

```
i := 0;
WHILE NOT EOF(rd) AND i # NUMBER(str) DO
  str[i] := GetChar(rd);
  INC(i)
END;
RETURN i
```

```
PROCEDURE GetSubLine(
    rd: T;
    VAR (*out*) str: ARRAY OF CHAR): CARDINAL
  RAISES {Failure, Alerted, Error};
```

Read from `rd` into `str` until a newline is read, `rd` is exhausted, or `sub` is filled. More precisely, this is equivalent to the following, in which `i` is a local variable:

```
i := 0;
WHILE
  NOT EOF(rd) AND
  i # NUMBER(str) AND
  (i = 0 OR str[i-1] # '\n')
DO
  str[i] := GetChar(rd);
  INC(i)
END;
RETURN i
```

```
PROCEDURE GetText(
    rd: T;
    len: INTEGER)
    : TEXT
  RAISES {Failure, Alerted, Error};
```

Read from `rd` until it is exhausted or `len` characters have been read, and return the result as a `TEXT`. More precisely, this is equivalent to the following, in which `i` and `res` are local variables:

```
res := ""; i := 0;
WHILE NOT EOF(rd) AND i # len DO
  res := res & Text.FromChar(GetChar(rd));
  INC(i)
END;
RETURN res
```

```
PROCEDURE GetLine(rd: T): TEXT
  RAISES {EndOfFile, Failure, Alerted, Error};
```

If `EOF(rd)` then raise `EndOfFile`. Otherwise, read characters until a newline is read or `rd` is exhausted, and return the result as a `TEXT`—but discard the final newline if it is present. More precisely, this is equivalent to the following, in which `ch` and `res` are local variables:

```
IF EOF(rd) THEN RAISE EndOfFile END;
res := ""; ch := '\000'; (* any char but newline *)
WHILE NOT EOF(rd) AND ch # '\n' DO
  ch := GetChar(rd);
  IF ch # '\n' THEN res := res & Text.FromChar(ch) END
END;
RETURN res
```

```
PROCEDURE GetIndex(rd: T): CARDINAL RAISES {};
```

This is equivalent to:

```
IF closed(rd) THEN
  RAISE Error(Code.Closed)
ELSE
  RETURN cur(rd)
END
```

```
PROCEDURE GetLength(rd: T): CARDINAL
  RAISES {Failure, Alerted, Error};
```

This is equivalent to:

```
IF closed(rd) THEN
  RAISE Error(Code.Closed)
ELSIF intermittent(rd) THEN
  RAISE Error(Code.Intermittent)
ELSE
  RETURN len(rd)
END
```

```
PROCEDURE Seek(rd: T; n: CARDINAL) RAISES {Failure, Alerted, Error};
```

This is equivalent to:

```
IF closed(rd) THEN
  RAISE Error(Code.Closed)
ELSIF NOT seekable(rd) THEN
  RAISE Error(Code.Unseekable)
ELSE
  cur(rd) := MIN(n, len(rd))
END
```

```
PROCEDURE Close(rd: T) RAISES {Failure, Alerted};
```

Release any resources associated with `rd` and set `closed(rd) := TRUE`. The documentation of a procedure that creates a reader should specify what resources are released when the reader is closed. This leaves `rd` closed even if it raises an exception, and is a no-op if `rd` is closed.

```
PROCEDURE Intermittent(rd: T): BOOLEAN RAISES {};

PROCEDURE Seekable(rd: T): BOOLEAN RAISES {};

PROCEDURE Closed(rd: T): BOOLEAN RAISES {};
```

Return `intermittent(rd)`, `seekable(rd)`, and `closed(rd)`, respectively.

```
END Rd.
```

4 The Stdio and FileStream interfaces

The interface `Stdio` provides streams for standard input, standard output, and standard error:

```
INTERFACE Stdio;
IMPORT Rd, Wr;

VAR
  stdin: Rd.T;
  stdout: Wr.T;
  stderr: Wr.T;

END Stdio.
```

The initialization of these streams depends on the underlying operating system. If the output streams are directed to terminals, they should be unbuffered, so that explicit `Wr.Flush` calls are unnecessary for interactive programs. If the streams are directed to or from random-access files, they should be seekable. It is possible that `stderr = stdout`; therefore, programs that perform seek operations on `stdout` should take care not to destroy output data when writing error messages.

The `FileStream` interface provides simple routines for opening files. The detailed semantics of the file system vary greatly from system to system, so it is to be expected that this interface will grow in different directions in different systems. But all systems should be able to implement the following very weakly-specified interface, and thereby provide a measure of portability for simple clients.

The interface doesn't specify whether the readers and writers returned by the procedures are seekable or buffered. Probably readers and writers to disk files are seekable and buffered, but in general this depends on the system. Similarly, none of the procedures in the interface have raises clauses, since the errors they can raise are system-dependent.

Closing a file reader or writer closes the underlying file.

```
INTERFACE FileStream;
IMPORT Rd, Wr;

PROCEDURE OpenRead(n: TEXT): Rd.T;
```

Return a reader whose source is the contents of the file named `n`.

```
PROCEDURE OpenWrite(n: TEXT): Wr.T;
```

Return a writer whose target is the contents of the file named `n`. If the file does not exist it will be created; if it does exist it will be truncated to length zero.

```
PROCEDURE OpenAppend(n: TEXT): Wr.T;
```

Return a writer whose target is the contents of the file named `n`. If the file does not exist it will be created; if it does exist then the writer will be positioned to append to the existing contents of the file.

```
END FileStream.
```

5 The WrClass interface

There is no end to the number of useful classes of readers and writers. Here are a few examples from SRC's standard libraries:

- Tee writers, which write copies of their stream to each of two other writers. The name comes from the Unix program "tee", which performs a similar function in the realm of pipes. The most common use is to write to a terminal and to a logfile at the same time.
- Various ways to make new readers from old readers: for example, by concatenation, subsequencing, duplication, and filtering.
- Split writers, which are intended for use by applications that use parallel threads writing to a single writer. Split writers keep the output from each thread separate; this creates the illusion that one thread writes all of its output before the next thread starts writing its output.
- Local pipes, in which a reader is connected to a writer so that its source is the writer's target.
- Formatted writers, in which the client can mark the start and end of logical objects and specify desirable places to break the objects into lines. Formatted writers are basic tools for building pretty printers.

It is beyond the scope of this paper to describe these classes in detail. Instead we will describe the interfaces that allow you to define new classes.

The basic idea is that readers and writers are objects whose method suites are determined by their class. In the most naive version of this idea, a writer class's `putChar` method would determine the effect of `Wr.PutChar` for writers of the class:

```
PutChar(wr, ch) = wr.putChar(ch)
```

The `putChar` method for a terminal writer would send characters to the terminal; while the method for a disk file writer would send characters to the disk, etc.

There are two reasons for rejecting this naive version. The first reason is that it is inefficient to call a method for every `PutChar`. The second and more important reason is that most writers are buffered, and it is undesirable to force every client to reimplement buffering.

We implement `PutChar` and `GetChar` by class-independent code operating on a buffer; class-dependent code is invoked only when the buffer fills up (in the case of a writer) or empties (in the case of a reader).

In this section we define the `WrClass` interface, which reveals the buffer structure in a writer object. New writer classes are created by importing `WrClass` (to gain access to the buffer and the methods) and then defining a subclass of `Wr.T` whose methods provide the new class's behavior. The private fields that are needed by the class-independent code but are irrelevant to the buffer structure are lumped together into the opaque type `Private`.

```
INTERFACE WrClass;
IMPORT Wr;
FROM Thread IMPORT Alerted;
FROM Wr IMPORT Failure, Error;

TYPE
  Private <: ROOT;

REVEAL
  Wr.T = Private BRANDED OBJECT
    buff: REF ARRAY OF CHAR;
    st: CARDINAL; (* index into buff *)
    lo, hi, cur: CARDINAL; (* indexes into c(wr) *)
    closed, seekable, buffered: BOOLEAN
  METHODS
    seek (n: CARDINAL) RAISES {Failure, Alerted, Error};
    length(): CARDINAL
        RAISES {Failure, Alerted, Error} := LengthDefault;
    flush () RAISES {Failure, Alerted, Error} := FlushDefault;
    close () RAISES {Failure, Alerted, Error} := CloseDefault
  END;
```

Let `wr` be a writer, which abstractly is given by `c(wr)`, `target(wr)`, `cur(wr)`, `closed(wr)`, `seekable(wr)`, `buffered(wr)`. The actual representation of `wr` is an object of type `Wr.T`. The `wr.cur`, `wr.closed`, `wr.seekable`, and `wr.buffered` fields in the object represent the corresponding abstract attributes of `wr`. The `wr.buff`, `wr.st`, `wr.lo`, and `wr.hi` fields in the object represent a buffer containing the unflushed part of `c(wr)`. The target of the writer is represented in some class-specific way, which is not specified by this interface.

More precisely, we say that the state of the writer object `wr` is *valid* if the following conditions V1 through V4 hold:

V1. the `cur` field and the booleans are correct:

```
wr.cur = cur(wr) AND
wr.closed = closed(wr) AND
wr.buffered = buffered(wr) AND
wr.seekable = seekable(wr)
```

V2. the indexes of any unflushed characters are in the range `[lo..cur-1]`. That is, for all `i` not in `[wr.lo..wr.cur-1]`,

```
c(wr)[i] = target(wr)[i]
```

V3. the (possibly) unflushed characters are stored in `buff` starting with `buff[st]`. That is, for all `i` in `[wr.lo..wr.cur-1]`,

```
c(wr)[i] = wr.buff[wr.st + i - wr.lo]
```

(Usually `st` is zero. Non-zero values may be useful to satisfy buffer alignment constraints.)

V4. the current position is either contained in the buffer, or just past the buffer:

```
wr.lo <= wr.cur <= wr.hi
```

It is possible that `buff = NIL` in a valid state, since the range of `i`'s in V3 can be empty; for example, in case `lo = cur`.

We say that the state is *ready* if the buffer contains the current position; that is, if

```
    NOT wr.closed
AND wr.buff # NIL
AND wr.lo <= cur(wr) < wr.hi
```

If the state is ready, then `Wr.PutChar` can be implemented by storing into the buffer. The class-independent code in `WrRep` does exactly this, until the buffer is full, at which point it calls a class method to consume the buffer and provide a new one.

In general, the class-independent code modifies `cur` and `buff[i]` for `i` in the range `[st..st+(hi-1)-lo]`, but not the `buff` reference itself, `st`, `lo`, or `hi`. The class-independent code locks the writer before calling any methods; therefore, no two method activations initialized by the class-independent code will be concurrent. A method must not apply operations from the `Wr` interface to the writer itself, or deadlock will result.

Here are the specifications for the methods:

The method call `wr.seek(n)` treats `n` as a position to seek to, and moves the buffer to contain this position. More precisely:

> Given a valid state, `wr.seek(n)` must produce a valid ready state in which `wr.cur = MIN(n, len(wr))` and `c(wr)` is unchanged.

An important special case is when `n = wr.cur = wr.hi`; that is, when the buffer has overflowed and the effect of the seek is simply to advance from the last character of a buffer to the first character of a new buffer. Every writer class (seekable or not) must provide a seek method that supports this special case. The method must support the general case only if the writer is seekable.

The `flush` method updates the underlying target of the writer. That is:

> Given a valid state, `wr.flush()` must produce a valid state in which `c(wr)` and `cur(wr)` are unchanged and `target(wr) = c(wr)`.

If a writer is unbuffered, the class-independent code will call the flush method after every modification to the buffer.

The `close` method releases all resources associated with a writer. That is:

> Given a valid state in which `target(wr) = c(wr)`, the call `wr.close()` must release all resources associated with `wr`.

The exact meaning is class-specific. Validity is not required when the method returns, since after it returns, the class-independent code will set the `closed` bit in the writer, which makes the rest of the state irrelevant, even if it is invalid.

The `length` method returns the length of the writer. That is:

> Given a valid state, `wr.length()` must return `len(wr)`, leaving a valid state in which `c(wr)` and `cur(wr)` are unchanged.

The next two procedures are needed to code class-specific operations.

```
PROCEDURE Lock(wr: Wr.T) RAISES {};
```

> The writer `wr` must be unlocked; lock it and make its state valid.

```
PROCEDURE Unlock(wr: Wr.T);
```

> The writer `wr` must be locked and valid; unlock it and restore the private invariant of the writer implementation.

A class-specific operation on a writer `wr` should use the following template:

```
Lock(wr); TRY ... FINALLY Unlock(wr) END
```

The methods don't have to do this, since the class-independent code automatically locks and unlocks the writer around method calls. The next section provides examples of the use of `Lock` and `Unlock`.

The last declarations in the interface are for the default methods:

```
PROCEDURE LengthDefault(wr: Wr.T): CARDINAL RAISES {};

PROCEDURE CloseDefault(wr: Wr.T) RAISES {};

PROCEDURE FlushDefault(wr: Wr.T) RAISES {};
```

`LengthDefault` returns `wr.cur`, while `CloseDefault` sets `wr.buff` to `NIL` and `FlushDefault` is a no-op.

```
END WrClass.
```

6 Text writers

As an example of a writer class implementation, this section describes a simplified version of text writers.

The target of a text writer is an internal buffer whose contents can be retrieved as a `TEXT`. Retrieving the buffer resets the target to be empty.

Text writers are buffered and unseekable, and never raise `Failure` or `Alerted`. The fact that they are buffered is essentially unobservable, since there is no way for the client to access the target except through the text writer. The interface is:

```
INTERFACE TextWr;
IMPORT Wr;

TYPE T <: Wr.T;

PROCEDURE New(): T;
```

Return a new text writer with `c = "", cur = 0`.

```
PROCEDURE ToText(wr: T): TEXT;
```

Return `c(wr)`, resetting `c(wr)` to `""` and `cur(wr)` to zero.

```
END TextWr.
```

Next we describe a very simple implementation of text writers. A fast implementation would probably import the private representation of the Text interface.

```
MODULE TextWr;
IMPORT Wr, WrClass, Text;
FROM Wr IMPORT Failure;
EXCEPTION FatalError;

REVEAL
  T = Wr.T BRANDED OBJECT text: TEXT END;

CONST BuffSize = 500;
```

A single buffer of the given size is used; each time it fills up, its characters are appended to `text`. That is, the representation invariant for a text writer `wr` is

```
target(wr) = wr.text & SUBARRAY(wr.buff^, 0, wr.cur-wr.lo)
```

Notice that since `wr` is unseekable, `len(wr)` is always equal to `wr.cur`.

```
PROCEDURE New(): T =
  BEGIN
    RETURN
      NEW(T,
        st := 0,
        lo := 0,
        cur := 0,
        hi := BuffSize,
        buff := NEW(REF ARRAY OF CHAR, BuffSize),
        closed := FALSE,
        seekable := FALSE,
        buffered := TRUE,
        seek := Seek,
        close := Close,
        text := "")
  END New;
```

```
PROCEDURE Seek(wr: T; n: CARDINAL) RAISES {Failure} =
  BEGIN
    IF wr.cur # n THEN
      RAISE FatalError (* Bug in WrRep *)
    END;
    wr.text := wr.text &
      Text.FromStr(SUBARRAY(wr.buff^, 0, wr.cur - wr.lo));
    wr.lo := wr.cur;
    wr.hi := wr.lo + NUMBER(wr.buff^)
  END Seek;

PROCEDURE Close(wr: T) RAISES {} =
  BEGIN wr.buff := NIL; wr.text := NIL END Close;


PROCEDURE ToText(wr: T): TEXT =
  VAR
    result: Text.T;
  BEGIN
    WrClass.Lock(wr);
    TRY
      wr.seek(wr.cur);
      result := wr.text;
      wr.text := "";
      wr.cur := 0;
      wr.lo := 0;
      wr.hi := NUMBER(wr.buff^)
    FINALLY
      WrClass.Unlock(wr)
    END;
    RETURN result
  END ToText;

BEGIN END TextWr.
```

7 The unsafe interfaces

The routines in the `UnsafeWr` and `UnsafeRd` interfaces are like the corresponding routines in the `Wr` and `Rd` interfaces, but it is the client's responsibility to lock the stream before calling them. The lock can be acquired once and held for several operations, which is faster than acquiring the lock for each operation, and also makes the whole group atomic. Danger is the price of speed: it is an unchecked runtime error to call one of these operations without locking the stream.

The `UnsafeWr` interface also provides routines for formatted printing of integers and reals. Using them is more efficient but less convenient than using the `Fmt` interface (described in the first edition of the Modula-3 report [3]). For example, the statement

```
Wr.PutText(wr, "Line " & Fmt.Int(n) & " of file " & f)
```

could be replaced with the following faster code:

```
LOCK wr DO
  FastPutText(wr, "Line ");
  FastPutInt (wr, n);
  FastPutText(wr, " of file ");
  FastPutText(wr, f)
END
```

If several threads are writing characters concurrently to the same writer, treating each `PutChar` as an atomic action is likely to produce inscrutable output—it is usually preferable if the units of interleaving are whole lines, or even larger. It is therefore convenient as well as efficient to import `UnsafeWr` and use `LOCK` clauses like the one above to make small groups of output atomic. But don't forget to acquire the lock! If you call one of the routines in this interface without it, then the unsafe code in `WrRep` might crash your program in a rubble of bits. A historical note: the main public interface to Modula-2+ writers used the unsafe, unmonitored routines. Errors were more frequent than expected, mostly because of concurrent calls to `Wr.Flush` or `Wr.Close`, which often occur as implicit finalization actions when the programmer doesn't expect them. In the Modula-3 design we have therefore made the main interfaces safe.

Here is the interface:

```
UNSAFE INTERFACE UnsafeWr;
IMPORT Wr, Thread;
FROM Thread IMPORT Alerted;
FROM Wr IMPORT Failure, Error;
FROM Fmt IMPORT Base, Style;

REVEAL
  Wr.T <: Thread.Mutex;
```

Thus an importer of `UnsafeWr` can write code like `LOCK wr DO ... END`.

```
PROCEDURE FastPutChar(wr: Wr.T; ch: CHAR)
  RAISES {Failure, Alerted, Error};
```

Like `Wr.PutChar`, but `wr` must be locked (as in all routines in the interface).

```
PROCEDURE FastPutText(wr: Wr.T; t: TEXT)
  RAISES {Failure, Alerted, Error};
```

Like `Wr.PutText`.

```
PROCEDURE FastPutString(wr: Wr.T; a: ARRAY OF CHAR)
  RAISES {Failure, Alerted, Error};
```

Like `Wr.PutString`.

```
PROCEDURE FastPutInt(wr: Wr.T; n: INTEGER; base := 10)
  RAISES {Failure, Alerted, Error};
```

Like `Wr.PutText(wr, Fmt.Int(n, base))`.

```
PROCEDURE FastPutReal(
  wr: Wr.T;
  r: REAL;
  precision: CARDINAL := 6;
  style := Style.Mix)
  RAISES {Failure, Alerted, Error};
```

Like `Wr.PutText(wr, Fmt.Real(wr, precision, style))`.

```
PROCEDURE FastPutLongReal(
  wr: Wr.T;
  r: LONGREAL;
  precision: CARDINAL := 6;
  style := Style.Mix)
  RAISES {Failure, Alerted, Error};
```

Like `Wr.PutText(wr, Fmt.LongReal(wr, precision, style))`.

```
END UnsafeWr.
```

The `UnsafeRd` interface is similar, but `GetChar` and `Eof` are the only operations that are sufficiently performance-critical to be included:

```
UNSAFE INTERFACE UnsafeRd;
IMPORT Rd, Thread;
FROM Thread IMPORT Alerted;
FROM Rd IMPORT Failure, Error, EndOfFile;

REVEAL
  Rd.T <: Thread.Mutex;
```

```
PROCEDURE FastGetChar(rd: Rd.T): CHAR
  RAISES {EndOfFile, Failure, Alerted, Error};
```

Like `Rd.GetChar`, but rd must be locked.

```
PROCEDURE FastEOF(rd: Rd.T): BOOLEAN
  RAISES {Failure, Alerted, Error};
```

Like `Rd.EOF`, but rd must be locked.

```
END UnsafeRd.
```

8 The WrRep module

Finally we come to the machine-dependent part of the design: the unsafe modules that make the common operations fast. These modules can be reprogrammed to take advantage of the character manipulation instructions available on a particular machine. The versions of the modules presented here assume that bytes are addressable, and achieve efficiency by doing arithmetic on byte pointers. They also assume that the garbage collector is not relocating, that concurrent assignments to references are atomic, and that character arrays are packed.

```
UNSAFE MODULE WrRep EXPORTS Wr, WrClass, UnsafeWr;
IMPORT Thread, Fmt, Text;
FROM Thread IMPORT Alerted;
EXCEPTION FatalError;

REVEAL
  Private =
    Thread.Mutex BRANDED OBJECT
      next, stop: UNTRACED REF CHAR := NIL;
      buffP: REF ARRAY OF CHAR
    END;
```

Recall that a `Wr.T` was defined in `WrClass` to consist of the `Private` fields followed by the buffer structure. The `Private` fields start with a `Thread.Mutex`, which is as expected, since `UnsafeWr` revealed that `Wr.T` is a subtype of `Thread.Mutex`.

The basic idea is that `wr.next` points at the character of `wr.buff` that will be written by the next call to `PutChar`. The fast path through `FastPutChar` writes this character and advances `wr.next`, until `wr.next` = `wr.stop`, at which point the code takes a slower path:

```
<*INLINE*> PROCEDURE FastPutChar(wr: T; ch: CHAR)
  RAISES {Failure, Alerted, Error} =
```

```
(* wr is clean (see below) and locked. *)
BEGIN
  IF wr.next # wr.stop THEN
    wr.next^:= ch;
    INC(wr.next, ADRSIZE(CHAR))
  ELSE
    SlowPutChar(wr, ch)
  END
END FastPutChar;
(* wr is clean and locked *)
```

Notice that `FastPutChar` does not update `wr.cur`, and therefore does not maintain the validity of `wr`. This saves time, and the correct value for `wr.cur` can be computed from `wr.next` whenever a valid state is required.

We call a writer "clean" if it satisfies the invariant of `FastPutChar`; we will derive the precise definition of this invariant bit by bit. First, since the fast path through `FastPutChar` implements `PutChar` by storing into the buffer and not flushing afterwards, we conclude that a clean writer `wr` must satisfy the following condition:

C1. If `wr.next` $\neq$ `wr.stop`, then `wr` is buffered and ready, and

```
wr.next = ADR(wr.buff[wr.st + cur(wr) - wr.lo]))
```

Notice the use of `cur(wr)` instead of `wr.cur`, since the latter value may be invalid.

A noteworthy consequence of C1 is that in a clean writer, `wr.next` = `NIL` implies `wr.stop` = `NIL`. (If `wr.next` were `NIL` but `wr.stop` were not, then C1 would imply that `NIL` was a buffer address, which is nonsense.) Because both fields default to `NIL`, a newly-allocated writer will satisfy C1. The first call to `FastPutChar` on a new writer will take the slow path, which can set up the pointers so that subsequent calls will be fast.

Next, consider that when the fast path of `FastPutChar` fills the buffer it must preserve C1; therefore it must make `next` = `stop` if it fills the buffer. Thus a clean writer `wr` must satisfy

C2. If `wr.next` $\neq$ `wr.stop`, then

```
wr.stop =
  ADR(wr.buff[wr.st + (wr.hi - 1) - wr.lo]) + ADRSIZE(CHAR)
```

You might think that this equation could be simplified by removing the "`- 1`" from inside the subscript and the "`+ ADRSIZE(CHAR)`" from outside, but this would access a non-existent array element if `stop` points just past the end of `buff`. The fast path through `FastPutChar` maintains C2, since it doesn't affect the consequent, and it can only make the antecedent false.

Next, consider that it must be possible to make a clean writer valid, for example, in order to call its methods. We will do this by updating the `cur` field. It follows that the lagging `cur` field must be the only violation of validity; that is, a clean writer `wr` satisfies

C3. All the validity conditions V1 through V4 defined in `WrClass` hold for `wr`, except that the equation for `wr.cur` in V1 may fail.

Inspection shows that the fast path through `FastPutChar` maintains C3.

To make a clean writer valid we will compute the correct value for `wr.cur` from `wr.next` using the equation in C1. Unfortunately, C1 requires that this equation hold only when `wr.next` $\neq$ `wr.stop`, but we will often need to make a clean writer valid when these pointers are equal; for example, when the buffer fills. We therefore add a condition that says that the equation holds whenever `wr.next` is not `NIL`:

C4. If `wr.next` $\neq$ `NIL`, then

```
(wr.next = ADR(wr.buff[wr.st + cur(wr) - wr.lo])))
```

The fast path through `FastPutChar` maintains C4, since it increments both sides of the equality by one.

Finally, we must deal with the case `wr.next = NIL`, which is the case in a writer that is newly allocated by the runtime system. Such a writer will be valid, since it was given to us by a class implementation, and we have not yet invalidated it by any calls to `FastPutChar`. Thus we conclude that a clean writer `wr` satisfies:

C5. If `wr.next` $=$ `NIL`, then `wr` is valid.

The fast path through `FastPutChar` maintains C5, since it maintains the stronger invariant `wr.next` $\neq$ `NIL`.

We define a writer to be *clean* if it satisfies C1–C5.

Conditions C3, C4, and C5 justify the following procedure for making a clean writer valid:

```
<*INLINE*> PROCEDURE MakeValid(wr: T) =
  (* wr is locked and clean. *)
  BEGIN
    IF wr.next # NIL THEN
      wr.cur :=
        wr.lo + (wr.next - ADR(wr.buff[wr.st])) DIV ADRSIZE(CHAR)
    END
  END MakeValid;
  (* wr is locked, clean, and valid *)
```

The reverse operation, `MakeClean`, sets the `next` and `stop` pointers to produce a clean state. It also returns a boolean indicating whether the writer is ready. Here is its spec:

```
PROCEDURE MakeClean(wr: T): BOOLEAN
```

> Assuming `wr` is valid and locked, set `wr.next` and `wr.stop` to produce a valid clean state; furthermore if `wr` is ready and buffered, make `wr.next` different from `wr.stop`. Return `TRUE` if and only if the state is ready.

`MakeClean` has two uses: to reset the `next` and `stop` pointers after a class method has accessed the buffers, and to reset the pointers from their initial `NIL` values the first time a writer is encountered by this module. (In the second case, `MakeClean` is being applied to a writer that is already clean, in spite of its name.)

Before listing the implementation of `MakeClean`, we will see how it is used in the code for `SlowPutChar`, which is a long but straightforward case analysis, as is usual for the slow path that takes care of all the cases that are ignored in the fast path:

```
PROCEDURE SlowPutChar(wr: T; ch: CHAR)
  RAISES {Failure, Alerted, Error} =
  (* wr is clean and locked; wr.next = wr.stop. *)
  BEGIN
    IF wr.closed THEN
      RAISE Error(Code.Closed)
    END;
    (* First goal is to make wr valid *)
    IF wr.next # NIL THEN
      MakeValid(wr)
    ELSE
      (* wr is already valid; but might be newly allocated. *)
      EVAL MakeClean(wr)
      (* wr is valid and clean, and if wr is ready
        and buffered, then wr.next is non-NIL *)
    END;
    (* wr is valid and clean *)
    IF wr.cur = wr.hi THEN
      (* wr is valid, clean, and full *)
      wr.seek(wr.cur);
      (* wr is valid and ready *)
      IF NOT MakeClean(wr) THEN
        RAISE FatalError (* Seek method erred *)
      END
      (* wr is valid, clean, and ready *)
    END;
    (* wr is valid, clean, and ready *)
    IF wr.next # wr.stop THEN
      wr.next^ := ch;
      INC(wr.next, ADRSIZE(CHAR))
```

```
    ELSE
      (* wr is unbuffered *)
      wr.buff[wr.st + wr.cur - wr.lo] := ch;
      INC(wr.cur);
      wr.flush()
    END
  END SlowPutChar;
```

Here is the implementation of `MakeClean`, which is short but tricky:

```
PROCEDURE MakeClean(wr: T): BOOLEAN =
  BEGIN
    wr.buffP := wr.buff;
    IF (wr.lo <= wr.cur) AND (wr.cur < wr.hi)
        AND (wr.buffP # NIL) AND (NOT wr.closed)
    THEN
      (* wr is ready *)
      wr.next := ADR(wr.buffP[wr.st + wr.cur - wr.lo]);
      wr.stop :=
        ADR(wr.buffP[wr.st + wr.hi - wr.lo - 1]) + ADRSIZE(CHAR);
      IF wr.stop < wr.next THEN
        RAISE FatalError (* Who changes wr without the lock? *)
      END;
      IF NOT wr.buffered THEN wr.stop := wr.next END;
      RETURN TRUE
    ELSE
      (* wr is not ready *)
      wr.stop := NIL;
      wr.next := NIL;
      RETURN FALSE
    END
  END MakeClean;
```

The language requires that this procedure avoid unchecked runtime errors even if a buggy class implementation is modifying the writer without holding the lock. The unsafe operations in this module are the computations of `wr.next` and `wr.stop`, together with the increment to `wr.next`. The danger is that errors in the address arithmetic could make `wr.next` point somewhere outside of `wr.buff`, causing `PutChar` to spray characters randomly into memory. To prevent this, it suffices to ensure that these two pointers both point into the array `wr.buff^` (or immediately after the array) and that they are in the proper order. `MakeClean` guarantees this, since

1. After copying `wr.buff` into `wr.buffP`, it uses `wr.buffP` for the rest of the computation, so it won't matter if `wr.buff` changes concurrently. (Recall that we are assuming that reads and writes of references are atomic.)

2. In the computation of `wr.next` and `wr.stop`, the subscripts into `wr.buffP` will be checked, and a runtime error will occur if they are out of range, even if `wr.st`, `wr.cur`, `wr.hi`, and `wr.lo` are changing concurrently.

3. The program checks that `wr.next` precedes `wr.stop` after computing them.

4. The program maintains `wr.buffP` equal to `wr.buff`, which guarantees that `wr.buff^` will not be collected, even if a buggy class implementation changes `wr.buff` without locking the writer.

All of this may seem like paranoia, but the rule is that a module exporting a safe interface must guarantee that *no programing error* by a safe client of that interface can lead to an unchecked runtime error. Changing the buffer structure without locking the writer is a possible programming error by a client of `WrClass`. We therefore must program `WrRep` in such a way that this error cannot lead to an unchecked runtime error. Otherwise we would have to add the word "UNSAFE" to the `WrClass` interface.

A client of `UnsafeWr` could call `FastPutChar` concurrently from two threads, which could advance `next` past `stop` and clobber memory. We have no defense against this, which is why `UnsafeWr` is unsafe.

The remainder of the program is straightforward:

```
PROCEDURE Lock(wr: T) =
  BEGIN
    Thread.Acquire(wr);
    MakeValid(wr)
  END Lock;

PROCEDURE Unlock(wr: T) =
  BEGIN
    EVAL MakeClean(wr);
    Thread.Release(wr)
  END Unlock;

<*INLINE*> PROCEDURE PutChar(wr: T; ch: CHAR)
  RAISES {Failure, Alerted, Error} =
    (* wr must be unlocked. *)
  BEGIN
    LOCK wr DO FastPutChar(wr, ch) END
  END PutChar;
```

We won't present the code for the procedures `PutText`, `PutString`, `FastPutText`, `FastPutString`, `FastPutInt`, `FastPutReal`, or `FastPutLongReal`, since they don't illustrate any interesting new points.

```
PROCEDURE Seek(wr: T; n: CARDINAL) RAISES {Failure, Alerted} =
  BEGIN
    LOCK wr DO
      IF NOT wr.seekable THEN
        RAISE Error(Code.Unseekable)
      END;
      MakeValid(wr);
      TRY wr.seek(n) FINALLY EVAL MakeClean(wr) END
    END
  END Seek;

PROCEDURE GetIndex(wr: T): CARDINAL RAISES {} =
  BEGIN LOCK wr DO MakeValid(wr); RETURN wr.cur END END GetIndex;

PROCEDURE Length(wr: T): CARDINAL RAISES {Failure, Alerted} =
  BEGIN
    LOCK wr DO
      MakeValid(wr);
      TRY RETURN wr.length() FINALLY EVAL MakeClean(wr) END
    END
  END Length;

PROCEDURE Flush(wr: T) RAISES {Failure, Alerted} =
  BEGIN
    LOCK wr DO
      MakeValid(wr);
      TRY wr.flush() FINALLY EVAL MakeClean(wr) END
    END
  END Flush;

PROCEDURE Close(wr: T) RAISES {Failure, Alerted} =
  BEGIN
    LOCK wr DO
      IF NOT wr.closed THEN
        MakeValid(wr);
        TRY
          wr.flush();
          wr.close()
        FINALLY
          wr.closed := TRUE;
          wr.next := wr.stop;
          wr.buffP := NIL
```

```
        END
      END
    END
  END Close;

PROCEDURE Seekable(wr: T): BOOLEAN RAISES {} =
  BEGIN
    LOCK wr DO RETURN wr.seekable END
  END Seekable;

PROCEDURE Closed(wr: T): BOOLEAN RAISES {} =
  BEGIN
    LOCK wr DO RETURN wr.closed END
  END Closed;

PROCEDURE Buffered(wr: T): BOOLEAN RAISES {} =
  BEGIN
    LOCK wr DO RETURN wr.buffered END
  END Buffered;

PROCEDURE CloseDefault(wr: T) RAISES {} =
  BEGIN wr.buff := NIL END CloseDefault;

PROCEDURE FlushDefault(wr: T) RAISES {} =
  BEGIN END FlushDefault;

PROCEDURE LengthDefault(wr: T): CARDINAL RAISES {} =
  BEGIN RETURN wr.cur END LengthDefault;

BEGIN END WrRep.
```

The reader may feel that our uncompromising pursuit of safety and efficiency has led to a design that is too complex. The program would be much simpler if `WrRep` kept the writer valid at all times, and the cost would be only a few instructions per operation. The point is that our design allows a range of implementations of `WrRep`. We have presented one that illustrates the issues that arise at the boundary between safe and unsafe code. Substituting a simpler `WrRep` would not affect clients of `Wr` or of `WrClass`.

9 The RdClass interface

The `RdClass` interface is analogous to the `WrClass` interface. It reveals that every reader contains a buffer of characters together with methods for managing the buffer. New reader classes are created by importing `RdClass` (to gain access to the buffer and the methods) and then defining a subclass of `Rd.T` whose methods provide the new class's behavior. The opaque type `Private` hides irrelevant details of the class-independent code.

```
INTERFACE RdClass;
IMPORT Rd;
FROM Thread IMPORT Alerted;
FROM Rd IMPORT Failure, Error;

TYPE
  Private <: ROOT;
  SeekResult = {Ready, WouldBlock, Eof};

REVEAL
  Rd.T = Private BRANDED OBJECT
    buff: REF ARRAY OF CHAR;
    st: CARDINAL; (* index into buff *)
    lo, hi, cur: CARDINAL; (* indexes into src(rd) *)
    closed, seekable, intermittent: BOOLEAN;
  METHODS
    seek(dontBlock: BOOLEAN): SeekResult
      RAISES {Failure, Alerted, Error};
    length(): CARDINAL RAISES {Failure, Alerted, Error}
      := LengthDefault;
    close() RAISES {Failure, Alerted, Error}
      := CloseDefault;
  END;
```

Let `rd` be a reader, abstractly given by `len(rd)`, `src(rd)`, `cur(rd)`, `avail(rd)`, `closed(rd)`, `seekable(rd)`, and `intermittent(rd)`. The data fields `cur`, `closed`, `seekable`, and `intermittent` in the object represent the corresponding abstract attributes of `rd`. The `buff`, `st`, `lo`, and `hi` fields represent a buffer that contains part of `src(rd)`, the rest of which is represented in some class-specific way.

More precisely, we say that a reader `rd` is *valid* if V1 through V3 hold:

V1. the characters of `buff` starting with `st` accurately reflect `src`. That is, for all `i` in `[rd.lo .. rd.hi-1]`,

```
rd.buff[rd.st + i - rd.lo] = src(rd)[i]
```

V2. if the `cur` field is in range, it is up-to-date:

```
cur(rd) = MIN(rd.cur, len(rd))
```

(This equation implies that `rd.cur > len(rd)` has exactly the same meaning as `rd.cur = len(rd)`. This convention allows the implementation to use "lazy seeking"; that is, `Rd.Seek` can simply update `rd.cur`, without calling any class methods.)

V3. the reader does not claim to be both intermittent and seekable:

```
NOT (rd.intermittent AND rd.seekable)
```

It is possible that `buff = NIL` in a valid state, since the range of `i`'s in V1 may be empty; for example, in case `lo = hi`.

There is no requirement that `cur(rd)` be anywhere near `rd.lo` or `rd.hi` in a valid state. If in fact `cur(rd)` lies between these values, we say the reader is *ready*. More precisely, `rd` is ready if:

```
NOT rd.closed AND
rd.buff # NIL AND
rd.lo <= rd.cur < rd.hi
```

If the state is ready, then `Rd.GetChar` can be implemented by fetching from the buffer.

The class-independent code modifies `rd.cur`, but no other variables revealed in this interface. The class-independent code locks the reader before calling any methods.

Here are the specifications for the methods:

The basic purpose of the `seek` method is to make the reader ready. To seek to a position `n`, the class-independent code sets `rd.cur := n`; then if it is necessary to make the reader ready, it calls `rd.seek`. As in the case of writers, the seek method can be called even for an unseekable reader in the special case of advancing to the next buffer.

There is a wrinkle to support the implementation of `CharsReady`. If `rd` is ready, the class-independent code can handle the call to `CharsReady(rd)` without calling any methods (since there is at least one character ready in the buffer), but if `rd.cur = rd.hi`, then the class independent code needs to find out from the class implementation whether any characters are ready in the next buffer. Using the `seek` method to advance to the next buffer won't do, since this could block, and `CharsReady` isn't supposed to block. Therefore, the `seek` method takes a boolean argument saying whether blocking is allowed. If blocking is forbidden and the next buffer isn't ready, the method returns the special value `WouldBlock`; this allows the class-independent code to return zero from `CharsReady`.

More precisely,

> Given a valid state with `rd.seekable` or `rd.cur = rd.hi`, the effect of the call `res := rd.seek(dontBlock)` is to leave `rd` valid without changing

> the abstract state of `rd`. Furthermore, if `res = Ready` then `rd` is ready and `cur(rd) = rd.cur`; while if `res = Eof`, then `cur(rd) = rd.cur = len(rd)`; and finally if `res = WouldBlock` then `dontBlock` was `TRUE` and `avail(rd) = cur(rd)`.

The `length` method returns the length of a non-intermittent reader. That is:

> Given a valid state in which `rd.intermittent` is `FALSE`, the call `rd.length()` returns `len(rd)` without changing the state of `rd`.

The `close` method releases all resources associated with `rd`. The exact meaning of this is class-specific. When the method is called the state will be valid; validity is not required when the method returns (since after it returns, the class-independent code will set the closed bit in the reader, which makes the rest of the state irrelevant).

The remainder of the interface is similar to the corresponding part of the `WrClass` interface:

```
PROCEDURE Lock(rd: Rd.T) RAISES {};
```

> The reader `rd` must be unlocked; lock it and make its state valid.

```
PROCEDURE Unlock(rd: Rd.T) RAISES {};
```

> The reader `rd` must be locked and valid; unlock it and restore the private invariant of the reader implementation.

```
PROCEDURE LengthDefault(rd: Rd.T): CARDINAL
  RAISES {Failure, Alerted, Error};

PROCEDURE CloseDefault(rd: Rd.T) RAISES
  {Failure, Alerted, Error};
```

> The procedure `LengthDefault` causes a checked runtime error, representing the failure to supply a length method for a non-intermittent reader. The procedure `CloseDefault` sets `rd.buff` to `NIL`.

```
END RdClass.
```

10 The RdRep module

This module is very similar to the `WrRep` module, so we will list its code with only a few comments. We omit the straightforward implementations of the procedures `GetSub`, `GetSubLine`, `GetText`, `GetLine`, `Intermittent`, `Seekable`, and `Closed` from the `Rd` interface, and of all the procedures in the `RdClass` interface.

```
UNSAFE MODULE RdRep EXPORTS Rd, RdClass, UnsafeRd;
IMPORT Thread, Text;
FROM Thread IMPORT Alerted;
EXCEPTION FatalError;

REVEAL
  Private =
    Thread.Mutex BRANDED OBJECT
      next, stop: UNTRACED REF CHAR := NIL;
      buffP: REF ARRAY OF CHAR;
  END;
```

The implementation of `Rd.Seek` is lazy. When a client calls `Rd.Seek` with an index that does not lie within the buffer, `Rd.Seek` simply records the destination index in `rd.cur` and sets both `rd.next` and `rd.stop` to `NIL`. When `rd.next` $\neq$ `NIL`, the `Rd` implementation ignores the value of `rd.cur` in determining `cur(rd)`, but when `rd.next` = `NIL` the `Rd` implementation uses the value of `rd.cur`.

A reader `rd` is "clean" if the following conditions hold (see the `WrRep` module for more explanation):

C1. If `rd.next # rd.stop`, then

```
Ready(rd) AND
  (rd.next = ADR(rd.buff[rd.st + cur(rd) - rd.lo])
```

C2. If `rd.next # rd.stop`, then

```
rd.stop =
  ADR(rd.buff[rd.st + (rd.hi - 1) - rd.lo]) + ADRSIZE(CHAR)
```

C3. The validity conditions V1 and V3 hold for rd.

C4. If `rd.next # NIL` then

```
(rd.next = ADR(rd.buff[rd.st + cur(rd) - rd.lo])
```

C5. If `rd.next = NIL`, then rd is valid.

```
<*INLINE*> PROCEDURE MakeValid(rd: T) =
  (* rd locked and clean *)
  BEGIN
    IF rd.next # NIL THEN
      rd.cur :=
        rd.lo + (rd.next - ADR(rd.buff[rd.st])) DIV ADRSIZE(CHAR)
    END
  END MakeValid;
  (* rd is locked, clean, and valid.  Furthermore, if rd.next#NIL,
  then rd.cur=cur(rd); this is important for the implementation
  of GetIndex. *)


PROCEDURE MakeClean(rd: T) =
  BEGIN
    rd.buffP := rd.buff;
    IF (rd.lo <= rd.cur) AND
       (rd.cur < rd.hi)  AND
       (rd.buffP # NIL)
    THEN
      rd.next := ADR(rd.buffP[rd.st + rd.cur - rd.lo]);
      rd.stop :=
        ADR(rd.buffP[rd.st + rd.hi - rd.lo - 1]) + ADRSIZE(CHAR);
      IF rd.stop < rd.next THEN
        RAISE FatalError (* Who's changing rd without the lock? *)
      END
    ELSE
      rd.stop := NIL;
      rd.next := NIL
    END
  END MakeClean;


PROCEDURE SlowGetChar(rd: T): CHAR
    RAISES {EndOfFile, Failure, Alerted, Error} =
    (* rd is locked and clean; rd.next = rd.stop *)
  VAR res: CHAR;
  BEGIN
    IF rd.closed THEN RAISE Error(Code.Closed) END;
    TRY
      MakeValid(rd);
      IF rd.seek(dontBlock := FALSE) = SeekResult.Eof THEN
        RAISE EndOfFile
      END
```

```
    FINALLY
      MakeClean(rd)
    END;
    IF rd.next = rd.stop THEN
      RAISE FatalError (* Seek method didn't make reader ready *)
    END;
    res := rd.next^;
    INC(rd.next, ADRSIZE(CHAR));
    RETURN res
  END SlowGetChar;

<*INLINE*> PROCEDURE GetChar(rd: T): CHAR
    RAISES {EndOfFile, Failure, Alerted, Error} =
    (* rd is unlocked *)
  BEGIN
    LOCK rd DO RETURN FastGetChar(rd) END
  END GetChar;


<*INLINE*> PROCEDURE FastGetChar(rd: T): CHAR
    RAISES {EndOfFile, Failure, Alerted, Error} =
    (* rd is locked *)
  VAR res: CHAR;
  BEGIN
    IF rd.next # rd.stop THEN
      res := rd.next^;
      INC(rd.next, ADRSIZE(CHAR))
    ELSE
      res := SlowGetChar(rd)
    END;
    RETURN res
  END FastGetChar;

<*INLINE*> PROCEDURE EOF(rd: T): BOOLEAN
    RAISES {Failure, Alerted, Error} =
    (* rd is unlocked *)
  BEGIN
    LOCK rd DO RETURN FastEOF(rd) END
  END EOF;
```

```
<*INLINE*> PROCEDURE FastEOF(rd: T): BOOLEAN
    RAISES {Failure, Alerted, Error} =
    (* rd is locked *)
  BEGIN
    IF rd.next # rd.stop THEN RETURN FALSE
    ELSE RETURN SlowEOF(rd)
    END
  END FastEOF;

PROCEDURE SlowEOF(rd: T): BOOLEAN RAISES {Failure, Alerted} =
  (* rd is locked; rd.next = rd.stop *)
  VAR res: CHAR;
  BEGIN
    IF rd.closed THEN
      RAISE Error(Code.Closed)
    ELSE
      MakeValid(rd);
      TRY
        RETURN rd.seek(dontBlock := FALSE) = SeekResult.Eof
      FINALLY
        MakeClean(rd)
      END
    END
  END SlowEOF;

PROCEDURE UnGetChar(rd: T) RAISES {Error} =
  BEGIN
    LOCK rd DO
      IF rd.closed THEN RAISE Error(Code.Closed) END;
      IF (rd.next = NIL) OR (rd.next = ADR(rd.buff[rd.st])) THEN
        RAISE Error(Code.CantUnget)
      END;
      DEC(rd.next)
    END
  END UnGetChar;

PROCEDURE CharsReady(rd: T): CARDINAL
RAISES {Failure, Alerted, Error} =
  BEGIN
    LOCK rd DO
      IF rd.closed THEN RAISE Error(Code.Closed) END;
      MakeValid(rd);
```

```
      IF NOT (rd.lo <= rd.cur AND rd.cur < rd.hi) THEN
        TRY
          IF rd.seek(dontBlock := TRUE) = SeekResult.Eof
            THEN RETURN 1
          END
        FINALLY
          MakeClean(rd)
        END;
        IF rd.cur > rd.hi THEN
          RAISE FatalError (* Seek method erred *)
        END
      END;
      RETURN rd.hi - rd.cur
    END
  END CharsReady;

PROCEDURE GetIndex(rd: T): CARDINAL
  RAISES {Failure, Alerted, Error} =
  BEGIN
    LOCK rd DO
      IF rd.closed THEN RAISE Error(Code.Closed) END;
      MakeValid(rd);
      IF rd.seekable AND (rd.next = NIL) THEN
        rd.cur := MIN(rd.cur, rd.length())
      END;
      RETURN rd.cur
    END
  END GetIndex;


PROCEDURE GetLength(rd: T): CARDINAL
  RAISES {Failure, Alerted, Error} =
  BEGIN
    LOCK rd DO
      IF rd.closed THEN
        RAISE Error(Code.Closed)
      ELSIF rd.intermittent THEN
        RAISE Error(Code.Intermittent)
      ELSE
        TRY
          MakeValid(rd);
          RETURN rd.length()
        FINALLY
          MakeClean(rd)
```

```
        END
      END
    END
  END GetLength;

PROCEDURE Seek(rd: T; n: CARDINAL)
  RAISES {Failure, Alerted, Error} =
  BEGIN
    LOCK rd DO
      IF rd.closed THEN
        RAISE Error(Code.Closed)
      ELSIF NOT rd.seekable THEN
        RAISE Error(Code.Unseekable)
      ELSE
        rd.cur := n;
        MakeClean(rd)
      END
    END
  END Seek;

PROCEDURE Close(rd: T) RAISES {Failure, Alerted, Error} =
  BEGIN
    LOCK rd DO
      IF NOT rd.closed THEN
        TRY
          MakeValid(rd);
          rd.close()
        FINALLY
          rd.closed := TRUE;
          rd.next := NIL;
          rd.stop := NIL;
          rd.buffP := NIL
        END
      END
    END
  END Close;

BEGIN END RdRep.
```

11 Concluding remarks

We have heard programmers say "there is no way to give a formal specification for an object-oriented interface, since the different subclass methods can do different things". We hope this paper presents a less superficial view. To give a formal specification for an object-oriented interface, the key is to distinguish the abstraction represented by an object from the object itself, as we have distinguished `cur(wr)` from `wr.cur`, for example. The implementer of a subclass has considerable freedom to "instantiate" the abstraction (for example, by choosing the target of a class of writers), but no additional freedom to change the meaning of the operations, which are defined once and for all in terms of the abstraction. Admittedly there may be operations (like `Close`) that leave considerable freedom to the class implementer, but if all the operations are like this, the abstraction is not likely to be very useful.

Our treatment has been "formal" only in a very pragmatic way. The stream interfaces would surely benefit from being translated into Larch [4], or some equally formal specification language. Nevertheless, we feel that many programs being written today could be improved by a dose of specification of the pragmatic sort illustrated by this paper.

It is interesting that the traditional technique of program verification via invariants was most useful in the lowest level of the system. The desire to optimize the fast path introduced a case analysis into the slow path, which was best managed by carefully writing invariants. The pattern of reasoning we used applies in many similar situations: we began by coding the fast path based on efficiency considerations; from this code we derived the global cleanliness invariant; from this we derived the case analysis on the slow path.

Specifying the interfaces was harder than coding the implementation. We used interfaces in layers to hide dangerous information from safe clients, while revealing it to unsafe clients. There are many views of a `Wr.T`: a client of `Wr` sees a pure opaque type, a client of `WrClass` sees only the buffer structure, a client of `UnsafeWr` sees the mutex, and the implementation sees everything. A client that defines a new class sees the class fields and the buffer fields, but not the mutex or the private fields.

To achieve this pattern of information-hiding without partially opaque object types, it would be necessary to allocate each group of fields separately and link them together with additional references. This would require several allocations per writer, which would be costly. Partial opacity makes it possible to achieve this information-hiding with essentially no runtime penalty. In our design, creating a writer requires allocating a single ten-word object (assuming one-word mutexes, references, and integers). The method suite does not have to be allocated dynamically, since its contents are known at compile time, and different instances of a class all point at the same statically allocated method suite.

The least methodical part of the design is the delicate code required to export a safe interface from an unsafe module. Writing this code is a little like writing a secure operating system without any help from the virtual memory system. At present, will power seems more useful than methodology for avoiding errors in this kind of code. We hope we managed to illustrate the pitfalls without falling into any.

Acknowledgments

Kenneth Brooks, Butler Lampson, and Kai Li contributed to the development of the Modula-2+ streams package. Eric Muller implemented the basic stream classes for the Modula-3 runtime.

Bibliography

[1] Graham M. Birtwistle, Ole-Johan Dahl, Bjorn Myhrhaug, and Kristen Nygaard. *Simula Begin*. Auerbach, Philadelphia, PA, 1973.

[2] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. Modula-3 Report (revised). Research Report 52, Digital Systems Research Center, November 1989.

[3] Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson. Modula-3 Report. Research Report 31, Digital Systems Research Center, August 1988.

[4] J. V. Guttag, J. J. Horning, and J. M. Wing. Larch in Five Easy Pieces. Research Report 5, Digital Systems Research Center, July 1985.

[5] J. E. Stoy and C. Strachey. OS6—an experimental operating system for a small computer. Part 2: input/output and filing system. *The Computer Journal*, 15(3), 1972.

Index

SRC Reports

"A Kernel Language for Modules and Abstract Data Types."
R. Burstall and B. Lampson.
Research Report 1, September 1, 1984.

"Optimal Point Location in a Monotone Subdivision."
Herbert Edelsbrunner, Leo J. Guibas, and Jorge Stolfi.
Research Report 2, October 25, 1984.

"On Extending Modula-2 for Building Large, Integrated Systems."
Paul Rovner, Roy Levin, John Wick.
Research Report 3, January 11, 1985.

"Eliminating **go to**'s while Preserving Program Structure."
Lyle Ramshaw.
Research Report 4, July 15, 1985.

"Larch in Five Easy Pieces."
J. V. Guttag, J. J. Horning, and J. M. Wing.
Research Report 5, July 24, 1985.

"A Caching File System for a Programmer's Workstation."
Michael D. Schroeder, David K. Gifford, and Roger M. Needham.
Research Report 6, October 19, 1985.

"A Fast Mutual Exclusion Algorithm."
Leslie Lamport.
Research Report 7, November 14, 1985.
Revised October 31, 1986.

"On Interprocess Communication."
Leslie Lamport.
Research Report 8, December 25, 1985.

"Topologically Sweeping an Arrangement."
Herbert Edelsbrunner and Leonidas J. Guibas.
Research Report 9, April 1, 1986.

"A Polymorphic λ-calculus with Type:Type."
Luca Cardelli.
Research Report 10, May 1, 1986.

"Control Predicates Are Better Than Dummy Variables For Reasoning About Program Control."
Leslie Lamport.
Research Report 11, May 5, 1986.

"Fractional Cascading."
Bernard Chazelle and Leonidas J. Guibas.
Research Report 12, June 23, 1986.

"Retiming Synchronous Circuitry."
Charles E. Leiserson and James B. Saxe.
Research Report 13, August 20, 1986.

"An $O(n^2)$ Shortest Path Algorithm for a Non-Rotating Convex Body."
John Hershberger and Leonidas J. Guibas.
Research Report 14, November 27, 1986.

"A Simple Approach to Specifying Concurrent Systems."
Leslie Lamport.
Research Report 15, December 25, 1986.
Revised January 26, 1988

"A Generalization of Dijkstra's Calculus."
Greg Nelson.
Research Report 16, April 2, 1987.

"*win* and *sin*: Predicate Transformers for Concurrency."
Leslie Lamport.
Research Report 17, May 1, 1987.
Revised September 16, 1988.

"Synchronizing Time Servers."
Leslie Lamport.
Research Report 18, June 1, 1987.
Temporarily withdrawn to be rewritten.

"Blossoming: A Connect-the-Dots Approach to Splines."
Lyle Ramshaw.
Research Report 19, June 21, 1987.

"Synchronization Primitives for a Multiprocessor: A Formal Specification."
A. D. Birrell, J. V. Guttag, J. J. Horning, R. Levin.
Research Report 20, August 20, 1987.

"Evolving the UNIX System Interface to Support Multithreaded Programs."
Paul R. McJones and Garret F. Swart.
Research Report 21, September 28, 1987.

"Building User Interfaces by Direct Manipulation."
Luca Cardelli.
Research Report 22, October 2, 1987.

"Firefly: A Multiprocessor Workstation."
C. P. Thacker, L. C. Stewart, and E. H. Satterthwaite, Jr.
Research Report 23, December 30, 1987.

"A Simple and Efficient Implementation for Small Databases."
Andrew D. Birrell, Michael B. Jones, and Edward P. Wobber.
Research Report 24, January 30, 1988.

"Real-time Concurrent Collection on Stock Multiprocessors."
John R. Ellis, Kai Li, and Andrew W. Appel.
Research Report 25, February 14, 1988.

"Parallel Compilation on a Tightly Coupled Multiprocessor."
Mark Thierry Vandevoorde.
Research Report 26, March 1, 1988.

"Concurrent Reading and Writing of Clocks."
Leslie Lamport.
Research Report 27, April 1, 1988.

"A Theorem on Atomicity in Distributed Algorithms."
Leslie Lamport.
Research Report 28, May 1, 1988.

"The Existence of Refinement Mappings."
Martín Abadi and Leslie Lamport.
Research Report 29, August 14, 1988.

"The Power of Temporal Proofs."
Martín Abadi.
Research Report 30, August 15, 1988.

"Modula-3 Report."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 31, August 25, 1988.
This report has been superseded by Research Report 52.

"Bounds on the Cover Time."
Andrei Broder and Anna Karlin.
Research Report 32, October 15, 1988.

"A Two-view Document Editor with User-definable Document Structure."
Kenneth Brooks.
Research Report 33, November 1, 1988.

"Blossoms are Polar Forms."
Lyle Ramshaw.
Research Report 34, January 2, 1989.

"An Introduction to Programming with Threads."
Andrew Birrell.
Research Report 35, January 6, 1989.

"Primitives for Computational Geometry."
Jorge Stolfi.
Research Report 36, January 27, 1989.

"Ruler, Compass, and Computer: The Design and Analysis of Geometric Algorithms."
Leonidas J. Guibas and Jorge Stolfi.
Research Report 37, February 14, 1989.

"Can fair choice be added to Dijkstra's calculus?"
Manfred Broy and Greg Nelson.
Research Report 38, February 16, 1989.

"A Logic of Authentication."
Michael Burrows, Martín Abadi, and Roger Needham.
Research Report 39, February 28, 1989.

"Implementing Exceptions in C."
Eric S. Roberts.
Research Report 40, March 21, 1989.

"Evaluating the Performance of Software Cache Coherence."
Susan Owicki and Anant Agarwal.
Research Report 41, March 31, 1989.

"WorkCrews: An Abstraction for Controlling Parallelism."
Eric S. Roberts and Mark T. Vandevoorde.
Research Report 42, April 2, 1989.

"Performance of Firefly RPC."
Michael D. Schroeder and Michael Burrows.
Research Report 43, April 15, 1989.

"Pretending Atomicity."
Leslie Lamport and Fred B. Schneider.
Research Report 44, May 1, 1989.

"Typeful Programming."
Luca Cardelli.
Research Report 45, May 24, 1989.

"An Algorithm for Data Replication."
Timothy Mann, Andy Hisgen, and Garret Swart.
Research Report 46, June 1, 1989.

"Dynamic Typing in a Statically Typed Language."
Martín Abadi, Luca Cardelli, Benjamin C. Pierce, and Gordon D. Plotkin.
Research Report 47, June 10, 1989.

"Operations on Records."
Luca Cardelli and John C. Mitchell.
Research Report 48, August 25, 1989.

"The Part-Time Parliament."
Leslie Lamport.
Research Report 49, September 1, 1989.

"An Efficient Algorithm for Finding the CSG Representation of a Simple Polygon."
David Dobkin, Leonidas Guibas, John Hershberger, and Jack Snoeyink.
Research Report 50a, September 10, 1989.

"Boolean Formulæ for Simple Polygons" (video).
John Hershberger and Marc H. Brown.
Research Report 50b, September 10, 1989.

"Experience with the Firefly Multiprocessor Workstation."
Susan Owicki.
Research Report 51, September 15, 1989.

"Modula-3 Report (revised)."
Luca Cardelli, James Donahue, Lucille Glassman, Mick Jordan, Bill Kalsow, Greg Nelson.
Research Report 52, November 1, 1989.

Mark R. Brown and Greg Nelson

Systems Research Center
130 Lytton Avenue
Palo Alto, California 94301