SOFTWARE VERIFICATION RESEARCH CENTRE

DEPARTMENT OF COMPUTER SCIENCE

THE UNIVERSITY OF QUEENSLAND

Queensland 4072
Australia

TECHNICAL REPORT

No. 95-10

The datalogger case study in Care

Peter A. Lindsay

June 1995

Phone: +61 7 365 1003
Fax: +61 7 365 1533

Note: Most SVRC technical reports are available via anonymous ftp, from `ftp.cs.uq.edu.au` in the directory `/pub/SVRC/techreports`.

The Data Logger case study in CARE

Peter A. Lindsay*
Software Verification Research Centre
email: pal@cs.uq.edu.au

Abstract

This paper presents an extended case study in the use of the CARE language for formally verified software development. The case study concerns storage of variable-length records into a fixed-size memory space. The problem is specified, and a design given, using the CARE language. The solution assumes the existence of certain low-level library modules whose specifications (only) are given here. The design is shown rigorously to meet its specification.

1 Introduction

1.1 Background

The case study is based on a software module for logging events in a medical embedded device such as described in [3]. The device has a limited battery life, so a primary safety requirement for the software is that it be time (CPU) efficient.

The device has a fixed-size memory, which means that in normal use the memory will sometimes be filled to capacity. In such a case, when a new event occurs that requires logging, it will be necessary to overwrite some of

*The CARE project is a collaboration between the SVRC and Telectronics Pacing Systems Pty Ltd, supported by Generic Technology Grant No. 16038 from the Industry Research and Development Board of the Australian Government's Department of Industry, Science and Technology.

the other event records. The more recent events are the more important: the oldest records should be overwritten first. A design is required which maintains as many of the records as practicable.

From time to time, the complete stored event log will be uploaded from the device and the logger will be reinitialized.

1.2 Informal description of software requirements

The application is concerned with storing variable-length records into a fixed-size memory space. (Memory size is determined at compile time, so a symbolic constant should be used in the design.) The design is required to be time (CPU) efficient foremost, but the more records that can be stored the better. When free memory space is no longer available, new records should overwrite the oldest records first. Each new record to be stored is given in full at the time of insertion. There should be operations for reinitializing the log, for inserting a new record, and for reading (uploading) the complete stored event log in a last-in first-out manner.

1.3 This paper

Section 2 gives a formal specification of the data logger in CARE verbose notation [2].[1] Z [4] notation is used for giving mathematical definitions.

Section 3 describes the basic theory for storing records in a fixed-size array. Section 4 gives specifications of the CARE "primitives" that will be assumed (for records, arrays, etc). Section 5 describes a design for the data logger. Section 6 sketches a rigorous verification that the design meets the specification.

2 Formal specification

Let *Record* represent the mathematical set of all possible event records. There will be a CARE type for records:

Type `Record` has specification: *Record*.

[1] A verbose notation for data refinement is introduced.

The following CARE type is used for the log of records stored in the device:

Type `DataLog` has specification: seq *Record* .

The following CARE type is used for the sequence of records read from the log:

Type `RecordSeq` has specification: seq *Record* .

Initially the data log is empty:

Fragment `initialize()` has specification:
output `m:DataLog` such that $\#m = 0$.

Here is the specification of the fragment for reinitializing the log:

Fragment `reinitialize(d:DataLog)` has specification:
output `m:DataLog` such that $\#m = 0$.

Here is the specification of the fragment which reads from the log:

Fragment `readDataLog(m:DataLog)` has specification:
output `s:RecordSeq` such that $s = rev(m)$.

Here is the specification of the fragment for inserting a new record into the log:

Fragment `insertRecord(x:Record,d:DataLog)` has specification:
output `m:DataLog` such that $\exists\, lost : Record \bullet d \frown \langle x \rangle = lost \frown m$.

Note that, while the specification of `insertRecord` is underdetermined, the implementation is expected to do the best it can: i.e., it should return the "longest possible" such m.

3 Mathematical modelling of the problem

This section describes the basic theory for storing records in a low-level data structure (a fixed-size array).

3.1 Records

This section describes a basic mathematical theory of records. For storage, records will be decomposed into sequences of individual ("byte-sized") data items. Let *DataPieces* be the mathematical set of all possible data pieces. The following functions will be defined on records: a function *decompose* for decomposing a record into its constituent data pieces; a function *numcells* which gives the size (in number of data pieces) of a record. The number *maxrecsize* represents a (fixed) upper bound on the size of records.

$$
\begin{array}{|l}
decompose : Record \rightarrow \mathrm{seq}\, DataPieces \\
maxrecsize : \mathbb{N}_1 \\
numcells : Record \rightarrow (1 \,..\, maxrecsize) \\
\hline
\forall\, x : Record \bullet numcells(x) = \#\, decompose(x)
\end{array}
$$

3.2 Storage arrays

This section describes a basic mathematical theory of arrays: *memsize* is the number of cells in the array (assumed to be large enough to store at least one record); *Index* is the set of indexes of the array.

$$
\begin{array}{|l}
memsize : \mathbb{N}_1 \\
\hline
maxrecsize < memsize
\end{array}
$$

$$Index == (1 \,..\, memsize)$$
$$Index^{+} == (1 \,..\, memsize + 1)$$

Cells can contain size data, data pieces, or other kinds of thing (left unspecified here) – see Fig. 1:

$$Cell \ ::= \ \underline{scell}\langle\langle 1 \,..\, maxrecsize \rangle\rangle \mid \underline{dcell}\langle\langle DataPieces \rangle\rangle \mid \ldots$$

An array is modelled as a function from indexes to cells:

$$Array == Index \rightarrow Cell$$

3.3 Storing records

A record will be written into the array from a certain point by recording its size in the first cell and then its decomposition into individual data pieces consecutively in the following cells: see Fig. 1. This section describes some basic mathematical theory associated with such a model.

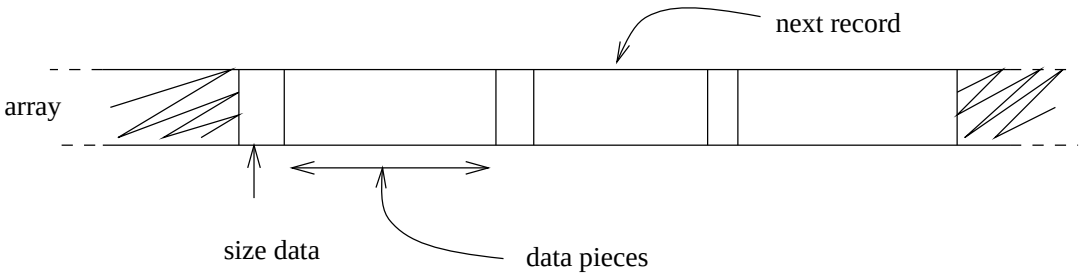

Figure 1: Record pieces are stored in consecutive cells of an array.

$hasRecord(a, i)$ is a predicate which indicates whether or not array a has a record in the cells from index i onwards.

$$
\begin{array}{|l}
hasRecord : Array \leftrightarrow Index \\
\hline
hasRecord(a, i) \Leftrightarrow \\
\quad \exists\, x : Record \bullet i + \#s \leq memsize \land a(i) = \underline{scell}\, \#s \\
\qquad\qquad \land\, \forall k : 1 \,..\, \#s \bullet a(i + k) = \underline{dcell}\, s(k) \\
\qquad \text{where } s = decompose(x)
\end{array}
$$

$hasRecords(a, i, j)$ is a predicate which indicates whether or not array a has a (possibly empty) sequence of records in the cells from index i to $j - 1$.

$$
\begin{array}{|l}
hasRecords : Array \leftrightarrow Index^+ \times Index^+ \\
\hline
\forall\, i : Index^+ \bullet hasRecords(a, i, i) \\
\forall\, i, j : Index^+ \bullet i < j \Rightarrow \\
\quad (hasRecords(a, i, j) \Leftrightarrow \\
\quad hasRecord(a, i) \land hasRecords(a, i + \underline{scell}^{\sim}\, a(i) + 1, j))
\end{array}
$$

$getRecords(a, i, j)$ finds the sequence of records (if any) in array a from index i up to index $j - 1$.

$$
\begin{array}{|l}
getRecords : Array \times Index^+ \times Index^+ \pfun \mathrm{seq}\, Record \\
\hline
\mathrm{dom}\, getRecords = hasRecords \\
\forall\, i : Index^+ \bullet getRecords(a, i, i) = \langle\,\rangle \\
\forall\, x : Record;\ i : Index^+ \bullet \mathrm{let}\, s = decompose(x)\, \mathrm{in} \\
\quad a(i) = \underline{scell}\, \#s \land \forall\, j : 1 \,..\, \#s \bullet a(i+j) = \underline{dcell}\, s(j) \Rightarrow \\
\quad\quad getRecords(a, i, i + \#s + 1) = \langle x \rangle \\
\forall\, i, j, k : Index^+ \bullet i < j < k \Rightarrow \\
\quad getRecords(a, i, k) = getRecords(a, i, j) \frown getRecords(a, j, k)
\end{array}
$$

The following useful lemma is a logical consequence of the definition of *getRecords*:

$$
\begin{array}{l}
\forall\, a, b : Array;\ p, e : Index \bullet \\
\quad (hasRecords(a, p, e+1) \land \forall\, i : Index \bullet p \le i \le e \Rightarrow a(i) = b(i)) \\
\quad \Rightarrow hasRecords(b, p, e+1) \land getRecords(b, p, e+1) = getRecords(a, p, e+1)
\end{array}
$$

4 CARE primitives for the problem domain

This section gives specifications of the types and fragments which will be used as primitives in the design. Note that, since only the specifications of primitives will be given here, the design is independent of how the primitives are implemented.

4.1 Types

The following CARE type will be used for passing size data:

Type `NumCells` has specification: $(1 \,..\, maxrecsize)$

The following CARE types will be used for arrays and indexes:

Type `Array` has specification: $Array$
Type `Index` has specification: $Index$
Type `Index+` has specification: $Index^+$

4.2 Fragments for records and arrays

The following fragment finds the size (in number of "bytes") of a given record:

Fragment `numcells(x:Record)` has specification:
output `n:NumCells` such that $n = numcells(x)$.

The following fragment returns the size of the array:

Fragment `arraysize(a:Array)` has specification:
output `n:Nat` such that $n = memsize$.

The following fragment returns an arbitrary array (for initialization):

Fragment `arbArray()` has specification:
output `a:Array`.

The following fragment extracts size data from array a at index p:

Fragment `getsize(a:Array,p:Index)` has specification:
precondition $hasRecord(a, p)$
output `s:NumCells` such that $a(p) = \underline{scell}(s)$.

4.3 Reading and writing individual records

The following fragment reads a single record from the cells following index p in array a (assuming such a record exists):

Fragment `readRecord(a:Array,p:Index)` has specification:
precondition $hasRecord(a, p)$
output `x:Record` such that $\exists\, s : \mathbb{N}_1 \bullet getRecords(a, p, p + s + 1) = \langle x \rangle$.

The following fragment writes a record into the cells following index p in array a:

Fragment `writeRecord(a:Array,p:Index,x:Record)` has specification:
precondition $p + numcells(x) \leq memsize$
output `b:Array` such that
$hasRecords(b, p, p + s + 1) \land getRecords(b, p, p + s + 1) = \langle x \rangle \land$
$(p \,..\, p + s) \lhd a = (p \,..\, p + s) \lhd b$
where $s = numcells(x)$.

4.4 Fragments for reporting

The following fragments will be used for reporting the sequence of records read from the log:

Fragment `emptyRecordSeq()` has specification:
output `s:RecordSeq` such that $\#s = 0$.

Fragment `apndlRecordSeq(x:Record,s:RecordSeq)` has specification:
output `t:RecordSeq` such that $t = \langle x \rangle \frown s$.

Fragment `concatRecordSeq(s:RecordSeq,t:RecordSeq)` has specification:
output `r:RecordSeq` such that $r = s \frown t$.

4.5 Arithmetic

The following type will be used for natural numbers:

Type `Nat` has specification: $\mathbb{N}$

The following fragments will be needed for performing integer arithmetic:[2]
`0()` `0:Nat`
`1()` `1:Nat`
`_+_(m:Nat,n:Nat)` $m + n$`:Nat`
`_-_(m:Nat,n:Nat)` $max\{0, m - n\}$`:Nat`
`equal(m:Nat,n:Nat)` if $m = n$ then report `yes` else report `no`
`lessthaneq(m:Nat,n:Nat)` if $m \leq n$ then report `yes` else report `no`

5 The Design

5.1 The data model

Informally, the data logger will be designed as follows: After reinitialization, the first new record will be written into the array in the cells following index 1. After that, new records will be written into the array consecutively until

[2]Terse CARE notation is used here.

no more fit; writing then begins from index 1 again, overwriting existing records.

Thus, the "state" of the device at any time during its operation will be one in which there is a sequence of "newer" records starting at index 1 in the array, followed by a sequence of "older" records towards the end, with the two sequences possibly separated by a gap: see Fig. 2. In what follows, the sequence of newer records will be called the *low set* and the sequence of older records the *high set*.

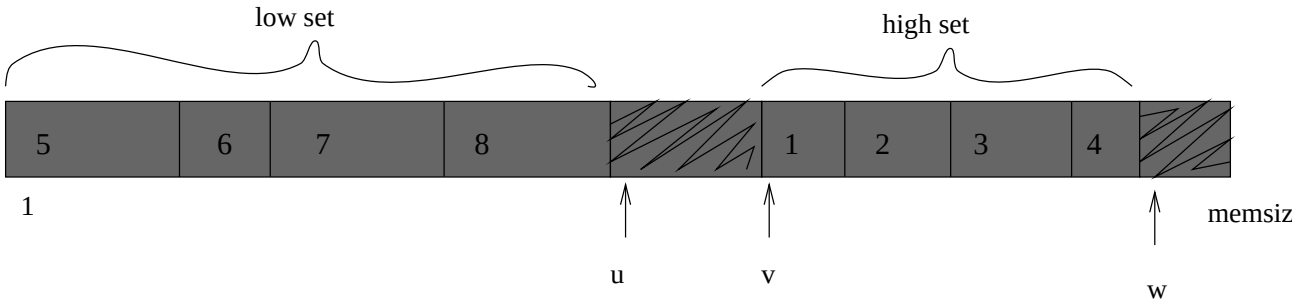

Figure 2: Design of the data log.

As a CARE data refinement, the `DataLog` will be implemented in terms of an array `a` with three pointers as follows: `u` indicates the end of the low set (the next cell after the last cell corresponding to the last record of the low set); `v` indicates the start of the high set (size data cell corresponding to the first record of the high set); and `w` indicates the end of the high set. This is represented by a CARE data refinement as follows:

Type `DataLog` has
specification: seq *Record*
refinement:
value `d:DataLog`
is refined by `a:Array,u,v,w:Index+`
with invariant $u \leq v \leq w \wedge hasRecords(a, 1, u) \wedge hasRecords(a, v, w)$
with refinement relation $d = getRecords(a, v, w)^\frown getRecords(a, 1, u)$.

The invariant defines which quadruples (a, u, v, w) represent data logs. The refinement relation relates the "abstract" value of the data log d (a sequence of records) to the "concrete" values a, u, v, w.

5.2 Initialization and reinitialization

Initially the values of `u`, `v` and `w` will be 1.

> Fragment `initialize()` has
> specification:
> output `m:DataLog` such that $\#m = 0$.
> implementation:
> compose `arbArray,1,1,1` into `m:DataLog`;
> return `m`.

The log gets reinitialized by setting all three indexes to 1.

> Fragment `reinitialize(d:DataLog)` has
> specification:
> output `m:DataLog` such that $\#m = 0$.
> implementation:
> decompose `d` to `a:Array,u,v,w:Index+`;
> compose `a,1,1,1` into `m:DataLog`;
> return `m`.

5.3 Reading the data log

Here is the implementation of the fragment for reading the log:

> Fragment `readDataLog(d:DataLog)` has
> specification:
> output `r:RecordSeq` such that $r = rev(d)$
> implementation:
> decompose `d` to `a:Array,u,v,w:Index+`;
> return `concatRecordSeq(readRecords(a,1,u),readRecords(a,v,w))`.

where

> Fragment `readRecords(a:Array,p,e:Index+)` has
> specification:
> precondition $p \leq e \wedge hasRecords(a, p, e)$
> output `r:RecordSeq` such that $r = rev(getRecords(a, p, e))$.
> implementation:

return `readAccum(a,p,e,emptyRecordSeq)`.

Fragment `readAccum(a:Array,p,e:Index+,u:RecordSeq)` has specification:
precondition $p \leq e \wedge hasRecords(a, p, e)$
output `r:RecordSeq` such that $r = rev(getRecords(a, p, e))^\frown u$.
implementation:

```
cases equal(p,e) of:
  yes: return u.
  no:  assign apndlRecordSeq(readRecord(a,p),u) to w:RecordSeq;
       return readAccum(a,p+getsize(a,p)+1,e,w).
```

variant: $p - e$

5.4 Inserting a new record

The algorithm for inserting a new record into the log can be described informally as follows:

- If there is enough room for the new record after the low set, then append the record to the end of the low set.
  - If the record fits into the gap between the low set and the high set, then the high set can remain unchanged.
  - Otherwise, check to see if any records from the high set will *not* be overwritten:
    * If there are any remaining records, they become the new high set. (The other records get overwritten.)
    * Otherwise, the new high set is empty. (The `v` and `w` pointers are both set equal to the new `u`.)
- Otherwise, the new record gets written at the start. The new low set consists of the latest record alone. The old high set gets thrown away. The new high set is defined from the old low set as follows:
  - If there are any remaining records, they become the new high set.

– Otherwise, the new high set is empty.

This algorithm is expressed as a CARE fragment implementation in Fig. 3 below, where `findRemRecords(a,p,e,n)` is used for finding the index of the first (if any) record which starts after index `p+n` and ends at or before index `e` in array `a` (Fig. 4).

6 Verification

This section sketches a justification of the correctness of the design.

6.1 The data refinement

In order for the refinement to be valid, each concrete value should correspond to exactly one abstract value (the "representation proof obligation"). Since the refinement relation is functional in the concrete values, this obligation is trivially true.

Strictly, there is also an obligation ("adequacy" – cf. [1]) to show that every abstract value has at least one corresponding concrete value. In our case this obligation clearly cannot be discharged: there is a fixed bound on the number of records the array can hold. To address this problem we need to put a constraint on the length of sequences that are used to model the data log; unfortunately this cannot be done in any straightforward manner.[3]

6.2 Initialization and reinitialization

Partial correctness of `initialize` follows from the refinement relation and the fact that $getRecords(a, 1, 1) = \langle\,\rangle$:

$$\#(getRecords(a, 1, 1) \frown getRecords(a, 1, 1)) = \#(\langle\,\rangle \frown \langle\,\rangle) = \#\langle\,\rangle = 0$$

Well-formedness of 'compose `a,1,1,1`' requires showing that the invariant holds:

$$1 \leq 1 \leq 1 \wedge hasRecords(a, 1, 1) \wedge hasRecords(a, 1, 1)$$

This follows easily from the fact that $hasRecords(a, i, i)$ for all i.

The proof obligations for `reinitialize` are similar.

[3] This is yet another example of the constraint propagation problem.

Fragment `insertRecord(x:Record,d:DataLog)` has specification:

output `m:DataLog`

such that $\exists\, lost : \mathrm{seq}\ Records \bullet d \frown \langle x \rangle = lost \frown m$

implementation:

```
decompose d to a:Array,u,v,w:Index+;
assign numcells(x) to s:NumCells;
cases lessthaneq(u+s,arraysize) of:
  yes: cases lessthaneq(u+s+1,v) of:
         yes: assign writeRecord(a,u,x) to a':Array;
              compose a',u+s+1,v,w into m:DataLog;
              return m.
         no:  cases findRemRecords(a,v,w,u+s-v)
              of:
                found: assign output(s) to v':Index;
                       assign writeRecord(a,u,x) to a':Array;
                       compose a',u+s+1,v',w into m:DataLog;
                       return m.
                none:  assign writeRecord(a,u,x) to a':Array;
                       assign u+s+1 to u':Index+;
                       compose a',u',u',u' into m:DataLog;
                       return m.
  no:  cases findRemRecords(a,1,u,s) of:
         found: assign output(s) to v':Index;
                assign writeRecord(a,1,x) to a':Array;
                compose a',s+1,v',u into m:DataLog;
                return m.
         none:  assign writeRecord(a,1,x) to a':Array;
                assign s+1 to u':Index+;
                compose a',u',u',u' into m:DataLog;
                return m.
```

Figure 3: Design of the function for inserting a new record into the log.

Branching fragment `findRemRecords(a:Array,p,e:Index+,n:Nat)` has specification:

precondition $p \leq e \wedge 1 \leq n \wedge p + n \leq memsize \wedge hasRecords(a, p, e)$

result defined by cases

if $\exists\, i : Index \bullet p + n \leq i < e \wedge hasRecords(a, i, e)$

then report `found` and return `i:Index`

such that $p + n \leq i < e \wedge hasRecords(a, i, e)$

else report `none`

implementation:

```
cases equal(p,e) of:
  yes: report none
  no:  cases lessthaneq(e,p+n) of:
         yes: report none
         no:  assign getsize(a,p) to s:NumCells;
              assign p+s+1 to k:Index+;
              cases equal(k,e) of:
                yes: report none
                no:   cases lessthaneq(n,s+1)
                      of:
                        yes: report found and return k
                        no:
                             findRemRecords(a,k,e,n-(s+1))
```

variant: n

Figure 4: The auxiliary fragment for finding the (index of the) first record which will not be overwritten.

6.3 Reading the data log

We examine each of the fragments in turn:

`readDataLog`:

Partial correctness of `readDataLog` amounts to showing

$$rev(getRecords(a, 1, u)) \frown rev(getRecords(a, v, w)) = rev(d)$$

which follows easily from the refinement relation.

Well-formedness of the calls to `readRecords` amounts to showing

$$1 \leq u \wedge hasRecords(a, 1, u),\ v \leq w \wedge hasRecords(a, v, w)$$

which follow from the refinement invariant.

`readRecords`:

Partial correctness and well-formedness are straightforward.

`readAccum`:

Partial correctness of the first path follows from the fact that $p = e$ on this path:

$$rev(getRecords(a, p, p)) \frown u = (rev\langle\,\rangle) \frown u = \langle\,\rangle \frown u = u$$

as required.

Partial correctness of the second path amounts to showing

$$rev(getRecords(a, p + s + 1, e)) \frown w = rev(getRecords(a, p, e)) \frown u$$

from the following facts about the path:

$$p \neq e,\ a(p) = \underline{scell}(s),\ getRecords(a, p, p + s + 1) = \langle x \rangle,\ w = \langle x \rangle \frown u$$

With a little effort, the desired result can be shown to follow from the following fact:

$$getRecords(a, p, e) = getRecords(a, p, p + s + 1) \frown getRecords(a, p + s + 1, e)$$

Well-formedness of the calls to `readRecord` and `getsize` on the second path follows from the fact that $p < e$ on this path, using the following lemma:

$$p < e \wedge hasRecords(a, p, e) \Rightarrow$$
$$hasRecord(a, p) \wedge p + s + 1 \leq e \wedge hasRecords(a, p + s + 1, e)$$

where $s = \underline{scell}^{\sim}\ a(p)$.

Termination of the recursion follows from the fact that

$$0 \leq e - (p + s + 1) < e - p$$

6.4 Inserting a new record

`insertRecord:`

Let us first consider partial correctness of the algorithm for inserting a record into the log. From the refinement relation we can assume

$$d = getRecords(a, v, w) \frown getRecords(a, 1, u)$$
$$m = getRecords(a', v', w') \frown getRecords(a', 1, u')$$

where a', u', v' and w' are the new values of the array and pointers defined in the algorithm. We are required to show that $d \frown \langle x \rangle = lost \frown m$ for some sequence of records $lost$. Let us consider each of the paths in turn:

1. Room for x at the end of the existing low set before the high set starts. The following facts hold at the end of this path:

   $$u + s \leq memsize,\ u + s + 1 \leq v,\ u' = u + s + 1,\ v' = v,\ w' = w,$$
   $$getRecords(a', 1, u) = getRecords(a, 1, u),\ getRecords(a', u, u') = \langle x \rangle$$

   It is easy to show that $getRecords(a', 1, u') = getRecords(a, 1, u) \frown \langle x \rangle$. Thus the desired result holds, with $lost = \langle\,\rangle$.

2. Room at end of low set but some (not all) of high set gets overwritten.

   $$u + s \leq memsize,\ \neg\,(u + s + 1 \leq v),$$
   $$u + s \leq v' < w,\ hasRecords(a, v', w),\ u' = u + s + 1,\ w' = w,$$
   $$getRecords(a', 1, u') = getRecords(a, 1, u) \frown \langle x \rangle$$

It is easy to show that $getRecords(a, v, w) = getRecords(a, v, v') \frown getRecords(a', v', w')$.
Thus the desired result holds, this time with $lost = getRecords(a, v, v')$.

3. Room at end of low set and all of high set gets overwritten.

$$u + s \leq memsize,\ \neg\,(u + s + 1 \leq v),\ u' = v' = w' = u + s + 1,$$
$$getRecords(a', 1, u') = getRecords(a, 1, u) \frown \langle x \rangle$$

The desired result holds, this time with $lost = getRecords(a, v, w)$.

4. No room at end of low set and some (but not all) of low set gets overwritten.

$$\neg\,(u + s \leq memsize),\ 1 \leq v' < u,\ hasRecords(a, v', u),$$
$$u' = s + 1,\ w' = u,\ getRecords(a', 1, s + 1) = \langle x \rangle$$

It is easy to show that $getRecords(a, 1, u) = getRecords(a, 1, v') \frown getRecords(a', v', w')$.
The desired result holds, with $lost = getRecords(a, v, w) \frown getRecords(a, 1, v')$.

5. No room at end of low set and all of low set gets overwritten.

$$\neg\,(u + s \leq memsize),\ s + 1 \leq v' < u,\ hasRecords(a, v', u),$$
$$u' = v' = w' = s + 1,\ getRecords(a', 1, u') = \langle x \rangle$$

The desired result holds, with $lost = d$.

Now consider the well-formedness of fragment calls:

1. For the first three occurrences of calls to `writeRecord`, the precondition is $u + numcells(x) \leq memsize$, which is a consequence of one of the path tests on the paths concerned. For the other two occurrences, the precondition is $1 + numcells(x) \leq memsize$, and this follows from the fact that

$$numcells(x) \leq maxrecsize < memsize$$

2. The precondition of `findRemRecords(a,p,e,n)` is

$$p \leq e \wedge 1 \leq n \wedge p + n \leq memsize \wedge hasRecords(a, p, e)$$

   It is straightforward to check both of the cases that arise.

3. The data abstraction 'compose `a,u,v,w`' is well-formed provided the following holds:

$$u \leq v \leq w \wedge hasRecords(a, 1, u) \wedge hasRecords(a, v, w)$$

   It is straightforward to check each of the cases that arise.

`findRemRecords:`

For partial correctness of `findRemRecords` there are five paths to consider:

1. Correctness of the first path follows from

$$1 \leq n \wedge p = e \Rightarrow \neg\ \exists\, i : Index \bullet p + n \leq i < e$$

2. Correctness of the second path follows from

$$e \leq p + n \Rightarrow \neg\ \exists\, i : Index \bullet p + n \leq i < e$$

3. Correctness of the third path follows from the following lemma

$$p + s + 1 = e \Rightarrow \neg\ \exists\, i : Index \bullet p < i < e \wedge hasRecords(a, i, e)$$

   where $s = \underline{scell}^{\sim}\ a(p)$.

4. Correctness of the fourth path follows from the fact that

$$p + n \leq p + s + 1 < e \wedge hasRecords(a, p + s + 1, e)$$

   on this path.

5. Correctness of the fifth path is obvious upon noting that $k+n-(s+1) = p + n$.

Termination and well-formedness of the recursive call to `findRemRecords` are straightforward.

7 Conclusion

The design has been shown to satisfy the specification, in terms of its intended functionality. The algorithm for inserting records into the log is of reasonable complexity and, in the absense of CARE , would take some concerted thought to convince oneself of its correctness. The case study thus demonstrates how the CARE method can be used to increase assurance in the correctness of designs.

The author would like to thank his colleagues on the CARE project for their useful contributions to this case study.

References

[1] C.B. Jones. *Systematic Software Development Using VDM.* Prentice-Hall International, second edition, 1990.

[2] P.A. Lindsay. The CARE method of verified software development. Technical Report 95-9, Software Verification Research Centre, University of Queensland, 1995.

[3] R. Mojdehbakhsh, W-T. Tsai, S. Kirani, and L. Elliott. Retrofitting software safety in an implantable medical device. *IEEE Software*, pages 41–50, January 1994.

[4] J.M. Spivey. *The Z Notation: a Reference Manual.* Prentice-Hall, New York, 1989.