Applying Formal Methods to the Design of Smart Card Software

Michael Butler, Pieter Hartel, Eduard de Jong and Mark Longley

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-97-08
July 1997

`www.dsse.ecs.soton.ac.uk/techreports/`

Department of Electronics and Computer Science
University of Southampton
Highfield, Southampton SO17 1BJ, United Kingdom

Applying Formal Methods to the Design of Smart Card Software

Michael Butler, Pieter Hartel, and Mark Longley
Department of Electronics and Computer Science, University of Southampton

Eduard de Jong
Integrity Arts, San Mateo, USA

Deliverable of the

Smart Card Software Generator Project

July 1997

Contents

Section 1

Management Summary

The goal of this work is the design of a language for the implementation of smart card applications, specifically an operating system, as high integrity software. The integrity of a piece of software is demonstrated by proving various properties of the software. The language must therefore exclude any constructs that would make such proofs unreasonably difficult. An untyped language is not only very difficult to reason about formally but also allows many unchecked run-time errors that are eliminated in a, suitably, typed language. We would like the type system of the language to be strong, expressive and simple. Unfortunately the language is required to be able implement certain routines that might normally be part of the run-time system, notably the storage allocation routines. This requirement is likely to force the adoption of a weaker type system than we would ideally prefer.

In order to understand the consequences of this requirement we first had to understand in more detail the storage allocation system required. To this end Michael Butler prepared an initial **Z** specification of the existing implementation (See 2). Pieter Hartel then produced an executable specification and Mark Longley a Miranda implementation (See 3). These led to a modified **Z** specification for the existing implementation in both an abstract and refined form. (See 4). The refined form of the modified **Z** specification was further refined to a detailed design (See 5). This was followed by some thought about the general requirements and implications of a storage allocation function (See 6) and an example implementation in **Modula-3** (See 7).

Finally a proposal for a type system was prepared, describing the advantages of certain choices and the problems introduced by others (See 8).

1.1 Conclusion

- The formal specification, and example implementations, enabled discussions concerning the existing implementation of the tagged memory system to proceed without confusion or ambiguity.
- Consideration of the general problem of a functional interface to a storage allocator, the tagged memory specification and the example **Modula-3** implementation led to a type system proposal.
- The proposed type system has a number of significant features:
  1. It is not so strong as to preclude the implementation of a storage allocator.
  2. By distinguishing between type coercions we have shown that it is possible to limit the "*type-unsafe*" type coercion to a single instance in the storage allocator.
  3. By introducing range types and associated language constructs we have shown that static checking of array indexing can be achieved in many cases.
  4. As well as describing the inherently type-unsafe operations we have also described those operations that may still require a run-time type check. These run-time checks could also be eliminated if the appropriate proof obligations were satisfied.

Section 2

Initial Z Specification of Memory Manager

2.1 Introduction

The following is a LaTeX presentation of a hand written **Z** specification for the **CLASP** tagged memory implementation presented by Michael Butler. We use a non-standard form of **Z** to which the various **Z** tools will not be applicable.

2.2 Abstract Specification

We leave the details of the information being stored and the tags it is associated with unspecified.

$$[Tag, Page]$$

We have a limit on the size of the memory system and a given set of available tags.

$$\begin{array}{|l} msize: \mathbb{N} \\ tags: \mathbb{F}\ Tag \end{array}$$

The abstract specification of the memory system specifies the association between tags and sequences of information as a partial function. The only property we require is that the total number of pages associated with all the tags is no greater than the size of the memory.

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\ AMemSys\ }} \\ data: tags \rightarrow\!\!\!\!\!+ \operatorname{seq} Page \\ \hline (\sum t \mid t \in \operatorname{dom}(data) \bullet \#data(t)) \leqslant msize \\ \hline \end{array}$$

We have an operation that creates a new association, for a new tag, for a given sequence length.

$$\begin{array}{|l} \multicolumn{1}{l}{\underline{\ ANewObject\ }} \\ \Delta AMemSys \\ n?: \mathbb{N} \\ t!: tags \\ \hline t! \notin \operatorname{dom}(data) \\ (\sum t \mid t \in \operatorname{dom}(data) \bullet \#data(t)) \leqslant (msize - n?) \\ \exists\, d: \operatorname{seq} Page \mid \#d = n? \bullet data' = data \cup \{t! \mapsto d\} \\ \hline \end{array}$$

We have an operation that returns the information associated with a tag.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{AReadObject} \quad}} \\
\Xi \mathit{AMemSys} \\
t? : \mathit{tags} \\
d! : \mathrm{seq}\ \mathit{Page} \\
\hline
t? \in \mathrm{dom}(\mathit{data}) \\
d! = \mathit{data}(t?) \\
\hline
\end{array}
$$

We have an operation that writes a sequence of information to a tag.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{AWriteObject} \quad}} \\
\Delta \mathit{AMemSys} \\
t? : \mathit{tags} \\
d? : \mathrm{seq}\ \mathit{Page} \\
\hline
t? \in \mathrm{dom}(\mathit{data}) \wedge \# d? = \mathit{data}(t?) \\
(\sum t \mid t \in \mathrm{dom}(\mathit{data}) \bullet \# \mathit{data}(t)) \leqslant (\mathit{msize} - \# d?) \\
\mathit{data}' = \mathit{data} \oplus \{t? \mapsto d?\} \\
\hline
\end{array}
$$

We have an operation that releases a tag.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{AReleaseObject} \quad}} \\
\Delta \mathit{AMemSys} \\
t? : \mathit{tags} \\
\hline
t? \in \mathrm{dom}(\mathit{data}) \\
\mathit{data}' = \{t?\} \lhd \mathit{data} \\
\hline
\end{array}
$$

2.3 Concrete Specification

The concrete specification requires some more complex types so we introduce some abbreviations.

$$
\begin{array}{l}
\mathit{Loc} == 0 \ldots (\mathit{msize} - 1) \\
\mathit{Gen} == \mathbb{N} \\
\mathit{PageNo} == \mathbb{N} \\
\mathit{CPage} == \mathit{tags} \times \mathit{PageNo} \times \mathit{Gen} \times \mathit{Page} \\
\mathit{CMem} == \mathit{Loc} \rightarrow \mathit{CPage}
\end{array}
$$

The concrete specification of the memory system introduces a number of relations, most of which are implicit in the others. We have more complex conditions controlling the various generations of information that may be associated with a tag.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{CMemSys} \quad}} \\
\mathit{cmem} : \mathit{CMem} \\
\mathit{locs} : \mathit{tags} \pfun (\mathrm{seq}\ \mathit{Loc}) \\
\mathit{size} : \mathit{tags} \pfun \mathbb{N} \\
\mathit{avail} : \mathbb{P}\ \mathit{Loc} \\
\hline
\mathrm{dom}(\mathit{locs}) = \mathit{dom}(\mathit{size}) \\
\forall t : \mathit{tags} \mid t \in \mathrm{dom}(\mathit{locs}) \bullet \\
\qquad \mathit{locs}(t) = \mathit{recent}(\mathit{good\_matches}(t, \mathit{size}(t), \mathit{cmem})) \\
\mathit{avail} = \mathit{Loc} \setminus \\
\qquad \{ i \mid \exists t \in \mathrm{dom}(\mathit{size}) \bullet \exists j : \mathbb{N} \mid \\
\qquad\qquad 0 \leqslant j < \mathit{size}(t) \wedge \mathit{locs}(t)(j) = i \\
\qquad \} \\
\hline
\end{array}
$$

The data refinement used when deriving the concrete specification from the abstract specification is as follows.

$$
\begin{array}{|l}
\hline
\textit{Abstraction} \\
\textit{AMemSys} \\
\textit{CMemSys} \\
\hline
\mathrm{dom}(\textit{data}) = \mathrm{dom}(\textit{size}) \\
\forall\, t \in \mathrm{dom}(\textit{data}) \bullet \\
\quad \#\textit{data}(t) = \textit{size}(t) \\
\quad \textit{data}(t) = \textit{page} \circ \textit{cmem} \circ (\textit{locs}(t)) \\
\hline
\end{array}
$$

We have an operation to write to the memory system.

$$
\begin{array}{|l}
\hline
\textit{CWriteObject} \\
\Delta \textit{CMemSys} \\
t?: \textit{Tag} \\
d?: \mathrm{seq}\ \textit{Page} \\
\hline
t? \in \mathrm{dom}(\textit{size}) \\
\#d? = \textit{size}(t?) \\
\exists\, \textit{lc}: \mathrm{seq}\ \textit{Loc} \mid \mathrm{injective}(\textit{lc}) \wedge (\mathrm{ran}\ \textit{lc}) \subseteq \textit{avail} \wedge \#\textit{lc} = \textit{size}(t) \bullet \\
\quad \textit{cmem}' = \textit{cmem} \oplus \{j \mid 0 \leqslant j < \textit{size}(t?) \bullet \\
\qquad\qquad \textit{lc}(j) \mapsto (t?, j, g+1, d?(j)) \\
\qquad\qquad \underline{\textbf{where}} \\
\qquad\qquad g = \mathrm{fst}(\textit{max\_gen}(\textit{good\_matches}(t?, \\
\qquad\qquad\qquad\qquad \textit{size}(t?), \\
\qquad\qquad\qquad\qquad \textit{cmem}))) \\
\qquad\quad \} \\
\quad \textit{locs}' = \textit{locs} \oplus \{t? \mapsto \textit{lc}\} \\
\quad \textit{avail}' = (\textit{avail} \setminus (\mathrm{ran}\ \textit{lc})) \cup \mathrm{ran}(\textit{locs}(t?)) \\
\quad \textit{size}' = \textit{size} \\
\hline
\end{array}
$$

We have an operation to read the information associated with a tag.

$$
\begin{array}{|l}
\hline
\textit{CReadObject} \\
\Xi \textit{CMemSys} \\
t?: \textit{tags} \\
d!: \mathrm{seq}\ \textit{Page} \\
\hline
t? \in \mathrm{dom}(\textit{size}) \\
d! = \textit{page} \circ \textit{cmem} \circ (\textit{locs}(t?)) \\
\hline
\end{array}
$$

We have an operation to release a tag.

$$
\begin{array}{|l}
\hline
\textit{CReleaseObject} \\
\Delta \textit{CMemSys} \\
t?: \textit{tags} \\
\hline
t? \in \mathrm{dom}(\textit{size}) \\
\textit{size}' = \{t?\} \lhd \textit{size} \\
\textit{locs}' = \{t?\} \lhd \textit{locs} \\
\textit{avail}' = \textit{avail} \cup \mathrm{ran}(\textit{locs}(t?)) \\
\textit{cmem}' = \textit{cmem} \\
\hline
\end{array}
$$

We have an operation that returns the pages associated with a tag.

$$
\begin{array}{|l}
good\_matches : tags \times \mathbb{N} \times CMem \rightarrow \mathbb{P}(Gen \times \mathrm{seq}\ Loc) \\
\hline
good\_matches(t, n, cmem) = \\
\quad \{(g, lc) \mid \# lc = n \wedge \\
\quad\quad \forall i \mid 0 \leqslant i < n \bullet \\
\quad\quad\quad tag(cmem(lc(i))) = t \\
\quad\quad\quad gen(cmem(lc(i))) = g \\
\quad\quad\quad pageno(cmem(lc(i))) = i \\
\quad \}
\end{array}
$$

We have a operation that returns the maximum generation associated with a tag.

$$
\begin{array}{|l}
max\_gen : \mathbb{P}(Gen \times \mathrm{seq}\ Loc) \rightarrowtail Gen \times \mathrm{seq}\ Loc \\
\hline
\mathrm{dom}(max\_gen) = (Gen \times \mathrm{seq}\ Loc) \setminus \{\} \\
\forall s \in \mathrm{dom}(max\_gen) \bullet \\
\quad max\_gen(s) \in s \\
\quad \forall x \bullet x \in s \Rightarrow \mathrm{fst}(max\_gen(s)) \geqslant \mathrm{fst}(x)
\end{array}
$$

We have an operation to return the most recent information associated with a tag.

$$
\begin{array}{|l}
recent : \mathbb{P}(Gen \times \mathrm{seq}\ Loc) \rightarrowtail Loc \\
\hline
\mathrm{dom}(recent) = \mathrm{dom}(max\_gen) \\
recent(s) = \mathrm{snd}(max\_gen(s))
\end{array}
$$

We have four operations that extract the fields from a concrete page.

$$
\begin{array}{|l}
tag : CPage \rightarrow tags \\
gen : CPage \rightarrow \mathbb{N} \\
pageno : CPage \rightarrow \mathbb{N} \\
page : CPage \rightarrow Page \\
\hline
tag(t, pn, g, p) = t \\
gen(t, pn, g, p) = g \\
pageno(t, pn, g, p) = pn \\
page(t, pn, g, p) = p
\end{array}
$$

2.4 Conclusion

The details of the actual implementation still remain unclear. The relationship between the New and Write operations and the precise data structures required by the implementation require further explanation.

Section 3

Memory System Implementation

3.1 Introduction

Starting from the initial **Z** specification of a memory system I produced a Miranda[1] implementation of the operations specified in order to clarify some of the questions raised by the specification.

I chose to define an abstract type that provided the **New**, **Read**, **Write** and **Release** operations from the abstract specification. The implementation (type and operations) of this abstract type was intended to illuminate some of the issues raised by the concrete specification.

3.2 Parameterised Specification

The first problem with implementing the specification is its use of the unspecified types **Tag** and **Page** and the values **msize** and **tags**. I chose to use the Miranda `%free` mechanism to produce a script that was paremeterised by the types `info` and `tag` and the values `memsys_size` and `tags`[2].

```
%free
{     info :: type
      memsys_size :: num
      tag :: type
      tags :: [tag]
}
```

I use the name `info` for the "chunk" of information associated with a `tag` rather than **Page** as I find this less confusing.

This does not completely model the **Z** specification as it is not possible to constrain `memsys_size` to be a natural number or `tags` to be a finite list. We do, however, obtain an implementation that is independent of the types of the **Tag** and **Page**.

3.3 Abstract Type Signature

The signature of the abstract type does not capture any of the conditions specified by the abstract specification, it simply declares the types of the functions. As the **ANewObject**, **AWriteObject** and **AReleaseObject** operations modify a memory system we must add an extra operation to provide an initial memory system[3].

As the operations over a memory system may fail for a variety of reasons I include a simple form of exception handling using the following Miranda algebraic type:

```
ok * ::= OK * | Error [[char]]
```

This allows us to return an error message when an exception occurs.

[1] Miranda is a trademark of Research Software Ltd.
[2] The script is also parameterised by display functions for the types.
[3] The signature also contains a function to display a memory system.

```
abstype memsys with
      initial_memsys ::  memsys
      new_assoc ::  num -> memsys -> ok (tag,memsys)
      read_assoc ::  tag -> memsys -> ok [info]
      write_assoc ::  tag -> [info] -> memsys -> ok memsys
      release_assoc ::  tag -> memsys -> ok memsys
```

3.4 Implementation Type

The concrete specification defines a number of relations (**cmem**, **locs**, **size** and **avail**) most of which, as Michael Butler noted, are included for ease of presentation. I have chosen to represent a memory system as a simple list of pages (that is `page` not **Page**!) where all these relations are implicit in the housekeeping information associated with the pages.

```
location, generation, page_No == num
page ::= Page bool bool tag generation page_No info

memsys == [page]
```

A real implementation may well include auxiliary data structures that record information for each tag. Lacking information about what these data structures might be I chose an implementation that examined the extreme case where no such data structures are available. This means that all operations over the relations defined in the concrete specification will require repeated searches over the entire memory system in this implementation.

The `page` type contains two boolean flags that indicate whether the page is currently in use (associated with some tag) and whether it has been written to yet. The first flag is required so that we can generate a new list of pages to associate with a tag and the second so that we can write into pages returned by `new_assoc` and still maintain a trace of the generations of pages associated with a tag. Each page also records the tag it is associated with, if any, and its generation and page number for that tag.

In the implementation equations for the abstract type I have chosen to manipulate the locations of pages rather than the pages themselves to give some sense of the problems a real implementation might face.

3.5 `initial_memsys`

The initial memory system is simply a list of pages, each of which is marked as unused. In a real implementation the data in a page (housekeeping and information) will have some, arbitrary, initial value. Errors in the implementation of the memory system could cause this initial information to be returned in some circumstances. There is no way in Miranda to provide arbitrary initial data in the initial memory system. The best we can do is use initial values that will cause a fatal error if they are referenced. This means that a Miranda implementation can never properly model a real system where erroneously referencing initial data would not cause an error.

3.6 ANewObject/`new_assoc`

The `new_assoc` function creates a new association between a tag and a list of pages of the required length. This will fail if there are no unused tags or there are not enough unused pages. We implement existential quantifiers in the specification as searches (`filter`) followed by choices (`hd` and `take`). The association between tags and pages is modified by updating the housekeeping information in the pages newly associated with the tag. While these pages are now marked as used they are not yet marked as written.

3.7 AReadObject/`read_assoc`

The `read_assoc` function returns the list of information, if any, associated with a tag. This will fail if there are no pages associated with the tag or they are unwritten pages. Because we are mainting a trace of the generations of pages associated with a tag this function must ensure that it returns those pages associated with the most recent generation of the tag.

3.8 AWriteObject/`write_assoc`

The `write_assoc` function writes a list of information to a tag. This will fail if there are no pages associated with the tag, the number of pages to write differ from those associated with the tag or there are not enough unused pages. Each successive write to a tag obtains a new generation count, the most recent generation being that returned by a read. When we write some information to a tag we must distinguish the first write to newly allocated storage from all subsequent writes to the tag. We use the second boolean flag in the pages to do this.

3.9 AReleaseObject/`release_assoc`

The `release_assoc` function frees all the pages associated with a tag. This releases all the generations of pages associated with a tag. As there is no limit on the number of generations that may be associated with a tag a sequence of writes to the same tag can exhaust the memory system.

3.10 Conclusion

- There are many data structures that could be chosen to represent the relations employed in the concrete specification. They have various advantages and disadvantages in terms of the storage used and the ease with which functions over them may be implemented. This implementation uses the simplest possible data structure resulting in complex function definitions.
- There are various properties of real implementations that can't be captured in a Miranda implementation.
- The **ANewObject** operation doesn't seem to be entirely neccessary. A suitably sophisticated `write_assoc` function could achieve the same effect (generational backup) without the complexities this operation introduces.

Section 4

Modified Z Specification

4.1 Introduction

This modification of the initial **Z** specification incorporates the new features required of the memory system that arose during discussions of the original version. The following changes have been made in the specification:

- The **ANewObject** operation is not required in its original form. We can instead use an operation that returns an unused tag and a more sophisticated write operation.
- The generation marking of the information associated with a tag is now controlled by a new "commit" operation. After a commit operation a subsequent write will aquire a new generation mark. All following writes, assumming no commit occurs, will have this same generation mark.
- The read operation may now specify a generation relative to the current generation.
- A limit on the number of generations that may be associated with a tag is introduced.
- Operations that manipulate the generations of information associated with a tag may be introduced.

These new requirements alter the abstract specification significantly, introducing new operations and making explicit some of the lower level details of the memory system.

4.2 Abstract Specification

As in the original specification we don't need to know anything about tags and the information associated with them so we parameterise our specification by the types *Tag* and *Info* types.

$$[Tag, Info]$$

We also have a given set of available tags and limits on the size of the memory system and the maximum generation of information that may be associated with a tag:

$$\begin{array}{|l} tags : \mathbb{F}\, Tag \\ msize : \mathbb{N}_1 \\ maxgen : \mathbb{N}_1 \end{array}$$

The memory system is specified by two partial functions and a set, we include a derived value to aid the presentation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AMemSys \quad}} \\
assoc : tags \rightarrowtail\!\!\!\!\!\!\!\!\not\;\;\;\;\; \mathrm{seq}(\mathrm{seq}\ Info) \\
size : tags \nrightarrow \mathbb{N}_1 \\
committed : \mathbb{P}\ tags \\
usage : \mathbb{N} \\
\hline
\mathrm{dom}\ assoc = \mathrm{dom}\ size \\
committed \subseteq \mathrm{dom}\ assoc \land (\forall t : tags \mid t \in committed \bullet assoc\ t \neq \langle\,\rangle) \\
\forall t : tags \mid t \in \mathrm{dom}\ assoc \bullet \\
\quad \#(assoc\ t) \leqslant maxgen \land \\
\quad \forall i : \mathbb{N}_1 \mid 1 \leqslant i \leqslant \#(assoc\ t) \bullet \\
\quad\quad \#(assoc\ t\ i) = size\ t \\
usage = \big(\sum t : tags \mid t \in \mathrm{dom}\ assoc \bullet \#(assoc\ t) \times size\ t\big) \\
usage \leqslant msize \\
\hline
\end{array}
$$

The *assoc* function associates a tag with a sequence of sequences of information, the most recent generation is at the head of the sequence. The *size* function gives the length of the information sequences associated with a tag. The *committed* set records those tags whose most recent generation of information has been committed. We require the two functions to have the same domain, the committed set to be a subset of this set and all the information sequences associated with a tag to be of the length given by the *size* function. Finally, we require that the total amount of information associated with all the tags should not exceed the size of the memory system.

We need to describe the initial state of the memory system:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AInitialMemSys \quad}} \\
AMemSys \\
\hline
\mathrm{dom}\ assoc = \varnothing \\
\hline
\end{array}
$$

We simply require the *assoc* function to have an empty domain.

We have an operation that returns an unused tag, we have chosen to specify the size of the information sequences we expect to write to the tag as an argument to this operation instead of letting the first write determine the sequence length:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad ANewTag \quad}} \\
\Delta AMemSys \\
n? : \mathbb{N}_1 \\
t! : tags \\
\hline
t! \notin \mathrm{dom}\ assoc \\
assoc' = assoc \cup \{t! \mapsto \langle\,\rangle\} \\
size' = size \cup \{t! \mapsto n?\} \\
committed' = committed \\
\hline
\end{array}
$$

We return an unused tag (one that has no associated sequence of information sequences), record the expected length of the information sequences and mark the most recent generation as We have an operation to read the information sequence, of a given generation, associated with a tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AReadGeneration \quad}} \\
\Xi AMemSys \\
t? : tags \\
g? : \mathbb{N} \\
info! : \mathrm{seq}\ Info \\
\hline
t? \in \mathrm{dom}\ assoc \land assoc\ t \neq \langle\,\rangle \land g \leqslant (\#(assoc\ t) - 1) \\
info! = assoc\ t\ (g + 1) \\
\hline
\end{array}
$$

The tag must have an associated information sequence of the given generation, numbered relative to the current generation.

We have a schema that constrains a generation argument to the current generation:

$$\begin{array}{|l}
\hline
\textit{CurrentGeneration} \\
g? : \mathbb{N} \\
\hline
g? = 0 \\
\hline
\end{array}$$

Using schema conjunction and hiding we can specify an operation that reads the current generation of information associated with a tag:

$$\textit{ARead} \mathrel{\widehat{=}} (\textit{AReadGeneration} \land \textit{CurrentGeneration}) \setminus (g?)$$

We have an operation that releases all the information associated with a tag:

$$\begin{array}{|l}
\hline
\textit{ARelease} \\
\Delta \textit{AMemSys} \\
t? : \textit{tags} \\
\hline
t? \in \operatorname{dom} \textit{assoc} \\
\textit{assoc}' = \{t?\} \mathbin{\lhd\!\!\!-} \textit{assoc} \\
\textit{size}' = \{t?\} \mathbin{\lhd\!\!\!-} \textit{size} \\
\textit{committed}' = \textit{committed} \setminus \{t?\} \\
\hline
\end{array}$$

We simply remove the tag from the domains of the two functions and from the committed set.

We have an operation that commits the current generation of information associated with a tag:

$$\begin{array}{|l}
\hline
\textit{ACommit} \\
\Delta \textit{AMemSys} \\
t? : \textit{tags} \\
\hline
t? \in \operatorname{dom} \textit{assoc} \land \textit{assoc}\ t? \neq \langle\,\rangle \\
\textit{committed}' = \textit{committed} \cup \{t?\} \\
\hline
\end{array}$$

The tag must have an associated information sequence, which we mark as committed.

We have an operation that writes a sequence of information to a tag. This operation has a number of different cases depending on the state of the sequence of generations associated with the tag and whether the current generation has been committed.

The first write to a tag, after $\textit{ANewTag}$, must make sure there is enough room to write the new information:

$$\begin{array}{|l}
\hline
\textit{AWriteFirst} \\
\Delta \textit{AMemSys} \\
t? : \textit{tags} \\
\textit{info}? : \operatorname{seq} \textit{Info} \\
\hline
t? \in \operatorname{dom} \textit{assoc} \land \#\textit{info}? = \textit{size}\ t? \\
\textit{assoc}\ t? = \langle\,\rangle \\
(\textit{usage} + \#\textit{info}?) \leqslant \textit{msize} \\
\textit{assoc}' = \textit{assoc} \oplus \{t? \mapsto \{1 \mapsto \textit{info}?\}\} \\
\textit{size}' = \textit{size} \\
\textit{committed}' = \textit{committed} \\
\hline
\end{array}$$

We override the association for the tag with a singleton sequence containing the new information sequence.

Writing to a tag whose current generation is not committed doesn't need any extra room[1]:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AWriteUncommitted \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta AMemSys \\
t? : tags \\
info? : \mathrm{seq}\, Info \\
\hline
t? \in \mathrm{dom}\, assoc \wedge \# info? = size\ t? \\
assoc\ t? \neq \langle\rangle \\
t? \notin committed \\
assoc' = assoc \oplus \{t? \mapsto (assoc\ t?\ \oplus \{1 \mapsto info?\})\} \\
size' = size \\
committed' = committed
\end{array}
$$

We override the association for the tag with a new sequence of information sequences obtained by overriding its first element with the new information sequence.

Writing to a tag whose current generation has been committed requires extra room for the new information:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad AWriteCommitted \quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad}} \\
\Delta AMemSys \\
t? : tags \\
info? : \mathrm{seq}\, Info \\
\hline
t? \in \mathrm{dom}\, assoc \wedge \# info? = size\ t? \\
assoc\ t? \neq \langle\rangle \\
t? \in committed \\
(usage + \# info?) \leqslant msize \\
assoc' = assoc \oplus \{t? \mapsto (1 \mathrel{..} maxgen \lhd (\{1 \mapsto info?\} \frown assoc\ t?))\} \\
size' = size \\
committed' = committed \setminus \{t?\}
\end{array}
$$

We again override the association for the tag with a sequence of information sequences obtained from the current value. In this case the sequence of sequences is obtained by concatenating the new seqeunce in front of the existing one and then cropping the sequences of sequences by the maximum allowed generation.

We can now use schema disjunction to specify the write operation:

$$AWrite \mathrel{\widehat{=}} AWriteFirst \vee AWriteUncommitted \vee AWriteCommitted$$

4.3 Concrete Specification

In the concrete specification the functions of the abstract specification are now implicit in the lower level functions that describe locations and the pages associated with them.

We introduce the following abbreviation and values to simulate a boolean type:

$$\mathbb{B} == \mathbb{N}$$

$$
\begin{array}{|l}
true : \mathbb{B} \\
false : \mathbb{B} \\
\hline
true \neq false
\end{array}
$$

[1] This may not capture the desired behaviour but can easily be changed.

We will associate a piece of housekeeping data with each tag, this will record whether it is in use, the length of information sequences associated with it, whether its current generation of information has been committed, the number of generations currently associated with it and the index of the current generation:

$$TagData == \mathbb{B} \times \mathbb{N}_1 \times \mathbb{B} \times \mathbb{N} \times \mathbb{N}$$

We adopt a peculiar generation numbering scheme that is, apparently, that used in the real implementation.

We introduce some projection operations over this type:

$$\begin{array}{|l}
tdu : TagData \to \mathbb{B} \\
tds : TagData \to \mathbb{N}_1 \\
tdc : TagData \to \mathbb{B} \\
tdg : TagData \to \mathbb{N} \\
tdx : TagData \to \mathbb{N} \\
\hline
tdu\ (inuse, size, committed, generations, genindex) = inuse \\
tds\ (inuse, size, committed, generations, genindex) = size \\
tdc\ (inuse, size, committed, generations, genindex) = committed \\
tdg\ (inuse, size, committed, generations, genindex) = generations \\
tdx\ (inuse, size, committed, generations, genindex) = genindex
\end{array}$$

We introduce a new constraint that is required by the generation indexing scheme:

$$\begin{array}{|l}
maxindex : \mathbb{N} \\
\hline
maxindex \geqslant maxgen
\end{array}$$

The following abbreviation will be useful:

$$Indices == 0 \mathrel{..} maxindex$$

The abstract association between tags and sequences of sequences of information is now implicit in a function from locations to pages. We first introduce some useful abbreviations:

$$\begin{array}{l}
Loc == 0 \mathrel{..} (msize - 1) \\
Page == Info \times \mathbb{B} \times tags \times \mathbb{N} \times \mathbb{N} \\
Memory == Loc \to Page
\end{array}$$

The memory will consist of $msize$ locations each of which addresses a page. Each page contains a piece of information and housekeeping data that records whether it is in use, the tag it is associated with, its generation index for that tag and its page number (relative position in the information sequence) for that tag.

We again introduce some projection operations:

$$\begin{array}{|l}
pi : Page \to Info \\
pu : Page \to \mathbb{B} \\
pt : Page \to tags \\
px : Page \to \mathbb{N} \\
pn : Page \to \mathbb{N} \\
\hline
pi\ (info, inuse, tag, genindex, pageno) = info \\
pu\ (info, inuse, tag, genindex, pageno) = inuse \\
pt\ (info, inuse, tag, genindex, pageno) = tag \\
px\ (info, inuse, tag, genindex, pageno) = genindex \\
pn\ (info, inuse, tag, genindex, pageno) = pageno
\end{array}$$

We have an operation that takes a memory and returns all the locations whose pages are associated with a given tag:

$$\begin{array}{|l}
taglocs : tags \times Memory \to \mathbb{P}\ Loc \\
\hline
taglocs\ (tag, mem) = \\
\quad \{ l : Loc \mid pu\ (mem\ l) = true \land pt\ (mem\ l) = tag \}
\end{array}$$

We need only consider those pages that are marked as in use.

Given this we can specify an operation that takes a memory and returns a set of sets of locations. The pages of the locations in each set of will have the same generation index:

$$
\begin{array}{|l}
\mathit{gentaglocs} : \mathit{tags} \times \mathit{Memory} \rightarrow \mathbb{P}(\mathbb{P}\,\mathit{Loc}) \\
\hline
\exists\, \mathit{lsets} : \mathbb{P}(\mathbb{P}\,\mathit{Loc}) \mid \mathit{lsets} = \mathit{gentaglocs}\ (\mathit{tag}, \mathit{mem}) \bullet \\
\quad \bigcup \mathit{lsets} = \mathit{taglocs}\ (\mathit{tag}, \mathit{mem}) \land \\
\quad \forall\, \mathit{lset} : \mathbb{P}\,\mathit{Loc} \mid \mathit{lset} \in \mathit{lsets} \bullet \\
\qquad \forall\, l_1, l_2 : \mathit{Loc} \mid l_1 \in \mathit{lset} \land l_2 \in \mathit{lset} \bullet \\
\qquad\quad \mathit{px}\ (\mathit{mem}\ l_1) = \mathit{px}\ (\mathit{mem}\ l_2)
\end{array}
$$

We can also specify an operation that returns the generation indices associated with a tag:

$$
\begin{array}{|l}
\mathit{tagindices} : \mathit{tags} \times \mathit{Memory} \rightarrow \mathbb{P}\,\mathbb{N} \\
\hline
\mathit{tagindices}\ (\mathit{tag}, \mathit{mem}) = \\
\quad \{ l : \mathit{Loc} \mid \mathit{pu}\ (\mathit{mem}\ l) = \mathit{true} \land \mathit{pt}\ (\mathit{mem}\ l) = \mathit{tag} \bullet \mathit{px}\ (\mathit{mem}\ l) \}
\end{array}
$$

Before we can specify the concrete memory system we must first characterise the acceptable sets of generation indices that may be associated with a tag. This (partial) function returns the oldest and newest index in an acceptable, non-empty, index set. An acceptable index set is contiguous modulo the fact that it may wrap around at *maxindex*.

$$
\begin{array}{|l}
\mathit{indices} : \mathbb{P}\,\mathbb{N} \rightarrow\!\!\!\!\!+ \mathbb{N} \times \mathbb{N} \\
\hline
\mathit{indices} = \\
\quad \{ i : \mathbb{P}\,\mathbb{N};\ \mathit{old}, \mathit{new} : \mathbb{N} \mid \\
\qquad i \subseteq \mathit{Indices} \land \mathit{old} \in \mathit{Indices} \land \mathit{new} \in \mathit{Indices} \land \\
\qquad (\mathit{old} \leqslant \mathit{new} \land i = \mathit{old} \mathinner{..} \mathit{new} \lor \\
\qquad\ \mathit{old} > \mathit{new} \land i = 0 \mathinner{..} \mathit{new} \cup \mathit{old} \mathinner{..} \mathit{maxindex}) \bullet \\
\qquad\quad i \mapsto (\mathit{old}, \mathit{new}) \\
\quad \}
\end{array}
$$

The concrete memory system is specified by two functions, we include two derived values to aid the presentation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{CMemSys}} \\
\hline
data : tags \rightarrow TagData \\
mem : Memory \\
freetags : \mathbb{P}\, tags \\
freelocs : \mathbb{P}\, Loc \\
\hline
\forall t : tags \mid taglocs\ (t, mem) \neq \varnothing \bullet tdu\ (data\ t) = true \\
\forall t : tags \mid tdu\ (data\ t) = true \bullet \\
\quad tdg\ (data\ t) = \#glocs \wedge tdg\ (data\ t) \leqslant maxgen \\
\qquad \wedge \\
\quad tdc\ (data\ t) = true \Rightarrow tdg\ (data\ t) > 0 \\
\qquad \wedge \\
\quad tdg\ (data\ t) > 0 \Rightarrow \\
\qquad (\exists\, old : \mathbb{N} \bullet indices\ tidxs = (old, tdx\ (data\ t))) \\
\qquad \wedge \\
\quad \forall\, lset : \mathbb{P}\, Loc \mid lset \in glocs \bullet \\
\qquad \#lset = tds\ (data\ t) \wedge \\
\qquad \{l : Loc \mid l \in lset \bullet pn\ (mem\ l) + 1\} = 1 \mathrel{..} \#lset \\
\quad \textbf{where} \\
\quad glocs = gentaglocs\ (t, mem) \\
\quad tidxs = tagindices\ (t, mem) \\
freetags = \{t : tags \mid tdu\ (data\ t) \neq true\} \\
freelocs = \{l : Loc \mid pu\ (mem\ l) \neq true\} \\
\hline
\end{array}
$$

The conditions over these two functions are more complex than in the abstract specification. This is partly due to the consistency conditions required by the relationship between the two functions. The greatest complexity arises from the conditions on the *memory* required to make it behave as a representation of a $\mathrm{seq}(\mathrm{seq}\ Info)$. The generation indexing scheme introduces even more complexity. The only simplification is that the memory size constraint is clearly satisfied by the *memory* function.

The data refinement from the abstract to concrete specifications is specified by the following *Abstraction* schema:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{Abstraction}} \\
\hline
AMemSys \\
CMemSys \\
\hline
assoc = \\
\quad \{t : tags \mid tdu\ (data\ t) = true \bullet \\
\qquad t \mapsto \{g : \mathbb{N};\ glocs : \mathbb{P}(\mathbb{P}\, Loc);\ tidxs : \mathbb{P}\,\mathbb{N} \mid \\
\qquad\qquad glocs = gentaglocs\ (t, mem) \wedge \\
\qquad\qquad tidxs = tagindices\ (t, mem) \wedge \\
\qquad\qquad g \leqslant tdg\ (data\ t) \bullet \\
\qquad\qquad g \mapsto \{lset : \mathbb{P}\, Loc;\ l : Loc \mid \\
\qquad\qquad\qquad lset \in glocs \wedge \\
\qquad\qquad\qquad l \in lset \wedge \\
\qquad\qquad\qquad offset\ (tidxs, px\ (meml)) = g \bullet \\
\qquad\qquad\qquad\quad pn\ (mem\ l) + 1 \mapsto pi\ (mem\ l) \\
\qquad\qquad\qquad \} \\
\qquad\qquad \} \\
\quad \} \\
\forall t : tags \mid t \in \mathrm{dom}\, size \bullet size\ t = tds\ (data\ t) \\
\forall t : tags \mid t \in committed \bullet tdc\ (data\ t) = true \\
\hline
\end{array}
$$

The data refinement above requires some functions over generation indices. We can increment and decrement generation indices:

$$
\begin{array}{|l}
incrindex : Indices \rightarrow Indices \\
decrindex : Indices \rightarrow Indices \\
\hline
incrindex = \{i, j : Indices \mid i = maxindex \land j = 0 \lor \\
\qquad\qquad\qquad\qquad\qquad i \neq maxindex \land j = i + 1 \bullet \\
\qquad\qquad\qquad\qquad\qquad i \mapsto j\} \\
decrindex = \{i, j : Indices \mid i = 0 \land j = maxindex \lor \\
\qquad\qquad\qquad\qquad\qquad i \neq 0 \land j = i - 1 \bullet \\
\qquad\qquad\qquad\qquad\qquad i \mapsto j\}
\end{array}
$$

Given these functions we can specify a function that takes a generation index and returns the offset of that generation relative to the newest generation:

$$
\begin{array}{|l}
offset : \mathbb{P}\, Indices \times Indices \rightarrow\!\!\!\!\!\!+ \ \mathbb{N} \\
\hline
offset = \\
\quad \{i : \mathbb{P}\, Indices;\ x : Indices;\ o : \mathbb{N} \mid \\
\qquad x \in i \land \\
\qquad (\exists\, old, new : \mathbb{N} \bullet \\
\qquad\quad indices\ i = (old, new) \land \\
\qquad\quad x = new \land o = 0 \lor \\
\qquad\quad x \neq new \land o = offset\ (i, incrindex\ x) + 1) \bullet \\
\qquad (i, x) \mapsto o \\
\quad \}
\end{array}
$$

The initial state of the memory system is:

$$
\begin{array}{|l}
\underline{\ CInitialMemSys\ } \\
CMemSys \\
\hline
freetags = tags \\
freelocs = Loc \\
\hline
\end{array}
$$

There are no used tags or memory locations. The remaining fields in the tag data and pages take arbitrary initial values.

We can return an unused tag:

$$
\begin{array}{|l}
\underline{\ CNewTag\ } \\
\Delta CMemSys \\
n? : \mathbb{N}_1 \\
t! : tags \\
\hline
t! \in freetags \\
data' = data \oplus \{t! \mapsto (true, n?, false, 0, tdx\ (data\ t!))\} \\
mem' = mem \\
freetags' = freetags \setminus \{t!\} \\
freelocs' = freelocs \\
\hline
\end{array}
$$

This simply updates the tag data, the memory remains unchanged.

We can read the information sequence, of a given generation, associated with a tag:

$$
\begin{array}{l}
\textit{CReadGeneration} \\
\hline
\Xi \mathit{CMemSys} \\
t? : \mathit{tags} \\
g? : \mathbb{N} \\
\mathit{info}! : \mathrm{seq}\, \mathit{Info} \\
\hline
\mathit{tdu}\ (\mathit{data}\ t?) = \mathit{true} \land \mathit{tdg}\ (\mathit{data}\ t?) \neq 0 \land g? < \mathit{tdg}\ (\mathit{data}\ t?) \\
\mathit{info}! = \\
\quad \{\mathit{lset} : \mathbb{P}\, \mathit{Loc};\ l : \mathit{Loc} \mid \\
\quad\quad \mathit{lset} \in \mathit{gentaglocs}\ (t?, \mathit{mem}) \land \\
\quad\quad l \in \mathit{lset} \land \\
\quad\quad \mathit{offset}\ (\mathit{tagindices}\ (t?, \mathit{mem}), \mathit{px}\ (\mathit{mem}\ l)) = g? \bullet \\
\quad\quad\quad \mathit{pn}\ (\mathit{mem}\ l) + 1 \mapsto \mathit{pi}\ (\mathit{mem}\ l) \\
\quad \}
\end{array}
$$

Using schema conjunction and hiding we can specify an operation that reads the current generation of information associated with a tag:

$$\mathit{CRead} \mathrel{\widehat=} (\mathit{CReadGeneration} \land \mathit{CurrentGeneration}) \setminus (g?)$$

We can release all the information associated with a tag:

$$
\begin{array}{l}
\textit{CRelease} \\
\hline
\Delta \mathit{CMemSys} \\
t? : \mathit{tags} \\
\hline
\mathit{tdu}\ (\mathit{data}\ t?) = \mathit{true} \\
\mathit{data}' = \mathit{data} \oplus \{t? \mapsto (\mathit{false}, \\
\qquad\qquad\qquad\qquad \mathit{tds}\ (\mathit{data}\ t?), \\
\qquad\qquad\qquad\qquad \mathit{tdc}\ (\mathit{data}\ t?), \\
\qquad\qquad\qquad\qquad \mathit{tdg}\ (\mathit{data}\ t?), \\
\qquad\qquad\qquad\qquad \mathit{tdx}\ (\mathit{data}\ t?))\} \\
\mathit{mem}' = \mathit{release}\ (\mathit{mem}, \mathit{taglocs}\ (t?, \mathit{mem})) \\
\mathit{freetags}' = \mathit{freetags} \cup \{t?\} \\
\mathit{freelocs}' = \mathit{freelocs} \cup \mathit{taglocs}\ (t?, \mathit{mem})
\end{array}
$$

This uses the following operation to release the locations associated with the tag:

$$
\begin{array}{l}
\mathit{release} : \mathit{Memory} \times \mathbb{P}\, \mathit{Loc} \to \mathit{Memory} \\
\hline
\mathit{release}\ (\mathit{mem}, \mathit{lset}) = \mathit{mem} \oplus \{l : \mathit{Loc} \mid l \in \mathit{lset} \bullet \\
\qquad\qquad\qquad\qquad\qquad l \mapsto (\mathit{pi}\ (\mathit{mem}\ l), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{false}, \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pt}\ (\mathit{mem}\ l), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{px}\ (\mathit{mem}\ l), \\
\qquad\qquad\qquad\qquad\qquad\quad \mathit{pn}\ (\mathit{mem}\ l)) \\
\qquad\qquad\qquad\qquad \}
\end{array}
$$

We can commit the current generation of information associated with a tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;CCommit\;}} \\
\Delta CMemSys \\
t? : tags \\
\hline
tdu\ (data\ t?) = true \land tdg\ (data\ t) > 0 \\
data' = data \oplus \{t? \mapsto (tdu\ (data\ t?), \\
\qquad\qquad\qquad\qquad tds\ (data\ t?), \\
\qquad\qquad\qquad\qquad true, \\
\qquad\qquad\qquad\qquad tdg\ (data\ t?), \\
\qquad\qquad\qquad\qquad tdx\ (data\ t?))\} \\
mem' = mem \\
freetags' = freetags \\
freelocs' = freelocs \\
\hline
\end{array}
$$

We can perform the first write to a tag after $CNewTag$:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\;CWriteFirst\;}} \\
\Delta CMemSysy \\
t? : tags \\
info? : \mathrm{seq}\ Info \\
\hline
tdu\ (data\ t?) = true \land tds\ (data\ t?) = \# info? \\
tdg\ (data\ t?) = 0 \\
\# freelocs \geqslant \# info? \\
data' = data \oplus \{t? \mapsto (tdu\ (data\ t?), \\
\qquad\qquad\qquad\qquad tds\ (data\ t?), \\
\qquad\qquad\qquad\qquad false, \\
\qquad\qquad\qquad\qquad 1, \\
\qquad\qquad\qquad\qquad 0)\} \\
\exists\, lset : \mathbb{P}\, Loc \mid lset \subseteq freelocs \land \# lset = \# info? \bullet \\
\qquad mem' = update\ (mem, lset, info?, t?, 0) \land \\
\qquad freelocs' = freelocs \setminus lset \\
freetags' = freetags \\
\hline
\end{array}
$$

This uses the following function to update the set of locations newly associated with the tag:

$$
\begin{array}{|l}
update : Memory \times \mathbb{P}\, Loc \times \mathrm{seq}\ Info \times tags \times \mathbb{N} \to Memory \\
\hline
\forall\, mem : Memory;\ lset : \mathbb{P}\, Loc;\ info : \mathrm{seq}\ Info;\ t : tags;\ x : \mathbb{N} \bullet \\
\qquad \exists\, mem' : \mathbb{P}\, Loc \to Page \mid \mathrm{dom}\ mem' = lset \bullet \\
\qquad\qquad \forall\, n : 1 \mathrel{..} \# info \bullet \\
\qquad\qquad\qquad \exists\, l : Loc \mid l \in lset \bullet \\
\qquad\qquad\qquad\qquad pi\ (mem'\ l) = info\ n \land \\
\qquad\qquad\qquad\qquad pu\ (mem'\ l) = true \land \\
\qquad\qquad\qquad\qquad pt\ (mem'\ l) = t \land \\
\qquad\qquad\qquad\qquad px\ (mem'\ l) = x \land \\
\qquad\qquad\qquad\qquad pn\ (mem'\ l) = n - 1 \\
\qquad\qquad \land \\
\qquad\qquad update\ (mem, lset, info, t, x) = mem \oplus mem'
\end{array}
$$

We write to a tag whose current generation is uncommitted:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CWriteUncommitted \quad}} \\
\Delta CMemSysy \\
t? : tags \\
info? : \mathrm{seq}\, Info \\
\hline
tdu\ (data\ t?) = true \land tds\ (data\ t?) = \#info? \\
tdg\ (data\ t?) > 0 \\
tdc\ (data\ t?) \neq true \\
data' = data \\
\exists\, lset : \mathbb{P}\, Loc \mid lset \in gentaglocs\ (t?, mem) \land \\
\qquad\qquad (\forall\, l : Loc \mid l \in lset \bullet px\ (mem\ l) = tdx\ (data\ t?)) \bullet \\
\qquad mem' = update(mem, lset, info?, t?, tdx\ (data\ t?)) \\
freetags' = freetags \\
freelocs' = freelocs \\
\hline
\end{array}
$$

When we write to a tag whose current generation has been committed we add a new generation to those associated with the tag. If this results in more than the maximum allowed number of generations the oldest generation must be dropped.

If we have not reached the maximum allowed number of generations we needn't drop the oldest generation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CWriteCommittedAddGen \quad}} \\
\Delta CMemSysy \\
t? : tags \\
info? : \mathrm{seq}\, Info \\
\hline
tdu\ (data\ t?) = true \land tds\ (data\ t?) = \#info? \\
tdg\ (data\ t?) > 0 \land tdg\ (data\ t?) < maxgen \\
tdc\ (data\ t?) = true \\
\#freelocs \geqslant \#info? \\
data' = data \oplus \{t? \mapsto (tdu\ (data\ t?), \\
\qquad\qquad\qquad\qquad tds\ (data\ t?), \\
\qquad\qquad\qquad\qquad tdc\ (data\ t?), \\
\qquad\qquad\qquad\qquad tdg\ (data\ t?) + 1, \\
\qquad\qquad\qquad\qquad incrindex\ (tdx\ (data\ t?)))\} \\
\exists\, lset : \mathbb{P}\, Loc \mid lset \subseteq freelocs \land \#lset = \#info? \bullet \\
\qquad mem' = update\ (mem, lset, info?, t?, incrindex\ (tdx\ (data\ t?))) \land \\
\qquad freelocs' = freelocs \setminus lset \\
freetags' = freetags \\
\hline
\end{array}
$$

If we have reached the maximum allowed number of generations we must drop the oldest generation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad CWriteCommittedMaxGen \quad}} \\
\Delta CMemSysy \\
t? : tags \\
info? : \mathrm{seq}\ Info \\
\hline
tdu\ (data\ t?) = true \land tds\ (data\ t?) = \#info? \\
tdg\ (data\ t?) = maxgen \\
tdc\ (data\ t?) = true \\
\#freelocs \geqslant \#info? \\
data' = data \oplus \{t? \mapsto (tdu\ (data\ t?), \\
\qquad\qquad\qquad\qquad tds\ (data\ t?), \\
\qquad\qquad\qquad\qquad tdc\ (data\ t?), \\
\qquad\qquad\qquad\qquad tdg\ (data\ t?), \\
\qquad\qquad\qquad\qquad incrindex\ (tdx\ (data\ t?)))\} \\
\exists\, old, new : \mathbb{N};\ oldlset : \mathbb{P}\, Loc;\ mem'' : Memory;\ lset : \mathbb{P}\, Loc \mid \\
\quad (old, new) = indices\ (tagindices\ (t?, mem)) \land \\
\quad oldlset = \{l : Loc \mid l \in taglocs\ (t?, mem) \land px\ (mem\ l) = old\} \land \\
\quad mem'' = release\ (mem, oldlset) \land \\
\quad lset \subseteq freelocs \land \#lset = \#info? \bullet \\
\qquad mem' = update\ (mem'', lset, info?, t?, incrindex\ (tdx\ (data\ t?))) \land \\
\qquad freelocs' = freelocs \setminus lset \\
freetags' = freetags \\
\hline
\end{array}
$$

We can now use schema disjunction to specify the write operation:

$$CWrite \mathrel{\widehat{=}} CWriteFirst \lor CWriteUncommitted \lor CWriteCommittedNewGen \lor CWriteCommittedMaxGen$$

4.4 Conclusion and Future Work

This specification demonstrates how we can refine an abstract specification into a concrete specification that is fairly close to a real implementation. It does, however, reveal that an implementation of sequences via the data in the pages of the memory requires a complex specification. This specification successfully captured our current understanding of the real implementation of the tagged memory system and allowed us to discuss it at length with Eduard de Jong. This discussion revealed a number of ways in which the implementation differed from the specification and suggested future work on the specification:

- The relationships between the *CNewTag* and *CWrite* operations must be refined to capture the correct allocation of locations to a tag.
- As it stands this specification is too strong as it assumes that the concrete operations are atomic. We must introduce a further level of refinement in terms of the truly atomic operations: writing a page and reading a byte.
- Once we accept that the concrete operations used in this specification are not atomic we must then allow for the memory system being in states not allowed by this specification. This means we must include detection of these non-standard states allowing recovery or error reporting.
- If we weaken the concrete specification to allow error reporting we must rewrite the abstract specification in a similar fashion.
- We would like to be able to extend the basic specification in a modular way to encompass these enhancements.

Section 5

Z Specification of Memory Module Design

We describe a further refinement of the *Concrete* specification of the memory management system. This refinement introduces the atomic operations over pages in terms of which all operations must be described. We also describe the new error states that may arise when a sequence of atomic operations is interrupted. By elaborating the housekeeping data stored in the pages in memory we are able to constrain these error states and ensure that any "*lost*" pages can subsequently be reclaimed.

5.1 Introduction

In figure 5.1 we describe the various specifications of the memory management system and the documents in which they appear. The initial *Abstract* and *Concrete* specifications appeared in 2 of the first deliverable.

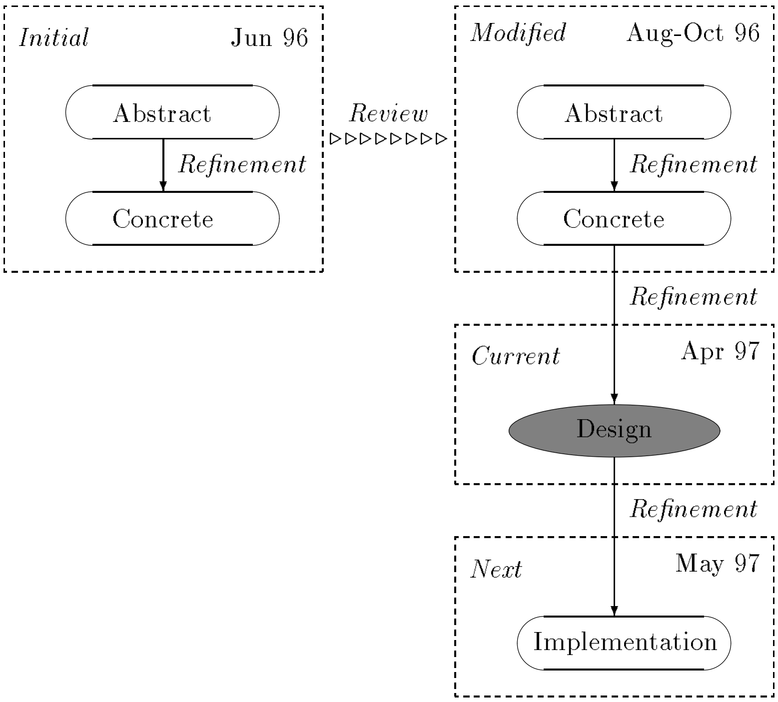

Figure 5.1: **Z** Specifications

Through a process of review and discussion with Eduard de Jong these were modified to produce the versions that appeared in 4 of that document. This document describes the *Design* level specification of the memory management system, which is a refinement of the *Concrete* specification. This document is also the product of an ongoing process of review and discussion with Eduard de Jong. This specification will itself be refined into the *Implementation* specification which will appear in a future document.

This specification introduces some of the low-level implementation details corresponding to the real hardware/software as described by Eduard de Jong. The inclusion of these implementation details gave rise to new error states which were not considered in the abstract and concrete specifications. This could have required the modification of these specifications to include these new error states. However, by careful specification of the handling of these new error states in this specification we were able to avoid the need for changes to the abstract and concrete specifications. The new implementation details are described below:

- Repeated updates of the same memory location cannot be allowed. The major consequence of is that we must simplify the data associated with each tag in the *TagData* mapping in order to remove values that can be calculated from the memory.
- The atomic operations over the memory are page based and are sequenced.
- The sequences of atomic operations may be interrupted at any point. This means that the memory can be left in error states not found during normal operation.

5.2 Sequences of Atomic Operations

In the actual implementation the basic operations over the memory are the writing of a page into tagged memory and the reading of a byte from tagged memory. The number of bytes in each page will be constant but may vary from implementation to implementation. We can assume that these atomic operations must either succeed or fail as this requirement can easily be satisfied by the actual implementation. Sequences of atomic operations can, however, be interrupted at any point by the removal of the card from its power supply. If a sequence of atomic write operations is interrupted the tagged memory can be left a state that is not captured by the concrete specification. This design specification describes all operations that write to the memory in terms of sequences of the atomic page writing operation that may be interrupted at any point. This design specification must capture the new error states that arise, allow for their subsequent detection and hopefully support recovery from these erroneous states. We do not describe operations that read from the memory in terms of sequences of atomic read operations as their interruption cannot introduce new states of the memory.

5.3 Handling Erroneous States

There are two different ways to handle erroneous states. The first approach I considered was to modify both the concrete and abstract specifications to allow for such erroneous states. The design specification could then simply allow such states but avoid discussing how they might be handled. The problem with this approach is that while error states can be detected, by the absence or duplication of pages, there is no way to recognise the cause of the error and therefore no way to perform error recovery. To solve this problem the memory manager would have to record some indication of its current state in the memory in such a way as to allow for subsequent error recovery. My attempts to specify the recording of such a state in a form that relates to the memory operations as seen by an application all required repeated writes of the state information to some page in memory.

This problem, along with the need to modify the abstract and concrete specifications, lead me to investigate an approach in which all the error detection and recovery could be contained within the new specification and hidden at some level within the final implementation of the system. This is method I adopted and which is described in the remainder of this paper.

5.4 Specification Constraints

There are a number of new constraints that I used as goals when designing the new specification. The first constraint was actually the motivation for the development of the tagged memory management system. However, the abstract and concrete specifications did not take this into account and in that sense it is new in this specification:

- We should write to a given page as few times as possible. This basically means that we should only write to a page when we have no choice:
  - When writing new pages of information.
  - When superseding pages of information.
  - When removing an association between a page and a tag.

  All the information required to track the state of the memory manager should be stored utilising only these write operations.

I was not certain that I would be able to satisfy all these constraints but in the event I believe I have succeeded in doing so while imposing only a slight memory cost on the memory manager.

- Memory is limited so the memory management system should use as little as possible itself.
- The only write operation we may perform on the memory is the atomic writing of a page.
- The card can be pulled at any time, thus any sequence of atomic write operations can be interrupted at any point. We should be able to detect the resulting erroneous state and then tidy up the memory.
- We should ensure that we can always recover the memory lost when an atomic operation sequence is interrupted. This is because one of the major results of verification will the memory boundedness of programs which will only hold if we can recover lost memory.
- We, of course, retain all the constraints employed in the previous specifications, such as requiring that the information read from a tag is equal to that previously written to that tag.

5.5 Error States

There are four contexts in which a sequence of atomic operations can be interrupted to give rise to a distinct error state:

- When writing a new generation of information we may fail to write all the required pages.
- When writing a new version of the current generation we may fail to write all the pages of the new version or to supersede all the pages of the old version.
- When releasing the pages of an old generation in order to provide space for a new generation we may fail to release all the pages of the old generation.
- When deallocating all the pages for a tag for the `Release` operation we may fail to deallocate them all.

We can't record a separate flag to track the current state of the memory manager for a tag as we would have to pick a page to keep it in which would then suffer from repeated writes as the state changed. Instead we use the prescence of page zero to indicate the prescence of all the other pages of a generation, as described by Eduard de Jong, and elaborate the information otherwise stored in a page by a further piece of data:

- A cyclic three state flag that allows us to determine the relative age of two versions of the same generation. Each page in a given version will have the same value in this flag, the pages of a new version will all take the successor state to that of the current version.

5.6 Error Recovery

As checking for and remedying error states before each operation would be expensive we instead wait until we have no choice but reclaim the memory lost due to disrupted operations. Thus the prescence of an error state in the memory manager will be noted by a `Write` operation failing to find sufficient free pages. We can then invoke an operation to tidy up the memory, releasing the lost pages for reuse. By performing some inexpensive local housekeeping in the operations we can restrict the complexity of the error states that can arise from repeated disruptions. This greatly simplifies the error recovery task. We describe the different forms of error recovery, how they are tidied up, the error states that invoke them and the housekeeping required of the operations:

1. If there are pages marked as in use by a tag but the tag data does not mark it as in use they can all be marked as not in use. This will only occur due to a disruption while releasing all the information associated with a tag. The `New` operation are required to tidy up any pages marked as in use by the new tag.

2. If there are pages for a given generation and version with no page zero they can all be marked as not in use. This will occur due to disruptions while writing new generations and versions and while superseding old versions and releasing old generations. The `Commit` and `Write` operations are required to tidy up incomplete versions and generations for the given tag.

3. If there are two complete sets of pages for a tag with the same generation the pages of the older version can be marked as not in use. This will only occur due to a disruption while writing a new version of the current generation. The `Commit` and `Write` operations are required to tidy up out-of-date versions for the given tag. Given this housekeeping we can ensure that only the current generation can ever have multiple versions.

4. If there are more generations associated with a tag than the maximum allowed then the pages of the oldest generation can be marked as not in use. This will only occur due to a disruption while writing a new generation when the maximum number of generations already exist. The `Write` operation are required to tidy up excessively old generations for the given tag.

Given this localised housekeeping we can easily calculate a conservative estimate of the number of locations currently in use before each `Write` operation. This estimate is conservative in the sense that, in the error states, it may conclude that more locations are in use that in fact are marked as in use. During normal operation this estimate will correspond exactly to the number of pages required by all the tags currently in use. If this estimate indicates that there are not enough free locations we can tidy up the memory, recovering locations lost due to interruption of a memory update, and try again. If there are still not enough free locations this indicates an unrecoverable error due to an application requiring more than the available memory. We make no attempt to handle this error, we instead require the user to avoid calling operations in such a manner as would cause this error. This may well require that memory boundedness constraints are verified for all applications.

This is an instance of a general issue concerning the limits of our specification. We are assuming that certain operations will only be called when it is sensible to do so. This allows us to avoid the additional complexity that would be required in the specifications if we were to consider these additional sources of errors. In a development process involving verification of the use of operations such simplifying assumptions can be formally justified.

5.7 Design Specification

The design specification follows the concrete specification in its general structure and retains some elements from both the abstract and concrete specification. The specification is parameterised by the following types:

$$[Tag, Info]$$

We have a set of tags, a memory size and a maximum generations count:

$$\begin{array}{|l} tags : \mathbb{F}\ Tag \\ msize : \mathbb{N}_1 \\ maxgen : \mathbb{N}_1 \end{array}$$

We have a simulated boolean type:

$$\mathbb{B} == \mathbb{N}$$

$$\begin{array}{|l} true : \mathbb{B} \\ false : \mathbb{B} \\ \hline true \neq false \end{array}$$

We have a constraint on the generation indices:

$$\begin{array}{|l} maxindex : \mathbb{N} \\ \hline maxindex \geqslant maxgen \end{array}$$

We have two useful abbreviations:

$$Indices == 0 \mathrel{..} maxindex$$
$$Loc == 0 \mathrel{..} (msize - 1)$$

We have the following characterisation of acceptable sets of generation indices:

$$\begin{array}{|l} indices : \mathbb{P}\,\mathbb{N} \rightarrow\!\!\!\!\!+ \mathbb{N} \times \mathbb{N} \\ \hline indices = \\ \quad \{ i : \mathbb{P}\,\mathbb{N};\ old, new : \mathbb{N} \mid \\ \quad\quad i \subseteq Indices \land old \in Indices \land new \in Indices \land \\ \quad\quad (old \leqslant new \land i = old \mathrel{..} new \lor \\ \quad\quad\ old > new \land i = 0 \mathrel{..} new \cup old \mathrel{..} maxindex) \bullet \\ \quad\quad\quad i \mapsto (old, new) \\ \quad \} \end{array}$$

In the following I derive the names employed for types and functions from those employed in the concrete specification, prefixing them with a 'D' or 'd'. In some cases I have also shortened the resulting name.

We modify the tag data, removing the generation count and index as these values can be determined from the memory.

$$DTagData == \mathbb{B} \times \mathbb{N}_1 \times \mathbb{B}$$

We have projections over this type:

$$\begin{array}{|l} dtdu : DTagData \rightarrow \mathbb{B} \\ dtds : DTagData \rightarrow \mathbb{N}_1 \\ dtdc : DTagData \rightarrow \mathbb{B} \\ \hline dtdu\ (inuse, size, committed) = inuse \\ dtds\ (inuse, size, committed) = size \\ dtdc\ (inuse, size, committed) = committed \end{array}$$

The abbreviations defining a page in memory are now altered to included the extra state information required by the design specification.

$$DPage == \mathbb{B} \times tags \times Info \times \mathbb{N} \times \mathbb{N} \times Version$$
$$Version ::= V_A \mid V_B \mid V_C$$
$$DMemory == Loc \rightarrow DPage$$

The memory will consist of *msize* locations each of which identifies a page. Each page contains a piece of information and housekeeping data that records whether it is in use, the tag it is associated with, its generation index, its page number and its version. The page versions form a cycle that allows us to order successive versions of of a generation:

$$
\begin{array}{|l}
\_ \prec \_ : Version \leftrightarrow Version \\
nextV : Version \to Version \\
\hline
V_A \prec V_B \land V_B \prec V_C \land V_C \prec V_A \\
nextV\ \ V_A = V_B \\
nextV\ \ V_B = V_C \\
nextV\ \ V_C = V_A
\end{array}
$$

We introduce some projection operations on $DPage$:

$$
\begin{array}{|l}
dpu : DPage \to \mathbb{B} \\
dpt : DPage \to tags \\
dpi : DPage \to Info \\
dpx : DPage \to \mathbb{N} \\
dpn : DPage \to \mathbb{N} \\
dpv : DPage \to Version \\
\hline
dpu\ (inuse, tag, info, genindex, pageno, version) = inuse \\
dpt\ (inuse, tag, info, genindex, pageno, version) = tag \\
dpi\ (inuse, tag, info, genindex, pageno, version) = info \\
dpx\ (inuse, tag, info, genindex, pageno, version) = genindex \\
dpn\ (inuse, tag, info, genindex, pageno, version) = pageno \\
dpv\ (inuse, tag, info, genindex, pageno, version) = version
\end{array}
$$

We have an operation that takes a memory and returns all the locations whose pages are marked as being associated with a given tag. We need only consider those pages that are marked as in use:

$$
\begin{array}{|l}
dtaglocs : tags \times DMemory \to \mathbb{P}\,Loc \\
\hline
dtaglocs\ (tag, mem) = \\
\qquad \{l : Loc \mid dpu\ (mem\ l) = true \land dpt\ (mem\ l) = tag \bullet l\}
\end{array}
$$

We can also specify an operation that returns the generation indices associated with a tag:

$$
\begin{array}{|l}
dtagindices : tags \times DMemory \to \mathbb{P}\,\mathbb{N} \\
\hline
dtagindices\ (tag, mem) = \\
\qquad \{l : Loc \mid l \in dtaglocs\ (tag, mem) \bullet dpx\ (mem\ l)\}
\end{array}
$$

In the following characterisations of the sets of locations corresponding various classes of error we use the numbering scheme from page 24.

As it is possible for the deallocation of pages for a tag to be interrupted we must be able to detect this condition. These are simply the locations that are marked as being in use by a tag that is itself marked as not being in use:

$$
\begin{array}{|l}
badlocs1 : (tags \to DTagData) \times DMemory \to \mathbb{P}(\mathbb{P}\,Loc) \\
\hline
badlocs1\ (data, mem) = \\
\qquad \{lset : \mathbb{P}\,Loc \mid (\exists\, tag : tags \mid lset = dtaglocs\ (tag, mem)) \land \\
\qquad\qquad\qquad\qquad\qquad dtdu\ (data\ tag) = false \bullet lset\}
\end{array}
$$

We can specify an operation that takes a tag and a memory and returns a set of sets of locations. The pages of the locations in each set of will have the same generation index for that tag:

$$
\begin{array}{|l}
gentlocs : tags \times DMemory \to \mathbb{P}(\mathbb{P}\,Loc) \\
\hline
gentlocs\ (tag, mem) = lsets \\
\textbf{where} \\
\qquad \bigcup lsets = \{l : Loc \mid l \in dtaglocs\ (tag, mem) \bullet l\} \land \\
\qquad \forall\, lset : \mathbb{P}\,Loc \mid lset \in lsets \bullet \\
\qquad\qquad \forall\, l_1, l_2 : Loc \mid l_1 \in lset \land l_2 \in lset \bullet \\
\qquad\qquad\qquad dpx\ (mem\ l_1) = dpx\ (mem\ l_2)
\end{array}
$$

We can further refine this by separating the pages of different versions for a given generation:

$$
\begin{array}{|l}
vergtlocs : tags \times DMemory \rightarrow \mathbb{P}(\mathbb{P}(\mathbb{P}\,Loc)) \\
\hline
vergtlocs\ (tag, mem) = lsetss \\
\textbf{where} \\
\quad \{lsets : \mathbb{P}(\mathbb{P}\,Loc) \mid lsets \in lsetss \bullet \bigcup lsets\} = gentlocs\ (tag, mem) \land \\
\quad \forall\, lsets : \mathbb{P}(\mathbb{P}\,Loc);\ lset : \mathbb{P}\,Loc \mid lsets \in lsetss \land lset \in lsets \bullet \\
\qquad \forall\, l_1, l_2 : Loc \mid l_1 \in lset \land l_2 \in lset \bullet \\
\qquad\quad dpv\ (mem\ l_1) = dpv\ (mem\ l_2)
\end{array}
$$

As it is possible for the writing or release of pages for a given generation to be interrupted we must be able to detect this condition. We always write page zero of a generation last and release it first so that its absence indicates an incomplete generation:

$$
\begin{array}{|l}
badlocs2 : DMemory \rightarrow \mathbb{P}(\mathbb{P}\,Loc) \\
\hline
badlocs2\ mem = \\
\quad \{lset : \mathbb{P}\,Loc \mid (\exists\, tag : tags;\ lsets : \mathbb{P}(\mathbb{P}\,Loc) \bullet \\
\qquad\qquad lsets \in vergtlocs\ (tag, mem) \land lset \in lsets) \land \\
\qquad \neg\,(\exists\, l : Loc \bullet l \in lset \land dpn\ (mem\ l) = 0)\}
\end{array}
$$

It is also possible for two complete versions of the same generation to exist at the same time. The version numbers allow us to distinguish the locations of the two versions and determine the out-of-date version:

$$
\begin{array}{|l}
badlocs3 : DMemory \rightarrow \mathbb{P}(\mathbb{P}\,Loc) \\
\hline
badlocs3\ mem = \\
\quad \{lset : \mathbb{P}\,Loc \mid (\exists\, tag : tags;\ lset' : \mathbb{P}\,Loc \bullet \\
\qquad\qquad \{lset, lset'\} \in vergtlocs\ (tag, mem) \land \\
\qquad\qquad \{lset, lset'\} \cap badgtlocs\ mem = \varnothing \land \\
\qquad\qquad (\forall\, l, l' : Loc \mid l \in lset \land l' \in lset' \bullet \\
\qquad\qquad\quad dpv\ (mem\ l) \prec dpv\ (mem\ l')))\}
\end{array}
$$

Finally, its possible for there to be one more complete generation than the maximum allowed:

$$
\begin{array}{|l}
badlocs4 : DMemory \rightarrow \mathbb{P}(\mathbb{P}\,Loc) \\
\hline
badlocs4\ mem = \\
\quad \{lset : \mathbb{P}\,Loc \mid \exists\, old, new : \mathbb{N};\ tag : tags \mid \\
\qquad\qquad (old, new) = indices\ (dtagindices\ (tag, mem)) \land \\
\qquad\qquad lset = \{l : Loc \mid l \in dtaglocs\ (tag, mem) \land \\
\qquad\qquad\qquad dpx\ (mem\ l) = old\} \land \\
\qquad\qquad \#(dtagindices\ (tag, mem)) > maxgen\}
\end{array}
$$

Given these characterisations of the erroneous states of the memory system we can describe how to tidy up a memory system.

All operations that modify the memory will be expressed in terms of sequences of the following atomic operation which updates a single location in the memory:

$$
\begin{array}{|l}
write : DMemory \times Loc \times DPage \rightarrow DMemory \\
\hline
write\ (mem, l, p) = mem \oplus \{l \mapsto p\}
\end{array}
$$

To tidy up a page in the memory we simply mark it as not in use with the following procedure:

PROCEDURE

$release : DMemory \times \mathbb{P}\, Loc \rightarrow DMemory$

$release\ (mem, lset) =$
  FOR l **IN** $lset$ **DO**
    $write\ (mem, l, (false,$
      $dpt\ (mem\ l),$
      $dpi\ (mem\ l),$
      $dpx\ (mem\ l),$
      $dpn\ (mem\ l),$
      $dpv\ (mem\ l)))$

It is important that we retain the housekeeping data in the page after we release it as it may be required when calculating the conservative estimate of free locations, described on page 31, before attempting subsequent `Write` operations. The construct

FOR l **IN** $lset$ **DO** $write(mem, l, p)$

is used to capture the fact that an update of a set of locations is achieved by a sequence of updates that may be interrupted at any point. In this case we are not concerned about the order in which the elements of the set of locations are released. In some cases we do wish to be sure that the first page we release from a generation is page zero:

PROCEDURE

$release0 : DMemory \times \mathbb{P}\, Loc \rightarrow DMemory$

$release0\ (mem, lset) =$
  $release(release\ (mem, lset0), lset \setminus lset0)$
  where
  $lset0 = \{l : Loc \mid l \in lset \wedge dpn\ (mem\ l) = 0\}$

We can now describe how to tidy up the various sets of erroneous pages:

$tidy1 : (tags \rightarrow DTagData) \times DMemory \times tags \rightarrow DMemory$
$tidy2 : DMemory \times tags \rightarrow DMemory$
$tidy3 : DMemory \times tags \rightarrow DMemory$
$tidy4 : DMemory \times tags \rightarrow DMemory$

$tidy1\ (data, mem, tag)$
  $= release\ (mem, dtaglocs\ (tag, mem) \cap \bigcup badlocs1\ (data, mem))$
$tidy2\ (mem, tag)$
  $= release\ (mem, dtaglocs\ (tag, mem) \cap \bigcup badlocs2\ mem)$
$tidy3\ (mem, tag)$
  $= release0\ (mem, dtaglocs\ (tag, mem) \cap \bigcup badlocs3\ mem)$
$tidy4\ (mem, tag)$
  $= release0\ (mem, dtaglocs\ (tag, mem) \cap \bigcup badlocs4\ mem)$

It should be noted that we are not concerned with the order in which locations are tidied up unlike when releasing locations during normal operation.

The memory system design is specified by two functions as in the concrete specification:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DMemSys \quad}}\\
ddata : tags \rightarrow DTagData \\
dmem : DMemory \\
\hline
\forall\, t : tags;\ lsetss : \mathbb{P}(\mathbb{P}(\mathbb{P}\, Loc)) \mid lsetss = vergtlocs\ (t, dmem) \bullet \\
\quad \#lsetss \leq maxgen + 1 \wedge \\
\quad lsetss \neq \varnothing \Rightarrow dtagindices\ (t, dmem) \in \mathrm{dom}\, indices \\
\quad \forall\, lsets : \mathbb{P}(\mathbb{P}\, Loc) \mid lsets \in lsetss \bullet \\
\qquad \#lsets \leq 2 \wedge \\
\qquad \forall\, lset : \mathbb{P}\, Loc \mid lset \in lsets \bullet \\
\qquad\quad \#lset \leq dtds\ (ddata\ t) \wedge \\
\qquad\quad (\exists\, l : Loc \mid l \in lset \bullet dpn\ (dmem\ l) = 0) \\
\qquad\qquad \Rightarrow \{l : Loc \mid l \in lset \bullet dpn\ (dmem\ l) + 1\} = 1 ... \#lset \\
\hline
\end{array}
$$

The conditions over the two functions are weaker than in the concrete specification. This is required to allow for the possible interruption of a sequence of atomic operations over the memory which can leave it in any of the, previously described, error states.

The data refinement from the concrete to design specifications is specified in terms of the tidied up design memory system:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad Abstraction \quad}}\\
CMemSys \\
DMemSys \\
\hline
\exists\, tdydmem : DMemory;\ lset \in \mathbb{P}\, Loc \mid \\
\quad lset = \bigcup badlocs1\ (ddata, dmem) \cup \\
\qquad\quad \bigcup badlocs2\ dmem \cup \\
\qquad\quad \bigcup badlocs3\ dmem \cup \\
\qquad\quad \bigcup badlocs4\ dmem\ \wedge \\
\quad tdydmem = release\ (dmem, lset) \wedge \\
\quad \forall\, t : tags \bullet \\
\qquad tdu\ (data\ t) = dtdu\ (ddata\ t) \wedge \\
\qquad tdu\ (data\ t) = true \Rightarrow \\
\qquad\quad tds\ (data\ t) = dtds\ (ddata\ t) \wedge \\
\qquad\quad tdc\ (data\ t) = dtdc\ (ddata\ t) \wedge \\
\qquad\quad tdg\ (data\ t) = \#vergtlocs\ (t, tdydmem) \wedge \\
\qquad\quad tdx\ (data\ t) = 0 \wedge dtdx\ (ddata\ t) = 0 \vee \\
\qquad\quad \exists\, o, n : \mathbb{N} \bullet \\
\qquad\qquad (o, n) = indices\ (tagindices\ (t, mem)) \wedge \\
\qquad\qquad (o, n) = indices\ (dtagindices\ (t, tdydmem)) \\
\quad \wedge \\
\quad \forall\, l : Loc \bullet \\
\qquad pi\ (mem\ l) = dpi\ (tdydmem\ l) \wedge \\
\qquad pu\ (mem\ l) = dpu\ (tdydmem\ l) \wedge \\
\qquad pt\ (mem\ l) = dpt\ (tdydmem\ l) \wedge \\
\qquad px\ (mem\ l) = dpx\ (tdydmem\ l) \wedge \\
\qquad pn\ (mem\ l) = dpn\ (tdydmem\ l) \\
\hline
\end{array}
$$

The initial state of the memory system is:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DInitialMemSys \quad}}\\
DMemSys \\
\hline
\forall\, l : Loc \mid dpu\ (dmem\ l) = false \wedge \\
\forall\, t : tags \mid dtdu\ (ddata\ t) = false \\
\hline
\end{array}
$$

We can return an unused tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DNewTag}} \\
\Delta \mathit{DMemSys} \\
n? : \mathbb{N}_1 \\
t! : \mathit{tags} \\
\hline
\mathit{dtdu}\ (\mathit{ddata}\ t!) = \mathit{false} \\
\mathit{dmem}' = \mathit{tidy1}\ (\mathit{ddata}, \mathit{dmem}, t!) \\
\mathit{ddata}' = \mathit{ddata} \oplus \{t! \mapsto (\mathit{true}, n?, \mathit{false})\}
\end{array}
$$

This not only updates the tag data but also tidies up any locations associated with the tag in order to control the range of possible error states. We are assumming here that the update of the tag data is an atomic operation that takes place after we tidy up the memory.

We can read the information sequence, of a given generation, associated with a tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DReadGeneration}} \\
\Delta \mathit{DMemSys} \\
t? : \mathit{tags} \\
g? : \mathbb{N} \\
\mathit{info}! : \mathrm{seq}\ \mathit{Info} \\
\hline
\mathit{dtdu}\ (\mathit{ddata}\ t?) = \mathit{true} \\
\mathit{dmem}' = \mathit{tidy4}\ (\mathit{tidy3}\ (\mathit{tidy2}\ (\mathit{dmem}, \{t?\}), \{t?\}), \{t?\}) \\
\mathit{info}! = \\
\quad \{\mathit{lset} : \mathbb{P}\ \mathit{Loc};\ l : \mathit{Loc} \mid \\
\qquad \mathit{lset} \in \mathit{gentlocs}\ (t?, \mathit{dmem}') \land \\
\qquad l \in \mathit{lset} \land \\
\qquad \mathit{offset}\ (\mathit{dtagindices}\ (t?, \mathit{dmem}'), \mathit{dpx}\ (\mathit{dmem}'\ l)) = g? \bullet \\
\qquad\quad \mathit{dpn}\ (\mathit{dmem}'\ l) + 1 \mapsto \mathit{dpi}\ (\mathit{dmem}'\ l) \\
\quad \} \\
\mathit{ddata}' = \mathit{ddata}
\end{array}
$$

It should be noted that this `Read` operation can modify the memory. This is because it must first tidy up the locations associated with the tag before it can determine the locations corresponding to the requested generation.

We can again use schema conjunction and hiding to specify an operation that reads the current generation:

$$\mathit{DRead} \mathrel{\widehat{=}} (\mathit{DReadGeneration} \land \mathit{CurrentGeneration}) \setminus (g?)$$

We can release all the information associated with a tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\textit{DRelease}} \\
\Delta \mathit{DMemSys} \\
t? : \mathit{tags} \\
\hline
\mathit{tdu}\ (\mathit{ddata}\ t?) = \mathit{true} \\
\mathit{ddata}' = \mathit{ddata} \oplus \{t? \mapsto (\mathit{false}, \\
\qquad\qquad\qquad\qquad \mathit{dtds}\ (\mathit{ddata}\ t?), \\
\qquad\qquad\qquad\qquad \mathit{dtdc}\ (\mathit{ddata}\ t?))\} \\
\mathit{dmem}' = \mathit{release}\ (\mathit{dmem}, \mathit{dtaglocs}\ (t?, \mathit{dmem}))
\end{array}
$$

We are assuming here that the update of the tag data is an atomic operation that takes place before we release any of the memory locations.

We can commit the current generation of information associated with a tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DCommit \quad}} \\
\Delta DMemSys \\
t? : tags \\
\hline
tdu\ (ddata\ t?) = true \\
dmem' = tidy3\ (tidy2\ (dmem, \{t?\}), \{t?\}) \\
dtaglocs\ (t?, dmem') \neq \varnothing \\
ddata' = ddata \oplus \{t? \mapsto (true, \\
\qquad\qquad\qquad\qquad tds\ (ddata\ t?), \\
\qquad\qquad\qquad\qquad true)\} \\
\hline
\end{array}
$$

It should be noted that we must tidy up the locations associated with the tag before we can be sure is is safe to mark the current generation as committed.

Before we can perform any of the `Write` operations we must first define how we calculate the conservative estimate of the number of locations in use. In normal operation, where the memory is not in an error state, the maximum number of locations in use between operations can be calculated by multiplying the sizes associated with each tag that is in use by *maxgen*. The `Write` operations require a further generation's worth of locations for the tag being written to. Thus the memory must be large enough to handle the maximum number of generations for all tags that will be in use at the same time plus one generation of the tag with the largest size. When errors states are considered each tag may have $maxgen + 1$ generations associated with it. These error states mean that `Write` operations cannot guarantee that there will be enough free locations unless they check the memory and tidy up any error states. We need a way to calculate an estimate of the number of locations marked as in use that will correspond to the actual number in normal operation and provide a conservative estimate in the error states. We estimate the number of locations in use by a tag by assuming that the existence of each location with a unique generation and version pair implies the existence of an entire generation of locations. This calculation applies even if the tag data marks the tag as not in use and requires that the tag data record the size of the generations previously associated with a tag. The conservative estimate we use is:

$$
\begin{array}{|l}
usedlocs : DTagData \times DMemory \rightarrow \mathbb{N} \\
\hline
usedlocs\ (ddata, dmem) = \\
\qquad \sum_{t:tags} dtds\ (ddata\ t) * max\ (maxgen, \# \bigcup vergtlocs\ (t, dmem))
\end{array}
$$

Given this we can check there is enough room to perform a `Write` operation. We express this as a schema that performs the check and if it fails tidies up the memory and performs the check again.

$$DWriteOK \mathrel{\widehat{=}} DCheckOK \lor (DCheckNotOK \mathbin{⨾} DTidy \mathbin{⨾} DCheckOK)$$

To check the `Write` operation we simply subtract our estimate of locations in use from the memory size and check there is room for a generation for the tag:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DCheckOK \quad}} \\
\Xi CMemSys \\
t? : tags \\
info? : \mathrm{seq}\ Info \\
\hline
msize - usedlocs\ (ddata, dmem) \geq dtds\ (ddata\ t?) \\
\hline
\end{array}
$$

We can't simply use schema negation to produce the negation of this check because its property is partly expressed by its declaration.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DCheckNotOK \quad}} \\
\Xi CMemSys \\
t? : tags \\
info? : \mathrm{seq}\ Info \\
\hline
msize - usedlocs\ (ddata, dmem) < dtds\ (ddata\ t?) \\
\hline
\end{array}
$$

To tidy up the memory we simply tidy up all the error states:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{DTidy}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta DMemSys \\
t? : tags \\
info? : \mathrm{seq}\, Info \\
\hline
ddata' = ddata \\
dmem' = tidy4\ (tidy3\ (tidy2\ (tidy1\ (ddata, dmem, \{t?\}), \{t?\}), \{t?\}), \{t?\}) \\
\hline
\end{array}
$$

We can perform the first write to a tag after $DNewTag$:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\textit{DWriteFirst}\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta DMemSys \\
t? : tags \\
info? : \mathrm{seq}\, Info \\
\hline
dtdu\ (ddata\ t?) = true \wedge dtds\ (ddata\ t?) = \#info? \\
tdymem = tidy4\ (tidy3\ (tidy2\ (dmem, \{t?\}), \{t?\}), \{t?\}) \\
dtaglocs\ (t?, tdymem) = \varnothing \\
ddata' = ddata \\
\exists\, lseq : \mathrm{seq}\, Loc \mid l \in \mathrm{ran}\, lseq \Rightarrow dpu\ (tdymem\ l) = false \wedge \\
\qquad\qquad\qquad \#lseq = \#info? \wedge \\
\qquad\qquad\qquad \#(\mathrm{ran}\, lseq) = \#info? \\
\qquad\qquad\qquad \bullet\ dmem' = update\ (tdymem, lseq, info?, t?, 0, V_A) \\
\hline
\end{array}
$$

We must tidy up the locations associated with the tag before we perform the memory update in order to control the range of possible error states. This uses the following procedure to update the set of locations newly associated with the tag:

$$
\begin{array}{|l}
\underline{\mathbf{PROCEDURE}} \\
update : Memory \times \mathrm{seq}\, Loc \times \mathrm{seq}\, Info \times tags \times \mathbb{N} \times Version \rightarrow Memory \\
update\ (mem, lseq, info, t, x, v) = \\
\quad \underline{\mathbf{FOR}}\ n = 1\ \underline{\mathbf{TO}}\ \#info\ \underline{\mathbf{DO}} \\
\qquad write\ (mem, lseq\ n, (true, \\
\qquad\qquad\qquad\qquad t, \\
\qquad\qquad\qquad\qquad info\ (\#info - n + 1), \\
\qquad\qquad\qquad\qquad x, \\
\qquad\qquad\qquad\qquad \#info - n, \\
\qquad\qquad\qquad\qquad v))
\end{array}
$$

The **FOR** loop can interrupted at any point and it is important that the last page written is page zero.

We can write to a tag whose current generation is uncommitted:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad \mathit{DWriteUncommitted} \quad}} \\
\Delta \mathit{DMemSys} \\
t? : \mathit{tags} \\
\mathit{info}? : \mathrm{seq}\, \mathit{Info} \\
\hline
\mathit{dtdu}\ (\mathit{ddata}\ t?) = \mathit{true} \land \mathit{dtds}\ (\mathit{ddata}\ t?) = \#\mathit{info}? \\
\mathit{tdymem} = \mathit{tidy4}\ (\mathit{tidy3}\ (\mathit{tidy2}\ (\mathit{dmem}, \{t?\}), \{t?\}), \{t?\}) \\
\mathit{dtaglocs}\ (t?, \mathit{tdymem}) \neq \varnothing \\
\mathit{dtdc}\ (\mathit{ddata}\ t?) \neq \mathit{true} \\
\mathit{ddata}' = \mathit{ddata} \\
(\mathit{old}, \mathit{new}) = \mathit{indices}\ (\mathit{dtagindices}\ (t?, \mathit{tdymem})) \\
\exists\, \mathit{olseq} : \mathrm{seq}\, \mathit{Loc};\ v : \mathit{Version};\ \mathit{lseq} : \mathrm{seq}\, \mathit{Loc} \mid \\
\quad \#\mathit{olseq} = \#\mathit{info}? \land \\
\quad \mathrm{ran}\, \mathit{olseq} \in \mathit{gentlocs}\ (t?, \mathit{tdymem}) \land \\
\quad (\forall\, l : \mathit{Loc} \mid l \in \mathrm{ran}\, \mathit{olseq} \Rightarrow \mathit{dpx}\ (\mathit{tdymem}\ l) = \mathit{new} \land \mathit{dpv}\ (\mathit{tdymem}\ l) = v) \land \\
\quad \mathit{dpn}\ (\mathit{olseq}\ 1) = 0 \land \\
\quad (\forall\, l : \mathit{Loc} \mid l \in \mathrm{ran}\, \mathit{lseq} \Rightarrow \mathit{dpu}\ (\mathit{tdymem}\ l) = \mathit{false}) \land \\
\quad \#\mathit{lseq} = \#\mathit{info}? \land \\
\quad \#(\mathrm{ran}\, \mathit{lseq}) = \#\mathit{info}? \bullet \\
\qquad \mathit{udmem} = \mathit{update}\ (\mathit{tdymem}, \mathit{lseq}, \mathit{info}?, t?, \mathit{new}, \mathit{nextV}\ v) \land \\
\qquad \mathit{dmem}' = \mathit{free}\ (\mathit{udmem}, \mathit{olseq}) \\
\hline
\end{array}
$$

We must again tidy up the locations associated with the tag before we perform the memory update in order to control the range of possible error states. We are assuming here that the memory update is performed before the memory release. The new version of the current generation has the same generation index but its version is that following that of the current version of the current generation. This uses the following procedure to release the locations of the old version of the current generation, releasing page zero first:

$$
\begin{array}{|l}
\underline{\textbf{PROCEDURE}} \\
\mathit{free} : \mathit{Memory} \times \mathrm{seq}\, \mathit{Loc} \to \mathit{Memory} \\
\mathit{update}\ (\mathit{mem}, \mathit{lseq}) = \\
\quad \underline{\textbf{FOR}}\ n = 1\ \underline{\textbf{TO}}\ \#\mathit{lseq}\ \underline{\textbf{DO}} \\
\qquad \mathit{write}\ (\mathit{mem}, \mathit{lseq}\ n, (\mathit{false}, \\
\qquad\qquad\qquad\qquad \mathit{dpt}\ (\mathit{mem}\ (\mathit{lseq}\ n)), \\
\qquad\qquad\qquad\qquad \mathit{dpi}\ (\mathit{mem}\ (\mathit{lseq}\ n)), \\
\qquad\qquad\qquad\qquad \mathit{dpx}\ (\mathit{mem}\ (\mathit{lseq}\ n)), \\
\qquad\qquad\qquad\qquad \mathit{dpn}\ (\mathit{mem}\ (\mathit{lseq}\ n)), \\
\qquad\qquad\qquad\qquad \mathit{dpv}\ (\mathit{mem}\ (\mathit{lseq}\ n))))
\end{array}
$$

When we write to a tag whose current generation has been committed we add a new generation to those associated with the tag. If this results in more than the maximum allowed number of generations the oldest generation must be dropped.

If we have not reached the maximum allowed number of generations we needn't drop the oldest generation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DWriteCommittedAddGen \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta DMemSys \\
t? : tags \\
info? : \mathrm{seq}\ Info \\
\hline
dtdu\ (ddata\ t?) = true \land dtds\ (ddata\ t?) = \#info? \\
tdymem = tidy4\ (tidy3\ (tidy2\ (dmem, \{t?\}), \{t?\}), \{t?\}) \\
\#gentlocs\ (t?, tdymem) < maxgen \\
dtdc\ (ddata\ t?) = true \\
(old, new) = indices\ (dtagindices\ (t?, tdymem)) \\
\exists\, lseq : \mathrm{seq}\ Loc \mid \\
\quad \forall\, l : Loc \mid l \in \mathrm{ran}\ lseq \Rightarrow dpu\ (tdymem\ l) = false \land \\
\quad \#lseq = \#info? \land \\
\quad \#(\mathrm{ran}\ lseq) = \#info? \bullet \\
\qquad dmem' = update\ (tdymem, lseq, info?, t?, incrindex\ new, V_A) \\
ddata' = ddata \oplus \{t? \mapsto (true, tds\ (data\ t?), false)\} \\
\hline
\end{array}
$$

We again tidy up the locations associated with the tag before updating the memory. We are assuming here that the memory update is performed before the tag data update[1].

If we have reached the maximum allowed number of generations we must drop the oldest generation:

$$
\begin{array}{|l}
\multicolumn{1}{l}{\underline{\quad DWriteCommittedMaxGen \qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad}}\\
\Delta DMemSys \\
t? : tags \\
info? : \mathrm{seq}\ Info \\
\hline
dtdu\ (ddata\ t?) = true \land dtds\ (ddata\ t?) = \#info? \\
tdymem = tidy4\ (tidy3\ (tidy2\ (dmem, \{t?\}), \{t?\}), \{t?\}) \\
\#gentlocs\ (t?, tdymem) = maxgen \\
dtdc\ (ddata\ t?) = true \\
(old, new) = indices\ (dtagindices\ (t?, tdymem)) \\
\exists\, olseq : \mathrm{seq}\ Loc;\ lseq : \mathrm{seq}\ Loc \mid \\
\quad \#olseq = \#info? \land \\
\quad \mathrm{ran}\ olseq \in gentlocs\ (t?, tdymem) \land \\
\quad \forall\, l : Loc \mid l \in \mathrm{ran}\ olseq \Rightarrow dpx\ (tdymem\ l) = old \land \\
\quad dpn\ (olseq\ 1) = 0 \land \\
\quad \forall\, l : Loc \mid l \in \mathrm{ran}\ lseq \Rightarrow dpu\ (tdymem\ l) = false \land \\
\quad \#lseq = \#info? \land \\
\quad \#(\mathrm{ran}\ lseq) = \#info? \bullet \\
\qquad udmem = update\ (tdymem, lseq, info?, t?, incrindex\ new, V_A) \land \\
\qquad dmem' = free\ (udmem, olseq) \\
ddata' = ddata \oplus \{t? \mapsto (true, tds\ (data\ t?), false)\} \\
\hline
\end{array}
$$

We again tidy up before the memory update and assume that the update precedes the release and they both precede the tag data update

We can now use schema disjunction to specify the write operation:

$$
\begin{aligned}
DWrite \mathrel{\widehat{=}} DWriteOK \mathbin{;} (&DWriteFirst \lor \\
&DWriteUncommitted \lor \\
&DWriteCommittedNewGen \lor \\
&DWriteCommittedMaxGen)
\end{aligned}
$$

[1] It is worth noting that the committed flag in tag the data introduces a possible data inconsistency between the memory and the tag data which is not present in the real system.

5.8 Conclusion

In this specification we have successfully described the operations that write to the memory in terms of sequences of atomic page writing operations. The possible interruption of these sequences of atomic operations introduced a number of possible error states not present in the previous specifications. By elaborating the data stored in a page and adding some limited housekeeping to the standard operations we have been able to restrict the number of possible error states. Given this restricted set of new error states we are able to specify how the operations may detect and recover from all such errors. This approach has the advantage that we have not had to revise the abstract and concrete specifications to handle error returns from the standard operations. Furthermore, by replacing data stored with each tag by searches over the memory, we have avoided the previous requirement for multiple updates of single memory locations.

Section 6

"Safe" Storage Allocation

To perform storage allocation for objects of a variety of types the storage allocation routines must be able to perform a number of operations we would not like to see used throughout a program:

- Determine the size (in bytes or bits say) of an object of any type.
- Determine the address boundary constraints of objects of any type.
- Coerce storage of any type into storage of some low-level type (an array of bytes or bits say).
- Given allocated storage of some low-level type of the required size and alignment for a type coerce it into storage of that type.

If the storage allocator is to be written in the language itself then these dangerous constructs must be present in the language itself. In order to limit the parts of a program that must be considered "unsafe" we must restrict the occurrences of all these operations.

The problems that arise if we don't can be seen in **C** where `malloc(3)` performs the storage allocations, handling all the alignment constraints. Each call of `malloc` will, however, appear as something like the following:

```
(Type *) malloc(Len * sizeof(Type))
```

Thus the entire program will be littered with `sizeof` operations and type casts. The problem **C** has is that it can't parameterize the storage allocation routines with the type of the object being allocated.

Modula-3 solves this problem by providing a `NEW(T, ...)` operation that is paremeterized by the type to be allocated. A similar effect could be achieved in **Modula-3** by utilising its generic modules. This would allow **Modula-3** to paremeterise a module of storage allocation routines by the type being allocated and restrict *all* the unsafe operations to this module. Each type could import an instance of this generic module to give a type specific function for allocating storage of that type.

Section 7

Modula-3 Memory Management

7.1 Introduction

We describe an implementation of a general storage allocation scheme for structures of any type in **Modula-3**.

7.2 Type System

Modula-3 is strongly typed but requires run-time type checking. This has both performance and storage implications because of the manipulations of the housekeeping information that must be associated with each value.

The types corresponding to "*pointers*" are called references in **Modula-3** and two different kinds of reference are supplied. The first, called "*traced*" references, refer to storage that is managed by the garbage collector. The second, called "*untraced*" references, refer to storage that is not garbage collected, though it may still have associated housekeeping information.

7.3 Safety

Certain operations, such as address arithmetic, are classified as unsafe and may only appear in modules that are explicitly marked as unsafe. Arbitrary type casts, such as from an arbitrary reference to a char pointer, are classified as unsafe. It is not possible for an unchecked run-time error to occur in a safe module.

7.4 Generic Modules

Modula-3 modules can be parameterised but only by the name of an interface. This means that if what we are really trying to implement is a type parameterisation we must explicitly code interfaces that do nothing more than contain the desired type for *each* type that we wish to pass to the parameterised module.

7.5 `BYTE` Module

We start by defining a module for the type `BYTE`. For this we require only an interface:

```
INTERFACE BYTE;
IMPORT ASCII;
TYPE T = BITS 8 FOR ASCII.Range;
TYPE PTR = UNTRACED REF ARRAY OF T;
END BYTE;
```

This requires the library module `ASCII` which contains a definition of the sub-range of the type `CHAR` that corresponds to the 256 characters that can be represented in 8 bits. We define a packed[1] type for 8 bit characters[2].

7.6 MEM Module

Given the interface `BYTE` we can define a module that handles low-level operations over a block of storage. The interface for this module is:

```
INTERFACE MEM
IMPORT BYTE;
EXCEPTION IllMemRef(CARDINAL);
PROCEDURE Get(offset: CARDINAL): BYTE.T RAISES IllMemRef;
PROCEDURE Set(offset: CARDINAL; byte: BYTE.T) RAISES IllMemRef;
END MEM.
```

This is a safe module and the functions may raise an exception if the supplied offset is too large[3]. We have a function to return the byte at a given offset/address and a function to set the byte at a given offset The implementation of this module is straightforward:

```
MODULE MEM;
IMPORT  BYTE;
CONST Size: CARDINAL = 1024;
VAR Block: BYTE.PTR;
PROCEDURE Get(offset: CARDINAL) : BYTE.T RAISES IllMemRef =
BEGIN
  IF offset >= Size THEN
    RAISE IllMemRef(offset);
  ELSE
    RETURN Block[offset];
  END;
END Get;
PROCEDURE Set(offset: CARDINAL; byte: BYTE.T) RAISES IllMemRef =
BEGIN
  IF offset >= Size THEN
    RAISE IllMemRef(offset);
  ELSE
    Block[offset] := byte;
  END;
END Set;
BEGIN
  Block := NEW(BYTE.PTR,Size);
  FOR offset := 0 TO Size - 1 DO
    Block[offset] := '\000';
  END;
END MEM.
```

The functions `Get` and `Set` read and write bytes in the block of storage. They simply check the offset before accessing the array of bytes. We initialise the allocated block of storage to `NUL`'s to allow tracking of what has been written where in the buffer easier.

[1] The conversions between the packed and unpacked representations of values are performed automatically by **Modula-3**.

[2] Though `CHAR` seems to take only 8 bits on this implementation it is simply guaranteed to contain *at least 256 elements* corresponding to the 8 bit characters.

[3] **Modula-3** would raise its own exception if the array index was out of bounds.

7.7 Store Module

In order to store a value of an arbitrary type we must be able to regard it as a sequence of bytes. This requires us to perform an arbitrary type coercion so we will require an unsafe module. Furthermore, to parameterise a module by a type we must parameterise it by an interface that is expected to define a type of a known name. We adopt the **Modula-3** idiom of naming such types "`T`". The generic interface is:

```
GENERIC INTERFACE Store(I);
PROCEDURE WriteVAR(VAR v:I.T; offset: CARDINAL);
PROCEDURE ReadVAR(VAR v:I.T; offset: CARDINAL);
END Store.
```

We parameterise by an arbitrary interface `I` which is expected to contain a type named "`T`". Notice that the argument to the functions is a variable designator, not a value. The implementation is a little tricky as we must forge a byte pointer from the variable designator:

```
GENERIC MODULE Store(I);
IMPORT  BYTE, MEM;
PROCEDURE WriteVAR(VAR value:I.T; offset: CARDINAL) =
VAR addr: UNTRACED REF BYTE.T;
BEGIN
  FOR i := 0 TO BYTESIZE(I.T) - 1 DO
    addr := ADR(value) + i;
    MEM.Set(offset + i,addr^);
  END;
END WriteVAR;
PROCEDURE ReadVAR(VAR value:I.T; offset: CARDINAL) =
VAR addr: UNTRACED REF BYTE.T;
BEGIN
  FOR i := 0 TO BYTESIZE(I.T) - 1 DO
    addr := ADR(value) + i;
    addr^ := MEM.Get(offset + i);
  END;
END ReadVAR;
BEGIN
END Store.
```

These functions copy the value in the argument variable designator into and out of storage a byte at a time. By declaring a local variable of the required type we can avoid an explicit use of the `LOOPHOLE` function.

7.8 Store_INTEGER Module

Given the generic module `Store` we can instantiate it to get a storage module for a type. For example, to store `INTEGER` we have the interface:

```
UNSAFE INTERFACE Store_INTEGER = Store(Integer)
END Store_INTEGER.
```

Its implementation is:

```
UNSAFE MODULE Store_INTEGER = Store(Integer)
END Store_INTEGER.
```

This module could then be imported into a module:

```
IMPORT Store_INTEGER AS SINT;
```

In that module we could then store and retrieve integers:

```
SINT.WriteVAR(numin,offset);
SINT.ReadVAR(numout,offset);
```

For any given `offset` and `numin` after these two commands the values of `numin` and `numout` will be equal.

7.9 Conclusion

As a model for a smart-card language **Modula-3** has a number of obvious disadvantages:

- Run-time type checking.
- No simple way to provide access to the sequence of bytes that represent a value for an arbitrary type.
- *Personal opinion*: an awful syntax.

It does have some advantages:

- Strongly typed.
- Object oriented.
- Defined module/interface system[4].

[4] This can be contrasted with `C` where header file usage is merely a convention that is not enforced by the language.

Section 8

A Proposed Type System for CLASP

8.1 Introduction

When designing a type system for the new **CLASP** language a number of conflicting goals must be reconciled:

- Consideration of program development and verification would lead to the choice of a strong type system that supports generic constructs. In particular we would want a type system that has no "*holes*" in the type system, such as the `LOOPHOLE` construct in **Modula-3**. We would also choose a simple type system that can be described by a small set of generally applicable rules, unlike that of **Modula-3**.
- Because of the constraints imposed by the target hardware the type system must impose a minimal run-time cost, unlike the run-time type checking found in **Modula-3**.
- It is required that the language be capable of implementing its own storage allocator. Such storage allocators normally require the ability to perform arbitrary type coercions and arbitrary address arithmetic, as exemplified by the implementation of `malloc(3)` in **C**.
- We want to be able to be able to localise, and minimise, the possibility of unchecked run-time errors and check as many run-time errors as possible. This is similar to the features provided by the "*safe*" and "*unsafe*" modules in **Modula-3**. In particular we must check for illegal array indexing (See 8.3).

It seems clear that the most appropriate language for the task is some kind of Object-oriented language. Given a free choice of selecting a type system one would prefer that of the **Eiffel** to that of **Modula-3**. **Eiffel** is strongly typed with no type "*holes*", supports generic constructs that are parameterised by types (not interfaces) and requires no run-time type checking. Unfortunately the requirement that the language be capable of implementing its own storage allocator means that it must allow arbitrary type coercions. We regard the storage allocator as representing all the low-level routines that the language may be required to implement as any problems they raise for the type system will be raised by the storage allocator. The question then becomes, how much of the elegance and power of a type system such as that of **Eiffel** can we retain while supporting the low-level programming constructs required in the language?

8.2 Storage Allocation

The basic function of a storage allocator is to manage regions of memory in such a way as to support the creation and destruction of structures of any type. The standard approach is to regard the memory as a sequence of bytes over whose addresses arithmetic operations may be performed. Management of the memory involves locating and reserving regions of the memory in order to represent these structures. This management process will need to store housekeeping information in the memory along with the contents of the structures themselves. It may also be necessary to enforce address alignment restrictions when allocating storage of certain types. All these features can be seen in the various versions of `malloc(3)` in **C**.

Programming errors in the implementation of a storage allocator can lead to a variety of run-time errors. The most obvious is that the value returned from the memory may not be that initially stored there. The operations of a storage allocator are "*type-unsafe*" in the sense they can easily "*forge*" values (bit patterns) for a type that are not valid for that type. If the application employs an out-of-date *handle* to a structure previously stored in memory similar errors can ensue. These programming errors will lead to obscure, uncheckable run-time errors. We wish, at least, to isolate and minimise such type-unsafe operations and and hopefully eliminate all but one type-unsafe operation (See 8.2.1 and 8.6). This type-unsafe operation should, of course, be supported by a proof of its safety as it will be the most likely cause of run-time errors.

8.2.1 Type Coercions

There are in fact two different kinds of type coercion required by a storage allocator.

The first is required to turn a reference to a structure of some type into a reference to a sequence of bytes. This is required when the application passes a structure to the storage allocator. This operation is, in fact, type-safe, in the sense used above, as all bit patterns are included in the type of bytes. The type conformance rule for `Bit_type` in **Eiffel**regards such a coercion as acceptable. It becomes type-unsafe if the type coercion does not take a copy of the structure and subsequently modifies the sequence of bytes. If we either require this type coercion to copy its argument or ensure that the resulting byte sequence is marked as "*read-only*" then this becomes a type-safe type coercion.

The second is required to turn a reference to a sequence of bytes into a reference to a structure of some type. This is required when the storage allocator passes a structure back to the application. This coercion is type-unsafe and we should severely restrict its use, perhaps allowing only a single use in the entire system.

8.2.2 Addresses

In the simple model of the storage allocator we require a type of addresses over which we can perform arithmetic. We thus require an operation that returns the address of a structure so that by performing operations relative to this address we can perform operations on the contents of the structure. This separation can easily lead to type-unsafe programming errors. In the restricted context of the **CLASP** language we may be able to avoid the use of a separate address type.

In the simple model we already require the compiler to provide values for the base address of the memory and some measure of its size. If the compiler could instead be assumed to provide us with a predefined array of bytes that represents the memory we could then remove some of the programming errors that may occur with address arithmetic. In particular, as arrays will have index checking in our language (See 8.3), illegal addresses can be excluded.

8.2.3 Operations

In the tagged memory scheme employed in **CLASP** the operations over the persistent (EEPROM) memory are restricted. While it is possible to read a byte from any address one must write blocks of bytes of a given size to an appropriate block address. While representing memory as an array removes some errors we would still be dependent on the coding of the operations over this array respecting the restricted nature of the operations actually allowed over the memory.

If we could represent the memory as an object whose methods imposed the restrictions required by the actual operations over the memory we could remove another source of programming errors. This object could easily have a single instance provided by the compiler.

8.3 Arrays and Number Ranges

Our language must employ array index checking as illegal array accesses are a common source of run-time errors. In the general case this will require a run-time check imposing both time and storage[1] costs on our programs. Number ranges are a useful programming device as they can be used to capture more clearly the intent of a program as well as detecting some programming errors. By combining number ranges, arrays whose length is statically determined (similar to the *closed* arrays of **Modula-3**) and language constructs to iterate over ranges we can provide static array index checking in many cases.

[1] For the bounds of the array.

In particular we would would *not* want to perform run-time checks, or allocate housekeeping information, for the array representing the memory (See 8.2.2) We therefore introduce a kind of array which *requires* that all its indexing be statically checked. This requirement could be in the form of a proof obligation rather than a strict requirement on the compiler itself. We do, however, expect our compiler to be able to statically determine that

```
memory[I + J]
```

is a valid array access given:

```
memory :  ARRAY [0..MemLen]
I : [0..MemLen - BlockLen]
J : [0..BlockLen]
```

for any compile-time constants `MemLen` and `BlockLen` where `BlockLen` $\leq$ `MemLen`. Removal of run-time checks on arbitrary array indexing may be achieved if the appropriate proof obligations are satisfied.

8.4 Generic Modules

The generic modules in **Modula-3** suffer from a number of serious flaws.

- A generic module cannot be type checked! Only its instances may be type checked, unhelpful for program development.
- Modules are parameterised by interfaces not types. Thus all types that may be used to instantiate a generic module must have a trivial interface defined for them and all these interfaces and generic modules must adopt a consistent naming scheme[2].

The approach adopted in **Eiffel** is much neater and avoids all these problems.

8.5 Tagged Memory

The tagged memory implemented in the persistent (EEPROM) memory provides special problems for the design of the language. The basic question is, should the application be aware that certain data is actually stored in tagged memory?

If the application is aware that a piece of data is in tagged memory it will need to explicitly read the data into RAM before treating as a value of the appropriate type. The types of values that may be read into RAM is therefore limited by the size of the RAM buffer used by the read operation. In this approach the tagged memory is regarded very much as the file system is in most programming languages[3]. The advantage of this approach is that the compiler need know nothing about the implementation of the tagged memory. If the compiler is required to know the details of the tagged memory implementation this raises the possibility of all sorts of problems arising from inconsistencies between the implementation and the compiler's model of the implementation.

If the storage of data in tagged memory is supposed to be invisible to the application, at least when reading values, then the compiler must have some knowledge of the implementation of the tagged memory. By making suitable choices in the implementation of the interface to the tagged memory allocator we can reduce this knowledge to its absolute minimum.

The basic idea is to implement a read from tagged memory in an application as:

$$Read(tag, type, field) : type.field$$

where *type* is the type of the data stored for the tag and $field$ is the field we want to read, which may be the entire value. The types of fields that may be read into RAM are again limited by the buffer size. The interface to the tagged memory will transform this into a call to the storage allocator:

$$ReadMemory(tag, size, offset, length)$$

[2] The **Modula-3** idiom seems to be to name types in the interface `T`.

[3] This interpretation of tagged memory makes it clear why it would not be safe to store structures containing pointers in tagged memory.

Where *size* is the length in bytes of values of type *type* and *offset* and *length* specify how to extract the specified *field* from values of that *type* when it is stored in contiguous memory. Both type casts, to and from byte arrays, are performed within the *Read* function, the storage allocator itself handling only arrays of bytes. Thus our language must provide operations that return the size in bytes of values of any type (like the `sizeof` operator in **C**), return the type of any field of a type, return the offset of any field of a type and return the length of any field of a type. With this interface the implementation of the tagged memory can be varied at will without the compiler having to change at all. Thus the compiler can invisibly transform source language expressions of the form `value.field` into the appropriate calls to the *Read* function.

In either of these approachs if the size of the RAM buffer is decreased it may become necessary to rewrite programs that attempt to read values, or fields, that will no longer fit in the buffer.

8.6 Views

Views are a generalisation of the concept of a type-safe type coercion. When viewing a value of one type as a value of another type we must, in general, perform run-time checks to ensure that the operation is type-safe. For instance, a value of type `[0..5]` may only be viewed as of type `[0..3]` if the value is actually less than 4. The reverse view would require no run time check. Views can be implemented either by copying or by ensuring that the resulting value is read-only. One possible use for views is when interpreting input character sequences. The language can clearly implement any protocol it wishes over an input stream, building any desired byte sequence from the input. A view of the resulting byte array both allows the application to regard the input as a structured value and to perform a run-time check, via the exception generated by a failed view, that it is a valid value of that type. We should not therefore need to use the type-unsafe type coercion when reading input.

8.7 Orthogonal Issues

Many issues in the design of the language are orthogonal to the design of its type system. One such issue is the interpretation of values of structured types. These can be interpreted as either references to values or as simply being values. Either approach can also be adopted when passing arguments to functions. Whichever choice is made the type system, and storage allocator, described can be implemented for the language.

8.8 Conclusion

By modelling memory as arrays of statically determined sizes and adopting the appropriate interface in the implementation of the storage allocator we can avoid the need for a type of addresses and for run-time checks on memory accesses. We can also restrict the type-unsafe type coercion to the interface between the applications and the storage allocator, which itself performs no type-unsafe operations. References to values stored in tagged memory can be rendered invisible to applications by the compiler without it requiring detailed knowledge of the tagged memory scheme. By using views with run-time checks we may be able to restrict the type-unsafe type coercion to its single use in the memory allocator interface.