Transacted Memory for Smart Cards[1]

Pieter H. Hartel, Michael J. Butler, Eduard de Jong, and Mark Longley
{phh,mjb}@ecs.soton.ac.uk and Eduard.deJong@Sun.COM

Declarative Systems and Software Engineering Group
Technical Report DSSE-TR-2000-9
August 16 2000

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

[1]This work was supported by Sun Microsystems Inc, USA and by Senter, The Netherlands under contract nr ITG94130

Transacted Memory for Smart Cards[†]

Pieter H. Hartel[‡], Michael J. Butler[‡], Eduard de Jong[§] and Mark Longley[‡]

August 17, 2000

Abstract

A transacted memory that is implemented using EEPROM technology offers persistence, undoability and auditing. The transacted memory system is formally specified in Z, and refined in two steps to a prototype C implementation / SPIN model. Conclusions are offered both on the transacted memory system itself and on the development process involving multiple notations and tools.

1 Introduction

The purpose of transaction processing [1] is to provide atomic updates of arbitrarily sized information. Smart cards need such a facility as any transaction can easily be aborted by pulling the smart card out of the Card Acceptance Device (CAD). Smart cards provide limited resources. High-end smart cards today offer 64KB of ROM, 64KB of EEPROM and 2KB of RAM. These limitations make techniques developed for mainstream transaction processing systems inappropriate for smart cards.

Current smart card solutions, including Java [1] Card implementations [13] typically maintain a log of old values, while an updated value is being constructed [4]. The log is cleared once the transaction is committed. If required, the logs can be used to provide the audit trail for security.

Current smart card implementations, by their very nature, view the memory as a resource, used to support a transaction processing API. We present a novel (patented) view [3], which embeds the transaction capabilities into the memory system itself. Transacted memory allows an arbitrary sequence of items to be written as a single transaction to the memory. The space required for such a sequence may even exceed the size of the RAM. An audit trail is automatically provided. The disadvantage of our system is an increased EEPROM requirement to twice the size of the data. The permanent RAM requirements are NIL, transient RAM requirements are of the order of a few bytes.

Transacted memory does not impose structural constraints on the information stored, nor does it provide marshaling and unmarshalling capabilities. These are intended to be implemented, for instance by an API on top of the transacted memory.

The current work is part of a series of formally specified components [7, 6] of smart card systems. We hope that we will eventually be able to design, specify and implement a complete smart card operating system kernel that can be subjected to Common Criteria at evaluation level EAL7 [12].

Transacted memory is not to be confused with transactional memory [8], which is a technique for supporting lock free data structures on multi processor architectures. The implementation of transactional memory is an extension of the cache coherence protocol of such machines [8]. We consider a different problem domain with severe resource constraints.

We present a high level specification of the system (using Z) and discuss two refinements (in Z) of the system, ultimately leading to executable code (using C). A number of properties of the high level specifications have been proved (by hand), and the prototype implementation has been subjected to assertion checking (using SPIN).

The contributions of the paper are:

- A presentation of the novel smart card transacted memory manager.
- A discussion of the lessons learned by systematically translating a Z specification with proofs into C code with assertion checking. This complements the results reported in our earlier paper [5].

1.1 The process

Figure 1 describes the specifications and the prototype implementation of the memory management system. Z was chosen as the specification language because at the time the project was started, (Summer of 1996) this appeared to be the specification language most acceptable by industry.

[†]This work was supported by Sun Microsystems Inc, USA and by Senter, The Netherlands under contract nr ITG94130

[‡]Dept. of Electronics and Computer Science, Univ. of Southampton, UK, Email: {phh, mjb}@ecs.soton.ac.uk

[§]Sun Microsystems, Inc. Palo Alto, CA 94043 USA, Email: Eduard.deJong@Sun.COM

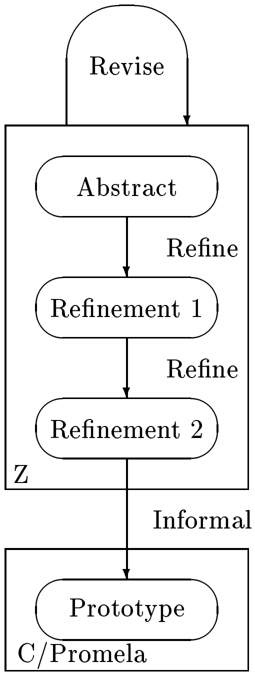

Figure 1: The process

The abstract specification was produced after initial discussions between the inventor of transacted memory (Eduard de Jong) and the specification team (the other authors). After further rounds of consultation the abstract specification was revised, and a first refinement was produced to reflect the reality of the EEPROM technology as documented in Section 3.

In 1997 a second data refinement was produced to reflect the possibilities of errors arising by interrupting EEPROM write operations. In 2000, the final specification labelled "prototype" was produced manually by interpreting the second refinement as literally as possible. The prototype is at the same time an executable specification (because it is a SPIN model) and a C program. Some macros are used to transfer from a common notation to either SPIN or C.

The prototype is a proper implementation, it is as memory efficient as possible. It is not as time efficient as possible, because often-used information is recomputed instead of cached. However the prototype is a useful yardstick to measure progress of further implementations by, which would explore space / time tradeoffs. The prototype also allows for a considerable degree of parallelism to be exploited in a hardware implementation of the memory system.

1.2 The Idea

Transacted memory is designed around two notions: a tag and an information sequence. A tag is merely a unique address, e.g. identifier of a particular information sequence. An information sequence is the unit of data stored and retrieved. An information sequence would be used to store a collection of objects that are logically part of a transaction.

There may be several generations of the information associated with a tag. Operations are provided to write a new generation, and to read the current or older generations. All generations associated with a tag have the same size, although this could be generalised.

The transaction processing capability of the memory is supported by a commit operation, which makes the most recently written information the current generation. The oldest generation is automatically purged should the number of generations for a tag exceed a preset maximum.

Transacted memory thus provides undoability (by being able to revert to a previous generation) and persistence (by using EEPROM technology). These are precisely the ingredients necessary to support transactions [11].

To provide this functionality, transacted memory maintains a certain amount of book-keeping information. In its most abstract form, the book-keeping information records three items:

- The size of the information sequence that may be associated with the tag.
- The different generations of information associated with each tag. It is possible that there is no information associated with a tag.
- Which tags are currently committed.

Having sketched the ideas, we will now make this precise by presenting an abstract Z specification.

2 Abstract Specification

The abstract specification assumes the existence of tags used to address the memory, and the existence of information to be stored in the memory. No further assumptions are made about either.

$$[Tag, Info]$$

The existence of a finite set of available tags is assumed ($tags$), as well as limits on the size of the memory ($msize$) and the maximum number of generations that may be associated with any tag ($maxgen$):

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

Two partial functions *assoc* and *size* and a set *committed* specify the memory system. The derived value *usage* is included to aid the presentation:

$$
\begin{array}{|l}
\textit{AMemSys} \\
\hline
assoc : tags \nrightarrow \mathrm{seq}(\mathrm{seq}\, Info) \\
size : tags \nrightarrow \mathbb{N}_1 \\
committed : \mathbb{P}\, tags \\
usage : \mathbb{N} \\
\hline
\mathrm{dom}\, assoc = \mathrm{dom}\, size \\
committed \subseteq \mathrm{dom}\, assoc \\
\forall\, t : tags \mid t \in committed \bullet assoc\ t \neq \langle\rangle \\
\forall\, t : tags \mid t \in \mathrm{dom}\, assoc \bullet \\
\quad \#(assoc\ t) \leq maxgen \land \\
\quad (\forall\, i : \mathbb{N}_1 \mid 1 \leq i \leq \#(assoc\ t) \bullet \\
\qquad \#(assoc\ t\ i) = size\ t) \\
usage = \Sigma(t : tags \mid t \in \mathrm{dom}\, assoc \bullet \\
\quad \#(assoc\ t) * size\ t) \\
usage \leq msize \\
\hline
\end{array}
$$

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. The two functions must have the same domain, the committed set must be a subset of this domain and all the information sequences associated with a tag must be of the length given by the size function. The total amount of information associated with all the tags should not exceed the size of the memory system. The non-standard Z construct Σ sums all values of the expression $\#(assoc\ t) * size\ t$.

The initial state of the memory system is described as follows:

$$
\begin{array}{|l}
\textit{AInitialMemSys} \\
\hline
AMemSys \\
\hline
\mathrm{dom}\, assoc = \varnothing \\
\hline
\end{array}
$$

The *ANewTag* operation returns an unused tag. The size of the information sequences to be written to the tag is specified as an argument $n?$ to this operation.

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

The operation returns an unused tag (one that has no associated sequence of information sequences), marks the most recent generation as empty, and records the expected length of the information sequences.

The *AReadGeneration* operation reads the information sequence of a given generation $g?$ associated with a tag. The tag $t?$ must have an associated information sequence of the given generation, numbered relative to the current generation.

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

The schema *CurrentGeneration* constrains a generation argument to the current generation:

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

The *ARead* operation reads the current generation of information associated with a tag. It is specified using schema conjunction and hiding.

$$
\begin{array}{l}
ARead \mathrel{\widehat=} (AReadGeneration \land \\
\qquad CurrentGeneration) \setminus (g?)
\end{array}
$$

The *ARelease* operation releases all the information associated with a tag. The operation does this by removing the tag from the domains of the functions *assoc* and *size*, and from the *committed* set.

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

The operation *ACommit* commits the current generation of information associated with a tag. The tag must have an associated information sequence, which is flagged as committed.

$$
\begin{array}{|l}
\multicolumn{1}{l}{\_\, ACommit\, \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_} \\
\Delta AMemSys \\
t? : tags \\
\hline
t? \in \operatorname{dom} assoc \\
assoc\ t? \neq \langle\rangle \\
committed' = committed \cup \{t?\} \\
\hline
\end{array}
$$

The operation $AWrite$ 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 by $AWriteFirst$, after $ANewTag$, must make sure there is enough room to write the new information. The association for the tag with a singleton sequence containing the new information sequence is replaced.

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

Writing to a tag whose current generation is not committed, by $AWriteUncommitted$, does not need any extra room.

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

Writing to a tag whose current generation has been committed by $AWriteCommitted$ requires extra room for the new information. In this case the new association is obtained by concatenating the new sequence in front of the existing one and then cropping the sequences of sequences by the maximum allowed generation.

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

Using schema disjunction the $AWrite$ operation is specified as follows:

$$
\begin{array}{l}
AWrite \mathrel{\widehat=} AWriteFirst \lor \\
\qquad AWriteUncommitted \lor AWriteCommitted
\end{array}
$$

This completes the presentation of the abstract specification of the transacted memory. In the following sections we will present the principles and the data structures of two refinements. The full specifications may be found in our technical report [2].

3 Refinement 1

EEPROM technology normally supports byte reads but only block writes. The block size is typically of the order of 8 ... 32 bytes. EEPROM technology allows a full block to be written efficiently, and we assume that a block is written atomically. It may be necessary to use a low level block write operation to achieve this. EEPROM lifetime is limited, so repeated writes to the same block must be avoided.

3.1 Data structures

To acknowledge these technological constraints, the first refinement introduces atomic operations over "pages" in terms of which all operations must be described. The two mappings *assoc* and *size*, and the set *committed* of the abstract specification are refined by four more concrete data structures. Before describing these, we introduce the definitions needed by the refinement. The first definition introduces a boolean flag.

$$\mathbb{B} ::= False \mid True$$

The EEPROM is treated as a sequence of pages, where each page contains a small amount of book-keeping information and a payload consisting of a single item of information from one of the original information sequences. The page size would typically be the block size of the

EEPROM technology. The type *Loc* below represents the locations of the pages in the memory. The type *Page* represents the actual data stored in each page, together with the book-keeping:

$$Loc == 0 \mathbin{..} (msize - 1)$$
$$Page == Info \times \mathbb{B} \times tags \times \mathbb{N} \times \mathbb{N}$$

The type *Page* contains five components:

1. *Info* represents one item from an information sequence, the actual payload.
2. The boolean flag states whether the page is actually in use.
3. *tags* represents the tag with which the current information sequence is associated.
4. The fourth component gives the generation index of the current information sequence.
5. The fifth component gives the page number of the item within the information sequence.

The refinement needs a small table, which records the essential data for each tag as type *TagData*.

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

The type *TagData* contains five components:

1. The first boolean flag states whether the tag is actually in use.
2. The second component states the size of the information sequence associated with this tag.
3. The third component states whether the current generation associated with the tag has been committed.
4. The fourth component gives the number of generations associated with the tag.
5. The fifth component gives the generation index of the current information sequence.

Having introduced the relevant types we are now in a position to show the four data structures that represent the state of the transacted memory.

MemSys

$$\begin{array}{l} data : tags \rightarrow TagData \\ mem : Loc \rightarrow Page \\ freetags : \mathbb{P}\, tags \\ freelocs : \mathbb{P}\, Loc \\ \hline \ldots \end{array}$$

The two sets *freetags* and *freelocs* are introduced for convenience. We refer to a technical report [2] for a description of the remainder of the refinement, and for a discussion of the abstraction invariant linking the state variables of the first refinement with the state variables of the abstract specification. The present refinement has not been verified, but we have verified an earlier refinement for a system without generations.

4 Refinement 2

The second data refinement describes the error states that may arise when a sequence of atomic page writes is interrupted. This may happen at any point, leaving the memory in error states not found during normal operation. These error states are therefore not present in the abstract specification or in the first refinement.

There are two different ways to handle erroneous states. The first approach is to modify the higher level specifications to allow for such erroneous states. The second refinement 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. The recording of such a state in a form that relates to the memory operations as seen by an application require repeated writes of the state information to some page in memory. This has to be avoided, so as not to wear out the EEPROM.

The second way to cope with erroneous states would allow all the error detection and recovery to be contained within the second refinement and hidden at some level within the final implementation of the system. This has been adopted and is described below.

4.1 Realistic Constraints

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

- a given page should be written as few times as possible. This means that a page should only be written to when there is 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 using only these write operations. The second refinement satisfies all these constraints 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 that may be performed on the memory is the atomic writing of a page.
- Any sequence of atomic write operations can be interrupted at any point. It should be possible to detect the resulting erroneous state and then to tidy up the memory.
- Lost memory should be recoverable when an atomic operation sequence is interrupted.
- All the constraints employed in the previous specifications should be retained, such as the main correctness requirement that the information read from a tag is equal to that previously written to that tag.

4.2 Causes of 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 not all the required pages may be successfully written.
- Writing a new version of the current generation 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 some of the pages of the old generation may not be released.
- When deallocating all the pages for a tag for the Release operation, some of the pages may not be released.

It is not possible to record a separate flag to track the current state of the memory manager as we would have to pick a page to keep it in which would then suffer from repeated writes as the state changed. Instead the presence of page zero has been chosen to indicate the presence of all the other pages of a generation. In addition, the information otherwise stored in a page is elaborated by a further piece of data:

- A cyclic, three state flag that makes it possible to determine the relative age of two versions of the same generation.

Here is the Z specification of the flag:

$$Version ::= VA \mid VB \mid VC$$

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.

4.3 Error Recovery

As checking for and remedying error states before each operation would be expensive, it was decided to wait until there is no choice but to reclaim the memory lost due to disrupted operations. Thus the presence of an error state in the memory manager will be noted by a Write operation failing to find sufficient free pages. An operation to tidy up the memory is invoked, which releases the lost pages for reuse. By performing some inexpensive local housekeeping in the operations, the complexity of the error states that can arise from repeated disruptions is restricted. This greatly simplifies the error recovery task. Below we list 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 is 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 after 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 exists. The Write operation is required to tidy up excessively old generations for the given tag.

Given this localised housekeeping it becomes possible to 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, locations may be marked as in use that are in fact not 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 the memory can be tidied, recovering locations lost due to interruption of a memory update. If there are still not enough free locations this indicates an unrecoverable error due to an application requiring more than the available memory. No attempt has been made to handle this error. Instead the user is required 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 makes it possible to avoid the additional complexity that would be required in the specifications when considering these additional sources of errors. In a development process involving verification of the use of operations, it should be possible to formally justify such simplifying assumptions.

4.4 Data structures

The type $DPage$ represents the refinement of the type $Page$:

$$DPage == \mathbb{B} \times tags \times Info \times \mathbb{N} \times \mathbb{N} \times Version$$

The type $DPage$ contains six components, which is a little more than the information kept by the first refinement:

1. The boolean flag states whether the page is actually in use.

2. $tags$ represents the tag with which the current information sequence is associated.

3. $Info$ represents one item from an information sequence, the actual payload.

4. The first number gives the generation index of the current information sequence.

5. The second number gives the page number of the item within the information sequence.

6. $Version$ is the cyclic flag that we mentioned in Section 4.2.

The type $DTagData$ represents the refinement of the type $TagData$.

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

The type $DTagData$ contains three components, i.e. considerably less than the information kept for the same purpose in the first refinement.

1. The first boolean flag states whether the tag is actually in use.

2. The second, numeric component states the size of the information sequence associated with this tag.

3. The last boolean flag states whether the current generation associated with the tag has been committed. This flag is only false upto the occurrence of the first write.

The data structures of the second refinement show the mappings that represent the state of the memory. No further data structures are used to maintain the transacted memory. Both mappings are supposed to be stored in EEPROM.

DMemSys

$ddata : tags \rightarrow DTagData$
$dmem : Loc \rightarrow DPage$

...

The abstract Z specification of Section 2 is (almost) standard Z notation. In the two refinements we felt the need to deviate more from standard Z to express important constraints such as the writing of pages to memory in a particular order. While it would be possible to specify this in Z, the specifications we came up with contained some elements that were less intuitive than say a simple for loop. Therefore we will not present further details of the Z version of the two refinements here [2]. Instead we discuss the essential elements of the SPIN and C version of the second refinement.

5 SPIN and C Prototype

The Prototype implements the two mappings $ddata$ and $dmem$ that form the core of the memory system as arrays. This is efficient, both in Promela (the modelling language of the SPIN tool) and in C:

```
#define  msize    10
#define  tsize    2
#define  DTagData byte
#define  DPage    short

DTagData ddata[tsize] ;
DPage    dmem[msize] ;
```

The domains of the mappings, *tags* and *Loc*, are represented by integers. The types *DTagData* and *DPage* are represented as a `byte` and a `short` respectively. Depending on the actual size of an information item, and the number of tags in the system larger sizes would be required. In any case the information must be tightly packed, as in a production implementation. An alternative would have been to use a `struct`. This would have been made it difficult to achieve the same information density, and it would not model reality accurately.

As a consequence, the various fields of the range types, as specified in Section 4.4, are accessed by a collection of macros. These macros work equally well in Promela as they do in C. For example reading the 'in use' flag of an element of the `ddata` array, and writing an entry in the same array are modelled as follows:

```
#define read_ddata1(t, u) \
   u =(ddata[t] >> inuse_shift) & inuse_mask

#define write_ddata3(t, u, s, c) \
   ddata[t] = \
      ((u) << inuse_shift) | \
      ((s) << size_shift) | \
      ((c) << committed_shift)
```

The shifts and masks are appropriately defined to pack and unpack the information. The remaining access operations are defined in a similar way.

5.1 DNewTag in C

Below is the C version of the *DNewTag* operation. Noteworthy is the `for` loop, which (inefficiently) locates a tag that is not is use, as stipulated by the predicate $t! \notin \operatorname{dom}\ assoc$ in the Z specification.

```
Tag DNewTag( Size size ) {
   Tag  tag ;
   bool taginuse ;

   for( tag = 0; tag < tsize; tag++ ) {
      read_ddata1(tag, taginuse) ;
      if( ! taginuse ) {
         write_ddata3(tag,
            true, size, false) ;
         break ;
      }
   }
   return tag ;
}
```

The Z specification also states that the *DNewTag* operation is undefined if the preconditions are not met, i.e. if there is no available tag. The C and SPIN prototype refine this specification by returning a value for `tag` that is outside the permitted range of `0 .. tsize-1`.

Given the encapsulation of the memory read and write operations by the macros `read_ddata1` and `write_ddata3`, the C version of the *DNewTag* operation is clear and concise.

5.2 DNewTag in Promela

The next point of interest is to compare the C version of *DNewTag* to the Promela version shown below. The first issue to be addressed is that Promela does not offer a function call mechanism. Instead function call/return must be simulated trough process creation and message passing [9]. This requires four steps.

The first step introduces a number of tags to distinguish the various messages required:

```
mtype {MSize, MTag, Mabort, Mdone, ... } ;
```

The second step introduces two channels – one to pass arguments to the simulated procedure, and another to return the results:

```
chan go_DNewTag   = [0] of { mtype, Size } ;
chan done_DNewTag = [0] of { mtype, Tag } ;
```

The third step models a procedure as a process, which continually waits for a message on its `go_...` channel, and responds on its `done_...` channel. A typical call to a procedure would send on the `go_...` channel and receive from the `done_...` channel:

```
init {
   go_DNewTag ! MSeq, 2 ;
   done_DNewTag ? MTag, tag ;
   ...
}
```

To allow SPIN to help discover errors in the specification a fourth step is need. Each procedure call may either complete successfully or it may abort. An abort would be triggered by a failed write operation to the EEPROM. Actual calls to a procedure call must therefore be prepared for two different kinds of response:

```
init {
   go_DNewTag ! MSeq, 2 ;
```

```
    if
    :: done_DNewTag ? MTag, tag -> ...
    :: done_DNewTag ? Mabort -> ...
    fi
}
```

The Promela version of the *DNewTag* operation is shown below. A non-deterministic choice is made at the second `if` statement either to perform the write to the EEPROM, or to abort the operation. Otherwise the code is the same as in the C version.

```
active proctype DNewTag( ) {
   Size size ;
   Tag  tag ;
   bool taginuse ;

endloop:
   do
   :: go_DNewTag ? MSeq, size ->
      tag = 0 ;
      do
      :: tag < tsize ->
         read_ddata1( tag, taginuse ) ;
         if
         :: ! taginuse ->
            if
            :: done_DNewTag ! Mabort ;
               goto endloop
            :: write_ddata3( tag,
                  true, size, false ) ;
               break
            fi ;
         :: else -> skip
         fi ;
         tag = tag + 1
      :: else -> break
      od ;
      done_DNewTag ! MTag, tag
   od
}
```

It is apparent that loops and other control statements are a bit more verbose in Promela than they are in C.

The SPIN model uses processes only to simulate procedures, not to introduce concurrency. Otherwise there could be no simple correspondence between the SPIN model and the C implementation. The SPIN model does use the non-determinism offered by SPIN to choose between successful and failed EEPROM writes.

5.3 DTidy

A second operation of interest is the *DTidy* operation, which performs the four steps explained in Section 4.3. The *DTidy* operation should be used once only, upon restart of the system i.e. after an aborted write operation.

The first phase of the operation is shown below. It detects and frees the locations in the `dmem` array that are marked as being in use by a tag that is itself marked as not in use, or that is not committed.

```
void DTidy( ) {
   Loc    loc ;
   bool   pageinuse ;
   Tag    tag ;
   Info   dpi ;
   Gen    dpx ;
   PageNo dpn ;
   Ver    dpv ;
   bool   taginuse ;
   Size   size ;
   bool   committed ;

   for( loc = 0; loc < msize; loc++ ) {
      read_dmem6( loc,
         pageinuse, tag, dpi, dpx, dpn, dpv ) ;
      if( pageinuse ) {
         read_ddata3( tag,
            taginuse, size, committed ) ;
         if( ! taginuse || ! committed ) {
            write_dmem6( loc,
               false, tag, dpi, dpx, dpn, dpv ) ;
         }
      }
   }
   ...
}
```

Here the scan of the entire tag array and the memory is unavoidable as the `Tidy` operation is intended to be used when the memory system is restarted after an aborted write. Short of scanning the entire collection of pages there is no way of knowing which pages belong to an aborted transaction.

The salient aspects of the C prototype and the SPIN model have now been covered. The complete list of data structures and functions of the transacted memory is shown in Table 1. The write operations will write the complete information sequence only if sufficient space is available.

5.4 Testing and assertion checking

The interest of the development of the prototype is in testing (C) and assertion checking (SPIN). Assertion checking in SPIN involves executing all possible execution paths of a finite program and testing assertions at various points in the execution paths.

```
typedef struct { Gen old, new ;
                 byte cnt ; } GenGenbyte ;
```
structure used to hold the number of the oldest and newest generation, and the number of generations.

```
typedef struct { Size size ;
             Info data[ssize] ; } InfoSeq ;
```
structure used to hold an information sequence and its size.

`GenGenbyte DGeneration( Tag ) ;`
Return all available information for the given tag. The result is undefined if the tag is not in use.

`Tag DNewTag( Size ) ;`
Return an unused tag of the specified size. The result is undefined if no tag is available.

`void DTidy( ) ;`
Recover from all possible interrupted writes.

`InfoSeq DReadGeneration( Tag, Gen ) ;`
Read the information sequence of a given tag and generation. The information sequence is undefined if the tag is not in use.

`InfoSeq DRead( Tag ) ;`
Read the information sequence of the current generation associated with the given tag.

`void DCommit( Tag ) ;`
Commit the current generation for the given tag. The operation has no effect if the tag is already committed.

`void DRelease( Tag ) ;`
Release all information associated with the given tag. The operation has no effect if the tag is not in use.

`void DWriteFirst( Tag, InfoSeq ) ;`
Write the to a tag immediately after the $DNewTag$ operation. The result is undefined if insufficient space is available.

`void DWriteUncommitted( Tag, InfoSeq ) ;`
Write to a tag whose current generation is uncommitted.

`void DWriteCommittedAddGen( Tag, InfoSeq ) ;`
Write to a tag whose current generation has been committed, and whose maximum number of generations has *not* been reached.

`void DWriteCommittedMaxGen( Tag, InfoSeq ) ;`
Write to a tag whose current generation has been committed, and whose maximum number of generations *has* been written. The oldest generation will be dropped.

Table 1: Transacted Memory data structures and functions for C.

To gain confidence in the prototype we wrote a simple test program. After some initialisation, the test program writes 16 generations of information for one particular tag. After each write operation, the test program reads back all existing generations and asserts that the information read back is correct.

Each write operation will be interrupted at least once, leading to an error state. The `DTidy` operation is then called upon to recover from the error. Since `DTidy` performs write operations as part of the recovery process, these writes may be interrupted as well, leading to further calls to the `DTidy` operation.

The test performs over 2000 successful write operations and 65 aborted writes, without a single assertion violation.

The above protocol initially revealed a number of assertion failures, due to clerical errors made while interpreting the Z specification in the transition to the prototype. Once these errors were corrected a number of more serious issues were found. These will be discussed in the next section.

6 Lessons learned

A variety of lessons were learned, about the specification process, and about the transacted memory system itself. We will discuss each, starting with the process.

The specifications indicated in Figure 1 were made by different authors. Butler started with an abstract specification and a refinement of an initial version of the system. This refinement was formally verified by hand. The further refinements of the revised version were not formally verified but the development of abstraction invariants did help to increase the confidence in the refinements.

The next step in the development process was an evaluation of the specifications, leading to a revised version and further refinement of the specification (by Longley). Then the prototype was created (by Hartel) on the basis of the earlier specifications. At each stage there was a fresh opportunity for making mistakes, from which, of course, we learn.

The original Z specifications contain non-standard Z constructs. The abstract specification contains a summation construct Σ, and the two refinements contain procedures and for loops. This made it impossible for tools such as ZTC [10] to be used. To asses whether this would be a serious problem we commented the Σ construct out of the abstract specification and ran it through ZTC. This revealed a number of clerical errors:

- Three occurrences were found by the ZTC syntax check of missing parentheses, writing $\#f(x)$ instead of $\#(f(x))$.
- Five occurrences were found of a misspelled variable name by the ZTC syntax check.

- One occurrence of a missing operator was found by the ZTC type checking, where we wrote $\#x = y$ instead of $\#x = \#y$.

- One occurrence was found by the ZTC type checker of an incorrectly used operator ($a \times b$ instead of $a * b$).

The abstract specification is relatively small (2 pages, or 10 schemas and two auxiliary definitions). Yet we found 10 clerical errors.

During the manual translation from the second refinement into Promela a number of errors were found, some of which of a serious nature.

- Two occurrences were found of misspelled identifiers.

- One occurrence was found of an auxiliary function whose definition was missing.

- Three occurrences were found of an auxiliary function definition part of an earlier refinement that was implicit reused in a later refinement.

- Consider the following Z example, consisting of state and an operation on the state:

  $$\begin{array}{|l}\hline \multicolumn{1}{l}{A} \\ a : \mathbb{N} \\ b : \mathbb{N} \\ \hline \end{array}$$

  Assume that the operation *Inca* increments a but leaves b untouched.

  $$\begin{array}{|l}\hline \multicolumn{1}{l}{Inca} \\ \Delta A \\ \hline a' = a + 1 \\ \hline \end{array}$$

  Some operations in the two refinements explicitly constrain $b' = b$, and some omit this. This is clearly inconsistent.

Three serious problems were found:

- In the second refinement the two committed write operations made the tag uncommitted instead of committed. This error was found by inspection.

- Instead of the correct version given in Section 5.1, the second refinement stated the equivalent of this `if` statement:

```
if( ! taginuse ) {
   write_dmem6( loc,
      false, tag, dpi, dpx, dpn, dpv ) ;
}
```

  The incorrect version thus failed to release pages with uncommitted data. This error was found by C testing.

- The *ACommit* operation as specified in Section 2 lacks a predicate $t? \notin committed$ and thus permits the *ACommit* operation to be repeated. The interpretation leading to the prototype was created in a 'defensive' style, by systematically excluding all states in which an operation was not considered to be applicable. This lead to a discrepancy between the more permissive specification and the more restrictive prototype. The discrepancy was found by Spin's assertion checking.

Consistent with our earlier findings [5], we believe that using more than one tool/notation has benefits that contribute to the accuracy of the resulting specifications/implementations. In the present case, the prototype was created in 10 days, using the second refinement as a starting point. Creating the second refinement took several months to complete. The cost of providing a 'second opinion' on a specification by translating it into a different language/notation can thus be qualified as low.

7 Conclusion and Future work

Our ideas of the transacted memory manager have become more and more accurate as progress was made on the various, more and more detailed specifications. The specifications served as a clear and unambiguous basis for discussions.

The combination of creating high level specifications in Z and detailed specification in Promela worked well for the memory manager. The high level Z specifications are clearer than the SPIN models would have been and conversely, the detailed SPIN models are clearer than the detailed Z specifications. Where we would have used proofs to establish properties of the operations in Z, we used assertion checking for the SPIN model. The manual translation from the Detailed Z specification to a SPIN model is the weakest link in the chain of specifications.

An ad-hoc common notation has been used, from which both a SPIN model and C prototype are generated by expedient use of simple macros. This gives a reasonable degree of confidence that the C prototype is consistent with the SPIN model. SPIN's concurrency has not been used, but its facility for making a non-deterministic choice has.

Using different languages and associated tools to specify and prototype a specification automatically provides a second, and even a third opinion on various important issues. We discovered a considerable number of problems in earlier specifications when working on later specifications.

The cost of providing a second opinion on the transacted memory manager was low.

Each operation of the prototype requires only a small, constant amount of RAM space, proportional to the number of generations associated with a tag. No RAM space is retained in between operations. However, to speed up some of the operations, a further refinement could be made to cache often used data.

The transacted memory that we have presented offers the basic facilities for building a type safe transaction facility. In addition, such a facility would also require a marshalling and unmarshalling capability. We are currently pursuing this for the use with Java.

The many for loops in the prototype may seem to introduce considerable inefficiency. However, we intend to produce further refinements down to the hardware level, where for loops taking a fixed number of steps could be 'unrolled' to form parallel hardware structures. The intention is to develop an FPGA based prototype.

One of our goals is to achieve certification of the transacted memory manager at EAL7 of the Common Criteria. This would require verification of the two the refinement steps as well as further formal refinements down to the hardware level. At this stage we are not sure whether Z is the most appropriate notation for this purpose. We are considering to rewrite the specifications using B since it provides better support for the refinement to code.

References

[1] Ph. A. Bernstein and E. Newcomer. *Principles of Transaction Processing.* Morgan Kaufman, San Francisco, 1997.

[2] M. J. Butler, P. H. Hartel, E. K. de Jong, and M. Longley. Applying formal methods to the design of smart card software. Declarative Systems & Software Engineering Technical Reports DSSE-TR-97-8, University of Southampton, 1997. `www.dsse.ecs.soton.ac.uk/ techreports/ 97-8.html`.

[3] E. K. de Jong and J. Bos. *Arrangements for storing different versions of a set of data in separate memory areas and method for updating a set of data in a memory.* Dutch Patent Application, 2000.

[4] D. Donsez, G. Grimaud, and S. Lecomte. Recoverable persistant memory of smartcard. In J.-J. Quisquater and B. Schneier, editors, *3rd Int. Conf. Smart card research and advanced application (CARDIS), LNCS 1820*, page to appear, Louvain la Neuve, Belgium, Sep 1998. Springer-Verlag, Berlin.

[5] P. Hartel, M. Butler, A. Currie, P. Henderson, M. Leuschel, A. Martin, A. Smith, U. Ultes-Nitsche, and B. Walters. Questions and answers about ten formal methods. In S. Gnesi and D. Latella, editors, *4th Int. Workshop on Formal Methods for Industrial Critical Systems, Vol II*, pages 179–203, Trento, Italy, Jul 1999. ERCIM/CNR, Pisa, Italy. `www.dsse.ecs.soton.ac.uk/ techreports/ 99-1.html`.

[6] P. H. Hartel, M. J. Butler, and M. Levy. The operational semantics of a Java secure processor. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java, LNCS 1523*, pages 313–352. Springer-Verlag, Berlin, 1999. `www.dsse.ecs.soton.ac.uk/ techreports/ 98-1.html`.

[7] P. H. Hartel and E. K. de Jong Frz. Towards testability in smart card operating system design. In V. Cordonnier and J.-J. Quisquater, editors, *1st Int. Conf. Smart card research and advanced application (CARDIS)*, pages 73–88, Lille France, Oct 1994. Univ. de Lille, France. `ftp.wins.uva.nl/ pub/ computer-systems/ functional/ reports/ CARDIS94_smartcard.ps.Z`.

[8] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for Lock-Free data structures. In *Int. Symp. in Computer Architecture (ICSA)*, pages 289–300, San Diego, California, May 1993. Computer Architecture News, 21(2).

[9] G. J. Holzmann. *Design and Validation of Computer Protocols.* Prentice Hall, Englewood Cliffs, New Jersey, 1991.

[10] Xiaoping Jia. *ZTC: A Type Checker for Z – User's Guide.* Dept. of Comp. and Inf. Sci, DePaul Univ.,Chicago, Illinois, May 1995. `ftp.comlab.ox.ac.uk/ pub/ Zforum/ ZTC-1.3/ guide.ps.Z`.

[11] S. M. Nettles and J. M. Wing. Persistence+undoability=transactions. In *25th Hawaii International Conference on System Sciences (HICS)*, pages 832–43. IEEE Comput. Soc. Press., Los Alamitos, California, 1991.

[12] National Institute of Standards and Technology. *Common Criteria for Information Technology Security Evaluation.* U. S. Dept. of Commerce, National Bureau of Standards and Technology, Aug 1999. `http:// csrc.nist.gov/ cc/`.

[13] Sun. *Java Card 2.1 Runtime Environment (JCRE) Specification.* Sun Micro systems Inc, Palo Alto, California, Jun 1999. `http:// java.sun.com/ products/ javacard/`.