A *VDM*$^\clubsuit$ Study of Fault-Tolerant Stable Storage – Towards a Computer Engineering Mathematics

Andrew Butterfield

Department of Computer Science, Trinity College, Dublin 2, Ireland

Abstract. This paper presents early results of research work being carried out on applying formal methods to the analysis of Stable Storage (Lampson 1981), which is a particular form of Fault Tolerance (Johnson 1989) adopted for data storage systems A prime concern is the development of the methods of the *Irish School of VDM* (*VDM*$^\clubsuit$) (Mac an Airchinnigh 1990) as applied to this application as an effective engineering mathematics discipline. Early results of the modelling are reported involving both the use of the formalism and understanding of the application area gained from the modelling. Also emerging from the research are suggestions of possible new operators that might be added to the calculus to make it a more effective modelling tool, as well as new extensions to the formal method itself.

1 Introduction

This paper presents early results of research work being carried out on applying formal methods to the analysis of *fault-tolerance* (Johnson 1989) in general, and *Stable Storage* (Lampson 1981) in particular. The goals of the research are to produce adequate models of such fault tolerance to assist the FASST research project[1] and to improve and develop the formal method employed, which is the *VDM* (Bjørner and Jones 1978, Bjørner et al. 1987) as modified by the *Irish School* (Mac an Airchinnigh 1990, 1991). The *Irish School of VDM* (*VDM*$^\clubsuit$) places VDM in a framework of Applied Constructive Mathematics, basing its reasoning on proving the equality of expressions by substitution of equals, as used in conventional engineering mathematics. This should be contrasted with the approach of Jones (1990) which uses the Logic of Partial Functions (LPF) as its underlying mathematical base. It is the main contention of the Irish School that the constructive mathematics approach, having many similarities to conventional engineering mathematics, is easier to use and henceforth more effective than those formal methods which rely on some form of logic.

Both goals are seen as synergistic — the development of the stable storage sub-system is hard industrial research and development that requires very rigourous reasoning if it is to succeed — while the goal of developing a truly industrial strength set of models and methods will be assisted considerably by the fact that they are being developed in tandem with such a project.

[1] ESPRIT P 5212 Fault Tolerant Architectures using Stable Storage

Before proceeding with the details of concern in this paper, it is worthwhile mentioning various aspects of the VDM that are not mentioned here. This paper has no examples of reification (Andrews, 1987, Jones 1990, Ch. 8., p179) as it has been discussed elsewhere and is not the *present* focus of this work which involves the construction of high level abstractions. Nor do pre-conditions play a major rôle here as the intention of modelling fault tolerant systems is to come up with models valid under all circumstances. The absence of reification or pre-conditions must not be interpreted as meaning that they are absent from $\mathit{VDM}^{\clubsuit}$. They exist and are used in the same fashion as those found in other "Schools".

The notation used is largely that of VDM, except where it clashes with established mathematical notations. The $\mathit{VDM}^{\clubsuit}$ tends to adopt conventional notation where a clash arises, in keeping with its philosophy which eschews the use of automated tools and favours much use of hand-written analysis. A guide to the notation used here is given in Appendix A at the end of this paper.

2 Ideal Memory

We start with a brief description of an abstract model of ideal memory, complete with read and write operations. A more detailed discussion can be found in (Butterfield 1992b). The precise nature of addresses and the stored values is not important at this level of detail, so they will be treated simply as being drawn from appropriate sets, which are considered to be *finite*. In particular, values could be bytes or pages, in solid-state memory or on magnetic media. Memory is modelled as a mapping from addresses to values, with Read (R) and Write (W) operations modelled as map application and override respectively:

$$a \in \mathit{ADDR}\,, \quad v \in \mathit{VAL}$$
$$\mu \in \mathit{MEM} = \mathit{ADDR} \overset{m}{\to} \mathit{VAL}$$

$$R : \mathit{ADDR} \to \mathit{MEM} \to \mathit{VAL}$$
$$R(a)\mu \triangleq \mu(a) \tag{1}$$

$$W : \mathit{ADDR} \times \mathit{VAL} \to \mathit{MEM} \to \mathit{MEM}$$
$$W(a,v)\mu \triangleq \mu + \{a \mapsto v\} \tag{2}$$

Note 1. As we are considering general memory systems, that we hope will run correctly all the time, all of the the memory accessing operators defined in this paper are considered to have a TRUE pre-condition. Any erroneous events leading to some form of failure will be explicitly modelled as will be seen later, and will always be defined, regardless of the state of the memory.

Note 2. The operators are defined using constructive post-conditions which specify the results as expressions. This should be contrasted with post-conditions expressed in the form of predicates which must be satisfied by the outputs.

This latter approach gives rise to a *proof obligation* (Jones 1987, 1990) to show that outputs meeting post-condition do in fact exist. The former approach, as adopted by the $\mathit{VDM}^\clubsuit$ incorporates just such a proof, as the post-condition has been *constructed.*

Note 3. value The address (a) and data (v) arguments of these operators have been been separated from the memory arguments (μ) by the technique of *currying* (Schonfinkel 1924, Curry 1958). This allows use to interpret $R(a)$ as an operation that reads from address a of any memory, and $W(a, v)$ as an operator that writes v into address a of any memory.

Given this model it is easy to show some key properties regarding the effects of multiple Writes to the same or different addresses and the effect of Writes on subsequent Reads:

$$(R(a) \circ W(a', v))\mu = \textbf{if } a' = a \textbf{ then } v \textbf{ else } R(a)\mu \tag{3}$$

$$(W(a, v) \circ W(a', v'))\mu = \tag{4}$$

$$\textbf{if } a' = a \textbf{ then } W(a, v)\mu \textbf{ else } (W(a', v') \circ W(a, v))\mu$$

These properties are fairly obvious, but are presented here so that they can be contrasted and compared with later results.

2.1 Error Detecting Memory

2.2 The Model

The key feature of error-detecting memories is some form of encoding that builds in redundant error detection data, with an associated decoder that extracts the original data along with some indication of possible errors. Our first model of *error-detecting memory* (*EDM*) avoids any explicit mention of an encoding scheme—it presumes that the *VAL* component of perfect memory is replaced by a $\mathbf{B} \times \mathit{VAL}$ pair, where the boolean flag is set to TRUE if no error has been *detected* in the data. In all that follows it is important to note that the flag models the knowledge the memory has of the condition of its data. A TRUE flag does not necessarily signify that no error has occurred, and indeed won't do so if an undetected error has taken place.

$$\mu \in \mathit{EDM} = \mathit{ADDR} \xrightarrow{m} (\mathbf{B} \times \mathit{VAL})$$

The relationship between *MEM* and *EDM* is not one of reification involving abstraction and representation. Error Detecting Memory is viewed within the $\mathit{VDM}^\clubsuit$ as an *Elaboration* of the *MEM* model (Mac an Airchinnigh 1990). The "perfect" Read and Write operators are replaced by Lampson (1981) with "imperfect" analogues called Get (G) and Put (P). These take additional *event* parameters which model the possible changes that might occur to data during memory operations. These events are modelled as *total functions* which express how the data actually stored or retrieved is related to the original specified data.

$$\varepsilon_r \in \mathit{R\_EVT} = (\mathbf{B} \times \mathit{VAL}) \to (\mathbf{B} \times \mathit{VAL})$$
$$\varepsilon_w \in \mathit{W\_EVT} = \mathit{VAL} \to \mathit{R\_EVT}$$

Note 4. An instance of ε_w has the form $\varepsilon_w[\![v]\!](b, w) \triangleq (b', v')$ where the single value v represents the new value being written to the memory, while the $\mathbf{B} \times \mathit{VAL}$ pair (b, w) denotes the previous contents. The result (b', v') of the event function is the (boolean,value) pair that is actually written. The brackets used ($[\![\ ]\!]$) are simply intended to highlight the curried arguments.

The reason for choosing functions, rather than erroneous values, to represent events is that functions can capture context-dependent errors (such as bit-toggling) which cause the erroneous value to depend on the previous value.

$$\begin{aligned} G &: \mathit{R\_EVT} \to \mathit{ADDR} \to \mathit{EDM} \to \mathbf{B} \times \mathit{VAL} \\ G[\![\varepsilon_r]\!](a)\mu &\triangleq \varepsilon_r(\mu(a)) \end{aligned} \tag{5}$$

$$\begin{aligned} P &: \mathit{W\_EVT} \to \mathit{ADDR} \times \mathit{VAL} \to \mathit{EDM} \to \mathit{EDM} \\ P[\![\varepsilon_w]\!](a, v)\mu &\triangleq \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \end{aligned} \tag{6}$$

Note 5. The choice of the signatures of the Put and Get operator is dictated by a desire to separate the events from the specifics of the data and addresses being used as much as possible, as one aim of the model is to be able to consider events in isolation. However there some key areas where this is not straightforward or possible as will be discussed shortly.

Comparison with *MEM*. In practice, we hope that errors are few and far between! We need to be able to model situations when no errors are taking place within the same framework. This is very straightforward—Error-free Puts and Gets will use *Identity Event* functions ($\varepsilon_w^{\mathcal{I}}$ and $\varepsilon_r^{\mathcal{I}}$ respectively).

$$\begin{aligned} \varepsilon_w^{\mathcal{I}} &: \mathit{W\_EVT} \\ \varepsilon_w^{\mathcal{I}}[\![v]\!](b, w) &\triangleq (\textsc{true}, v) \end{aligned} \tag{7}$$

$$\begin{aligned} \varepsilon_r^{\mathcal{I}} &: \mathit{R\_EVT} \\ \varepsilon_r^{\mathcal{I}} &\triangleq \mathcal{I} \end{aligned} \tag{8}$$

The first thing that should be shown is that error-free Gets and Puts in *EDM* behave just like the Reads and Writes of *MEM*. The full detail of this is to be found in (Butterfield 1992b) and presents no great difficulties, as long as we restrict *EDM* to those cases where only TRUE occurs in the stored tuples, thus denoting memories where no errors have been *detected.* We just sketch the details here. Essentially we introduce the notion of a *Restricting Invariant* ($\mathit{inv}\text{–}D_I$) which limits a domain D to some subset (D_I) that has desirable properties. We also introduce a *Partial Retrieve* function ($\mathit{retr}_p\text{–}E$) from D_I to another domain (E) with which the intended comparison is being made. In effect, the restricting invariant acts as a pre-condition for the partial retrieve function.

$$\begin{aligned} \mathit{inv\text{–}EDM_r} &: \mathit{EDM} \to \mathbf{B} \\ \mathit{inv\text{–}EDM_r}(\mu) &\triangleq (^\wedge\!/ \circ \mathcal{P}(\pi_1) \circ \mathrm{rng})\mu \end{aligned} \tag{9}$$

$$\begin{aligned} \mathit{retr_p\text{–}MEM} &: \mathit{EDM_r} \to \mathit{MEM} \\ \mathit{retr_p\text{–}MEM}(\mu_r) &\triangleq (\mathcal{I} \overset{m}{\to} \pi_2)\mu_r \end{aligned} \tag{10}$$

The problem then reduces to proving the following identities (Butterfield 1992b):

$$W(a,v) \circ \mathit{retr_p\text{–}MEM} = \mathit{retr_p\text{–}MEM} \circ P[\![\varepsilon_w^{\mathcal{I}}]\!](a,v) \tag{11}$$

$$R(a) \circ \mathit{retr_p\text{–}MEM} = \pi_2 \circ G[\![\varepsilon_r^{\mathcal{I}}]\!](a) \tag{12}$$

What is of interest here is the notion that an *elaboration* of a model can be mapped back onto the original model if a suitable *Restricting Invariant* is found. However, it must also be stressed that the relationship here is not of reification. In particular, there is no requirement to show how elements of *EDM* that contain FALSE flags are related to elements of *MEM* as there is no correspondence in *MEM* to such erroneous values.

Properties of the Model. We now proceed to examine the *EDM* model in more detail. First note that the model does not include scope for addressing errors—other than by explicitly using an address that is declared to be 'wrong'. This is not a serious omission at present because address errors, are a disaster, as far as the fault tolerant stable storage systems in this paper are concerned.

Event Examples. Two important examples of Write Events are the Null Write (ε_w^{ϕ}), where no data is changed at all, and the set of Decay Events (ε_w^{δ}) which indicate the corruption of data while sitting in memory. Decay can be modelled by a Put operation with a Write Event that ignores the Put's *VAL* parameter

$$\varepsilon_w^{\phi}[\![v]\!](b,w) \triangleq (b,w) \tag{13}$$

$$\varepsilon_w^{\delta}[\![v]\!](b,w) \triangleq \delta(b,w)\,\mathbf{where}\,\delta \in \mathit{R\_EVT} \tag{14}$$

The Null Write event during a Put operation illustrates an important point regarding the interpretation of the *EDM* model. Such an event will normally be considered an error by any observer, even though the resulting contents of memory may be flagged with TRUE and actually be the previously correct data that was stored before the Put occurred. This data is incorrect as the correct outcome of the Put operation should have been a TRUE flag with the *new* data. To re-iterate: *the value of the flag only models the error detecting memory's own perception of the state of the data.*

Note that both examples above show that some classes of Write Event functions make use of existing values in memory, rather than overriding them the fashion an ideal Write operation. We have here a first classification of Write

Events which distinguishes between *History-Preserving* and *History-Breaking* Write Events. A History Breaking event is one where the resulting data is independent of the previous contents of memory, and can always be expressed in the form $\varepsilon_w[\![v]\!](b,w) \triangleq \varepsilon_r(\textsc{true}, v)$ where ε_r is the equivalent Read Event. The Identity Write function ($\varepsilon_w^{\mathcal{I}}$) is the most obvious (and hopefully most frequent) example of a History Breaking Event.

Operator Composition. As with the ideal memory model, it is now necessary to investigate the effects of composing Puts and Gets, with the expectation that the presence of Write Event functions that are history-preserving will complicate matters. We find that the effect of Get after Put is much the same as observed for Reads and Writes (3):

$$\begin{aligned}&(G[\![\varepsilon_r]\!](a) \circ P[\![\varepsilon_w]\!](a',v))\mu \\ &= \textbf{if}\, a' \neq a \,\textbf{then}\, G[\![\varepsilon_r]\!](a)\mu \,\textbf{else}\, (\varepsilon_r \circ \varepsilon_w[\![v]\!])\mu(a)\end{aligned} \tag{15}$$

However, the relationship between successive Puts to the same address is more complex. Using the definition of Put twice with events ε_w and ε'_w gives the following identity:

$$(P[\![\varepsilon_w]\!](a,v) \circ P[\![\varepsilon'_w]\!](a,v'))\mu = P[\![\varepsilon_w]\!](a,v)(P[\![\varepsilon'_w]\!](a,v')\mu) \tag{16}$$

However, the desired result is of the form

$$(P[\![\varepsilon_w]\!](a,v) \circ P[\![\varepsilon'_w]\!](a,v'))\mu = P[\![\varepsilon''_w]\!](a,v)\mu \tag{17}$$

where ε''_w is the single event that is equivalent to the afore-mentioned two. To achieve this we introduce a version of function composition that is generalised to handle the presence of curried arguments. The *General Function Composition* operator ($\odot$) is ternary, taking the two functions to be composed as well as the curried argument of the first function to be applied (Butterfield, 1992a). The following equations give a definition of this operator and illustrates one of its key properties (a form of Associativity):

$$(f \odot_{x'} g)[\![x]\!]y \triangleq (f[\![x]\!] \circ g[\![x']\!])y \tag{18}$$

$$f \odot_x (g \odot_{x'} h) = (f \odot_x g) \odot_{x'} h \tag{19}$$

This operator enables us to produce a combination of event functions and some context values in such a way as to produce an expression that is itself an event function (i.e. has the same signature). This allows us to maintain the desired separation of events from the data being inserted into memory. Given this operator we can then describe the effect of two successive Puts to the same address as follows:

$$P[\![\varepsilon_w]\!](a,v) \circ P[\![\varepsilon'_w]\!](a,v') = P[\![\varepsilon_w \odot_{v'} \varepsilon'_w]\!](a,v) \tag{20}$$

where we can say now, that $\varepsilon''_w = \varepsilon_w \odot_{v'} \varepsilon'_w$. This should be compared to (5). The key result of all of this is that the effect of a sequence of Puts to one address is given by a single Put with the appropriate composition of event functions:

$$P[\![\varepsilon^n_w]\!](a, v_n) \circ P[\![\varepsilon^{n-1}_w]\!](a, v_{n-1}) \circ \cdots \circ P[\![\varepsilon^2_w]\!](a, v_2) \circ P[\![\varepsilon^1_w]\!](a, v_1)$$
$$=$$
$$P[\![\varepsilon^n_w \odot_{v_{n-1}} \varepsilon^{n-1}_w \odot_{v_{n-2}} \cdots \odot_{v_2} \varepsilon^2_w \odot_{v_1} \varepsilon^1_w]\!](a, v_n) \tag{21}$$

Note that the composition depends on both the events $\varepsilon^1_w \ldots \varepsilon^n_w$ and the context $(v_1 \ldots v_{n-1})$ in which they occur. This context dependence is important, and the use of the $\odot$ operator highlights precisely what this dependence is. Despite an desire, expressed earlier, to separate the events from the data and addresses involved with Put operators, we see that cannot be achieved for successive Puts to one address. The outcome of a sequence of general events depends intimately on the values present in memory before the events occur. This is most clearly seen in the expression $\varepsilon^n_w \odot_{v_{n-1}} \varepsilon^{n-1}_w \odot_{v_{n-2}} \cdots \odot_{v_2} \varepsilon^2_w \odot_{v_1} \varepsilon^1_w$ which suggests visually the interleaving of the composition of the write events with the values that the Puts are attempting to write to the memory Adding a new operator should always be approached with care, lest it be too specialised to be of any use outside the problem domain for which it was devised. An indication of other possible uses for $\odot$ is given in Appendix C.

2.3 Careful Memory

In (Lampson 1981), the next step was to define "Careful" versions of Put and Get. There, they are viewed as more fault-tolerant versions implemented using Get and Put as building blocks, but we will treat them as additional operators over the same *EDM* model.

CarefulGet. The following quote describing CarefulGet is from Lampson (1981).

> "*CarefulGet* repeatedly does *Get* until it gets a *good* status, or until it has tried *n* times"

Note that this implementation makes no explicit mention of errors. To model the fault tolerant aspects of CarefulGet (*CG*) we need to introduce the notion of a *sequence of Read Events* which will be an extra argument to the CarefulGet operation. It is then straightforward to give a recursive definition of the CarefulGet operator in terms of Get, that matches the above implementation description, except that the premature exhaustion of the read events is interpreted as meaning that a crash occurred before the CarefulGet operation could return any results. This is indicated by $\perp$, which is used in the *VDM*♣to denote a "do not care" situation, as well as "undefined" (Mac an Airchinnigh 1990). The use of a pre-condition to exclude $\perp$ results is not appropriate, as this would exclude crash conditions from those deemed as "valid inputs" to *CG*. As the data returned is not defined should the flag be FALSE, this situation is denoted here by the form (FALSE, _). This is the equivalent to the non-deterministic post-condition of more conventional *VDM* (Jones, 1990, p104 for example) as the '_' marker indicates a slot where any value (of the appropriate type) will suffice.

$$\varsigma_r \in \mathit{R\_EVTS} = \mathit{R\_EVT}^\star \tag{22}$$

$$\begin{aligned} &\mathit{CG} \;:\; \mathit{R\_EVTS} \to \mathit{ADDR} \to \mathit{EDM} \to \mathbf{B} \times \mathit{VAL} \\ &\mathit{CG}[\![\varsigma_r]\!](a) \triangleq \mathit{CG}'[\![n, \varsigma_r]\!](a) \end{aligned} \tag{23}$$

$$\mathit{CG}'[\![0, \varsigma_r]\!](a)\mu \triangleq (\text{FALSE}, \_) \tag{24}$$

$$\mathit{CG}'[\![k, \Lambda]\!](a)\mu \triangleq \bot \tag{25}$$

$$\begin{aligned} &\mathit{CG}'[\![k, \varepsilon_r{:}\varsigma_r]\!](a)\mu \triangleq \\ &\qquad \mathbf{let}\,(b, v) = G[\![\varepsilon_r]\!](a)\mu\,\mathbf{in} \\ &\qquad \mathbf{if}\,b\,\mathbf{then}\,(b, v)\,\mathbf{else}\,\mathit{CG}'[\![k-1, \varsigma_r]\!](a)\mu \end{aligned} \tag{26}$$

A key property (whose proof is straightforward) can be immediately stated:

$$\mathit{CG}[\![\varsigma_r]\!](a) = \mathit{CG}[\![\varsigma_r[1 \ldots n]]\!](a) \tag{27}$$

where $[1 \ldots n]$ selects the first n elements of a sequence.

A more important property, that is discussed in more detail here, is that the result of a CarefulGet operation with a given Read Event Sequence can be reduced to that of a Get operation with an single equivalent Read Event. This equivalent Read Event is called the *Get-Equivalent Form* (*GEq*) of the sequence and is derived from the given sequence, as well as a consideration of the actual contents of memory. The only difference is the treatment of crashes, which will be discussed later.

We already have one result regarding the fact that only the first n elements of the sequence matter. The next result is obtained by noting that the address being read during a *CG* operation is always the same as is the (b, v) value being handled by the read events. So each event in the sequence has the same context. We also note that the following occasions when *CG* will terminate:

- at the first occurrence of an event that results in (TRUE, _).
- if the first n events result in (FALSE, _).

A case that needs to be examined is one where all the events result in (FALSE, _), but the number of those events is less than n. In other words what has occurred is a crash, after (so-far) persistent read errors. It can be shown that, in the event of a crash, there is no single read event equivalent to the sequence. We can define a predicate *Crsh* that indicates if a sequence will result in a crash, given the existing contents of memory:

$$\begin{aligned} &\mathit{Crsh} \;:\; \mathbf{B} \times \mathit{VAL} \to \mathit{R\_EVTS} \to \mathbf{B} \\ &\mathit{Crsh}[\![b, v]\!]\varsigma_r \triangleq \text{len}\varsigma_r < n \\ &\qquad\qquad \wedge \\ &\qquad\qquad \text{elems}((\mathit{Done}_G[\![b, v]\!])^\star \varsigma_r) \subseteq \{\text{FALSE}\} \\ &\qquad\qquad \mathbf{where}\,\mathit{Done}_G[\![b, v]\!]\varepsilon_r = \pi_1 \varepsilon_r(b, v) \end{aligned} \tag{28}$$

Note 6. When applied to a read event, $Done_G[\![b, v]\!]$ returns TRUE if CarefulGet would terminate after that event.

The *Crsh* predicate serves to act as a pre-condition for *GEq*. The Get-Equivalent Form is defined as follows:

$$GEq : \mathbf{B} \times VAL \to R\_EVTS \to R\_EVT$$

$$pre\text{–}GEq[\![b, v]\!]\varsigma_r \triangleq \neg Crsh[\![b, v]\!]\varsigma_r \tag{29}$$

$$GEq[\![b, v]\!]\varsigma_r \triangleq \varsigma_r[\min\{n, f_T\}] \tag{30}$$

$$\textbf{where}\, f_T = fstloc[\![\{\text{TRUE}\}]\!]\beta$$

$$\textbf{where}\, \beta = (Done_G[\![b, v]\!])^\star \varsigma'_r$$

$$\textbf{where}\, \varsigma'_r = \varsigma_r[1 \ldots n]$$

Note 7. When applied to a sequence, $fstloc[\![S]\!]$ returns the index of the first occurrence of a member of S in the sequence. If none are found it returns the sequence length plus one.

Note 8. The sequence β consists of booleans indicating whether each event would cause CarefulGet to halt. It is produced by mapping $Done_G[\![b, v]\!]$ onto every element of ς'_r.

The result is the single error which, if it occurred when a G was attempted, would have the same result as the CG attempted with the sequence of errors

The key property of Get-Equivalent Forms is as follows:

$$\neg Crsh[\![\mu(a)]\!]\varsigma_r \Rightarrow CG[\![\varsigma_r]\!](a)\mu = G[\![GEq[\![\mu(a)]\!]\varsigma_r]\!](a)\mu \tag{31}$$

The proof of this is quite extensive, by induction on n and ς_r, and can be found in (Butterfield 1993b).

CarefulPut. The following quote describing CarefulPut is from Lampson (1981).

> "*CarefulPut* repeatedly does *Put* followed by *Get* until the *Get* returns *good* with the data being written"

The most important thing to note here is the complete absence of the parameter n. CarefulPut keeps trying until it succeeds *or crashes.*

Question 9. How should errors and events be modelled here ? We have *alternating* Puts and Gets with the possibility of a crash inbetween at any point!

Various alternatives are discussed in (Butterfield 1993b), with the method of choice being to use sequences of Write Events. When the Write Events are being fed into the Get operator (every second event in the sequence), they are first applied to the value (v) that the CarefulPut (*CP*) is trying to write. This results in a Read Event which is context sensitive and can depend on both the existing memory contents *and* the value v. Given sequences of Write and Read Events, it is possible to produce such a single Write Event Sequence denoting their combined effect during a CarefulPut operation by:

1. "Lifting" every Read Event to produce an equivalent Write Event. Such lifted Read Events will be denoted by ε_r^w or ε_r^p: $\varepsilon_r^w[\![v]\!](b,w) \triangleq \varepsilon_r(b,w)$
2. Zipping together the lists, but alternating elements from each, starting with the original Write Events.

For the Get operator in general there is no "context" (what *VAL* entity would act as the first argument ?). However, in the case of CarefulPut, a natural choice for such an argument *is* present.

$$\varsigma_w \in W\_EVTS = W\_EVT^\star$$
$$CP : W\_EVTS \rightarrow ADDR \times VAL \rightarrow EDM \rightarrow EDM$$

$$CP[\![\Lambda]\!](a,v)\mu \triangleq \mu \tag{32}$$
$$CP[\![<\varepsilon_w>]\!](a,v)\mu \triangleq P[\![\varepsilon_w]\!](a,v)\mu \tag{33}$$
$$\begin{aligned} CP[\![<\varepsilon_w, \varepsilon_r^w> {}^\frown \varsigma_w]\!](a,v)\mu \triangleq\ & \mathbf{let}\ \mu' = P[\![\varepsilon_w]\!](a,v)\mu\ \mathbf{in} \\ & \mathbf{if}\ G[\![\varepsilon_r^w[\![v]\!]]\!](a)\mu' = (\textsc{true}, v) \\ & \mathbf{then}\ \mu' \\ & \mathbf{else}\ CP[\![\varsigma_w]\!](a,v)\mu' \end{aligned} \tag{34}$$

It might appear that CarefulPut is non-terminating, as a reading of the Lampson quote above would seem to imply. This is not the case however, as the specification presented above encodes explicitly what Lampson assumes implicitly, that CarefulPut, when faced with persistent errors, will run until a crash occurs *and that such a crash will always eventually happen.* The specification of *CP* above shows this simply because the parameter ς_w is a *finite* sequence of events, and two of them are consumed for each recursive iteration.

The goal here is to find a Put-Equivalent Form (*PEq*) for *W_EVTS*, that determines the single Put which has the same effect as CarefulPut, as already shown for CarefulGet. We introduce a binary version of the $\odot_x$ operator introduced earlier, that can be used when the curried arguments are the same (the subscript decoration denoting a curried argument is dropped). This is called the *Same Argument Composition* operator and is also discussed in (Butterfield 1992a) It has the following definition:

$$(f \odot g)[\![x]\!]y \triangleq (f[\![x]\!] \circ g[\![x]\!])y \tag{35}$$

We proceed by noting the condition under which *CP* terminates, in the absence of crashes. This can be shown to be the following:

$$\mathbf{if}\ (\varepsilon_r^w \odot \varepsilon_w)[\![v]\!]\mu(a) = (\textsc{true}, v) \tag{36}$$

The *CP* algorithm will iterate until this condition is met, where μ denotes the state of the memory at the start of each iteration. The state of memory at the end of each iteration is given by:

$$\mu' = \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \tag{37}$$

Assume a call of CP that iterates many times, due to some persistent combination of erroneous events ($<\varepsilon_w^1, \varepsilon_r^1, \ldots>$). The successive contents of $\mu(a)$, originally u (say), will appear as follows:

$$
\begin{aligned}
\mu^0(a) &= u && (38)\\
\mu^1(a) &= \varepsilon_w^1 [\![v]\!](u) && (39)\\
\mu^2(a) &= (\varepsilon_w^2 \odot \varepsilon_w^1)[\![v]\!]u && (40)\\
\vdots \quad & \vdots \\
\mu^k(a) &= (\varepsilon_w^k \odot \ldots \odot \varepsilon_w^2 \odot \varepsilon_w^1)[\![v]\!]u && (41)
\end{aligned}
$$

The derivation of a Put-Equivalent Form involves the recognition of the fact that, unlike CarefulGet, CarefulPut does return a meaningful result in the event of a crash—namely the state in which the memory is left by that crash. We therefore anticipate that an equivalent form will be found for any instance of W_EVTS, even if it denotes a crash situation. In particular, we discover that appending any arbitrary "lifted" Read Error (ε_r^w) to the end of a sequence that denotes a crash between a Put and a Get (odd number of errors), will have no net effect on the resulting contents of memory:

$$\mathrm{odd}(\mathrm{len}\varsigma_w) \Rightarrow CP[\![\varsigma_w]\!](a, v) = CP[\![\varsigma_w {}^\frown <\varepsilon_r^w>]\!](a, v) \tag{42}$$

The proof is presented as Appendix B of this paper. In effect, we have converted the situation to one in which the crash occurs just after the Get, which of course has no effect on the resulting contents of memory.

Note 10. We have assumed here that Gets cannot side-effect memory, regardless of what fault occurs. This assumption would not hold valid for memory technology like Integrated Circuit dynamic memories that perform destructive read and the restore on a whole row of memory as well as the periodic read and refresh of every row[2]. In the presence of faults this could lead to memory changes on read as well as changes to bits at other addresses.

However, introducing this issue at the level of abstraction presented in this paper will introduce implementation features that are inappropriate at this point. The proper way to handle such issues is as they arise during the data reification process, which is where such details start to emerge.

An even more important response to the above note arises when we observe that the effect of such erroneous writes to data other than at the addressed location is likely to produce faults that cannot be tolerated by the stable storage system. In many ways these events are analogous to addressing errors. A key feature of the stable storage algorithms seems to be that the error-detection mechanism must cover all the data that could be affected during a Get or Put operation.

[2] Thanks must be given to an anonymous referee for pointing this out

We can now proceed to illustrate the Put-Equivalent Form:

$$PEq \;:\; VAL \times (\mathbf{B} \times VAL) \rightarrow W\_EVTS \rightarrow W\_EVT$$

$$pre\text{–}PEq[\![v,(b,w)]\!]\varsigma_w \triangleq \text{even}(\text{len}\varsigma_w) \tag{43}$$

$$PEq[\![v,(b,w)]\!]\Lambda \triangleq \varepsilon_w^{\phi} \tag{44}$$

$$PEq[\![v,(b,w)]\!]\varsigma_w \triangleq \pi_1(\varsigma'_w[i]) \tag{45}$$

$$\textbf{where}\, i = \min\{\text{len}\varsigma_w, f_T\}$$
$$\textbf{where}\, f_T = (fstloc[\![\{\text{TRUE}\}]\!] \circ Done_{P}[\![v,b,w]\!]^{\star})\varsigma'_w$$
$$\textbf{where}\, \varsigma'_w = (\amalg^{\diamond} \circ \; \langle,\rangle)\varsigma_w$$

Note 11. We are excluding sequences of odd length, as they can be extended by appending any lifted Read Event.

Note 12. The equivalent of a null event sequence is the Null Write event, as nothing changes.

This description is best understood by observing how it was constructed. Assume that $\varsigma_w = < w_1, r_1, w_2, r_2, \ldots, w_m, r_m >$.

The $\langle,\rangle$ operator simply converts an list of even length $(2m)$ into one of half the length containing pairs thus:

$$\langle,\rangle < w_1, r_1, w_2, r_2, \ldots, w_m, r_m > = < (w_1, r_1), (w_2, r_2), \ldots, (w_m, r_m) > \tag{46}$$

Note that this step indicates that we could have chosen this form of pair-sequence to represent the events during CarefulPut, as was discussed earlier, without any radical difference in the underlying operator properties.

We want to replace every w_i by the composition of itself with every write event that occurs earlier. This reflects the fact that the effect of that event may depend on previous ones. We wish to convert

$$< (w_1, r_1), (w_2, r_2), \ldots, (w_m, r_m) > \tag{47}$$

to

$$< (w_1, r_1), (w_2 \odot w_1, r_2), \ldots, (w_m \odot \cdots \odot w_2 \odot w_1, r_m)) > \tag{48}$$

To do this we introduce a binary operator $\diamond$ defined as follows:

$$(w_1, r_1) \diamond (w_2, r_2) \triangleq (w_2 \odot w_1, r_2) \tag{49}$$

Another operator we introduce is $\amalg$ which is a combination of mapping and reduction. Given a binary operator $\oplus$ then $\amalg^{\oplus}$ converts a list of the form:

$$< x_1, x_2, x_3, \ldots, x_n > \tag{50}$$

to the following list:

$$< x_1, x_1 \oplus x_2, x_1 \oplus (x_2 \oplus x_3), \ldots, x_1 \oplus (x_2 \oplus \cdots \oplus x_n) > \tag{51}$$

This operator and its properties are discussed in more detail in (Butterfield 1993a) Applying $\amalg^{\diamond}$ has the desired effect.

We finally need a predicate to check to see if a Put-Get sequence was successful:

$$Done_P[\![v, b, w]\!](\varepsilon_w, \varepsilon_r^w) \triangleq (\varepsilon_r^w \odot \varepsilon_w)[\![v]\!](b, w) = (\textsc{true}, v) \tag{52}$$

Applying this to every element of the sequence produced in the last step results in a sequence of booleans which indicates which event pairs would have resulted in termination

fstloc is used to obtain an index in a similar manner to *GEq*.

The key property that we required for the Put-Equivalent Form is now stated:

$$CP[\![\varsigma_w]\!](a, v)\mu = P[\![PEq[\![v, \mu(a)]\!]\varsigma_w]\!](a, v)\mu \tag{53}$$

The proof is trivial for null sequences (Λ), while that for non-null sequences proceeds by a variant of structural induction with a somewhat counter-intuitive inductive step:

1. Base Case: $\varsigma_w =< \varepsilon_w, \varepsilon_r^w >$
2. Inductive Step: We assume that if it holds for an instance of ς_w of the form: $< \varepsilon_w \odot \varepsilon_w', \varepsilon_r^w > {}^\frown \varsigma_w$
   that from this it is possible to deduce that it holds for the following instance: $< \varepsilon_w', \varepsilon_r^p, \varepsilon_w, \varepsilon_r^w > {}^\frown \varsigma_w$

We will justify the induction step here by pointing out that it is possible, given any error list (of even length), to construct a chain of lists of decreasing length, matching the induction step, until the base case is reached. The proof details are omitted here but can be found in (Butterfield 1993b).

2.4 Degree of Coverage

As we have seen, the equivalence operators reduce the sequences of events used by CarefulGet and CarefulPut to the single event that would produce the same result if used by Get or Put. The natural question to ask here is:

Question 13. Is the set of events that can result from finding the equivalents all possible sequences a proper subset of the set of all possible events ? In other words, has the introduction of the Careful operators eliminated some events (hopefully the erroneous ones) ?

The answer is *NO*, as can be seen by the following identities — Let ε_r be such that it produces $(\textsc{false}, \_)$ when its context is some instance of $\mathbf{B} \times VAL$, denoted by (b, w). Then the following always holds: $GEq[\![b, w]\!] < \varepsilon_r, \varepsilon_r, \ldots, \varepsilon_r >= \varepsilon_r$where there are n occurrences of ε_r. For a given value v, let $\varepsilon_r^w[\![v]\!](b, w) = (\textsc{true}, v)$ be the lifted read event that always returns that value flagged as OK. The the following always holds for any ε_w: $PEq[\![v, (\_, \_)]\!] < \varepsilon_w, \lambda v \cdot \lambda(b, w) \cdot (\textsc{true}, v) >= \varepsilon_w$.

The Careful operators provide *quantitative* fault tolerance, in that they reduce the probability of some errors occurring. They do not provide *qualitative* fault tolerance, which requires the probability of some errors to be reduced to zero, thus indicating that they have been eliminated. It must be stressed that the model as presented here does not itself handle the quantitative aspects of Stable Storage. Work has been done on introducing probability into the model, but as this raises considerable foundational issues, there is no room here to give it the coverage required. Details of this modelling will be published separately.

2.5 Stable Memory

There is no room here to present a detailed discussion of the work done in applying the *VDM*$^{\clubsuit}$ to the Stable operations from (Lampson 1981). A salient point of the material presented in this paper is that it justifies a radical set of simplifications to the StableGet and StablePut models. This is a much desired outcome as the complexity of the model, if continued in the same vein, undergoes a considerable increase when the Stable operators are examined.

The radical simplifications are summarised below with a brief justification for each:

- Our studies examine the effect of sequences of Writes and Reads on *independent* memory locations. The independence was demonstrated earlier, and allows us to ignore the aspect of memory modelling that views memory as a mapping from addresses to values. We can concentrate instead on the contents of a single memory location, and examine what happens to it as a result of varying combinations of Puts and Gets (Careful, Stable or otherwise).
- The Careful operators only provide quantitative fault tolerance and so can be replaced by the conventional Put and Get, for the purposes of qualitative analysis.
- The aspects of the Careful operators that matter for *quantitative* analysis (such as assessing the likelihood of certain errors occurring) are encapsulated in the Equivalent Form operators, and can be considered separately.
- The definitions of the Put and Get operators are extended to return *the list of remaining errors*, as well as what is presently returned. This is to allow the use of a single error sequence to describe the events occurring during sequences of operations, and is the main motivation for using a single uniform sequence to represent both Read and Write Events.

The notion of separating out various parts of a complex model into several simpler but interrelated models is considered one of the key requirements for any tractable industrial strength formal method. The examples here are the separation of addressing and quantitative issues out of the original model to leave a simpler core which can be used to assess the qualitative (correctness) properties of Stable Storage.

3 Summary

3.1 Results to Date

The results produced by this research to the present date centre on the demonstration of memory models incorporating conventional (error-prone) operations as well as Careful and Stable analogues. These models have been developed and analysed using the constructive equational reasoning that is characteristic of the *VDM*$^\clubsuit$ (Mac an Airchinnigh 1991).

The emphasis here has been on *elaborating* existing models (Mac an Airchinnigh 1990) at a given level of abstraction rather than following the conventional *VDM* style of reification which involves examining successively more concrete versions of a starting model. A key achievement here is the extension of the VDM concepts of invariant and retrieval into areas where elaboration, not reification, is taking place.

The rigourous examination of the equivalence between single errors and sequences of errors has highlighted a key distinction between between qualitative and quantitative fault tolerance. This distinction was not apparent to the author before the research work had begun. It is important as it stresses the fact that the usefulness of the Stable Storage concepts hinges on the (hoped for) rarity of certain patterns of errors which would cause it to fail. It does not work by eliminating the possibility of certain errors. The discovery of this distinction also contributes to the issue of reducing complexity, because it allows the qualitative and quantitative aspects of the various operators to be considered separately.

From the point of view of developing the mathematical ideas needed for studying fault tolerance, the research has led to the "discovery" of two operators, $\odot_x$ and $\amalg$ which play an important rôle in the models.

3.2 Future Work

Much work remains to be done. The elaboration process has to be continued until all the key features described in (Lampson 1981) have been modelled at the abstract level presented in this paper.

A phase of conventional *VDM* reification is also required, to examine how the concepts carry over to more concrete models of fault tolerance, with particular emphasis on looking at real-world coding schemes used to implement the boolean flag in the abstract model, as well as complications such as pattern faults in memory that affect distinct but related words.

In the longer term, there is a need to collate and rationalise the resulting collection of "discovered" operators. The danger here is that every stage of the modelling process will throw up more convenient operators, or shorthand notations, until the users are swamped by the sheer variety available. A regrouping phase will be required to prune the set of discovered operators down to those that are really fundamental and worth studying in their own right.

4 Acknowledgements

Particular thanks must be given to Dr. Mícheál Mac an Airchinnigh of the University of Dublin, Trinity College for his continual support and assistance with the *VDM*$^\clubsuit$. Thanks is also especially due to the the anonymous referees whose comments helped improve the clarity and focus of this paper.

5 Appendix A - Notation

5.1 *VDM*$^{\clubsuit}$Notation

Symbol	Meaning
$X \overset{m}{\to} Y$	Map from X to Y
$f[\![x]\!]y$	Function f applied to (curried) x, applied to y
$+$	Map Override operator
$\mu(x)$	Map Lookup, returning the element in the range mapped to by x
$\mathcal{I}$	The Identity Function
$^{\oplus}/$	Reduction w.r.t. binary operation $\oplus$
$\wedge$	Logical And
$\circ$	Function Composition
$\mathcal{P}(f)$	Mapping function f
π_n	nth Projection Function
rng	Map Range
$(f \overset{m}{\to} g)$	Maps f and g to Domain and Range resp. of a Map
$X^{\star}$	*Finite* Sequences over X
$\mathit{\Lambda}$	The Null Sequence
$:$	The Sequence 'Cons' Operator
$[l \ldots h]$	Sequence Subrange operator
$f^{\star}$	Maps f into a Sequence (Kleene Star functor)
len	The Sequence Length operator
$\subseteq$	The Subset relation
$\neg$	Logical Negation
$\Rightarrow$	Logical Implication
$<x>$	Singleton sequence containing x.
$<x, \ldots, y>$	Sequence notation
$\frown$	Sequence Concatenation operator

5.2 Possible extensions to *VDM*$^{\clubsuit}$Notation

Symbol	Meaning
$\odot_x$	Context-Dependent Curried-Function Composition, with context x
fstloc$[\![S]\!]$	Returns index of first occurrence of a member of S in a sequence
$\odot$	Context-Free (Same Argument) Curried-Function Composition operator
$\langle,\rangle$	Adjacent Sequence Element Pairing operator
$\amalg$	Map/Accumulate operator (hybrid of Mapping and Reduction)

5.3 Stable Storage Model Notation

Symbol	Meaning
ADDR, a	Domain of Addresses, typical member
VAL, v	Domain of Stored Values, typical member
MEM, μ	Domain of Ideal Memory, typical member
W	Write operation on Ideal Memory
R	Read operation on Ideal Memory
EDM, μ	Domain of Error Detecting Memory, typical member
$\mathit{R\_EVT}, \varepsilon_r$	Domain of Read Events, typical member
$\mathit{W\_EVT}, \varepsilon_w$	Domain of Write Events, typical member
P	Put operation on Error Detecting Memory
G	Get operation on Error Detecting Memory
$\varepsilon_r^{\mathcal{I}}$	Identity Read Event, $\varepsilon_r^{\mathcal{I}} = \mathcal{I}$
$\varepsilon_w^{\mathcal{I}}$	Identity Write Event, $\varepsilon_w^{\mathcal{I}}[\![v]\!](b, w) = (\text{TRUE}, v)$
ε_w^{ϕ}	Null Write Event, $\varepsilon_w^{\phi}[\![v]\!](b, w) = (b, w)$
ε_w^{δ}	Decay Write Event, $\varepsilon_w^{\delta}[\![v]\!](b, w) = \delta(b, w)$
$\mathit{R\_EVTS}, \varsigma_r$	Domain of Read Event sequences, typical member
CG	CarefulGet operation on Error Detecting Memory
Done_G	Successful Get predicate $\mathit{Done}_G[\![b, v]\!]\varepsilon_r = \pi_1\varepsilon_r(b, v)$
GEq	Get-Equivalent Form function
Crsh	Read Crash predicate
$\mathit{W\_EVTS}, \varsigma_w$	Domain of Write Event sequences, typical member
CP	CarefulPut operation on Error Detecting Memory
$\varepsilon_r^w, \varepsilon_r^p$	'Lifted' Read Events (converted to Write Events)
$\diamond$	Write Event Accumulation operator
Done_P	Successful Put-Get predicate
PEq	Put-Equivalent Form function

6 Appendix B - Proof

The following is the proof that:

$$\text{odd}(\text{len}\varsigma_w) \Rightarrow \mathit{CP}[\![\varsigma_w]\!](a, v) = \mathit{CP}[\![\varsigma_w{}^\frown < \varepsilon_r^w >]\!](a, v) \tag{54}$$

Proofs in $\mathit{VDM}^\clubsuit$ are similar to conventional mathematics in that it involves proving em identities of the form *lhs-expr* = *rhs-expr*, by the process of *substitution of equals* (Mac an Airchinnigh 1990, 1991). Either one of the *lhs-expr* or the *rhs-expr* are transformed until they equal the other, or both are transformed into an identical third expression.

The proof is by structural induction with a base case of $< \varepsilon_w >$ and an induction step from ς_w to $< \varepsilon_w, \varepsilon_r^w > {}^\frown\varsigma_w$ This enables us to ignore the cases when $\neg\text{odd}(\text{len}\varsigma_w)$, and remove the implication. The identity being proved here is in fact:

$$\mathit{CP}[\![\varsigma_w]\!](a, v) = \mathit{CP}[\![\varsigma_w{}^\frown < \varepsilon_r^? >]\!](a, v) \tag{55}$$

First we restate the recursive case of the definition of CP (34), by replacing calls to Put and Get by their expansions, and simplifying where possible:

$$
\begin{aligned}
& CP[\![<\varepsilon_w, \varepsilon_r^w> {}^\frown \varsigma_w]\!](a,v)\mu \\
& = \\
& \textbf{if } (\varepsilon_r^w \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\textsc{true}, v) \\
& \textbf{then } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\
& \textbf{else } CP[\![\varsigma_w]\!](a,v)(\mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\})
\end{aligned}
\tag{56}
$$

Case $<\varepsilon_w>$*:*

$$
\begin{aligned}
CP[\![<\varepsilon_w>]\!](a,v)\mu &= CP[\![<\varepsilon_w, \varepsilon_r^w>]\!](a,v)\mu && (57) \\
&= \ldots \text{expand } CP \text{ in lhs:} \\
&\quad \textbf{if } (\varepsilon_r^w \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\textsc{true}, v) && (58) \\
&\quad \textbf{then } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\
&\quad \textbf{else } CP[\![\Lambda]\!](a,v)(\mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\}) \\
&= \ldots \text{expand } CP[\![\Lambda]\!] \text{ in lhs:} \\
&\quad \textbf{if } (\varepsilon_r^w \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\textsc{true}, v) && (59) \\
&\quad \textbf{then } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\
&\quad \textbf{else } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\
&= \ldots \text{collapse } \textbf{if}\text{-expression as both arms are identical:} \\
&\quad \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} && (60)
\end{aligned}
$$

The lhs is equal to the rhs above, by the definition of CP (33), thus completing this case.

Case ς_w *to* $<\varepsilon_w, \varepsilon_r^p> {}^\frown \varsigma_w$*:* We assume that

$$
CP[\![\varsigma_w]\!](a,v) = CP[\![\varsigma_w {}^\frown <\varepsilon_r^w>]\!](a,v)
$$

and then show that

$$
CP[\![<\varepsilon_w, \varepsilon_r^p> {}^\frown \varsigma_w]\!](a,v)\mu = CP[\![<\varepsilon_w, \varepsilon_r^p> {}^\frown \varsigma_w {}^\frown <\varepsilon_r^w>]\!](a,v)\mu \tag{61}
$$

where ε_r^p is another lifted Read Event. We first reduce the lhs:

$$
\begin{aligned}
& CP[\![<\varepsilon_w, \varepsilon_r^p> {}^\frown \varsigma_w]\!](a,v)\mu && (62) \\
& = \ldots \text{expand } CP\text{:} \\
& \textbf{if } (\varepsilon_r^p \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\textsc{true}, v) && (63) \\
& \textbf{then } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\
& \textbf{else } CP[\![\varsigma_w]\!](a,v)(\mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\})
\end{aligned}
$$

We then reduce the rhs:

$$CP[\![<\varepsilon_w, \varepsilon_r^p> {}^\frown \varsigma_w {}^\frown <\varepsilon_r^w>]\!](a,v)\mu \tag{64}$$

$= \ldots$ expand CP:

$$\begin{aligned}&\textbf{if } (\varepsilon_r^p \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\text{TRUE}, v) \\ &\textbf{then } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\ &\textbf{else } CP[\![\varsigma_w {}^\frown <\varepsilon_r^w>]\!](a,v)(\mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\})\end{aligned} \tag{65}$$

$= \ldots$ use induction hypothesis on **else**-clause:

$$\begin{aligned}&\textbf{if } (\varepsilon_r^p \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\text{TRUE}, v) \\ &\textbf{then } \mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\} \\ &\textbf{else } CP[\![\varsigma_w]\!](a,v)(\mu + \{a \mapsto \varepsilon_w[\![v]\!](\mu(a))\})\end{aligned} \tag{66}$$

We see that the condition, then-clauses and the else-clauses in the lhs and rhs are the same. This completes the proof.

7 Appendix C - The $\odot$ Operator

There are some reasons for considering this operator in more depth. The first is the observation already made, that it generalises function composition from the standard case:

$$\_ \circ \_ : (A \to B) \times (B \to C) \to A \to C \tag{67}$$

to one where some curried arguments are carried through:

$$\begin{aligned}\_ \odot_\_ \_ \quad &: \quad (XAB \times X \times YBC) \to YAC \\ \textbf{where } & XAB = X \to A \to B \\ \textbf{and } & YBC = Y \to B \to C \\ \textbf{and } & YAC = Y \to A \to C\end{aligned}$$

This operator and related ones are discussed at length in (Butterfield 1992a)

The second reason for considering the operator concerns describing the behaviour of Mealy finite-state machines (Holcombe, 1982 §2.5) in the following way: Let Q denotes the set of states, Σ the set of inputs and Θ the set of outputs. We denote the next-state function as $N : \Sigma \to Q \to Q$ and the output function as $Y : \Sigma \to Q \to \Theta$. Given the current state q and input σ, then the next state and output are given by

$$(q', \theta) = (N[\![\sigma]\!]q, Y[\![\sigma]\!]q) \tag{68}$$

The $\odot$ operator can be used to describe the next output (θ) of the machine resulting from inputting σ after a sequence of inputs $<\sigma_1, \ldots, \sigma_n>$ applied to some starting state q_0, as follows:

$$\theta = (Y \odot_{\sigma_n} N \odot_{\sigma_{n-1}} \cdots \odot_{\sigma_1} N)[\![\sigma]\!]q_0 \tag{69}$$

What is interesting here is that the expression in parentheses has the signature of, and behaves like the output function Y, except that it refers to a earlier state and takes account of the intervening input sequence.

References

Andrews, D.: Data Reification and Program Decomposition, in *VDM '87 VDM — A Formal Method at Work*, Volume 252 of *Lecture Notes in Computer Science*, pp 389–422, Springer Verlag, 1987.

Bjørner, D., Jones, C. B. Eds: *The Vienna Development Method: The Meta-Language*, Volume 61 of *Lecture Notes in Computer Science*, Springer Verlag, 1978.

Bjørner, D., et al. Eds: *VDM '87 VDM — A Formal Method at Work*, Volume 252 of *Lecture Notes in Computer Science*, Springer Verlag, 1987.

Butterfield, A.: on Curried Function Composition. Technical Report TCD-CS-92-15, Dept. of Comp. Science, Trinity College, Dublin, May 1992.

Butterfield, A.: Formal memory models — a formal analysis using *VDM*$^\clubsuit$. Technical Report TCD-CS-92-27, Dept. of Comp. Science, Trinity College, Dublin, April 1992.

Butterfield, A.: on Mapped Reduction. Technical Report, Dept. of Comp. Science, Trinity College, Dublin, to appear 1993.

Butterfield, A.: The Careful Memory abstraction in Stable Storage. Technical Report, Dept. of Comp. Science, Trinity College, Dublin, to appear 1993.

Curry, H. B., Feys, R. *Combinatory Logic*, Volume 1. North Holland, Amsterdam, 1958.

Holcombe, W., M., L.: *Algebraic automata theory*, Cambridge University Press, 1982.

Johnson, B. W.: *Design and Analysis of Fault Tolerant Digital Systems.* Series in Electrical and Computer Engineering. Addison Wesley, 1989.

Jones, C. B.: VDM Proof Obligations and their Justification, in *VDM '87 VDM — A Formal Method at Work*, Volume 252 of *Lecture Notes in Computer Science*, pp 260–286, Springer Verlag, 1987.

Jones, C. B.: *Systematic Software Development using VDM, 2nd Ed.*. Series in Computer Science. Prentice Hall, 1990.

Lampson, B. W.: Atomic transactions. In *Distributed Systems, Architecture and Implementation: an Advanced Course*, Volume 105 of *Lecture Notes in Computer Science*, Chapter 11, pages 246–265. Springer Verlag, 1981.

Mac an Airchinnigh, M.: Mathematical Structures and their Morphisms in META-IV, in *VDM '87 VDM — A Formal Method at Work*, Volume 252 of *Lecture Notes in Computer Science*, pp 287–320, Springer Verlag, 1987.

Mac an Airchinnigh, M.: *Conceptual Models and Computing.* PhD thesis, Dept. of Comp. Sci,, Trinity College Dublin, Ireland, 1990.

Mac an Airchinnigh, M.: The Irish School of VDM. In *VDM '91*, Volume 552 of *Lecture Notes in Computer Science.* Springer Verlag, 1991.

Schonfinkel, M.: Über die bausteine der mathematischen logik. *Mathematische Annalen*, 92:305–16, 1924.

This article was processed using the LaTeX macro package with LLNCS style