

# The Careful Memory abstraction in Stable Storage

Andrew Butterfield  
Department of Computer Science  
Trinity College, Dublin  
Ireland

April 16, 1993

**Keywords:** Memory Systems; Stable Storage; Fault Tolerance; Formal Models

## Abstract

This article presents models of the Careful operators in *Stable Memory* [Lam81], using *VDM\** as the modelling tool.

Stable Storage is obtained by using error correcting memory [But92a] to construct a system that performs *atomic transactions* [Lam81]. This paper examines a useful abstraction, that of Careful Memory operations, which is used as a stepping stone towards fully Stable memory. The material in here is based strongly on the contents of [But92b, But92a, But93].

A guide to the notation used is to be found in Appendix A at the rear of this paper.

We first investigate the “Careful” versions of Put and Get, and derive a new criterion for classifying errors based on whether or not they are handled properly.

The domain being considered is *EDM* from [But92a].

## 2.1 Careful—Get

We now *specify* what the desired effect of a *CarefulGet* operation should be:

**Definition 1** A *CarefulGet* operation ( $CG_S$ ) should behave like a *Read*, or *Get* with the identity event function:

- (1)  $CG_S : ADDR \rightarrow EDM \rightarrow \mathbf{B} \times VAL$
- (2)  $CG_S[[a]]\mu \overset{\Delta}{\rightsquigarrow} \mu(a)$

Note that we use an approximate definition operator ( $\overset{\Delta}{\rightsquigarrow}$ ) to indicate that this is an aspiration. The following shows how  $CG_S$  is *implemented* in [Lam81].

“*CarefulGet* repeatedly does *Get* until it gets a *good* status, or until it has tried  $n$  times”

```

imp-CG[[a]]  $\triangleq$  imp-CGn[[a]]
where imp-CGk[[a]] $\mu \triangleq$ 
    if  $k = 0$ 
    then (FALSE,  $\perp$ )
    else let  $(b, v) = \text{imp-G}[[a]]\mu$  in
        if  $b$ 
        then  $(b, v)$ 
        else imp-CGk-1[[a]] $\mu$ 

```

Note that this implementation makes no explicit mention of errors. To model the fault tolerant aspects of  $CG_S$  we need to introduce them somehow. We shall introduce the notion of a sequence of events which will be an extra argument to the *CarefulGet* operation.

**Domain 1** The events affecting a *CarefulGet* are modelled as a sequence of *Read Event* functions:

- (3)  $\varsigma_r \in R\_EVTS = R\_EVT^*$

**Invariant 1** There is no invariant concerning  $R\_EVTS$ .

- (4)  $\text{inv-R\_EVTS} : R\_EVTS \rightarrow \mathbf{B}$
- (5)  $\text{inv-R\_EVTS}(\varsigma_r) \triangleq \text{TRUE}$

An obvious “invariant” might be to limit the length of such sequences to  $n$ , the number of tries made by *imp*-CG, but we may wish to use a single sequence to denote all the events associated with several uses of CarefulGet at a later stage. For this reason we leave things as they are.

**Operator 1** A CarefulGet operation in *EDM* is modelled as a function (*CG*) taking a Read Error Sequence as its first carried argument, and an address as its second. Its behaviour closely matches that of *imp*-CG described above, except that the premature exhaustion of the read errors is interpreted as a form of crash:

$CG : R\_EVTS \rightarrow ADDR \rightarrow EDM \rightarrow \mathbf{B} \times VAL$   
 $CG[\varsigma_r](a) \triangleq CG'[n, \varsigma_r](a)$

where

$$\begin{aligned} CG'[0, \varsigma_r](a)\mu &\triangleq (\text{FALSE}, \perp), \\ CG'[k, \Lambda](a)\mu &\triangleq \perp, \\ CG'[k, \varepsilon_r : \varsigma_r](a)\mu &\triangleq \\ &\quad \mathbf{let } (b, v) = G[\varepsilon_r](a)\mu \mathbf{ in} \\ &\quad \mathbf{if } b \mathbf{ then } (b, v) \mathbf{ else } CG'[k - 1, \varsigma_r](a)\mu \end{aligned}$$

A few obvious properties (whose proofs are left as exercises) can be immediately stated:

**Property 1** The result returned by applying *CG* to any sequence is the same if all but the first  $n$  elements are removed.

$$(6) \quad CG[\varsigma_r](a) = CG[\varsigma_r[1 \dots n]](a)$$

Interestingly enough, this is proved by induction on  $n$ .

**Property 2** If the event sequence has length less than  $n$ , then the only possible results are success ( $\text{TRUE}, v$ ) or crash ( $\perp$ ).

**Property 3** If the event sequence has length greater than or equal to  $n$ , then the only possible results are success ( $\text{TRUE}, v$ ) or fail ( $\text{FALSE}, \perp$ ).

The key condition that successfully terminates the recursion of *CG'* is that the first component of the result of  $G[\varepsilon_r](a)$  is returned as  $\text{TRUE}$ . This seems to supply a means of classifying event functions but great care must be taken at this point. Consider the following functions:

- $(\varepsilon_r^x)$  : The occurrence of a perfect Read does not guarantee successful termination at that point – the memory may contain the data  $(\text{FALSE}, v')$  due to some previous *Write Error*.
- $(\lambda(b, v) \cdot (\text{TRUE}, v))$  : This function will always terminate with *apparent* success, even if the stored data was in error *and marked as such*.
- $(\lambda(b, v) \cdot (f_b(v), v))$  : One cannot predict the effect of this function without knowing the value of  $v$  stored in memory, and how the boolean result of  $f_b$  depends on it.

We can use the event classification scheme introduced previously in [But92a] to help us talk about what happens. First we classify the result of any given  $CG'$  as follows:

- OK : memory location  $a$  contains  $(\text{TRUE}, v)$  and  $CG'[[k, \varsigma_r]](a)$  returns  $(\text{TRUE}, v)$
- ERR : if  $CG'[[k, \varsigma_r]](a)$  returns  $(\text{FALSE}, \perp)$
- BAD : memory location  $a$  contains  $(b, v)$  or  $\perp$  and  $CG'[[k, \varsigma_r]](a)$  returns  $(\text{TRUE}, v')$ , where  $v' \neq v$  **if**  $b = \text{TRUE}$ .
- CRSH : If  $CG'[[k, \varsigma_r]](a)$  returns  $\perp$

We now want to be able to come up with a classification of any sequence of read events with respect to the attempt to “Carefully—Get” a given value from a given address. We expect this classification to depend on  $\kappa_G$ , the classification of single read event during an attempt to get a value. Just as the effect of a read event  $\varepsilon_r$  while attempting to perform a Get from a memory location containing  $(b, v)$ , is given by  $\kappa_G[[b, v]]\varepsilon_r$ , so the effect of a sequence of read events  $\varsigma_r$  while attempting a CarefulGet of the same value is given by  $\kappa_{CG}[[b, v]]\varsigma_r$ .

$$(7) \quad \kappa_{CG} : \mathbf{B} \times \text{VAL} \rightarrow \text{R\_EVTS} \rightarrow \text{ECLASS}$$

$$(8) \quad \kappa_{CG}[[b, v]]\varsigma_r \triangleq \dots$$

To fill in the dots (...) in this classification scheme we need some way to show which event sequences are equivalent. We anticipate some *Equivalent Form* to which any event sequence can be reduced, which would then make classification easier. 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 affected by the read event. So each event in the sequence is classified in the same way. We also note that the following occasions when  $CG$  will terminate:

- at the first occurrence of an event that results in OK.
- at the first occurrence of an event that results in BAD.
- if the first  $n$  events result in ERR.

This leads to the notion of a *Get-Equivalent Form of a Read-Error Sequence w.r.t. current memory contents*, which is obtained as follows: First define  $fstloc$  as returning the index of the first occurrence in a sequence of an element belong to a specified set, with the length of the set plus one returned if no such occurrence exists:

$$(9) \quad fstloc : \mathcal{P}\Sigma \rightarrow \Sigma^* \rightarrow \mathbf{N}_1$$

$$(10) \quad fstloc[S](\Lambda) \triangleq 1$$

$$(11) \quad fstloc[S](e \cdot \sigma) \triangleq \chi[e]S \rightarrow (1, 1 + fstloc[S]\sigma)$$

Then, follow these steps to the equivalent form:

1. First limit the sequence  $(\varsigma_r)$  to the first  $n$  elements:

$$\varsigma_r[1 \dots n]$$

2. Then determine the classification of each read event in that sequence w.r.t the actual contents  $(b, v)$  of the addressed location:

$$(\kappa_G[b, v])^* \varsigma_r[1 \dots n]$$

3. Then pick out the first location that contains either OK or BAD.

$$fstloc[\{\text{OK}, \text{BAD}\}]((\kappa_G[b, v])^* \varsigma_r[1 \dots n])$$

4. And finally pick out the relevant read event, which is the first resulting in OK or BAD, or else the last ( $n$ th) event, if they are all classified as ERR.

$$\varsigma_r[\min\{n, fstloc[\{\text{OK}, \text{BAD}\}]((\kappa_G[b, v])^* \varsigma_r[1 \dots n])\}]$$

The result is the single event which, if it occurred when a  $G$  was attempted, would have the same result as the  $CG$  attempted with the sequence of events!

Lets package this up:

**Operator 2** The *Get-Equivalence* operator ( $GEq$ ) determines, for given memory contents, the effect of a sequence of Read Events occurring during the use

of the  $CG$  operator, expressed as the single Read Event that would lead  $G$  to have the same effect:

$$(12) \quad GEq \quad : \quad \mathbf{B} \times VAL \rightarrow R\_EVTS \rightarrow R\_EVT$$

$$(13) \quad GEq[[b, v]]_{\varsigma_r} \triangleq \varsigma_r[\min\{n, fstloc[\{\text{OK}, \text{BAD}\}]((\kappa_G[[b, v]])^*_{\varsigma_r}[1 \dots n])\}]$$

Note that this operator is effectively a *property based lookup mechanism* — i.e. it returns the first element that satisfies certain properties.

A case that needs to be examined is one where all the events result in  $ERR$ , but the number of events is less than  $n$ . In other words what has occurred is a crash, after (so-far) persistent read errors. We introduce a shorthand —  $f_T = fstloc[\{\text{OK}, \text{BAD}\}]$ , and let  $n > 2$ ,  $\kappa_G[[b, v]]\varepsilon_r^1 = \kappa_G[[b, v]]\varepsilon_r^2 = ERR$  and  $\varsigma_r = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle$  in the following:

$$(14) \quad GEq[[b, v]]_{\varsigma_r} = GEq[[b, v]] \langle \varepsilon_r^1, \varepsilon_r^2 \rangle$$

$$(15) \quad = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [\min\{n, f_T((\kappa_G[[b, v]])^* \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [1 \dots n])\}]$$

$$(16) \quad = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [\min\{n, f_T((\kappa_G[[b, v]])^* \langle \varepsilon_r^1, \varepsilon_r^2 \rangle)\}]$$

$$(17) \quad = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [\min\{n, f_T(\langle \kappa_G[[b, v]](\varepsilon_r^1), \kappa_G[[b, v]](\varepsilon_r^2) \rangle)\}]$$

$$(18) \quad = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [\min\{n, f_T(\langle ERR, ERR \rangle)\}]$$

$$(19) \quad = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [\min\{n, 3\}]$$

$$(20) \quad = \langle \varepsilon_r^1, \varepsilon_r^2 \rangle [3]$$

$$(21) \quad = \perp$$

As expected, in the case of a crash, there is no single read event equivalent to the sequence.

**Definition 2** The sequences that lead to crashes are those of length less than  $n$  which contain only events that result in  $ERR$  when trying to Get the data.

$$(22) \quad Crsh \quad : \quad \mathbf{B} \times VAL \rightarrow R\_EVTS \rightarrow \mathbf{B}$$

$$(23) \quad Crsh[[b, v]]_{\varsigma_r} \triangleq \text{len}_{\varsigma_r} < n \wedge \text{elems}((\kappa_G[[b, v]])^*_{\varsigma_r}) \subseteq \{ERR\}$$

Note that  $Crsh[[b, v]]\Lambda = \text{TRUE}$ .

**Proof 1**

$$(24) \quad Crsh[[b, v]]\Lambda = \text{len}\Lambda < n \wedge \text{elems}((\kappa_G[[b, v]])^*\Lambda) \subseteq \{ERR\}$$

$$(25) \quad = 0 < n \wedge \text{elems}(\Lambda) \subseteq \{ERR\}$$

$$(26) \quad = \text{TRUE} \wedge \emptyset \subseteq \{ERR\}$$

$$(27) \quad = \text{TRUE} \wedge \text{TRUE}$$

$$(28) \quad = \text{TRUE}$$

♣

**Property 4** The result of a CarefulGet with a given read event sequence that does not crash, is equal to the result of a Get with the Read Event that is the Get-Equivalent of the sequence:

$$(29) \quad \neg Crsh[\mu(a)]_{\varsigma_r} \Rightarrow CG[\varsigma_r](a)\mu = G[GEq[\mu(a)]_{\varsigma_r}](a)\mu$$

First assume that the actual value in memory ( $\mu(a)$ ) is equal to  $(\mathbf{B}, \mathbf{v})$ . Introduce some shorthands and expand the definitions to some extent as indicated below:

$$(30) \quad (\mathbf{B}, \mathbf{v}) = \mu(a)$$

$$(31) \quad \kappa_{b_v} = \kappa_G[\mathbf{B}, \mathbf{v}]$$

$$(32) \quad f_T = fstloc[\{\text{OK}, \text{BAD}\}]$$

$$(33) \quad f_\kappa = f_T \circ (\kappa_{b_v})^*$$

$$(34) \quad CG[\varsigma_r](a)\mu = CG'[n, \varsigma_r](a)\mu$$

$$(35) \quad GEq[\mathbf{B}, \mathbf{v}]_{\varsigma_r} = \varsigma_r[\min\{n, f_T((\kappa_{b_v})^* \varsigma_r[1 \dots n])\}]$$

$$(36) \quad = \varsigma_r[\min\{n, f_\kappa(\varsigma_r[1 \dots n])\}]$$

$$(37) \quad G[\varepsilon_r](a)\mu = \varepsilon_r(\mu(a)) = \varepsilon_r(\mathbf{B}, \mathbf{v})$$

What are actually going to prove is the following:

$$CG'[n, \varsigma_r](a)\mu = (\varsigma_r[\min\{n, f_\kappa(\varsigma_r[1 \dots n])\}])(\mathbf{B}, \mathbf{v})$$

subject to the conditions that  $n \geq 1$  and  $Crsh[\mathbf{B}, \mathbf{v}]_{\varsigma_r} = \text{FALSE}$ . We can write this in shorthand as follows:

$$P(n, \varsigma_r) = \text{TRUE, if } n \geq 1 \wedge \neg Crsh[\mathbf{B}, \mathbf{v}]_{\varsigma_r}.$$

The proof is by induction on  $n$  and  $\varsigma_r$ . We show the following:

- $P(1, \varsigma_r)$
- $\neg Crsh[\mathbf{B}, \mathbf{v}] < \varepsilon_r > \Rightarrow P(n, < \varepsilon_r >)$
- $P(n, \varsigma_r) \Rightarrow P(n+1, \varepsilon_r : \varsigma_r)$

**Lemma 1**  $P(1, \varsigma_r)$ . First reduce the rhs using the fact that  $f_T \sigma \geq 1$  for any list  $\sigma$ :

$$(38) \quad CG'[1, \varsigma_r](a)\mu = (\varsigma_r[\min\{1, f_\kappa(\varsigma_r[1 \dots 1])\}])(\mathbf{B}, \mathbf{v})$$

$$(39) \quad = (\varsigma_r[\min\{1, f_T(\dots)\}])(\mathbf{B}, \mathbf{v})$$

$$(40) \quad = (\varsigma_r[1])(\mathbf{B}, \mathbf{v})$$

Now consider the lhs:

$$\begin{aligned}
CG'[[1, \varsigma_r]](a)\mu &= \\
&= \mathbf{let} (b, v) = G[[\varsigma_r[1]]](a)\mu \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG'[[0, \varsigma_r[2 \dots \text{len}\varsigma_r]]](a)\mu \\
&= \mathbf{let} (b, v) = \varsigma_r[1](\mathbf{B}, v) \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG'[[0, \varsigma_r[2 \dots \text{len}\varsigma_r]]](a)\mu
\end{aligned}$$

When  $\kappa_{bv}(\varsigma_r[1]) \in \{\text{OK}, \text{BAD}\}$ , we note that this means that  $\pi_1(\varsigma_r[1](\mathbf{B}, v)) = \text{TRUE}$ . The value of  $b$  in the above example is  $\text{TRUE}$  and the result returned is  $(b, v) = \varsigma_r[1](\mathbf{B}, v)$ .

When  $\kappa_{bv}(\varsigma_r[1]) \in \{\text{ERR}\}$ , we note that this means that  $\pi_1(\varsigma_r[1](\mathbf{B}, v)) = \text{FALSE}$ . The value of  $b$  in the above example is  $\text{FALSE}$  and the result returned is  $CG'[[0, \varsigma_r[2 \dots \text{len}\varsigma_r]]](a)\mu = (\text{FALSE}, \perp)$ . Interpreting  $\perp$  as “do not care” gives us the desired result. ♣

**Lemma 2**  $\neg \text{Crsh}[[\mathbf{B}, v]] \langle \varepsilon_r \rangle \Rightarrow P(n, \langle \varepsilon_r \rangle)$ . We start by clarifying which sorts of sequences are being considered here. The condition  $\text{Crsh}[[\mathbf{B}, v]]_{\varsigma_r} = \text{FALSE}$  arises when either of the following conditions hold:

- $\text{len}\varsigma_r \geq n$
- or
- $\text{elems}((\kappa_G[[\mathbf{B}, v]])^* \varsigma_r) \cap \{\text{OK}, \text{BAD}\} \neq \emptyset$

It should be noted that  $S_1 \not\subseteq S_2$  is not the same as  $S_1 \supset S_2$ . A simple counter example will suffice:

$$(41) \quad \{\text{OK}, \text{BAD}\} \not\subseteq \{\text{ERR}\}$$

$$(42) \quad \{\text{OK}, \text{BAD}\} \not\supset \{\text{ERR}\}$$

In this case,  $\text{len} \langle \varepsilon_r \rangle = 1$ . As the case of  $n = 1$  has already been covered, we will assume that  $n > 1$ , which then means that

$$\text{elems}((\kappa_G[[\mathbf{B}, v]])^* \langle \varepsilon_r \rangle) \cap \{\text{OK}, \text{BAD}\} \neq \emptyset,$$

as  $\text{len} \langle \varepsilon_r \rangle < n$ . The implication of this is that  $\kappa_{bv}(\langle \varepsilon_r \rangle [1]) \in \{\text{OK}, \text{BAD}\}$  as  $\langle \varepsilon_r \rangle$  only has one element! This leads to the results that  $\pi_1(\varepsilon_r(\mathbf{B}, v)) = \text{TRUE}$  and  $f_\kappa(\langle \varepsilon_r \rangle) = 1$ .

First reduce the rhs:

$$(43) \quad CG'[[n, \langle \varepsilon_r \rangle]](a)\mu = (\langle \varepsilon_r \rangle [\min\{n, f_\kappa(\langle \varepsilon_r \rangle [1 \dots n])\}])(\mathbf{B}, v)$$



$$\begin{aligned}
(44) &= (\langle \varepsilon_r \rangle [\min\{n, f_\kappa(\langle \varepsilon_r \rangle)\}]) (\mathbf{B}, \mathbf{v}) \\
(45) &= (\langle \varepsilon_r \rangle [\min\{n, f_\kappa(\langle \varepsilon_r \rangle)\}]) (\mathbf{B}, \mathbf{v}) \\
(46) &= (\langle \varepsilon_r \rangle [\min\{n, 1\}]) (\mathbf{B}, \mathbf{v}) \\
(47) &= (\langle \varepsilon_r \rangle [1]) (\mathbf{B}, \mathbf{v}) \\
(48) &= \varepsilon_r (\mathbf{B}, \mathbf{v}) \\
(49) &= (\text{TRUE}, \dots)
\end{aligned}$$

Then reduce the lhs:

$$\begin{aligned}
CG' \llbracket n, \langle \varepsilon_r \rangle \rrbracket (a) \mu &= \\
&= \mathbf{let} (b, v) = G \llbracket \varepsilon_r \rrbracket (a) \mu \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG' \llbracket 0, \Lambda \rrbracket (a) \mu \\
&= \mathbf{let} (b, v) = \varepsilon_r (\mathbf{B}, \mathbf{v}) \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG' \llbracket 0, \Lambda \rrbracket (a) \mu
\end{aligned}$$

In this case  $b$  must be TRUE so we get  $(b, v) = \varepsilon_r (\mathbf{B}, \mathbf{v})$  as a result.



**Lemma 3**  $P(n, \varsigma_r) \Rightarrow P(n+1, \varepsilon_r : \varsigma_r)$ . We start with  $P(n+1, \varepsilon_r : \varsigma_r)$  and show that the equality holds if  $P(n, \varsigma_r)$  holds. First we reduce the rhs of  $P(n+1, \varepsilon_r : \varsigma_r)$  a bit:

$$\begin{aligned}
&CG' \llbracket n+1, \varepsilon_r : \varsigma_r \rrbracket (a) \mu \\
(50) &= (\varepsilon_r : \varsigma_r [\min\{n+1, f_\kappa(\varepsilon_r : \varsigma_r [1 \dots n+1])\}]) (\mathbf{B}, \mathbf{v}) \\
(51) &= (\varepsilon_r : \varsigma_r [\min\{n+1, f_\kappa(\varepsilon_r : \varsigma_r [1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(52) &= (\varepsilon_r : \varsigma_r [\min\{n+1, f_T(\kappa_{bv})^*(\varepsilon_r : \varsigma_r [1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(53) &= (\varepsilon_r : \varsigma_r [\min\{n+1, f_T(\kappa_{bv}(\varepsilon_r) : (\kappa_{bv})^* \varsigma_r [1 \dots n])\}]) (\mathbf{B}, \mathbf{v})
\end{aligned}$$

And do the same to the lhs:

$$\begin{aligned}
CG' \llbracket n+1, \varepsilon_r : \varsigma_r \rrbracket (a) \mu &= \\
&= \mathbf{let} (b, v) = G \llbracket \varepsilon_r \rrbracket (a) \mu \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG' \llbracket n, \varsigma_r \rrbracket (a) \mu \\
&= \mathbf{let} (b, v) = \varepsilon_r (\mathbf{B}, \mathbf{v}) \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG' \llbracket n, \varsigma_r \rrbracket (a) \mu
\end{aligned}$$

Now we consider cases. First we assume that  $\varepsilon_r$  produces ERR. The rhs reduces as follows:

$$CG' \llbracket n+1, \varepsilon_r : \varsigma_r \rrbracket (a) \mu$$

$$\begin{aligned}
(54) &= (\varepsilon_r : \varsigma_r[\min\{n+1, f_T(\kappa_{bv}(\varepsilon_r) : (\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(55) &= (\varepsilon_r : \varsigma_r[\min\{n+1, f_T(\text{ERR} : (\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(56) &= (\varepsilon_r : \varsigma_r[\min\{n+1, 1 + f_T((\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(57) &= (\varepsilon_r : \varsigma_r[1 + \min\{n, f_T((\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(58) &= (\varsigma_r[1 + \min\{n, f_T((\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(59) &
\end{aligned}$$

The lhs reduces as follows, given that  $b = \text{FALSE}$ :

$$\begin{aligned}
CG' \llbracket n+1, \varepsilon_r : \varsigma_r \rrbracket (a) \mu &= \\
&= \mathbf{let} (b, v) = \varepsilon_r(\mathbf{B}, \mathbf{v}) \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG' \llbracket n, \varsigma_r \rrbracket (a) \mu \\
&= CG' \llbracket n, \varsigma_r \rrbracket (a) \mu
\end{aligned}$$

Comparing the lhs and rhs we see that we have  $P(n, \varsigma_r)$ . Secondly, we assume that  $\varepsilon_r$  produces OK or BAD. The rhs reduces as follows:

$$\begin{aligned}
&CG' \llbracket n+1, \varepsilon_r : \varsigma_r \rrbracket (a) \mu \\
(60) &= (\varepsilon_r : \varsigma_r[\min\{n+1, f_T(\kappa_{bv}(\varepsilon_r) : (\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(61) &= (\varepsilon_r : \varsigma_r[\min\{n+1, f_T(\text{D}\dots : (\kappa_{bv})^* \varsigma_r[1 \dots n])\}]) (\mathbf{B}, \mathbf{v}) \\
(62) &= (\varepsilon_r : \varsigma_r[\min\{n+1, 1\}]) (\mathbf{B}, \mathbf{v}) \\
(63) &= (\varepsilon_r : \varsigma_r[1]) (\mathbf{B}, \mathbf{v}) \\
(64) &= \varepsilon_r(\mathbf{B}, \mathbf{v}) \\
(65) &
\end{aligned}$$

The lhs reduces as follows, given that  $b = \text{TRUE}$ :

$$\begin{aligned}
CG' \llbracket n+1, \varepsilon_r : \varsigma_r \rrbracket (a) \mu &= \\
&= \mathbf{let} (b, v) = \varepsilon_r(\mathbf{B}, \mathbf{v}) \mathbf{in} \\
&\quad \mathbf{if} b \mathbf{then} (b, v) \mathbf{else} CG' \llbracket n, \varsigma_r \rrbracket (a) \mu \\
&= (b, v) = \varepsilon_r(\mathbf{B}, \mathbf{v})
\end{aligned}$$

Comparing the lhs and rhs we have equality.

♣

**Proof 2** results from the three previous Lemmas and induction on  $n$  and  $\varsigma_r$ . ♣

All of this can be summarised as follows: *CarefulGet* has the effect of skipping past upto  $n-1$  events resulting in ERR, and either returning the  $n$ th ERR result (persistent read error) or an apparently successful result, which may or may not be a BAD. It is an implementation of *Get* using repeated *Gets*, that eliminates every ERR except persistent ones.

## 2.2 Careful—Put

We now *specify* what the desired effect of a *CarefulPut* operation should be:

**Definition 3** A *CarefulPut* operation ( $CP_S$ ) should behave like a Write, or Put with the identity event function:

$$(66) \quad CP_S \quad : \quad ADDR \times VAL \rightarrow EDM \rightarrow EDM$$

$$(67) \quad CP_S[a, v]\mu \stackrel{\Delta}{\rightsquigarrow} \mu + [a \mapsto (\text{TRUE}, v)]$$

The following shows how  $CP_S$  is *implemented* in [Lam81].

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

$$\begin{aligned} \text{imp-CP}[a, v]\mu &\triangleq \\ &\mathbf{let} \mu' = \text{imp-P}[a, v]\mu \mathbf{in} \\ &\mathbf{if} \text{imp-G}[a] = (\text{TRUE}, v) \\ &\mathbf{then} \mu' \\ &\mathbf{else} \text{imp-CP}[a, v]\mu' \end{aligned}$$

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

**Question 1** How should errors and events be modelled here? We have *alternating* Puts and Gets with the possibility of a crash inbetween at any point!

First note that the effect of a *CarefulPut* is to modify the contents of memory. This is in contrast to *CarefulGet* which simply returns the contents of a given location, and returns  $\perp$  in the event of a crash. We want *CarefulPut* to return the state of memory always, even after a crash. The value returned in the event of a crash is the state in which the memory was left.

One obvious modelling approach is to use a sequence of write and read event pairs:

$$(68) \quad \varsigma_{rw} \in WRERRS = (W\_EVT \times R\_EVT)^*$$

$$(69) \quad \varsigma_{rw} = \langle (\varepsilon_w^1, \varepsilon_r^1), (\varepsilon_w^2, \varepsilon_r^2), \dots, (\varepsilon_w^n, \varepsilon_r^n) \rangle$$

but this leaves open the question of how to model a crash during or just before a Get. We could try using a read event that always returned  $(\text{TRUE}, v)$  for any attempt to Put  $v$ .

Another approach is to use a sequence consisting only of write events, but where every second such event is in fact a read event *in disguise*. This is achieved by *lifting* read events to the form  $\varepsilon_w^r = \lambda v \cdot \lambda(b, w) \cdot \varepsilon_r(b, w) = \lambda v \cdot \varepsilon_r = \mathbf{K}\varepsilon_r$  that ignore their first (*VAL*) argument. A crash before or during a Get is then modelled by any sequence whose length is odd. The problem with this approach is that a complex and *undecidable* invariant is required. Alternatively, the “read event” could be *context sensitive* in that it would not in general ignore the first argument. This would eliminate the need for an invariant. 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, such an argument is present — the current value that it is attempting to write to memory. This is more general, and permits the same event sequence to serve for both the Put and the Get operations. For this reason the second alternative is chosen.

**Domain 2** The events affecting a CarefulPut are modelled as a sequence of write events:

$$(70) \quad \varsigma_w \in W\_EVTS = W\_EVT^*$$

**A Methodological Aside** Another approach to using a write event sequence would be to introduce a ‘zip’ function which takes two lists and merges them into one by alternating elements from each. In this case it would be specialised a bit as follows:

**Operator 3** The Zip—Event binary operator ( $\ddagger$ ) combines two sequences, one of write events, the other of read events, to produce a write event list that consists alternately of members from each list, starting with the first write event. with read events being ‘lifted’ as write events.

A question that needs to be answered is what should happen if the lists are of different lengths. The following approach is simplest as it requires no invariant, and provides maximum flexibility. We specify the outcome of zipping write events ( $\varsigma_w$ ) and read events ( $\varsigma_r$ ) as a set of cases depending on how their lengths are related:

- $\text{len}\varsigma_r \geq \text{len}\varsigma_w$ : the result list is twice as long as the write event list, with alternating write and read events ending on the read event corresponding to the last write event.

$$\langle \varepsilon_w^1 \dots \varepsilon_w^n \rangle \ddagger \langle \varepsilon_r^1 \dots \varepsilon_r^n \dots \rangle = \langle \varepsilon_w^1, \mathbf{K}\varepsilon_r^1 \dots \varepsilon_w^n, \mathbf{K}\varepsilon_r^n \rangle$$

- $\text{len}_{\varsigma_r} < \text{len}_{\varsigma_w}$ : the result list is twice as long as the write event list, with alternating write and read events ending on the last write event. If the read events expire prematurely, then the Identity Read Event<sup>1</sup> is assumed.

$$\langle \varepsilon_w^1 \dots \varepsilon_w^i \dots \varepsilon_w^n \rangle \ddagger \langle \varepsilon_r^1 \dots \varepsilon_r^i \rangle = \langle \varepsilon_w^1, K\varepsilon_r^1 \dots \varepsilon_w^i, K\varepsilon_r^i \dots \varepsilon_w^{i+1}, K\varepsilon_r^i \dots K\varepsilon_r^i, \varepsilon_w^n \rangle$$

In other words, the length of the write event list is the main determining factor of the length of the final result.

$$(71) \quad -\ddagger- \in W\_EVTS \times R\_EVTS \rightarrow W\_EVTS$$

$$(72) \quad \Lambda \ddagger \varsigma_r \triangleq \Lambda$$

$$(73) \quad \langle \varepsilon_w \rangle \ddagger \Lambda \triangleq \langle \varepsilon_w \rangle$$

$$(74) \quad \varepsilon_w : \varsigma_w \ddagger \Lambda \triangleq \langle \varepsilon_w, K\varepsilon_r^x \rangle \frown (\varsigma_w \ddagger \Lambda)$$

$$(75) \quad \varepsilon_w : \varsigma_w \ddagger \varepsilon_r : \varsigma_r \triangleq \varepsilon_w : \varsigma_w \ddagger \varepsilon_r : \varsigma_r$$

where we use an auxillary operator ( $\ddagger$ ) for the case when both read and write events are present:

$$(76) \quad -\ddagger\ddagger- \in W\_EVTS \times R\_EVTS \rightarrow W\_EVTS$$

$$(77) \quad \Lambda \ddagger\ddagger \Lambda \triangleq \Lambda$$

$$(78) \quad \Lambda \ddagger\ddagger \varepsilon_r : \varsigma_r \triangleq \langle K\varepsilon_r \rangle$$

$$(79) \quad \langle \varepsilon_w \rangle \ddagger\ddagger \Lambda \triangleq \langle \varepsilon_w \rangle$$

$$(80) \quad \varepsilon_w : \varsigma_w \ddagger\ddagger \Lambda \triangleq \langle \varepsilon_w, K\varepsilon_r^x \rangle \frown (\varsigma_w \ddagger\ddagger \Lambda)$$

$$(81) \quad \varepsilon_w : \varsigma_w \ddagger\ddagger \varepsilon_r : \varsigma_r \triangleq \langle \varepsilon_w, K\varepsilon_r \rangle \frown (\varsigma_w \ddagger\ddagger \varepsilon_r)$$

Any use of write events with CarefulPut would then always use  $\varsigma_w \ddagger \varepsilon_r$  to construct a list that did meet the required invariant. This function is sufficiently useful that it will often be used when we want context free read events. We shall also refer to  $K\varepsilon_r$  as  $\varepsilon_r^w$ .

**Operator 4** A CarefulPut operation in *EDM* is modelled as a function (*CP*) which takes a Write Event sequence as its first argument, and an address, value pair as its second. Its behaviour closely matches that of *imp-CP* described above, except that the premature exhaustion of read events is interpreted as a form of crash which returns the current state of the memory unchanged. Note also that the read event may be context sensitive, in which case the context is that of the value being written.

<sup>1</sup>Note that lifting the Identity Read Event does *not* give the Identity Write Event ( $\varepsilon_r^x \neq \varepsilon_w^x$ ).

$$\begin{aligned}
(82) \quad & CP : W\_EVTS \rightarrow ADDR \times VAL \rightarrow EDM \rightarrow EDM \\
(83) \quad & CP[\Lambda](a, v) \triangleq \mathcal{I} \\
(84) \quad & CP[\langle \varepsilon_w \rangle](a, v) \triangleq P[\varepsilon_w](a, v) \\
(85) \quad & CP[\langle \varepsilon_w, \varepsilon_r^w \rangle \wedge \varsigma_w](a, v)\mu \triangleq \mathbf{let} \mu' = P[\varepsilon_w](a, v)\mu \mathbf{in} \\
(86) \quad & \quad \quad \quad \mathbf{if} G[\varepsilon_r^w[v]](a)\mu' = (\mathbf{TRUE}, v) \\
(87) \quad & \quad \quad \quad \mathbf{then} \mu' \\
(88) \quad & \quad \quad \quad \mathbf{else} CP[\varsigma_w](a, v)\mu'
\end{aligned}$$

The goal here is to find a *equivalent form* for  $W\_EVTS$ , that determines the single Put which has the same effect as CarefulPut, as already shown for CarefulGet. We proceed by noting the condition under which  $CP$  terminates, in the absence of crashes:

$$(89) \quad \mathbf{if} G[\varepsilon_r^w[v]](a)\mu' = (\mathbf{TRUE}, v)$$

Which expands, given the definition of  $G$  to:

$$(90) \quad \mathbf{if} \varepsilon_r^w[v](\mu'(a)) = (\mathbf{TRUE}, v)$$

Replacing  $\mu'$  by the expansion of the call to  $P$  that produces it gives:

$$(91) \quad \mathbf{if} \varepsilon_r^w[v](\mu + [a \mapsto \varepsilon_w[v](\mu(a))](a)) = (\mathbf{TRUE}, v)$$

Finally we reduce this using map properties to:

$$(92) \quad \mathbf{if} \varepsilon_r^w[v](\varepsilon_w[v](\mu(a))) = (\mathbf{TRUE}, v)$$

The  $CP$  algorithm will iterate until this condition is met, where  $\mu$  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:

$$(93) \quad \mu' = \mu + [a \mapsto \varepsilon_w[v](\mu(a))]$$

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

$$\begin{aligned}
(94) \quad \mu^0(a) &= u \\
(95) \quad \mu^1(a) &= \varepsilon_w^1 \llbracket v \rrbracket (u) \\
(96) \quad \mu^2(a) &= \varepsilon_w^2 \llbracket v \rrbracket (\varepsilon_w^1 \llbracket v \rrbracket (u)) \\
(97) \quad &\vdots \\
(98) \quad \mu^k(a) &= (\varepsilon_w^k \llbracket v \rrbracket \circ \dots \circ \varepsilon_w^2 \llbracket v \rrbracket \circ \varepsilon_w^1 \llbracket v \rrbracket)(u) \\
(99) \quad &= (\varepsilon_w^k \odot \dots \odot \varepsilon_w^2 \odot \varepsilon_w^1) \llbracket v \rrbracket (u)
\end{aligned}$$

where  $\odot$  is the *Same Argument Composition operator* introduced in [But92b]. The termination conditions appear as follows:

$$\begin{aligned}
(100) \quad &(\varepsilon_r^1 \odot \varepsilon_w^1) \llbracket v \rrbracket (u) = (\text{TRUE}, v) \\
(101) \quad &(\varepsilon_r^2 \odot \varepsilon_w^2 \odot \varepsilon_w^1) \llbracket v \rrbracket (u) = (\text{TRUE}, v) \\
(102) \quad &\vdots \\
(103) \quad &(\varepsilon_r^k \odot \varepsilon_w^k \odot \dots \odot \varepsilon_w^2 \odot \varepsilon_w^1) \llbracket v \rrbracket (u) = (\text{TRUE}, v)
\end{aligned}$$

It is easy enough to see from this that an equivalent form does exist, and is generated by a recursive function very similar in structure to *CP* itself. To see how this is derived we shall illustrate the step by step reasoning that leads to the equivalence. The first step 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 expect that:

**Property 5** Appending any arbitrary lifted Read Event ( $\varepsilon_r^2$ ) to the end of a sequence that denotes a crash after a Put and before a Get (odd number of events), will have no net effect on the resulting contents of memory.

$$(104) \quad \text{odd}(\text{lens}_{\zeta_w}) \Rightarrow CP \llbracket \zeta_w \rrbracket (a, v) = CP \llbracket \zeta_w \frown \langle \varepsilon_r^2 \rangle \rrbracket (a, v)$$

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.<sup>2</sup> In other words, crashes that occur inbetween the Put and Get of *CarefulPut*

---

<sup>2</sup>Note that we have assumed here that Gets cannot side-effect memory, regardless of what fault occurs. This assumption would not hold valid for memory read technology that erases memory contents which must then be refreshed, like old core memories or some bubble memory technologies

can be treated as if they occurred just after the Get (as far as memory contents are concerned)

**Proof 3** The proof is by structural induction with a base case of  $P(\langle \varepsilon_w \rangle)$  and an induction step  $P(\varsigma_w) \Rightarrow P(\langle \varepsilon_w, \varepsilon_r^w \rangle \frown \varsigma_w)$ . This enables us to ignore the even cases, and remove the implication:

$$(105) P(\varsigma_w) \triangleq CP[\![\varsigma_w]\!](a, v) = CP[\![\varsigma_w \frown \langle \varepsilon_r^? \rangle]\!](a, v)$$

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

$$(106) \quad CP[\![\langle \varepsilon_w, \varepsilon_r^w \rangle \frown \varsigma_w]\!](a, v)\mu$$

$$(107) \quad =$$

$$(108) \quad \mathbf{if} (\varepsilon_r^w \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\mathbf{TRUE}, v)$$

$$(109) \quad \mathbf{then} \mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))]$$

$$(110) \quad \mathbf{else} CP[\![\varsigma_w]\!](a, v)(\mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))])$$

Case  $P(\langle \varepsilon_w \rangle)$ :

$$(111) CP[\![\langle \varepsilon_w \rangle]\!](a, v)\mu = CP[\![\langle \varepsilon_w, \varepsilon_r^? \rangle]\!](a, v)\mu$$

$$(112) \quad =$$

$$(113) \quad \mathbf{if} (\varepsilon_r^? \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\mathbf{TRUE}, v)$$

$$(114) \quad \mathbf{then} \mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))]$$

$$(115) \quad \mathbf{else} CP[\![\Lambda]\!](a, v)(\mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))])$$

$$(116) \quad =$$

$$(117) \quad \mathbf{if} (\varepsilon_r^? \odot \varepsilon_w)[\![v]\!](\mu(a)) = (\mathbf{TRUE}, v)$$

$$(118) \quad \mathbf{then} \mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))]$$

$$(119) \quad \mathbf{else} \mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))]$$

$$(120) \quad =$$

$$(121) \quad \mu + [a \mapsto \varepsilon_w[\![v]\!](\mu(a))]$$

The lhs is easily shown equal to the rhs above, thus completing this case.

Case  $P(\varsigma_w) \Rightarrow P(\langle \varepsilon_w, \varepsilon_r^w \rangle \frown \varsigma_w)$ :



We assume  $CP[\llbracket \varsigma_w \rrbracket](a, v) = CP[\llbracket \varsigma_w \wedge \langle \varepsilon_r^? \rangle \rrbracket](a, v)$  and try to show that

$$(122) \quad CP[\llbracket \langle \varepsilon_w, \varepsilon_r^w \rangle \wedge \varsigma_w \rrbracket](a, v)\mu = CP[\llbracket \langle \varepsilon_w, \varepsilon_r^w \rangle \wedge \varsigma_w \wedge \langle \varepsilon_r^? \rangle \rrbracket](a, v)\mu$$

We first reduce the lhs:

$$(123) \quad CP[\llbracket \langle \varepsilon_w, \varepsilon_r^w \rangle \wedge \varsigma_w \rrbracket](a, v)\mu$$

$$(124) \quad =$$

$$(125) \quad \mathbf{if} (\varepsilon_r^w \odot \varepsilon_w) \llbracket v \rrbracket (\mu(a)) = (\text{TRUE}, v)$$

$$(126) \quad \mathbf{then} \mu + [a \mapsto \varepsilon_w \llbracket v \rrbracket (\mu(a))]$$

$$(127) \quad \mathbf{else} CP[\llbracket \varsigma_w \rrbracket](a, v)(\mu + [a \mapsto \varepsilon_w \llbracket v \rrbracket (\mu(a))])$$

We then reduce the rhs:

$$(128) \quad CP[\llbracket \langle \varepsilon_w, \varepsilon_r^w \rangle \wedge \varsigma_w \wedge \langle \varepsilon_r^? \rangle \rrbracket](a, v)\mu$$

$$(129) \quad =$$

$$(130) \quad \mathbf{if} (\varepsilon_r^w \odot \varepsilon_w) \llbracket v \rrbracket (\mu(a)) = (\text{TRUE}, v)$$

$$(131) \quad \mathbf{then} \mu + [a \mapsto \varepsilon_w \llbracket v \rrbracket (\mu(a))]$$

$$(132) \quad \mathbf{else} CP[\llbracket \varsigma_w \wedge \langle \varepsilon_r^? \rangle \rrbracket](a, v)(\mu + [a \mapsto \varepsilon_w \llbracket v \rrbracket (\mu(a))])$$

We see that the condition and then-clauses are the same. We also note that the else clauses are the same by the hypothesis  $P(\varsigma_w)$ . This completes the proof. ♣

We can now proceed to illustrate how the equivalent form is derived. We need to define some auxillary operators beforehand. The  $\langle \cdot \rangle$  operator takes a list of the form:

$$\langle x_1, x_2, \dots, x_{2n-1}, x_{2n} \rangle$$

and returns a list formed by pairing adjacent elements thus:

$$\langle (x_1, x_2), \dots, (x_{2n-1}, x_{2n}) \rangle$$

$$(133) \quad \langle \cdot \rangle : \Sigma^* \rightarrow \Sigma \times \Sigma^*$$

$$(134) \quad \langle \cdot \rangle \Lambda \triangleq \Lambda$$

$$(135) \quad \langle \cdot \rangle \langle x, y \rangle \wedge \sigma \triangleq (x, y) : (\langle \cdot \rangle \sigma)$$

with precondition:

$$(136) \text{ pre-}\langle \cdot \rangle : \Sigma^* \rightarrow \mathbf{B}$$

$$(137) \text{ pre-}\langle \cdot \rangle \sigma \triangleq \text{even}(\text{len}\sigma)$$

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

$$\langle x_1, x_2, x_3, \dots, x_n \rangle$$

to the following list:

$$\langle x_1, x_1 \oplus x_2, x_1 \oplus x_2 \oplus x_3, \dots, x_1 \oplus x_2 \oplus \dots \oplus x_n \rangle$$

$$(138) \quad \Pi : X \times X \rightarrow X \rightarrow X^* \rightarrow X^*$$

$$(139) \quad \Pi^\oplus \Lambda \triangleq \Lambda$$

$$(140) \quad \Pi^\oplus \langle x \rangle \triangleq \langle x \rangle$$

$$(141) \quad \Pi^\oplus (x : y : \sigma) \triangleq x : (\Pi^\oplus ((x \oplus y) : \sigma))$$

This operator and its properties are discussed in more detail in [But93]

Next we bring in a shorthand for an accumulating binary operator:

$$(142) \quad \_ \diamond \_ : (W\_EVT^2) \times (W\_EVT^2) \rightarrow W\_EVT^2$$

$$(143) \quad (w_1, r_1) \diamond (w_2, r_2) \triangleq (w_2 \odot w_1, r_2)$$

and finally, a predicate used to check to see if a Put — Get sequence was successful:

$$(144) \quad Done_P : VAL \times \mathbf{B} \times VAL \rightarrow W\_EVT \times W\_EVT \rightarrow \mathbf{B}$$

$$Done_P[v, b, w](\varepsilon_w, \varepsilon_r^w)$$

$$(145) \quad \triangleq (\varepsilon_r^w \odot \varepsilon_w)[v](b, w) = (\text{TRUE}, v)$$

Using these auxillary functions, the following steps are now taken to obtain the equivalent form of a write event sequence:

1. First append an arbitrary read event to the end of any sequence whose length is odd:

$$\zeta'_w \triangleq \text{odd}(\text{len}\zeta_w) \rightarrow (\zeta_w \frown \varepsilon_r^?, \zeta_w)$$

to give us a sequence  $(\zeta'_w)$  which will look something like this:

$$\zeta'_w = \langle \varepsilon_w^1, \varepsilon_r^1, \varepsilon_w^2, \varepsilon_r^2, \dots, \varepsilon_w^n, \varepsilon_r^n \rangle$$

2. As we now have a list whose length is even, we shall pair up elements to produce a list of half the length, each pair consisting of a write event and its subsequent read event:

$$\langle \cdot \rangle \zeta'_w = \langle (\varepsilon_w^1, \varepsilon_r^1), (\varepsilon_w^2, \varepsilon_r^2), \dots, (\varepsilon_w^n, \varepsilon_r^n) \rangle$$

3. We use the  $\Pi$  operator with  $\diamond$  to enable us to replace any given Write Event by the accumulated effect of itself after all the Write Events that preceded it:

$$\begin{aligned} (\Pi^\diamond \circ \langle \cdot \rangle) \zeta'_w = \\ \langle (\varepsilon_w^1, \varepsilon_r^1), (\varepsilon_w^2 \odot \varepsilon_w^1, \varepsilon_r^2), \dots, (\varepsilon_w^n \odot \dots \odot \varepsilon_w^1, \varepsilon_r^n) \rangle \end{aligned}$$

4. We can now use a conventional sequence mapping operator to evaluate whether or not each entry denotes the termination of the computation. This is the first time that we see the context  $(v, (b, w))$  in which the equivalence occurs:

$$(\text{Done}_P \llbracket v, b, w \rrbracket^* \circ \Pi^\diamond \circ \langle \cdot \rangle) \zeta'_w = \langle \dots n \text{ booleans} \dots \rangle$$

5. The *fstloc* operator is then used to identify the first occurrence of TRUE, if any. The result obtained is the index of that occurrence, or the length of the sequence if all are FALSE:

$$i = \min\{\text{len}\zeta'_w, (\text{fstloc} \llbracket \{\text{TRUE}\} \rrbracket \circ \text{Done}_P \llbracket v, b, w \rrbracket^* \circ \Pi^\diamond \circ \langle \cdot \rangle) \zeta'_w\}$$

(we use  $i$  to denote that index).

6. The equivalent form is simply obtained by indexing the accumulated sequence by  $i$  and taking the first element of the resulting pair:

$$\varepsilon'_w = \pi_1((\Pi^\diamond \circ \langle \cdot \rangle) \zeta'_w)[i]$$

Lets package this up as an operator. Note, however, that one boundary condition must be taken care of — the case when  $\zeta_w = \Lambda$ , i.e. an immediate crash. The result of CarefulPut in this case is that no change occurs to memory. The corresponding equivalence form of the empty sequence is therefore the Null Write Event.

**Operator 5** The *Put Equivalence* operator ( $PEq$ ) determines, for a given value to be written and given pre-existing memory contents, the effect of a sequence of Write Events occurring during the use of the  $CP$  operator expressed as the single Write Event that would have the same effect:

$$(146) \quad PEq \quad : \quad VAL \times (\mathbf{B} \times VAL) \rightarrow W\_EVTS \rightarrow W\_EVT$$

$$PEq[[v, (b, w)]]\Lambda$$

$$(147) \quad \triangleq \quad \varepsilon_w^\phi$$

$$PEq[[v, (b, w)]]\varsigma_w$$

$$(148) \quad \triangleq$$

$$(149) \quad \mathbf{let} \quad \varsigma'_w = \text{odd}(\text{len}\varsigma_w) \rightarrow (\varsigma_w \frown \varepsilon_r^?, \varsigma_w) \mathbf{in}$$

$$(150) \quad \mathbf{let} \quad \varsigma''_w = (\Pi^\diamond \circ \langle \cdot \rangle)\varsigma'_w \mathbf{in}$$

$$(151) \quad \mathbf{let} \quad i = \min\{\text{len}\varsigma'_w, (\text{fstloc}[\{\text{TRUE}\}] \circ \text{Done}_P[[v, b, w]]^x)\varsigma''_w\}$$

$$(152) \quad \mathbf{in} \quad \pi_1(\varsigma''_w[i])$$

**Property 6** The effect of a CarefulPut operation with value  $v$  and Write Event sequence  $\varsigma_w$  on a memory location  $\mu(a)$  containing  $(b, w)$  is the same as obtained by a Put operation whose Write Event is equal to the Put Equivalent Form of  $\varsigma_w$ , w.r.t.  $(v, (b, w))$

$$CP[[\varsigma_w]](a, v)\mu = P[[PEq[[v, \mu(a)]]\varsigma_w]](a, v)\mu$$

As we have already demonstrated the equivalence of odd-length sequences with those having one extra element, we will restrict our proof to even-length sequences only. This eliminates the need for the first conditional in the definition of  $PEq$ . The proof proceeds by a variant of structural induction with a somewhat counter-intuitive inductive step:

1.  $P(\Lambda)$ ,
2.  $P(\langle \varepsilon_w, \varepsilon_r^w \rangle)$
3.  $P(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r^w \rangle \frown \varsigma_w) \Rightarrow P(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r^w \rangle \frown \varsigma_w)$

We will justify it here by pointing out that it is possible, given any event list (of even length), to construct a chain of lists of decreasing length until the base case is reached. This effectively shows how to construct the proof for that instance:

$$(153) \quad P(\langle \varepsilon_w^1, \varepsilon_r^1, \varepsilon_w^2, \varepsilon_r^2, \varepsilon_w^3, \varepsilon_r^3, \dots, \varepsilon_w^{n-1}, \varepsilon_r^{n-1}, \varepsilon_w^n, \varepsilon_r^n \rangle)$$

$$(154) \quad \Leftarrow P(\langle \varepsilon_w^2 \odot \varepsilon_w^1, \varepsilon_r^2, \varepsilon_w^3, \varepsilon_r^3, \dots, \varepsilon_w^{n-1}, \varepsilon_r^{n-1}, \varepsilon_w^n, \varepsilon_r^n \rangle)$$

$$(155) \quad \Leftarrow P(\langle \varepsilon_w^3 \odot \varepsilon_w^2 \odot \varepsilon_w^1, \varepsilon_r^3, \dots, \varepsilon_w^{n-1}, \varepsilon_r^{n-1}, \varepsilon_w^n, \varepsilon_r^n \rangle)$$

$$\begin{aligned}
(156) \quad & \vdots \\
(157) \quad & \Leftarrow P(\langle \varepsilon_w^{n-1} \odot \dots \odot \varepsilon_w^3 \odot \varepsilon_w^2 \odot \varepsilon_w^1, \varepsilon_r^{n-1}, \varepsilon_w^n, \varepsilon_r^n \rangle) \\
(158) \quad & \Leftarrow P(\langle \varepsilon_w^n \odot \varepsilon_w^{n-1} \odot \dots \odot \varepsilon_w^3 \odot \varepsilon_w^2 \odot \varepsilon_w^1, \varepsilon_r^n \rangle) \\
(159) \quad & = \text{TRUE}
\end{aligned}$$

We also introduce the following shorthand:


$$(160) \quad f_t \triangleq \text{fstloc}[\{\text{TRUE}\}]$$

The proofs for the base cases  $(\Lambda, \langle \varepsilon_w, \varepsilon_r^w \rangle)$  are trivial and are left as exercises.

The proof of the induction step has two cases, corresponding to whether or not the first Put attempt by CarefulPut is successful. However, the proof is quite long and complicated and will only be sketched out here. It is based on a series of Lemmas, the proofs of which are left as exercises. The final proof is presented in terms of these Lemmas.


#### Lemma 4

$$\begin{aligned}
(161) \quad & CP[\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w](a, v)\mu \\
(162) \quad & = \text{if } Done_P[v, \mu(a)](\varepsilon'_w, \varepsilon_r^?) \\
(163) \quad & \quad \text{then } \mu + [a \mapsto \varepsilon'_w[v](\mu(a))] \\
(164) \quad & \quad \text{else } CP[\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown \varsigma_w](a, v)\mu
\end{aligned}$$

The proof is obtained by using the definitions of  $CP$ ,  $Done_P$  and map properties. The details are left as an exercise 

#### Lemma 5

$$\begin{aligned}
(165) \quad & (f_t \circ Done_P[v, \mu(a)])(\langle \varepsilon_w, \varepsilon_r^w \rangle : \varsigma_w) \\
(166) \quad & = \text{if } Done_P[v, \mu(a)](\varepsilon_w, \varepsilon_r^w) \\
(167) \quad & \quad \text{then } 1 \\
(168) \quad & \quad \text{else } 1 + (f_t \circ Done_P[v, \mu(a)])\varsigma_w
\end{aligned}$$

The proof is obtained by using the definitions of  $\text{fstloc}$  and  $Done_P$ . The details are left as an exercise 

#### Lemma 6

$$\begin{aligned}
(169) \quad & (\Pi^\diamond \circ \langle \cdot \rangle)(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w) \\
(170) \quad & = (\varepsilon'_w, \varepsilon_r^?) : (\Pi^\diamond \circ \langle \cdot \rangle)(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown \varsigma_w)
\end{aligned}$$

The proof is obtained by using the definitions of  $\Pi$ ,  $\diamond$  and  $\langle \cdot, \cdot \rangle$ . Also required is the following  $\Pi$  property:

$$\Pi^\oplus (x : y : \sigma) = x : \Pi^\oplus ((x \oplus y) : \sigma)$$

where follows directly from the definition of  $\Pi$ . The remaining details of this Lemma are left as an exercise  $\clubsuit$

Now, a reminder — we are trying to prove that:

$$\begin{aligned} CP[\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w](a, v)\mu \\ = P[PEq[v, \mu(a)] \langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w](a, v)\mu \end{aligned}$$

holds if the following is true:

$$\begin{aligned} CP[\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown \varsigma_w](a, v)\mu \\ = P[PEq[v, \mu(a)] \langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown \varsigma_w](a, v)\mu \end{aligned}$$

The proof now proceeds by considering the case:

$$Done_P[v, \mu(a)](\varepsilon'_w, \varepsilon_r^?) = \text{TRUE}$$

**Proof 4** The lhs reduces, by Lemma 4, to:

$$(171) \mu + [a \mapsto \varepsilon'_w[v](\mu(a))]$$

The value of  $PEq[v, \mu(a)] \langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle$  is evaluated as shown below, where the lhs are labels corresponding to the lhs in the definition of  $PEq$ :

$$\begin{aligned} (172) \quad \varepsilon''_w & : (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w) \\ (173) \quad & = (\varepsilon'_w, \varepsilon_r^?) : (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown \varsigma_w) \\ (174) \quad & \text{(Lemma 6)} \\ (175) \quad i & : \min\{\text{len}(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w), \\ (176) \quad & (f_t \circ Done_P[v, \mu(a)])(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w)\} \\ (177) \quad & = \min\{\text{len}(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown \varsigma_w), 1\} \\ (178) \quad & \text{(Lemma 5)} \\ (179) \quad & = 1 \\ (180) \quad \pi_1(\varepsilon''_w[i]) & : \pi_1(((\varepsilon'_w, \varepsilon_r^?) : (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown \varsigma_w))[1]) \\ (181) \quad & = \pi_1(\varepsilon'_w, \varepsilon_r^?) \\ (182) \quad & = \varepsilon'_w \end{aligned}$$

The result is used with  $P$  to give the value of the rhs:

$$(183) P[\varepsilon'_w](a, v)\mu$$

Examining the definitions of  $P$  and  $CP$  show that the lhs and rhs are equal. ♣

The rest of the proof considers the other case:

$$\neg \text{Done}_P[v, \mu(a)](\varepsilon'_w, \varepsilon_r^?)$$

**Proof 5** The lhs reduces, by Lemma 4, to:

$$(184) CP[\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w](a, v)\mu$$

The value of  $PEq[v, \mu(a)](\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \hat{\ } \zeta_w)$  is evaluated as shown below, where the lhs are labels corresponding to the lhs in the definition of  $PEq$ :

$$\begin{aligned} (185) \quad \varepsilon''_w & : (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \hat{\ } \zeta_w) \\ (186) & = (\varepsilon'_w, \varepsilon_r^?) : (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w) \\ (187) & \quad (\text{Lemma 6}) \\ (188) & = (\varepsilon'_w, \varepsilon_r^?) : \sigma'' \\ (189) & = \mathbf{where} \sigma'' =: (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w) \\ (190) \quad i & : \min\{\text{len}((\varepsilon'_w, \varepsilon_r^?) : \sigma'', (f_i \circ \text{Done}_P[v, \mu(a)])(\varepsilon'_w, \varepsilon_r^?) : \sigma'')\} \\ (191) & = \min\{1 + \text{len}(\sigma''), 1 + (f_i \circ \text{Done}_P[v, \mu(a)])\sigma''\} \\ (192) & \quad (\text{Lemma 5}) \\ (193) & = 1 + \min\{\text{len}(\sigma''), (f_i \circ \text{Done}_P[v, \mu(a)])\sigma''\} \\ (194) & = 1 + j \\ (195) & \quad \mathbf{where} j = \min\{\text{len}(\sigma''), (f_i \circ \text{Done}_P[v, \mu(a)])\sigma''\} \\ (196) \quad \pi_1(\varepsilon''_w[i]) & : \pi_1(((\varepsilon'_w, \varepsilon_r^?) : \sigma'')[1 + j]) \\ (197) & = \pi_1(\sigma''[j]) \end{aligned}$$

Let us expand  $PEq[v, \mu(a)](\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w)$ , (ignoring the conditional for odd length lists) and compare:

$$\begin{aligned} (198) \quad & PEq[v, \mu(a)](\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w) \\ (199) & = \mathbf{let} \varepsilon''_w = (\Pi^\diamond \circ \langle \cdot, \cdot \rangle)(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w) \mathbf{in} \\ (200) & \quad \mathbf{let} i = \min\{\text{len} \varepsilon''_w, (f_i \circ \text{Done}_P[v, \mu(a)])(\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \hat{\ } \zeta_w)\} \mathbf{in} \\ (201) & \quad \pi_1(\varepsilon''_w[i]) \end{aligned}$$

If we compare this with the previous expansion, we see that they are the same, noting the correspondence between  $\varepsilon''_w$  and  $\sigma''$ , and between  $i$  and  $j$ .

We have shown that we can get from  $\langle \varepsilon'_w, \varepsilon_r^?, \varepsilon_w, \varepsilon_r \rangle \frown_{\zeta_w}$  to  $\langle \varepsilon_w \odot \varepsilon'_w, \varepsilon_r \rangle \frown_{\zeta_w}$ , as far as  $PEq$  is concerned. It is a trivial matter to insert this into  $P$ , thus obtaining the rhs. Comparison of the lhs and rhs now shows that we have completed the inductive step. This completes the proof. ♣

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

**Question 2** Is the set of events that can result from determining the equivalences of 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?

The answer is NO. Presented here are the counter-examples that form the proof. In effect we demonstrate how to construct, for any event, an event sequence the given event as its equivalent.

- **CarefulGet:** The following Read Event sequence will always be equivalent to  $\varepsilon_r$ , regardless of the nature of that event:

$$\underbrace{\langle \varepsilon_r, \varepsilon_r, \dots, \varepsilon_r \rangle}_{n \text{ times}}$$

- **CarefulPut:** The following Write Event sequence will always be equivalent to  $\varepsilon_w$ , regardless of the nature of that event:

$$\langle \varepsilon_w, \lambda v \cdot \lambda(b, w) \cdot (\text{TRUE}, v) \rangle$$

- **CarefulPut of given value  $V$ :** If we restrict the “read events” in the sequence to be of the form  $K\varepsilon_r$ , representing lifted “real” Read Events, then a single counter-example is no longer possible. However, if we restrict our question to be concerned with particular instances of  $v$  being written, then it is possible to produce a family of counter-examples, one for each possible value of  $v$ . The following Write Event sequence will always be



equivalent to  $\varepsilon_w$ , regardless of the nature of that event, given that the value  $V$  is being written:

$$\langle \varepsilon_w, \mathbf{K}(\lambda(b, w) \cdot \text{TRUE}, V) \rangle \hat{=} \zeta_w$$

The Careful operators provide *quantitative* fault tolerance, in that they reduce the probability of some events occurring. They do not provide *qualitative* fault tolerance, which requires the probability of some events to be reduced to zero.

**Question 3** Lamson asserts, in [Lam81] on page 254 that:

“CarefulPut . . . eliminates soft read errors. CarefulPut . . . eliminates null writes; it also eliminates bad writes, provided there is no crash during the CarefulPut”

Does the negative answer to the previous question contradict this ?

Once more the answer is No. What Lamson refers to as a *soft read error* (page 250) is in fact a (sub-)sequence of persistently erroneous Read Events whose length is less than  $n$ , the number of tries made by CarefulGet. Any single null or bad write will be masked by CarefulPut if it is the only such error present. We have seen that any such event can be made manifest, but only by the occurrence of the pathological Read Event  $(\lambda v \cdot \lambda(b, w) \cdot (\text{TRUE}, v))$  immediately afterwards.

The confusion arises simply because Lamson calls some events “errors”, whereas we would refer to an equivalent “event sequence”

Another classic case considered in fault tolerant systems concerns *single failures*

**Question 4** Is the set of events that can result from determining the equivalents of all sequences that contain at most ONE real error, a proper subset of the set of all possible errors ? In other words, has the introduction of the Careful operators eliminated some SINGLE errors ?

Note that we are discussing sequences where every event, bar one, is either  $\varepsilon_w^x$  or  $\mathbf{K}\varepsilon_r^x$ , as appropriate. Also note that a crash counts as an error, strictly speaking, as does the occurrence of a previous event that left an erroneous value in memory. These two cases are important, as will be seen.

The answer to the question is YES, with some caveats. Here, we simply show the appropriate examples, and discuss the caveats.

- **CarefulGet:** If a crash occurs, then the Get-Equivalent Form of a Read Event Sequence is not defined, as no value is returned. Therefore, no single event function can be said to be equivalent to a sequence which denotes a crash. In the absence of crashes, we must consider if an erroneous value in memory prior to the CarefulGet is to be counted as the single event. A reasonable answer is to exclude such situations, because they concern the result of some previous CarefulPut, and considering them in conjunction with a specific CarefulGet only serves to muddle the issue. In effect, we have refined the question to consider the case when memory's current contents are correct and no crash occurs. The following identities show that, under these circumstances, CarefulGet eliminates all single events that would be classified as  $\text{ERR}(?)$ , but will fail to convert  $\text{BAD}(\times)$  to  $\text{OK}(\surd)$ :

$$(202) \quad \text{GEq}[\text{TRUE}, w] \varepsilon_r^\surd : \varsigma_r = \varepsilon_r^\surd$$

$$(203) \quad \text{GEq}[\text{TRUE}, w] \varepsilon_r^? : \varepsilon_r^\surd : \varsigma_r = \varepsilon_r^\surd$$

$$(204) \quad \text{GEq}[\text{TRUE}, w] \varepsilon_r^\times : \varsigma_r = \varepsilon_r^\times$$

Note that the case  $\text{GEq}[\text{TRUE}, w] < \varepsilon_r^? >$  is not allowed as this denotes a crash, while the case  $\text{GEq}[\text{FALSE}, w] \varsigma_r$  denotes the case when a prior event has left memory corrupted. We can conclude, given the above conditions, that CarefulGet eliminates single errors, as long as they do not constitute a BAD event.

- **CarefulPut:** In this case we get a much stronger result than was obtained for CarefulGet. The result is characterised by the following identities, which are independent of the context, or the fact that an event might be ERR or BAD in that context:

$$(205) \quad \text{PEq}[v, (b, w)] < \varepsilon_w^x, \mathbf{K}\varepsilon_r^x > \frown \varsigma_w = \varepsilon_w^x$$

$$(206) \quad \text{PEq}[v, (b, w)] < \varepsilon_w, \mathbf{K}\varepsilon_r^x, \varepsilon_w^x, \mathbf{K}\varepsilon_r^x > \frown \varsigma_w = \varepsilon_w^x$$

$$(207) \quad \text{PEq}[v, (b, w)] < \varepsilon_w^x, \varepsilon_r^w, \varepsilon_w^x, \mathbf{K}\varepsilon_r^x > \frown \varsigma_w = \varepsilon_w^x$$

Note that  $\varsigma_w$  denotes an arbitrary, possibly empty event list. Remember that only one error can occur, all the other events being  $\varepsilon_w^x$  or  $\mathbf{K}\varepsilon_r^x$  as appropriate. The only cases not covered above, containing only one real error, are those that denote a crash. The following identities show the results in the event of an immediate crash or a single Write or Read event followed by a crash:

$$(208) \quad \text{PEq}[v, (b, w)] \Lambda = \varepsilon_w^\phi$$

$$(209) \quad \text{PEq}[v, (b, w)] < \varepsilon_w, \mathbf{K}\varepsilon_r^x > = \varepsilon_w$$

$$(210) \quad \text{PEq}[v, (b, w)] < \varepsilon_w^x, \varepsilon_r^w > = \varepsilon_w^x$$

We can see that CarefulPut does eliminate all single errors, with the exception of a crash. We also note that single events which would be classified

as a BAD if they occurred during a Put operation are harmless when it comes to CarefulPut ! This is because the value written to memory is checked using a Get operation which obtains both the boolean flag *and* the value for cross-checking. This holds as long as no crash occurs before that particular Get.

In the previous discussion, the effects of prior operations on memory were ignored. We look briefly at the effect that a prior event has on the ability of Careful operations to cope. For CarefulPut with single error we see that previous erroneous values have no effect, but the situation is not so simple for CarefulGet. We know it cannot distinguish between OK and BAD when memory contains (TRUE,  $v$ ), but what about the case when memory is flagged as erroneous ? Consider the following identities, where  $\varsigma_r$  is an arbitrary read event list,  $\varsigma_r^?$  is a list of length less than  $n$  consisting solely of ERRs and noting that no OK event is possible ( $\kappa_G \llbracket \text{FALSE}, v \rrbracket \varepsilon_r^? = \text{ERR} !$ ):

$$(211) \quad GEq \llbracket \text{FALSE}, w \rrbracket \varepsilon_r^\times : \varsigma_r = \varepsilon_r^\times$$

$$(212) \quad GEq \llbracket \text{FALSE}, w \rrbracket \varsigma_r^? \frown \langle \varepsilon_r^\times \rangle \frown \varsigma_r = \varepsilon_r^\times$$

$$(213) \quad GEq \llbracket \text{FALSE}, w \rrbracket \varsigma_r^? \frown \langle \varepsilon_r^? \rangle \frown \varsigma_r = \varepsilon_r^?$$

$$(214)$$

We see that the result is BAD if any such event occurs before the  $n$  tries expire. If the single event is ERR given the context, then the effect of CarefulPut is that of the event ( $\varepsilon_r^?$  or otherwise) which occurs in the  $n$ th position.

We conclude by noting that the equivalence operators give us an easy way to classify event sequences given a context — we simply reduce them down to the equivalent single event (in that context) and classify the event:

$$(215) \quad \kappa_{CG} : \mathbf{B} \times \text{VAL} \rightarrow \text{R\_EVTS} \rightarrow \text{ECLASS}$$

$$(216) \quad \kappa_{CG} \llbracket b, w \rrbracket \varsigma_r \triangleq \kappa_G \llbracket b, w \rrbracket (GEq \llbracket b, w \rrbracket \varsigma_r)$$

$$(217) \quad \kappa_{CP} : \text{VAL} \times \mathbf{B} \times \text{VAL} \rightarrow \text{W\_EVTS} \rightarrow \text{ECLASS}$$

$$(218) \quad \kappa_{CP} \llbracket v, (b, w) \rrbracket \varsigma_r \triangleq \kappa_P \llbracket v, (b, w) \rrbracket (PEq \llbracket v, (b, w) \rrbracket \varsigma_r)$$

Note that both these definitions can be expressed more concisely using the  $\odot$  operator:

$$\kappa_{CG} = \kappa_G \odot GEq$$

and

$$\kappa_{CP} = \kappa_P \odot PEq$$

A Careful Storage abstraction has been presented that improves the reliability of error-prone memory Put and Get operations by means of repetition. It has been demonstrated how the effect of such “Careful” operators, given the sequence of events occurring during any use of the same, can be reduced down to a single equivalent event. It has also been shown that the Careful Storage abstraction does not eliminate the possibility of any events. However, it has been seen that certain events, namely those classified as erroneous (ERR), will only occur if they are *persistent*. In this case, persistence means that the error occurs  $n$  times in a row, when  $n$  is an implementation defined constant used by the CarefulGet algorithm. It must be stressed that a probabilistic analysis has not been performed at this juncture to produce a more fault tolerant implementation

## VDM<sup>\*</sup>Notation

This notation is derived from [Mac90, Mac91].

Symbol	Meaning
$X \xrightarrow{m} Y$	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 $f$
$\pi_n$	$n$ th Projection Function
rng	Map Range
$(f \xrightarrow{m} g)$	Maps $f$ and $g$ to Domain and Range resp. of a Map
$X^*$	<i>Finite</i> Sequences over $X$
$\Lambda$	The Null Sequence
$:$	The Sequence ‘Cons’ Operator (a la HASKELL[Com92])
$[l \dots h]$	Sequence Subrange operator
$f^*$	Maps $f$ into a Sequence (Kleene Star functor)
len	The Sequence Length operator
$\subseteq$	The Subset relation
$\neg$	Logical Negation
$\Rightarrow$	Logical Implication
$\langle x \rangle$	Singleton sequence containing $x$ .
$\langle x, \dots, y \rangle$	Sequence notation
$\frown$	Sequence Concatenation operator

## Possible extensions to VDM<sup>\*</sup>Notation

Symbol	Meaning
$\odot_x$	Curried-Function Composition, with context $x$ [But92b]
$\odot$	Curried-Function Composition (Context-Free) [But92b]
$\langle \cdot \rangle$	Adjacent Element Pairing operator
$\amalg$	Mapped Reduction (hybrid of Mapping and Reduction) [But93]

- [But92a] Andrew Butterfield. Memory models — a formal analysis using *VDM\**. Technical Report TCD-CS-92-27, Dept. of Comp. Science, Trinity College, Dublin, April 1992.
- [But92b] Andrew Butterfield. On curried function composition. Technical Report TCD-CS-92-15, Dept. of Comp. Science, Trinity College, Dublin, May 1992.
- [But93] Andrew Butterfield. On map reduction. Technical Report to appear, Dept. of Comp. Science, Trinity College, Dublin, 1993.
- [Com92] Haskell Committee. Report on the programming language haskell. Technical Report Version 1.2, Haskell Committee, March 1992.
- [Lam81] B. W. Lampson. 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.
- [Mac90] Mícheál Mac an Airchinnigh. *Conceptual Models and Computing*. PhD thesis, Dept. of Comp. Sci., Trinity College Dublin, Ireland, 1990.
- [Mac91] Mícheál Mac an Airchinnigh. The irish school of vdm. In *VDM '91*, volume 552 of *Lecture Notes in Computer Science*. Springer Verlag, 1991.