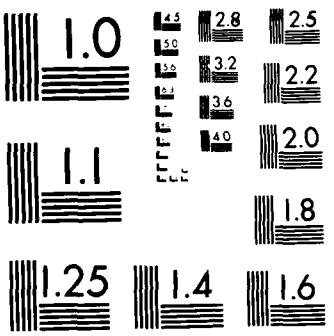
AD-A192 483 THE SOFTWARE REPEATER (AN EXERCISE IN Z SPECIFICATION) 1/1
(U) ROYAL SIGNALS AND RADAR ESTABLISHMENT MALVERN
(ENGLAND) C O'HALLORAN OCT 87 RSRE-MEMO-4090
UNCLASSIFIED DRIC-BR104504 F/G 12/5 NL



UNLIMITED

RSRE

MEMORANDUM No. 4090

ROYAL SIGNALS & RADAR ESTABLISHMENT

THE SOFTWARE REPEATER
(AN EXERCISE IN Z SPECIFICATION)

Author: C O'Halloran

PROCUREMENT EXECUTIVE,
MINISTRY OF DEFENCE,
R S R E MALVERN,
WORCS.

RSRE MEMORANDUM No. 4090

UNLIMITED

ROYAL SIGNALS AND RADAR ESTABLISHMENT

Memorandum 4098

Title : The software repeater
(an exercise in Z specification)

Author : C. O'Halloran

Date : October 1987

Summary

The specification of a software repeater in a language known as Z is presented. The specification is then refined into an implementation in the C programming language. The correctness of the implementation is shown by means of proof obligations and the program analysis tool Malpas. The C implementation was compiled and ran successfully on a 68010 processor.

INTRODUCTION

This memorandum arose from a problem posed by the panel on trustworthy computing technology under the auspices of the technical co-operation programme. This panel posed a simple problem in the specification of a process introducing concurrency, the software repeater. The process receives characters and at some later time transmits them. Since characters may arrive at a rate greater than the rate at which they can be transmitted an internal buffer is necessary.

Solutions of this problem illustrate techniques currently available for the formal specification, design and implementation of a concurrent process. The reason for choosing this particular problem was that if it could not be solved easily then it would not be feasible to use formal methods for more realistic problems. However formal methods would not normally be applied to a problem like this as it is so simple.The following specification was given:

1. Everything received by the repeater is re-transmitted, eight bit character by eight bit character.

2. Only received characters are transmitted.

3. The internal buffer used to store the characters must be finite.

4. Every error condition must be characterised, first by writing to an alternative port and then by halting.

5. Any simple processor may be used to implement the repeater. The repeater's design and implementation must allow its ports to run at a range of speeds.

This memo falls into three main sections. The first section deals with the specification of the repeater in Z. In this section the first two assertions about the repeater are shown to be valid. The other assertions such as the precise size of the buffer are design criteria. In the second section a design for the repeater is developed and proofs concerning the correspondence between specification and design are conducted. The third section deals with the implementation of the design in C for its eventual compilation and running on a processor. The C implementation is then translated into a language called MALPAS IL enabling it to be analysed for various purposes. There is one appendix listing the MALPAS IL model with comments on translation.

MALPAS IL is a program modelling language based upon directed graphs, for a description of the language see [RTP 87]. The equivalence between directed graphs and regular expressions allow abstract interpretations to be performed in various regular algebras. One analysis in particular yields the semantic relationship between the initial and the final state of a program. In conjunction with a form of verification condition generator and a simplifier, an automatic and independent proof that a program satisfies a specification may be obtained. For an overview see [Bramson 85, 87]. The ability to verify an IL model by the use of assertions gives MALPAS some of the properties of formal specification systems such as GYPSY or EVES, [Smith 87, Pase 83, Craigen 86].

For a description of Z and associated refinement see [Hayes 86, Spivey 86, Sufrin 86, Sanders 86, Sørensen 86]. Z is based on elementary set theory which provides a readily understandable basis for specification. It is structured using the language of schemas, which supports an incremental style of presentation and is explained in detail in [Morgan 84]. Z and the development method originated from the Oxford Programing Research Group.

"Thou whoreson Zed,
thou unnecessary letter!" King Lear

Data Objects

The fact that the characters are represented by eight bit characters is irrelevant to the specification at this level. Therefore the set of all possible characters is introduced as a given set.

[Char]

To describe the behaviour of the process sequences of characters which will record the history of the characters input or output are introduced by a schema called HISTORY.

HISTORY

$$input\_trace, output\_trace : seq\ Char$$

When a character is received then the identifier input_trace must be updated, similarly for output_trace. To denote some change in an identifier a prime "'" is appended to it. A schema name which is primed means that the identifiers declared within it are primed and the meaning of a schema which contains a schema name is given by expanding the included schema. Thus a change in the recorded history of input and output is denoted by the following schema.

ΔHISTORY

HISTORY
HISTORY'

---

input_trace prefix input_trace'
output_trace prefix output_trace'

The predicate below the line guarantees that the behaviour of the process is a continuation of the previous behaviour. For example input_trace cannot be updated by making it a shorter sequence. Note that there is an implicit conjunction between predicates on different lines. Finally an update to the internal buffer is defined by

ΔBUFFER

$$buffer, buffer' : seq\ Char$$

Repeater Operations

The maximum size for the buffer is max : ℕ. The initial and intermediate states of the repeater are given by the initial and final states of the operation RECITE. The schema RECITE specifies the behaviour of the repeater before one of two error conditions occur. The handling of errors in Z is specified separately by another schema, in this case OVERFLOW which is described later. One error condition is that a character collected at the input port has been overwritten by another incoming character. To model this a set of values of a status register for the input port is introduced before the operation. The second error condition is when the internal buffer overflows.

$Input\_status\_register \triangleq \{ready, collect, overflow\}$

RECITE

$\Delta BUFFER$
$HISTORY$
$status' : Input\_status\_register$

$input\_trace = \langle\rangle \land output\_trace = \langle\rangle \land buffer = \langle\rangle$
$input\_trace' = output\_trace' \frown buffer'$
$(\#buffer' \geq max \lor status' = overflow)$

This operation defines the initial state of the buffer and traces to be equal to the empty sequence. Buffer overflow occurs when the buffer contains 'max' characters. This guarantees that the repeater does not lose the last character. The final state constrains the input trace to be the concatenation of the output trace with the buffer.

When an error occurs it must be reported to the external world. Output communications are denoted by decorating an identifier by a shriek, '!'. Since a report is to be communicated, a set of reports must be introduced. The specification does not say what they are only that they exist. Two error reports from this set are declared afterwards.

$[Report]$

$buffer\_overflow, bit\_overflow : Report$

OVERFLOW

$buffer', input\_trace', output\_trace' : seq\ Char$
$rep! : Report$
$status' : Input\_status\_register$

$input\_trace' = output\_trace' \frown buffer'$
$((\#buffer' \geq max \land rep! = buffer\_overflow)$
$\lor$
$(status' = overflow \land rep! = bit\_overflow \land \#buffer' \neq max)$
$)$

The schemas RECITE and OVERFLOW may be conjoined to give a description of the repeater with error handling.

$$REPEATER_0 \triangleq RECITE \land OVERFLOW$$

To complete the specification an operation to transmit the characters held in the buffer must be defined.

CLEARDOWN

buffer : seq Char
ΔHISTORY

---

input_trace' = input_trace
input_trace = output_trace^buffer
output_trace' = output_trace^buffer

The total specification of the repeater is given by composing the operations $REPEATER_0$ and CLEARDOWN.

$$REPEATER \triangleq REPEATER_0 \fatsemi CLEARDOWN$$

The English specification given in the introduction is not complete with regard to errors. It states that after reporting an error the repeater must halt, yet earlier it stated that everything received must be re-transmitted. This specification gives a clear and unambiguous description of the software repeater. In addition properties of the formal description may be derived.

Satisfaction of Requirements

To show that the specification meets the relevant requirements the following theorem is proved.

$$REPEATER \vdash input\_trace' = output\_trace'$$

Proof:

By the definition of ⨾ REPEATER is

REPEATER

buffer : seq Char
ΔHISTORY
status, status' : Input_status_register
rep! : Report

---

input_trace = ⟨⟩ ∧ output_trace = ⟨⟩ ∧ buffer = ⟨⟩
∃ in, out, buff : seq Char .
in = out^buff
((#buff ≥ max ∧ rep! = buffer_overflow) ∨
(status' = overflow ∧ rep! = bit_overflow ∧ #buff ≠ max))
input_trace' = in ∧ output_trace' = out^buff

which, by elimination of the existential quantifier, simplifies to

input_trace', output_trace' : seq Char
status, status' : Input_register_port
rep! : Report

---

input_trace' = output_trace'

which contains the required predicate. □

THE DESIGN

Refinement

In the previous section a formal unambiguous specification of the repeater was developed. The next stage is to produce a design which is a lower level representation of the specification. The design arises when the specification is refined into a small number of components. Each component is a new specification of a smaller part of the program. The design also describes how the components are composed so that their contributions satisfy the original specification. In the final step into code each component is refined until it may be directly expressed in terms of the target programming language. At each stage of refinement correctness is assured by addressing certain proof obligations. This form of refinement is known as algorithmic refinement.

Control constructs

Our design will be expressed using the generalised control constructs described in [GRIES 81], these constructs are not part of the Z language. The general form of the alternative command is

$$
\begin{aligned}
&\textbf{if } B_1 \text{ ---> } S_1 \\
&\ \ [\!] \ B_2 \text{ ---> } S_2 \\
&\quad \ldots \\
&\ \ [\!] \ B_n \text{ ---> } S_n \\
&\textbf{fi}
\end{aligned}
$$

Where $n \in \mathbb{N}$.

Informally each B_i ---> S_i is a guarded command one of which may be executed if the guard B_i is true. The other important control construct used is the iterative command. The general form of the iterative command is

$$
\begin{aligned}
&\textbf{do } B_1 \text{ ---> } S_1 \\
&\ \ [\!] \ B_2 \text{ ---> } S_2 \\
&\quad \ldots \\
&\ \ [\!] \ B_n \text{ ---> } S_n \\
&\textbf{od}
\end{aligned}
$$

Where $n \in \mathbb{N}$. We shall only be using the singleton form of this command. In this case the iterative command is the usual notion of the WHILE DO construct in programming languages.

Design refinement

Since REPEATER was specified by the composition of $REPEATER_0$ and CLEARDOWN it may be decomposed into the sequential composition of two program operations specified by these schemas. The schema $REPEATER_0$ was itself a conjunction of the two schemas, RECITE and OVERFLOW. A natural decomposition of $REPEATER_0$ is therefore the sequential

... RECITE and the error handler OVERFLOW. The symbol denoting refinement is $\sqsubseteq$, so that the refinement of $REPEATER_0$ may be expressed as

$$REPEATER_0 \sqsubseteq RECITE ; OVERFLOW$$

Note that sequential composition is not schema composition, it is a composition operator of a program design. Also RECITE and OVERFLOW stand for operations satisfying these schemas. Also note that it is impossible to implement $REPEATER_0$ by reporting that its buffer had overflowed before repeating a single character. This is because to guarantee refinement certain proof obligations must be addressed. In this case they are

$REPEATER_0 \sqsubseteq RECITE ; OVERFLOW$
if and only if

(1) $\mathbf{pre}\ REPEATER_0 \vdash \mathbf{pre}\ RECITE$
(2) $\mathbf{pre}\ REPEATER_0 \wedge RECITE \vdash (\mathbf{pre}\ OVERFLOW)'$
(3) $\mathbf{pre}\ REPEATER_0 \wedge RECITE \wedge OVERFLOW \vdash REPEATER_0[\_''/\_']$

$P[\_''/\_']$ is a notational convention renaming all variables decorated with a single dash "'" to variables decorated with a double dash "''". The **pre** operator gives the precondition for the schema operation. The precondition in Z is something which guarantees that a final state can in fact be reached from a state satisfying it. These theorems are easy to prove but their proofs are omitted due to considerations of space, a more substantial example will be given later. The overall refinement so far is the following.

$$REPEATER \sqsubseteq RECITE ; OVERFLOW ; CLEARDOWN$$

Again RECITE, OVERFLOW and CLEARDOWN are operations satisfying the schema specifications. The first operation receives characters and re-transmits them at some later time until some error occurs. When this happens the RECITE operation halts and an error is reported by OVERFLOW. Finally the CLEARDOWN operation transmits the characters left in the buffer.

Because RECITE describes the operation of receiving and transmitting characters constantly, a natural development would be as an iterative command. To ensure that this construct behaves in the way described by RECITE, when their domains agree, then again some proof obligations must be satisfied. Normally these proof obligations would include showing the existence of a measure which decreases after each iteration. The measure guarantees the termination of the loop after a finite number of iterations. The specification of the repeater implies that after a finite number of characters have been received an error occurs. This is not a desirable property for an implementation to have so the proof obligations concerning the boundedness of the loop are ignored. With this simplification the proof obligations for the partial correctness of a refinement are:

RECITE $\sqsubseteq$ INV ;
do
GUARD ---> BODY (Guarded command loop)
od

if and only if

(1) $\mathbf{pre}\ RECITE \vdash \mathbf{pre}(INV \wedge \neg GUARD')$
(2) $\mathbf{pre}\ RECITE \wedge INV \wedge \neg GUARD' \vdash RECITE$

(3) $\mathbf{pre}\ RECITE \wedge INV \wedge GUARD' \vdash (\mathbf{pre}\ BODY)'$

(4) $\mathbf{pre}\ RECITE \wedge INV \wedge GUARD' \wedge BODY' \vdash INV[\_''/\_']$

The following schemas define the guard and invariant relation which acts as an initialisation operation. They are invented and then then tried to determine whether the proof obligations can be shown to hold. A few attempts were necessary before the present schemas were chosen.

GUARD

$buffer : \mathbf{seq}\ Char$
$status : Input\_status\_register$

$\#buffer < max \wedge status \neq overflow$

INV

$input\_trace', output\_trace', buffer' : \mathbf{seq}\ Char$

$input\_trace' = output\_trace'^\frown buffer'$

The body of the loop is defined by taking the disjunction of three schemas COLLECT, TRANSMIT and SKIP. The repeater knows when a character has been collected at the input port because the input status register has status 'collect'.

COLLECT

$status : Input\_status\_register$
$char? : Char$
$\Delta BUFFER$
$\Delta HISTORY$

$status = collect$
$buffer' = buffer^\frown(char?)$
$input\_trace' = input\_trace^\frown(char?)$
$output\_trace' = output\_trace$

The operation COLLECT specifies that the status register has the value 'collect' and the character at the input port is added to the buffer. The record of characters received is updated but no change is made to the record of characters output.

The operation TRANSMIT specifies that the status register for the output port indicates that it is ready to receive another character. The character at the head of the queue is recorded in the output trace but the input trace is unaltered. To specify the TRANSMIT operation the output port needs a status register. The status signifies when a character has been output so that a character from the buffer does not overwrite it before transmission. Hence the set of values for the output status register is introduced and afterwards the particular value 'empty' is declared.

$[Output\_status\_register]$

$empty : Output\_status\_register$

TRANSMIT

status_o : Output_status_register
char! : Char
ΔHISTORY
ΔBUFFER

---

status_o = empty
buffer' = tail(buffer)
char! = head(buffer)
output_trace' = output_trace^(char!)
input_trace' = input_trace

Finally the SKIP operation leaves the state of the buffer and the traces unchanged whenever the status of the output port is unready or the status of the input port is not collect. For example if the output port has not transmitted all the bits of a character then it is not ready to receive a new character from the buffer. Similarly the input port may still be collecting the bits of an incoming character.

SKIP

status : Input_status_register
status_o : Output_status_register
ΞHISTORY
ΞBUFFER

(status_o ≠ empty ∨ buffer = ())
status = ready

The meaning of the Ξ operator is that a state transformation has taken place leaving the state of the schema unchanged. The previous three schemas define the body as

$$BODY \triangleq COLLECT \vee TRANSMIT \vee SKIP$$

The following propositions discharge the proof obligations for the refinement of RECITE. The first proposition ensures that there exists a state where both the invariant relation holds and the guard is false. (The precondition of a schema may be calculated by removing all variables ending with a prime or shreik "!" from the signature. They are then renamed and existentially quantified in the predicate part of the schema.)

Proposition 1: $\mathbf{pre}\ RECITE \vdash \mathbf{pre}(INV \wedge \neg GUARD')$

Proof:
Calculating $\mathbf{pre}(INV \wedge \neg GUARD')$ gives

$\mathbf{pre}(INV \wedge \neg GUARD') =$
$\exists$ buff, in, out : seq Char; stat : Input_status_register •
in = out^buff $\wedge$ (#buff $\geq$ max $\vee$ stat = overlfow)

which simplifies to true. □

The next proposition ensures that RECITE is satisfied if the loop terminates.

Proposition 2: **pre** RECITE $\wedge$ INV $\wedge$ $\neg$GUARD' $\vdash$ RECITE

Proof:
Calculating the precondition for RECITE and conjoining it with INV and $\neg$GUARD' gives

$$
\begin{array}{|l}
\Delta\text{BUFFER} \\
\Delta\text{HISTORY} \\
\text{status'} : \text{Input\_status\_register} \\
\hline
\text{input\_trace} = \langle\rangle \wedge \text{output\_trace} = \langle\rangle \wedge \text{buffer} = \langle\rangle \\
\text{input\_trace'} = \text{output\_trace'}\frown\text{buffer'} \\
(\#\text{buffer'} \geq \text{max} \vee \text{status'} = \text{overflow}) \\
\exists\ \text{buf, in, out} : \text{seq Char; stat} : \text{Input\_status\_register} \bullet \\
(\#\text{buf} \geq \text{max} \vee \text{stat} = \text{overflow}) \wedge \text{in} = \text{out}\frown\text{buf}
\end{array}
$$

which may be simplified to

$$
\begin{array}{|l}
\Delta\text{BUFFER} \\
\Delta\text{HISTORY} \\
\text{status'} : \text{Input\_status\_register} \\
\hline
\text{input\_trace} = \langle\rangle \wedge \text{output\_trace} = \langle\rangle \wedge \text{buffer} = \langle\rangle \\
\text{input\_trace'} = \text{output\_trace'}\frown\text{buffer'} \\
(\#\text{buffer'} \geq \text{max} \vee \text{status'} = \text{overflow})
\end{array}
$$

which is the same schema as RECITE.
□

The following proposition ensures that the body of the loop is applicable to all states satisfying the invariant relation (in particular the states satisfying the initialisation) providing the guard allows the iteration to continue.

Proposition 3: **pre** RECITE $\wedge$ INV $\wedge$ GUARD' $\vdash$ (**pre** BODY)'
Proof:
The hypothesis of the theorem is pre RECITE $\wedge$ INV $\wedge$ GUARD', which is

$$
\begin{array}{|l}
\Delta\text{BUFFER} \\
\Delta\text{HISTORY} \\
\text{status'} : \text{Input\_status\_register} \\
\hline
\text{input\_trace} = \langle\rangle \wedge \text{output\_trace} = \langle\rangle \wedge \text{buffer} = \langle\rangle \\
\text{input\_trace'} = \text{output\_trace'}\frown\text{buffer'} \\
\#\text{buffer'} < \text{max} \wedge \text{status'} \neq \text{overflow}
\end{array}
$$

(**pre** BODY)' is equivalent to (**pre** COLLECT $\vee$ **pre** TRANSMIT $\vee$ **pre** SKIP)'. These preconditions may be calculated and then disjoined to give (**pre** BODY)'.

status' : Input_status_register status_o' : Output_status_register buffer' : seq Char
status' = collect ∨ (status_o' = empty ∧ #buffer' ≠ 0) ∨ (status_o' ≠ empty ∧ status' = ready) ∨ (buffer' = ⟨⟩ ∧ status' = ready)

The relevant predicate of (**pre** RECITE ∧ INV ∧ GUARD') is (#buffer < max ∧ status' ≠ overflow). To prove (**pre** BODY)' a case analysis is performed. Since the value of status' is drawn from the set Input_status_register it can either be equal to 'ready' or 'collect'.

CASE 1: Assume status' = collect. In this case proof of (**pre** BODY)' is immediate because the first disjunct is (status' = collect).

CASE 2: status' ≠ collect. By hypothesis status' ≠ overflow therefore status' = ready because this is the only other member of the set Input_status_register.

> CASE 2A: status' = ready ∧ status_o' = empty.
> In this case (**pre** BODY)' may be proved if #buffer' ≠ 0 because of the second disjunct of (**pre** BODY)'. If #buffer = 0 then buffer = ⟨⟩ and therefore the fourth disjunct of (**pre** BODY)' is true because in this limb of the proof status' = ready.
>
> CASE 2B: status' = ready ∧ status_o' ≠ empty.
> In this case the proof of (**pre** BODY)' is immediate because of the third disjunct.

All the possible cases have now been exhausted and in each case (**pre** BODY)' has been shown to be true. □

It was necessary to change the specification of SKIP and Input_status_register in order to prove the previous proposition. For example the type Input_status_register was previously just a given set, but a crucial part of the above proof is the fact that the set contains only three elements. Hence the set was re-defined to make it have only three elements. The last proof obligation ensures that the body of the loop preserves the invariant.

Proposition 4: **pre** RECITE ∧ INV ∧ GUARD' ∧ BODY' ⊢ INV[_''/_']
Proof:

The hypothesis (**pre** RECITE ∧ INV ∧ GUARD' ∧ BODY') may be calculated and simplified using substitution and properties of sequences to give

```
input_trace, output_trace, buffer : seq Char
input_trace'', output_trace'', buffer'' : seq Char
buffer' : seq Char
status' : Input_status_register
status_o' : Output_status_register
char? : Char
```

```
input_trace = ⟨⟩ ∧ output_trace = ⟨⟩ ∧ buffer = ⟨⟩
#buffer' < max ∧ status' ≠ overflow
input_trace'' = output_trace''^buffer''
( status' = collect ∧ buffer'' = buffer'^(char?) )
∨
( status_o' = empty ∧ buffer'' = tail(buffer') )
∨
( (status_o' ≠ empty ∨ buffer' = ⟨⟩)
  buffer'' = buffer'
  status' = ready )
```

Calculating INV[_''/_'] gives

```
input_trace'', output_trace'', buffer'' : seq Char
```

```
input_trace'' = output_trace''^buffer''
```

The proof is immediate because the third conjunct of the hypothesis is the predicate in INV[_''/_'].

□

The next stage in the development of the design is to refine the body of the previos loop. IF - FI programs often arise from specifications of the form

$$F = F1 \vee F2$$

in which the pre-conditions of the disjuncts are characterised by predicates which can be expressed as boolean expressions in the target language. If this is the case F can be refined at once into the IF - FI program structure, like so:

```
F ⊑ if  pre F1 ---> F1
    □   pre F2 ---> F2
    fi
```

The body of the previous loop is a disjuction of three schemas, so the body may be refined in this manner. The preconditions for the operations COLLECT, TRANSMIT and SKIP have been calculated earlier so

```
BODY
⊑
if status = collect                      ---> COLLECT
 □ status_o = empty ∧ #buffer ≠ 0        ---> TRANSMIT
 □ (status_o ≠ empty ∨ buffer = ⟨⟩)
   ∧ status = ready                      ---> SKIP
fi
```

The operation OVERFLOW may be refined directly into code so this is delayed until the next section. The operation CLEARDOWN takes a buffer that has overflowed and re-transmits each character within it. The obvious way to implement this is using a guarded loop again. This time the loop must be shown to terminate after a finite number of iterations. The guard for the second iterative command is

C_GUARD

buffer : seq Char

#buffer > 0

The invariant for CLEARDOWN is

C_INV

input_trace', output_trace', buffer' : seq Char

---

input_trace' = output_trace'^buffer'

The body of the iterative command is defined by the disjunction of two operations TRANSMIT2 and SKIP2

TRANSMIT2

status_o : Output_status_register
char! : Char
ΔHISTORY
ΔBUFFER

---

status_o = empty
buffer' = tail(buffer)
char! = head(buffer)
output_trace' = output_trace^(char!)
input_trace' = input_trace

SKIP2

status_o : Output_status_register
ΞHISTORY
ΞBUFFER

---

status_o ≠ empty

C_BODY ≙ TRANSMIT2 ∨ SKIP2

C_BODY can be refined directly like so

```
C_BODY
⊑
if status_o = empty ∧ buffer ≠ ()  ---> TRANSMIT2
 [] status_o ≠ empty              ---> SKIP2
fi
```

As in the previous operation RECITE there are proof obligations including obligations to show that the command will terminate after a finite number of iterations.

```
CLEARDOWN ≙ C_INV ;
            do
              C_GUARD ---> C_BODY     (Guarded command loop)
            od

        if and only if

(1)     pre CLEARDOWN ⊢ pre (C_INV ∧ ¬C_GUARD')
(2)     pre CLEARDOWN ∧ C_INV ∧ ¬C_GUARD' ⊢ CLEARDOWN
(3)     pre CLEARDOWN ∧ C_INV ∧ C_GUARD' ⊢ bound_fuction(S') ≥ 0
(4)     pre CLEARDOWN ∧ C_INV ∧ C_GUARD' ⊢ (pre C_BODY)'
(5)     pre CLEARDOWN ∧ C_INV ∧ C_GUARD' ∧ C_BODY' ⊢
        [C_INV[_''/_'] | bound_function(S'') < bound_function(S')]

        where S ≙ [ΔHISTORY;ΔBUFFER]
        and bound_function : S ⇸ ℤ
```

The proofs of these theorems are similar to the previous set which have been proved, their proofs however shall be omitted. In the next section the design is assembled and implemented in C.

THE IMPLEMENTATION

So far the repeater has described its function at a character level. Each character is received and transmitted at a port. At this level the character is dealt with as a series of eight bits. The specification does not describe the bit level interaction because this is dealt with by the hardware. The hardware chosen to support the implementation is the Microsys VME bus system incorporating a 68010 processor. Assembling the components of the design for the repeater gives

```
INV;
do
  GUARD ---> if  status = collect                    ---> COLLECT
             [] status_o = empty ∧ #buffer ≠ 0       ---> TRANSMIT
             [] (status_o ≠ empty ∨ buffer = ())
               ∧ status = ready                      ---> SKIP
             fi
od;
OVERFLOW;
C_INV;
do
  C_GUARD ---> if  status_o = empty ∧ buffer ≠ ()  ---> TRANSMIT2
               [] status_o ≠ empty                 ---> SKIP2
               fi
od
```

These non-deterministic constructs may be implemented by deterministic if-then-else, while-do type constructs that the usual programming languages support. Hence a possible implementation for this design could be the following C program.

```
#define PORT_A 0xF40000  /* Base address of SIO Channel A */
#define PORT_B 0xF40020  /* Base address of SIO Channel B */
#define CMDREG 0x1       /* Offset of Command Register */
```

```
#define STAT_0 0x11       /* Offset of status Register 0 */
#define STAT_1 0x11       /* Offset of status Register 1 */
#define DATA 0x13         /* Offset of Data Register */

#define io(port,offset)  *(char *)(port+offset)
/* set up address of register on a channel correctly */
#define TRUE 1
#define FALSE 0
#define Max 64              /* Size of internal character buffer */

int start, finish, buffer_overflow, bit_overflow;

char buffer[Max];  /* Internal buffer */
char c;

get()  { c = io(PORT_B,DATA); }

put()  { io(PORT_B,DATA) = c; }

printerr(str)
char str[];
{
 int i;
 for (i=0; i < 17; i++)
 {
   while ( (io(PORT_A,STAT_0) & 4) == 0) /* output not ready */
   ;                                     /* wait */
   io(PORT_A,DATA) = str[i];
 }
}

main()
{
   int i;
   io(PORT_B,RCVCTL) = '\301'; /* Set up Receiver ctrl reg on B */
   io(PORT_B,XMTCTL) = '\307'; /* Set up Transmit ctrl reg on B */
   io(PORT_B,CMDREG) = '\060'; /* Reset Command register */

   start = 0;
   finish = 0;

   for(i=0; i<5000; i++);
   /* pause to ensure initialisation has occurred */

   while ( ((finish + 1) & (Max - 1)) != start) /*buffer not full */
           &&
           ((io(PORT_B,STAT_1) & 32) == 0)  /* status /= overflow */
         )
   {
      if io(PORT_B,STAT_0) & 1 != 0           /* status = collect */
      {
       get();
       buffer[finish] = c;
       finish = (finish + 1) & (Max-1)  /*finish:=finish+1 MOD 64 */
      }
      else
      {
         if ( ((io(PORT_B,STAT_0) & 4) != 0)  /* transmit status */
              &&
              (start != finish)               /* non-empty buffer */
            )
         {
             c = buffer[start]
             put();
```

```
                start = (start+1) & (Max-1);
            }
        }
    }
    if  ( ((start + 1) & (Max - 1)) == finish)  /* buffer full */
    {
        printerr("buffer overflow\r\n")
    }
    else
    {
        printerr("bit    overflow\r\n")

    }
    while (start != finish)  /* while the buffer isn't empty */
    {
        if ((io(PORT_B,STAT_0) & 4) != 0)  /* transmit status */
        {
            c = buffer[start]
            put();
            start = (start+1) & (Max-1);
        }
    }

}
```

The internal buffer is implemented by a character array. The array is indexed by two integers denoting the start and finish of the buffer. Addition involving these integers is constrained to the interval zero to sixty three by a bitwise 'and' with sixty three. In other words the indexes may be increased modulo sixty three. The status values correspond to certain bits being set in a register. For example if the register, at offset STAT_0, has the first bit set then a character has arrived ready for collection.

This implementation bears little resemblance to the design above. However on closer inspection it can be seen that the if-then-else construct implements the alternative construct and likewise the while-do for the iterative command. If fragments of the program are examined then it can be seen that they implement schema operations. For example the schema

TRANSMIT2

```
status_o : Output_status_register
char! : Char
ΔHISTORY
ΔBUFFER
```

```
status_o = empty
buffer' = tail(buffer)
char! = head(buffer)
output_trace' = output_trace^(char!)
input_trace' = input_trace
```

may be implemented simply by

```
{   c = buffer[start]
    put();
    start = (start+1) & (Max-1);
}.
```

The trace identifiers are not implemented of course since they are

records of the characters received and transmitted. However the output character 'c' does become the head of the buffer and the buffer does become the tail of the buffer by incrementing the 'start' index. The implementation could be shown to be correct by hand using pre and post conditions derived from the schema, but another approach is taken.

The C program is translated into MALPAS IL for subsequent verification. By strategic planting of assertions derived from the design and specification the IL may be verified. The IL translation may be found in the appendix. The assertions at the two loop heads are the invariants INV and C_INV. Hence the invariants from the design give rise to the loop invariants used in the IL. The only other assertion in the main part is the post condition that the input trace equals the output trace. This is simply the desired property of the process that was proved in the second section. Note that 'input_trace' and 'output_trace' appear in the IL description to describe the overall state of the process that was implicit in the C implementation.

The IL listed in the appendix, along with the appropriate declarations, was submitted for compliance analysis. As stated in the introduction compliance analysis employs semantic analysis. The semantic analysis of the program together with the assertions gives, for each program path, the circumstances under which the program will violate its specification. This is given as a predicate for each path and the disjunction of all these predicates is called the threat to the program. For example the threat condition on the path from the start of the program to the first loop head is given by the following predicate.

(1) empty_sequence ≠ convtoseq(buffer,0,0)

This arises because the assertion at the first loop head is 'input_trace = output_trace^convtoseq(buffer,start,finish)'. Now input_trace and output_trace have been initialised to empty_sequence and both start and finish were initialised to zero. Thus the assertion is false if the predicate (1) is true and hence this is a threat to the program satisfying its specification. However with judicious replacement rules all the threat conditions can be shown to be false. In this way the model of the implementaion was verified. The C program was compiled and ran successfully on the 68010 processor.

CONCLUSION

A method has been presented which describes a software repeater as a high level specification and enables an implementation to be produced which conforms to this specification. This implementation also ran successfully on a 68010 processor. The length of this paper is due to its tutorial nature. In the development of a system details of the languages Z and MALPAS IL would be omitted along with details of refinement.

The formulation of the English specification into Z showed many areas of uncertainty. The refinement with the associated proof obligations also fed back into the development of the repeater causing changes in the design. Tools to aid specification in Z and a proof assistant for Z are being built as part of the FORESITE project.

The partial correctness of the implementation is not very satisfactory and this approach to specifying concurrent processes is limited. A better approach would have been to use CSP, [Hoare 86], to specify the repeater. Indeed Trevor King has elegantly specified the repeater using CSP, [King 85]. Even though he does not discuss implementation his specification may be readily implemented in Occam, the programming language for the Inmos transputer. However the implementation would not be so easy if one wanted to implement the specification on some

other processor. A promising topic of current research at the Oxford Programming Research Group is the integration of CSP and Z. The use of formal methods made the production of such a simple program more difficult than an implementation from the English specification. But by use of formal methods correspondence between specification and implementation was shown as well as properties of the specification.

ACKNOWLEDGEMENTS

I am greatly indebted to Chris Sennett for his comments on this paper and his suggestions on the design of the software repeater.

REFERENCES

[Bramson 85] B D Bramson "Finding errors in computer programs"
RSRE report 85001, 1985

[Bramson 87] B D Bramson "Tools for the specification, design, analysis and verification of software"
RSRE report 87005, 1987

[Craigen 86] D Craigen "A description of m-Verdi" Draft paper
I.P.Sharp Associates Limited.
September 1986

[Gries 81] David Gries "The science of programming"
Springer-Verlag

[Hayes 86] Ian Hayes (editor) "Specification Case studies"
Prentice-Hall 1986

[Hoare 86] C.A.R Hoare "Communicating Sequential Processes"
Prentice-Hall 1986

[King 85] T King "Comments on the RS232 software repeater problem"
RSRE Working paper, October 1985

[Morgan 84] Carroll Morgan, Ian Hayes, Ib Sørensen
"The Schema Language"
Programming Research Group, Oxford 1984

[Pase 83] Bill Pase, S Kromodimoeljo
"Overview of an EVES prover"
I.P.Sharp Associates Limited.
August 1983

[RTP 87] Rex, Thompson & Partners Limited.
"MALPAS Intermediate Language (Version 4.0)"
MALPAS IL manual 1987.
Rex, Thompson & Partners Limited,
'Newnhams',
West Street,
Farnhamn
Surrey GU9 7EQ.

[Sanders 86] J.W.Sanders "Refinement for Z"
Z course notes, 1986.

[Smith 87] P Smith "A tutorial guide to the GYPSY methodology"
System Designers PLC. Ref C3158/17
June 1987

[Sørensen 86] Ib Holm Sørensen "Structured programming with schemas" Z course notes, 1986.

[Spivey 86] J M Spivey "Understanding Z" D.Phil Thesis, Oxford, 1986

[Sufrin 86] Bernard Sufrin (editor) "Z handbook" Draft 1.1, Oxford University Computing Laboratory, 1986

APPENDIX

The translation of a program into MALPAS IL is itself a form of analysis because of IL's strong typing. For example bit expressions in C are implicitly coerced to integers if the identifier on the left hand side of an assignment has type integer. In IL however any such coercion has to be explicit. This means coercion functions have to be introduced into the IL description.

To aid the automatic simplification of predicates replacement rules may be introduced. For example to simplify the threat expresion given at the end of section three the following replacement, or rewrite, rule might be defined.

```
REPLACE (b : char-array; s,f : integer )
convtoseq(b,s,f) BY empty_sequence : char-sequence IF s = f;
```

This rule introduces bound identifiers b, s and f. These bound identifiers occur in the expression which is to be replaced, namely 'convtoseq(b,s,f)'. The expression will be replaced by empty_sequence only if the integer s equals the integer f.

The procedures that were defined in the C program may be translated into IL procedures. These may be defined in two parts. The first is the PROCSPEC declaration, this specifies the procedure in terms of input and output variables. The second part defines the actual body of the procedure and allows the verification of the body with respect to the PROCSPEC declaration.

The following is the IL translation of the C program given in section three along with implanted assertions.

```
TITLE repeater;

TYPE port, char, bits;
TYPE input_status = (ready,overflow,collect);

FUNCTION input(bits) : input_status;
FUNCTION output(bits) : boolean;

CONST max : integer = 64;
CONST buffer_overflow : char-array = "buffer overflow/R/N";
CONST bit_overflow : char-array =    "bit    overflow/R/N";
CONST rcvctl : bits = "0xD"
CONST xmtctl : bits = "0xB"
CONST stat_0 : bits = "0xF"
CONST stat_1 : bits = "0x11"
CONST data   : bits = "0x13"
CONST port_a : port;
CONST port_b : port;

FUNCTION character(bits) : char;
FUNCTION char_bits(char) : bits;
FUNCTION int_bits(integer) : bits;
FUNCTION bits_int(bits) : integer;
```

```
FUNCTION convtoseq(char-array,integer,integer ) : char-sequence;

INFIX &(bits,bits) : bits;  [Bitwise And]

FUNCTION rw_io(port,bits,bits,integer ) : bits;
[Read or write to port at offset (:bits) at time (:integer ) with
 value (:bits) ]

REPLACE (b : char-array; s,f : integer )
seq(b!s)^convtoseq(b,bits_int( int_bits(s+1)&int_bits(63) ), f )
BY convtoseq(b,s,f );

REPLACE (i, o : char-sequence;b : char-array; s,f : integer; c : char )
i^seq(c) = o^convtoseq( update(b,f,c),s,
                        bits_int(int_bits(f+1)&int_bits(63))
                      )
BY i = o^convtoseq(b,s,f );

REPLACE (b : char-array; s,f : integer )
convtoseq(b,s,f )
BY empty_sequence : char-sequence IF s = f;

FUNCTION length(char-array) : integer;

PROCSPEC io(IN p : port IN offset : bits INOUT time : integer
            INOUT reg : bits)
    DERIVES time AS time+1,
            reg AS rw_io(p,offset,reg,time)
    PRE true
    POST (time = 'time + 1) AND (reg = rw_io('p,'offset,'reg,'time));

PROCSPEC printerr(IN rep : char-array INOUT i : integer)
    DERIVES i AS i+16,
    PRE length(rep) = 17
    POST i = 'i+16;

PROCSPEC get(OUT c : char IN in : bits INOUT i : integer
             INOUT tr : char-sequence)
    DERIVES c AS character(rw_io(port_b, data, in, i)),
             i AS i + 1,
             tr AS tr ^ seq(character(rw_io(port_b,data,in,i)))
    PRE true
    POST (c = character(rw_io(port_b, data, 'in, 'i)) ) AND
         (i = 'i + 1) AND
         (tr = 'tr ^ seq(character(rw_io(port_b,data,'in,'i))));

PROCSPEC put(IN c : char INOUT i : integer INOUT tr : char-sequence)
    DERIVES i AS i + 1,
            tr AS tr ^ seq(c)
    PRE true
    POST (i = 'i + 1) AND
         (tr = 'tr ^ seq(c));

MAINSPEC (OUT input_trace : char-sequence
          OUT output_trace : char-sequence
         )
PRE true
POST input_trace = output_trace;

MAIN

VAR buffer : char-array;
VAR start, finish : integer;
VAR status : bits;
VAR in : bits;
VAR i : integer;
```

```
VAR init_ctrl : bits;
VAR init_tran : bits;
VAR c : char;

input_trace := empty_sequence : char-sequence;
output_trace := empty_sequence : char-sequence;
io(port_b,rcvctl,i,init_ctrl);
io(port_b,xmtctl,i,init_tran);
start := 0;
finish := 0;

LOOP ASSERT input_trace = output_trace^convtoseq(buffer,start,finish);

io(port_b,stat_1,i,status);

EXIT WHEN bits_int(int_bits(finish+1) & int_bits(max-1)) = start
          OR input(status) = overflow;

          io(port_b,stat_0,i,status);

          IF input(status) = collect
          THEN
              get(c,in,i,input_trace);
              buffer := update(buffer,finish,c);
              finish := bits_int(int_bits(finish+1)&int_bits(max-1));
          ELSE
              io(port_b,stat_0,i,status);
              IF output(status) AND start /= finish
              THEN
                  c := buffer!start;
                  put(c,i,output_trace);
                  start := bits_int( int_bits(start+1) &
                                     int_bits(max-1) );
              ENDIF;
          ENDIF;
ENDLOOP;

IF bits_int(int_bits(finish+1) & int_bits(max-1)) = start
THEN
     printerr(buffer_overflow,i)
ELSE
    printerr(buffer_overflow,i)
ENDIF;

LOOP ASSERT input_trace = output_trace^convtoseq(buffer,start,finish);

EXIT WHEN start = finish
     io(port_b,stat_0,i,status);
     IF output(status)
     THEN
          c := buffer!start;
          put(c,i,output_trace);
          start := bits_int( int_bits(start+1) & int_bits(max-1) );
     ENDIF;
ENDLOOP;

ENDMAIN

PROC io;
reg := rw_io(p,offset,reg,time);
time := time + 1;
ENDPROC;

PROC get;
io(port_b,data,i,in);
c := character(in);
```

```
tr := tr^seq(c);
ENDPROC;

PROC put;
VAR d : bits;
d := char_bits(c);
io(port_b,data,i,d);
tr := tr^seq(c);
ENDPROC;

PROC printerr;
VAR index : integer;
VAR c : bits;
index := 0;
LOOP
ASSERT i = 'i + index;
EXIT WHEN index = 16;
          c := char_bits(rep!index);
          io(port_a,stat_0,i,c);
          index := index + 1;
ENDLOOP;
ENDPROC

FINISH
```

DOCUMENT CONTROL SHEET

Overall security classification of sheet ..... UNCLASSIFIED .....

(As far as possible this sheet should contain only unclassified information. If it is necessary to enter classified information, the box concerned must be marked to indicate the classification eg (R) (C) or (S))

1. DRIC Reference (if known)	2. Originator's Reference	3. Agency Reference	4. Report Security Classification
	MEMO 4090		UNCLASSIFIED

5. Originator's Code (if known)	6. Originator (Corporate Author) Name and Location
778400	RSRE, SAINT ANDREWS ROAD, MALVERN, WORCS WR14 3PS

5a. Sponsoring Agency's Code (if known)	6a. Sponsoring Agency (Contract Authority) Name and Location

7. Title

THE SOFTWARE REPEATER (AN EXERCISE IN Z SPECIFICATION)

7a. Title in Foreign Language (in the case of translations)

7b. Presented at (for conference papers) Title, place and date of conference

8. Author 1 Surname, initials	9(a) Author 2	9(b) Authors 3,4...	10. Date	pp. ref.
O'HALLORAN C			1987.10	21

11. Contract Number	12. Period	13. Project	14. Other Reference

15. Distribution statement

Descriptors (or keywords)

continue on separate piece of paper

Abstract

The specification of a software repeater in a language known as Z is presented. The specification is then refined into an implementation in the C programming language. The correctness of the implementation is shown by means of proof obligations and the program analysis tool Malpas. The C implementation was compiled and ran successfully on a 68010 processor.

END

DATE

FILMED

6-1988

DTIC