Retrenchment and Punctured Simulation

R. Banach[a], M. Poppleton[a,b]
[a]Computer Science Dept., Manchester University, Manchester, M13 9PL, U.K.
[b]School of Mathl. and Inf. Sciences, Coventry University, Coventry, CV1 5FB, U.K.
banach@cs.man.ac.uk , m.r.poppleton@coventry.ac.uk

Abstract: Some of the shortcomings of using refinement alone as the means of passing from high level simple models to actual detailed implementations are reviewed. Retrenchment is presented as a framework for ameliorating these. In retrenchment the relationship between an abstract operation and its concrete counterpart is mediated by extra predicates, allowing most particularly the description of non-refinement-like properties, and the mixing of I/O and state aspects in the passage between levels of abstraction. Stepwise simulation, the ability of simulator to mimic a sequence of execution steps of the simulatee, is introduced as the reference point for discussing the broader semantic issues surrounding retrenchment. Punctured simulation is introduced as a naturally occurring phenomenon implicit in the ability of retrenchments to describe of non-refinement-like properties; and it is shown to have relevance even in some refinements. Two special cases of retrenchment, simple simulable retrenchment and memoryless regular retrenchment are introduced. Both enjoy a unique domain property for large maximal punctured simulations, and the former has an easy stepwise simulation property too. A simple case study is presented. The B-Method and notation are used throughout the paper.
Keywords: Retrenchment, Simulation, Punctured Simulation, B-Method.

1 Introduction

In [Banach and Poppleton (1998)], we proposed retrenchment in order to liberalise the notion of refinement. The purpose of this was to enable more of the informal aspects of development to be captured within a formal framework. A retrenchment step from a more abstract to a more concrete level of abstraction admits strengthening of the precondition and weakening of the postcondition, and the mingling of state and I/O information between the levels of abstraction in question. This is accomplished by having two extra predicates per retrenched operation, the WITHIN and CONCEDES clauses[1], the former expressing the precondition strengthening, and the latter expressing the postcondition weakening. Most specifically, non-refinement-like behaviour can be entertained within the framework via the weakened postcondition. This in turn permits inconvenient low level detail of the true system from interfering with an idealised model at a high level of abstraction, leading to hopefully cleaner, more comprehensible development routes.

[Banach and Poppleton (1998)] was concerned with making the engineering case for the retrenchment notion. We considered it important to proceed in this more pragmatic manner at the outset, as the experience with refinement showed in hindsight that too early an emphasis on mathematical elegance could have detrimental consequences on the development activity for realistic problems. In this paper we focus on some of the theoretical properties of retrenchment. Specifically, since retrenchment allows the retrieve relation to be violated in the after-state of an execution step, and also because the more

1. As we have called them in the concrete syntax of the B-Method.

concrete level of abstraction may contain operations that do not appear at the abstract level, the simulation relation between the two levels, familiar from refinement, may break down completely. We study this situation to see what can be recovered from it. Specifically we introduce punctured simulation, a behavioural notion describing simulations “with holes in them”, and show that in special cases of retrenchment, “large maximal” punctured simulations possess a unique domain property, which is about the best one could hope for in the context of such a general notion.

In more detail, the rest of this paper is as follows. In section 2 we briefly recall why refinement is too strong a notion to conveniently encompass much of realistic system building activity. For a much fuller treatment of such motivational issues see [Banach and Poppleton (1998)]. In section 3 we present retrenchment and the proof obligations that give it semantic content. In section 4 we define stepwise simulation, the central concept for discussing wider semanic issues for retrenchment, and briefly comment on our choice of definition. In section 5 we introduce punctured simulation formally and give some basic properties, showing in fact that it has relevance even in the familiar world of B refinement. In section 6, simple simulable retrenchment is introduced, a special case of retrenchment with properties quite close to those of refinement. For this we get a straightforward stepwise simulation result, and we see that the unique domain property mentioned above holds. In section 7, another special case is introduced, memoryless regular retrenchment, and a similar unique domain property is established. Section 8 presents a small case study of a model power generation plant to illustrate punctured simulation. Section 9 concludes. The B-Method and Abstract Machine Notation [Abrial (1996a), Wordsworth (1996), Lano and Haughton (1996)] are used throughout the paper. By providing in particular a fixed syntactic framework for refinement, they provide a very convenient structure into which the ideas of retrenchment can be placed. This also helps in comparing refinement and retrenchment although a thorough treatment of that topic is beyond the scope of this paper.

2 The Trouble with Refinement

We will start with a small example taken from an idealised process management subsystem. Suppose we must defensively implement the operation of adding a new process *newproc* to an existing pool of processes *procs* . In the B notation we would write (a little loosely for brevity) at the abstract level:

```
MACHINE          $Proc\_Machine$
VARIABLES        $procs$
INVARIANT        $procs \subseteq \mathsf{PROCS}$
INITIALISATION   $procs := \varnothing$
OPERATIONS
    $AddProc\ (\ newproc\ )\ \triangleq$
        IF
            $newproc \in \mathsf{PROCS}$
        THEN
            $procs := procs \cup \{\ newproc\ \}$
        END
END
```

We might want to refine this to a situation in which the set *procs* was represented by an injective sequence of the constituent processes, *procsseq* . For pragmatic reasons *proc-*

sseq would be limited in size to 15 elements say. Set union becomes represented by appending sequences. So the resulting refinement would turn out as follows:

```
REFINEMENT        Proc_Machine_R
REFINES           Proc_Machine
VARIABLES         procsseq
INVARIANT         $procsseq \in \mathrm{iseq}(\ \mathsf{PROCS}\ ) \wedge \mathrm{size}(\ procsseq\ ) \leq 15 \wedge$
                  $procs = \mathrm{rng}(\ procsseq\ )$
INITIALISATION    $procsseq := <\ >$
OPERATIONS
    $AddProc\ (\ newproc\ )\ \triangleq$
        IF
            $newproc \in \mathsf{PROCS} \wedge newproc \notin \mathrm{rng}(\ procsseq\ ) \wedge$
                $\mathrm{size}(\ procsseq\ ) < 15$
        THEN
            $procsseq := procsseq \frown [\ newproc\ ]$
        END
END
```

Unfortunately this is not a refinement in the technical sense. To see this let us examine the proof obligation of refinement as generated in the B-Method:

$$
\begin{aligned}
&INV_A \wedge INV_C \wedge \mathsf{trm}(AddProc_A) \\
&\qquad \Rightarrow\ \mathsf{trm}(AddProc_C) \wedge [AddProc_C] \neg\, [AddProc_A] \neg\, INV_C
\end{aligned}
$$

Here INV_A and INV_C are the abstract and concrete invariants respectively (and the concrete invariant contains the retrieve relation $procs = \mathrm{rng}(\ procsseq\ )$). Furthermore $\mathsf{trm}(AddProc_A)$ and $\mathsf{trm}(AddProc_C)$ are the abstract and concrete termination predicates (normally called preconditions in non-B parlance). And $[AddProc_C] \neg\, [AddProc_A] \neg\, INV_C$ asserts that for every step that the concrete operation $AddProc_C$ makes, there is a step that the abstract operation $AddProc_A$ can make that establishes the truth of the concrete invariant INV_C , in the abstract and concrete after-states.

Consider what this says when $|\ procs\ | = 15$ and $newproc \notin procs$. We have to show two things. Firstly that $\mathsf{trm}(AddProc_C)$, which is just true , can be derived from the hypotheses. But true holds anyway so there is nothing to prove here. However when we secondly check that for every concrete step we have to have an abstract step that emulates it, we run into trouble since as well as other things we have to prove:

$$
\begin{aligned}
&procs \subseteq \mathsf{PROCS} \wedge \\
&procsseq \in \mathrm{iseq}(\ \mathsf{PROCS}\ ) \wedge \mathrm{size}(\ procsseq\ ) \leq 15 \wedge procs = \mathrm{rng}(\ procsseq\ ) \wedge \mathit{true} \\
&\qquad \Rightarrow \\
&\quad newproc \in \mathsf{PROCS} \wedge newproc \notin procs \wedge \mathrm{size}(\ procsseq\ ) = 15 \\
&\qquad\quad \Rightarrow \\
&\qquad [\ \mathit{skip}\ ] \neg\, [\ procs := procs \cup \{\ newproc\ \}\ ] \neg\, (\ procs = \mathrm{rng}(\ procsseq\ )\ )
\end{aligned}
$$

This is evidently false since in the after-states *procs* will have an extra element compared to *procsseq* . To get a genuine refinement we would have to push the concrete level finiteness requirement up to the abstract level.

But this is a bad idea we claim. It acts only to obscure the simplicity and clarity of the abstract system as we originally wished to conceive it. This in turn weakens the usefulness of the abstract level by narrowing the gap between abstract and concrete levels.

This leads to a shallower development route, capturing less of the design activity within the formal framework.

One technical device that leads to a way out of the formal conundrum just identified, and one that is much favoured in the formal development community, is to amplify the key statement in the abstract operation to:

$procs := procs \cup \{ newproc \}$ [ ] *skip*

In this particular case, *skip* , ([ ] is the choice combinator in B), would do just what the concrete operation does in the (null) ELSE branch of the conditional, so we could rescue the situation and recover a refinement. More generally though, we couldn't just casually insert "[ ] *skip*" and leave it at that. We would need to write *skip* alone as the body of the abstract operation, and have a suitably trivial retrieve relation, so that we could rely on the fact that with a trivial retrieve relation, *any* terminating operation (that preserves it) refines *skip* . Obviously in such a situation the *skip* says nothing at all about the relationship between the "true" abstract and concrete levels, except to signal loud and clear that whatever it is, it is certainly *not* a refinement relationship. Therefore as a technique for capturing design decisions in the engineering of real systems in the manner illustrated (as opposed to its proper role as an identity for sequential composition in the calculus of generalised substitutions), we consider *skip* harmful.

Returning to our putative refinement above, when we are indeed in the $| procs | = 15$ and $newproc \notin procs$ situation, the concrete system simply does nothing. Would this be appropriate in reality? Of course not. We would want the operation to at least inform its caller that it was unable to fulfil the normal demand, and that it was taking an exceptional action. For this we would need a change of signature, eg.

$res \longleftarrow AddProc (newproc)$

The *res* output would indicate success or failure regarding the normal functioning of the operation. Evidently changing the signature is not within the province of refinement as it is conventionally understood, so incorporating a more concrete signature into the abstract level, particularly when its role in the ideal abstract model would be spurious, represents at best another unnecessary distraction from the simplicity of the abstract system, or at worst a further undesirable narrowing of the abstraction gap between abstract and concrete systems.

We see that there are drawbacks to using conventional refinement as the sole means of going from an abstract description of a system to a concrete one. Evidently the issues we have raised are rather trivial in the case of a small illustrative example, but it is not hard to imagine that in realistic situations, the level of detail that needs to be brought up to the abstract system in order for there to be a refinement between abstract and concrete worlds is so great that it overwhelms the underlying concepts of the abstract model. The supposedly abstract model then becomes little more than a restatement of the concrete model in another language. Such a thing is not terrible in itself of course — the different perspectives of the two descriptions can each illuminate the other — but the valuable goal of setting out how the real system definition is arrived at from the designer's original simplified ideas is lost. In this manner we briefly promote the introduction of a more liberal notion than refinement which we intend to bridge that gap. Such motivational issues are discussed much more extensively in [Banach and Poppleton (1998)].

3 Retrenchment

The top level system construct in B is the MACHINE which expresses the abstract model of the system being built. A MACHINE is refined via the top level REFINEMENT construct which roughly speaking contains similar components to a MACHINE , with the exception that its INVARIANT clause contains the retrieve relation which links abstract and concrete variables. The abstract variables in question are those to be found in the top level construct being refined, this being named in the REFINES clause of the REFINEMENT , and refers to either a MACHINE or a preceding REFINEMENT .

Retrenchment may be viewed as a variation on refinement. For flexibility we will allow either a MACHINE or a REFINEMENT to be retrenched, since the result of the development step is essentially the specification of a fresh problem. Here is the syntax of a RETRENCHMENT .

```
MACHINE          M ( a )          MACHINE          N ( b )
                                  RETRENCHES       M
VARIABLES        u                VARIABLES        v
INVARIANT        I ( u )          INVARIANT        J ( v )
                                  RETRIEVES        G ( u , v )
INITIALISATION   X ( u )          INITIALISATION   Y ( v )
OPERATIONS                        OPERATIONS
    o ⟵ OpName ( i ) ≙               p ⟵ OpName ( j ) ≙
        S ( u , i , o )                   BEGIN
END                                           T ( v , j , p )
                                          LVAR
                                              A
                                          WITHIN
                                              P ( i , j , u , v , A )
                                          CONCEDES
                                              C ( u , v , o , p , A )
                                          END
                                  END                              (3.1)
```

Thus we have a MACHINE $M(a)$, with typical operation given by the signature $o \longleftarrow OpName(i)$, the body of *OpName* being a generalised substitution $S(u, i, o)$. On the right we have a MACHINE $N(b)$, together with the RETRENCHES M clause and retrieve relation RETRIEVES $G(u, v)$. (We insist that the retrieve relation and invariant are given separately in a retrenchment in order to separate concerns.) The body of each operation $p \longleftarrow OpName(j)$ is now a ramified generalised substitution, that is to say a generalised substitution $T(v, j, p)$, together with its ramification, the LVAR , WITHIN , CONCEDES clauses. Each *OpName* of M must appear ramified within N , but we allow additional operations in N . (These could be specified trivially by *skip*s in M but we consider such uses of *skip* at least undesirable not to say harmful.) If we strip away the RETRENCHES clause, the RETRIEVES clause, and the ramifications, we end up with just a normal B MACHINE .

Speaking informally, the ramification of an operation allows us to describe how the concrete operation fails to refine its abstract counterpart. The LVAR A clause, which is optional, allows us to introduce logical variables A that remember before-values of variables and inputs, so that we may refer to them in the context of the after-state if necessary. The scope of the LVAR declaration is the WITHIN and CONCEDES clauses. The job of the WITHIN clause is to describe nontrivial relationships between the abstract and concrete before-values of variables u and v, and abstract and concrete input values i and j, and to define values for the logical variables A. It is used to strengthen the precondition as we will see below, and thus may contain any strengthening of the retrieve relation required in the retrenchment step. The purpose of the CONCEDES clause is to provide a similar storehouse for information concerning the after-state. In particular, the CONCEDES clause involves abstract and concrete variables, abstract and concrete outputs, and the logical variables A, and weakens the retrieve relation in the after-state allowing non-refinement like behaviour to be expressed. These ideas find more precise expression in the proof obligations which we now list.

We take for granted an environment where all the necessary identifiers are defined in terms of basic types. Then there are the conventional machine POs for M and N. Firstly the initialisation POs:

$$[\ X(u)\ ]\ I(u)$$

$$[\ Y(v)\ ]\ J(v)$$

and then the invariant preservation POs:

$$I(u) \wedge \mathsf{trm}(S(u, i, o)) \Rightarrow [\ S(u, i, o)\ ]\ I(u)$$

$$J(v) \wedge \mathsf{trm}(T(v, j, p)) \Rightarrow [\ T(v, j, p)\ ]\ J(v)$$

Next we have the sharp retrenchment initialisation PO which is just like the corresponding refinement initialisation PO:

$$[\ Y(v)\ ] \neg [\ X(u)\ ] \neg\ G(u, v)$$

and finally we have the retrenchment PO for operations which reads:

$$\begin{aligned}
&(I(u) \wedge G(u, v) \wedge J(v)) \wedge (\mathsf{trm}(T(v, j, p)) \wedge P(i, j, u, v, A))\\
&\qquad\Rightarrow\\
&\quad \mathsf{trm}(S(u, i, o)) \wedge [\ T(v, j, p)\ ] \neg [\ S(u, i, o)\ ] \neg\\
&\qquad\qquad (G(u, v) \vee C(u, v, o, p, A))
\end{aligned} \tag{3.2}$$

The antecedents of this PO contain the invariants and retrieve relation $(I(u) \wedge G(u, v) \wedge J(v))$, and moreover the $\mathsf{trm}(T(v, j, p))$ clause is strengthened by the WITHIN clause $P(i, j, u, v, A)$. These assumptions allow us to infer the abstract $\mathsf{trm}(S(u, i, o))$ clause, and also that the familiar "$[\ T(v, j, p)\ ] \neg [\ S(u, i, o)\ ] \neg$" structure establishes for the after-states either the retrieve relation, or the CONCEDES clause, the latter permitting reference to outputs as well as after-states, and to before-data as remembered in the A variables. The justification of the precise form of this PO was discussed at length in [Banach and Poppleton (1998)]. Beyond those considerations which continue to apply here, we will say that it leads to a clean notion of stepwise simulation which we introduce in the next section.

We give an example of retrenchment by redoing our failed refinement above within the new framework. Given our preceding comments about the appropriateness of a refinement step between the desired two levels of abstraction, the following seems a more natural development step.

```
MACHINE           $Proc\_Machine\_Ret$
RETRENCHES        $Proc\_Machine$
SETS              $\mathsf{RESPONSES} = \{added , notadded\}$
VARIABLES         $procsseq$
INVARIANT         $procsseq \in \mathrm{iseq}( \mathsf{PROCS} ) \wedge \mathrm{size}( procsseq ) \leq 15$
RETRIEVES         $procs = \mathrm{rng}( procsseq )$
INITIALISATION    $procsseq := < >$
OPERATIONS
    $res \longleftarrow AddProc ( newproc ) \triangleq$
        BEGIN
            IF
                $newproc \in \mathsf{PROCS} \wedge newproc \notin \mathrm{rng}( procsseq ) \wedge$
                $\mathrm{size}( procsseq ) < 15$
            THEN
                $procsseq := procsseq \frown [ newproc ] \parallel$
                $res := added$
            ELSE
                $res := notadded$
            END
        LVAR
            $LL , PP$
        WITHIN
            $LL = \mathrm{size}( procsseq ) \wedge PP = procsseq$
        CONCEDES
            $(LL = 15 \wedge procsseq = PP \wedge res = notadded) \Leftrightarrow \neg(res = added)$
        END
END
```

Note how the CONCEDES clause allows us to express what happens when the refinement relationship breaks down, as well as allowing us to say something about the output *res* , which did not exist at the abstract level. Note also how the occurrences of *procsseq* in the WITHIN clause refer to the before-value while the occurrence in the CONCEDES clause refers to the after-value, necessitating the use of *PP* .

4 Stepwise Simulation

The generality of the relationships we are prepared to admit as retrenchments, provokes the question of what is the fundamental semantic anchor point for the retrenchment notion. Our stance is, that this lies in the kinds of simulation that retrenchments support. The basic simulation notion that we focus on is that of stepwise simulation, by which we mean the simulation of a sequence of steps of the simulatee by an equal length sequence of steps of the simulator. This choice is dictated by various technical details regarding different notions of simulation that it is beyond the scope of this paper to discuss. We write a step of a machine such as M of (3.1) in the form:

$$u \text{ -}(i, m, o)\text{-}\rangle\ u'$$

where u and u' are the before and after states, m is the name of the operation (where it can help, we write S , the body of m , instead of m itself), and i and o are the input and output of m . This signifies that (u, i) satisfy $\mathsf{trm}(S)$, and that (u, i, u', o) satisfy the before-after predicate of m (which in B parlance says that (u', o) is a possible result from

(u, i)). When discussing properties of sequences of steps, $\mathsf{last}(\mathcal{T})$ will denote the index of the last state mentioned in $\mathcal{T}$, and $r \in \mathrm{dom}^\bullet(\mathcal{T})$ will mean $r \in [0 \ldots \mathsf{last}(\mathcal{T}) - 1]$ if $\mathcal{T}$ is finite, and $r \in \mathsf{NAT}$ otherwise. Similarly for sequences of any type. In general we need to distinguish Ops^M, the operation names at the abstract level, from Ops^N the operation names at the concrete level, where $\mathsf{Ops}^M \subseteq \mathsf{Ops}^N$.

Definition 4.1 Let (3.1) be a retrenchment. Suppose that $\mathcal{T} \equiv [\ v_0 \text{ -}(j_0, m_0, p_1)\text{-›}\ v_1 \text{ -}(j_1, m_1, p_2)\text{-›}\ v_2 \ldots\ ]$ is an execution sequence of N, and that $\mathcal{S} \equiv [\ u_0 \text{ -}(i_0, m_0, o_1)\text{-›}\ u_1 \text{ -}(i_1, m_1, o_2)\text{-›}\ u_2 \ldots\ ]$ is an execution sequence of M, where $[\ m_0, m_1, \ldots\ ]$ is a sequence over Ops^M. Then $\mathcal{S}$ is a stepwise simulation of $\mathcal{T}$ iff $G(u_0, v_0)$ holds, and for all $r \in \mathrm{dom}^\bullet(\mathcal{T})$ there is an A_r such that:

$$\begin{aligned} &G(u_r, v_r) \wedge P_{m_r}(i_r, j_r, u_r, v_r, A_r) \wedge \\ &\qquad (G(u_{r+1}, v_{r+1}) \vee C_{m_r}(u_{r+1}, v_{r+1}, o_{r+1}, p_{r+1}, A_r)) \end{aligned} \tag{4.1}$$

Note that it is the concrete sequence that we are taking as the simulatee, and the abstract one that is the simulator. This is definitely the more appropriate perspective for retrenchment, made more so by the considerations of the next and subsequent sections. The picture for conventional refinement is less clear: who should be the simulator and who the simulatee can be argued both ways, but again we do not have space to go into details.

5 Punctured Simulation

The last section defined stepwise simulation as the ability to simulate the whole of a sequence. But the CONCEDES clause in a retrenchment makes it suitable for cases where simulation simply breaks down as a result of incompatibility between abstract and concrete models, even though the retrenchment operation PO remains valid. We look at this more closely now.

Let us consider a simulation of a concrete execution which has just broken down. We have succesfully simulated some steps, arriving at states u_r and v_r, and having established $(G(u_r, v_r) \vee C(u_r, v_r, o_r, p_r, A_{r-1}))$. However we are unable to establish $(G(u_r, v_r) \wedge P_{m_r}(u_r, v_r, i_r, j_r, A_r))$ for the next concrete step $v_r \text{ -}(j_r, m_r, p_{r+1})\text{-›}\ v_{r+1}$. Common sense and some algebra shows that this could be because:

(I) $m_r \notin \mathsf{Ops}^M$, or
(II) $m_r \in \mathsf{Ops}^M$, but $P_{m_r}(u_r, v_r, i_r, j_r, A_r)$ does not hold, or
(III) $m_r \in \mathsf{Ops}^M$, $C(u_r, v_r, o_r, p_r, A_{r-1})$ holds but $G(u_r, v_r)$ does not hold.

Of course, (II) and (III) may hold simultaneously. Furthermore, after one or more non-simulable steps, non-simulability may be contributed to by the failure of the abstract invariant, i.e.

(IV) $\neg\ (\exists\ u_r \bullet I(u_r) \wedge G(u_r, v_r))$

These conditions (I)–(IV) delineate the ways in which retrenchment can describe the failure of simulability. Any or all of them may explain non-simulability at any point of a concrete execution $\mathcal{T}$, which we summarise as the failure of condition (S) thus:

(S) $(\exists\ u_r, i_r, A_r \bullet I(u_r) \wedge G(u_r, v_r) \wedge P_{m_r}(u_r, v_r, i_r, j_r, A_r))$

(The remaining condition for simulability, the trm condition, we take care of via $\mathcal{T}$.)

Definition 5.1 Let (3.1) be a retrenchment and suppose that v is a concrete state of N . If $(\exists\, u \bullet I(u) \wedge G(u, v) \wedge J(v))$ holds, we say that v is 0-recoverable. We say that v is s-recoverable iff at least s concrete steps are required to reach a 0-recoverable state from v . If there is no such s we call v irrecoverable.

The possibility of needing to recover from a faulty situation is a familiar one in real applications. Often a single *Reset* action is all that is required and 1-recoverability suffices. Then again there are systems (for example complex high power electrical networks) which require a much more carefully staged recovery, consisting of a larger number of steps. And just because a concrete state is s-recoverable does not mean that any particular concrete execution sequence indeed recovers from it in precisely s steps; the recovery may itself partially fail, requiring a more protracted route back to simulability. Equally one can imagine that at a sufficiently high level of abstraction, the possibility of failure and the necessity of recovery might be out of scope for the model. Retrenchment allows for these possibilities via the following very general concept.

Definition 5.2 Let (3.1) be a retrenchment and suppose that $\mathcal{T} \equiv [\, v_0 \text{ -}(j_0, m_0, p_1)\text{-›}\ v_1 \text{ -}(j_1, m_1, p_2)\text{-›}\ v_2 \ldots]$ is a concrete execution sequence of N . A punctured (stepwise) simulation $\mathcal{S}$ of $\mathcal{T}$ is a subset dom($\mathcal{S}$) of $\text{dom}^{\bullet}(\mathcal{T})$, and a map $\phi_{\mathcal{S}}$ from dom($\mathcal{S}$) to steps of M , $\phi_{\mathcal{S}}(r) = u_r \text{ -}(i_r, m_r, o_{r+1})\text{-›}\ u_{r+1}$, such that:

(1) For each $r \in$ dom($\mathcal{S}$) , $I(u_r)$ holds.

(2) For each $r \in$ dom($\mathcal{S}$) , for an appropriate A_r , (4.1) holds for the steps $u_r \text{ -}(i_r, m_r, o_{r+1})\text{-›}\ u_{r+1}$ and $v_r \text{ -}(j_r, m_r, p_{r+1})\text{-›}\ v_{r+1}$ indexed by r .

(3) For each adjacent pair $\{r, r+1\} \in$ dom($\mathcal{S}$) , the steps indexed by r and $r+1$ agree on the value of u_{r+1} .

Fig. 1 illustrates the general idea.

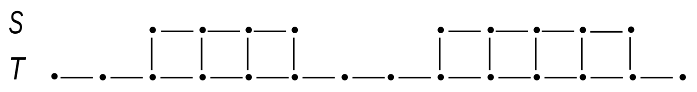

Fig. 1. A punctured simulation.

Definition 5.3 Let (3.1) be a retrenchment. Suppose that $\mathcal{T} \equiv [\, v_0 \text{ -}(j_0, m_0, p_1)\text{-›}\ v_1 \text{ -}(j_1, m_1, p_2)\text{-›}\ v_2 \ldots]$ is a concrete execution sequence of N , and let $\mathcal{S}$ be a punctured simulation of $\mathcal{T}$. A portion of dom($\mathcal{S}$) is a maximal interval of consecutive natural numbers in dom($\mathcal{S}$) . A portion π of $\mathcal{S}$ is a maximal run of consecutive execution steps of $\mathcal{S}$, the image under $\phi_{\mathcal{S}}$ of a portion, dom(π) , of dom($\mathcal{S}$) . Runs of consecutive execution steps of $\mathcal{T}$ indexed by maximal intervals of ($\text{dom}^{\bullet}(\mathcal{T})$ – dom($\mathcal{S}$)) are called exceptions.

Fig. 1 shows two portions of $\mathcal{S}$ and three exceptions. Normally in a punctured simulation we would expect the exceptions to consist exclusively of non-simulable parts of the concrete execution sequence, but there is nothing in Definition 5.2 to ensure this. This is addressed in the following definition.

Definition 5.4 Let (3.1) be a retrenchment. Suppose that $\mathsf{T} \equiv [\, v_0 \text{ -}(j_0, m_0, p_0)\text{-› } v_1 \text{ -}(j_1, m_1, p_1)\text{-› } v_2 \ldots \,]$ is a concrete execution sequence of N , and let $\mathsf{S}^{\sim}$ be a punctured simulation of T . S is a (punctured) subsimulation of $\mathsf{S}^{\sim}$ iff it is a punctured simulation of T in its own right, $\mathrm{dom}(\mathsf{S}) \subseteq \mathrm{dom}(\mathsf{S}^{\sim})$, and both simulations agree as maps on the common parts of their domains. S is a proper subsimulation of $\mathsf{S}^{\sim}$ iff $\mathrm{dom}(\mathsf{S}) \neq \mathrm{dom}(\mathsf{S}^{\sim})$. S is a subsimulation of $\mathsf{S}^{\sim}$ iff $\mathsf{S}^{\sim}$ is a supersimulation of S . A punctured simulation (of T) is maximal iff it is not a proper subsimulation of some other punctured simulation of T . A portion π of a punctured simulation S of T is maximal iff there is no proper supersimulation $\mathsf{S}^{\sim}$ of S containing a portion $\pi^{\sim}$ properly containing π . A portion π of a punctured simulation S of T is called large iff there is no punctured simulation $\mathsf{S}^{\sim}$ of T containing a portion $\pi^{\sim}$ such that $\mathrm{dom}(\pi) \subsetneq \mathrm{dom}(\pi^{\sim})$. A maximal punctured simulation whose portions are all large is called a large maximal punctured simulation.

Obviously a stepwise simulation is a maximal punctured simulation which is total on its domain (and so contains just one large portion). The following counterexample shows that the concept of punctured simulation is relevant even in the more familiar world of B refinement.

Counterexample 5.5 Consider the B refinement:

```
MACHINE          M                    REFINEMENT       N
                                      REFINES          M
VARIABLES        uu                   VARIABLES        vv
INVARIANT        uu : NAT ∧           INVARIANT        vv : NAT ∧
                 uu ∈ {1, 2}                           vv = 0
INITIALISATION   uu := 1              INITIALISATION   vv := 0
OPERATIONS                            OPERATIONS
    Op ≙                                  Op ≙ skip
        PRE uu = 1
        THEN uu := 2
        END
END                                   END
```

Fig. 2 shows that either the first or the second abstract step alone forms a maximal punctured simulation of the two step concrete execution sequence beneath, because the two abstract steps are unable to agree on a common intermediate abstract state. This shows that maximal punctured simulations are neither unique nor do they necessarily have a unique domain, even under apparently favourable circumstances. The counterexample also indicates that punctured simulation is not simply an alternative means of dealing with situations that need stuttering (i.e. the interspersing of arbitrary finite runs of *skip*s into abstract execution sequences — see eg. [Abadi and Lamport (1991)]), because in a punctured simulation the final abstract state of one portion need not coincide with the first abstract state of the next portion: in particular two consecutive portions must have an exception of at least one concrete step between them.

The main point of the rest of this paper, is to show that under suitable circumstances, a unique domain property can be proved for large maximal punctured simulations. We start with a very general construction.

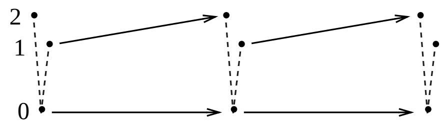

Fig. 2. Simulation steps not forming a stepwise simulation.

Theorem 5.6 Let (3.1) describe a retrenchment and let $\mathcal{T} \equiv [\ v_0 \text{ -}(j_0, m_0, p_1)\text{-›}\ v_1 \text{ -}(j_1, m_1, p_2)\text{-›}\ v_2 \ldots\ ]$ be an execution sequence of *N* . Then $\mathcal{T}$ has a maximal punctured simulation $\mathcal{S} \equiv [\ u_0 \text{ -}(i_0, m_0, o_1)\text{-›}\ u_1 \text{ -}(i_1, m_1, o_2)\text{-›}\ u_2 \ldots\ ]$.

Proof. Consider the steps of $\mathcal{T}$. Some of them may be simulable. Any simulation of such a simulable step forms a portion of a punctured simulation by itself. Candidate simulations of adjacent simulable steps can be combined into bigger portions if they agree on the abstract states at their interfaces. These portions are partially ordered by the supersimulation relation. Since $\text{dom}^\bullet(\mathcal{T})$ provides an upper bound for the domain of any such portion, maximal portions exist by Zorn's Lemma. Let the set of maximal portions be Π_{Max} . We use Π_{Max} to construct $\mathcal{S}$ as follows.

```
1.  $r := 0$
2.  REPEAT  steps 3, 4, 5
3.      WHILE  there is no element of $\Pi_{Max}$ starting at index $r$
        DO  $r := r + 1$
        END
4.      CHOOSE  an element $\pi$ of $\Pi_{Max}$ starting at $r$ and include it in $\mathcal{S}$
5.      $r := r + \text{size}(\pi) + 1$
6.  UNTIL  all indices of $\mathcal{T}$ have been considered
```

Evidently $\mathcal{S}$ so constructed is maximal. ☺

The above result cannot be strengthened to assert largeness (without strengthening the hypotheses) because nothing prevents a scenario in which there are two maximal portions in Π_{Max} which overlap but such that neither is included in the other; but which nevertheless are such that either could be included in a maximal punctured simulation. We leave the reader to construct easy counterexamples based on the following situation.

Example 5.7

MACHINE	*M*	RETRENCHMENT	*N*
		RETRENCHES	*M*
VARIABLES	*uu*	VARIABLES	*vv*
INVARIANT	$uu : \text{NAT}_1$	INVARIANT	$vv : \text{NAT} \wedge$ $vv = 0$
INITIALISATION	$uu :\in \text{NAT}_1$	INITIALISATION	$vv := 0$
OPERATIONS		OPERATIONS	

```
    Op ≙ uu := uu + 1                    Op ≙
END                                           BEGIN
                                                  skip
                                              WITHIN
                                                  uu < 4
                                              CONCEDES
                                                  false
                                              END
                                         END
```

6 Simple Simulable Retrenchment

In this section we define a class of retrenchments for which a stepwise simulation result can be proved, and then show that this carries through to a unique domain property for large maximal punctured simulations.

Definition 6.1 For a retrenchment like (3.1), suppose the joint initialisation establishes:

$$(G(u_0, v_0) \wedge \bigwedge_{m \in \mathsf{Ops}^M} (\forall\, j_m\ \exists\, i_m, A_m \bullet P_m(i_m, j_m, u_0, v_0, A_m))) \tag{6.1}$$

and suppose that each Ops^M operation $n \equiv (T_n, A_n, P_n, C_n)$ of N satisfies the operation compatibility PO:

$$\begin{aligned} &G(u, v) \vee C_n(u, v, o, p, B) \\ &\quad \Rightarrow \\ &\qquad (G(u, v) \wedge \bigwedge_{m \in \mathsf{Ops}^M} (\forall\, j_m\ \exists\, i_m, A_m \bullet P_m(i_m, j_m, u, v, A_m))) \end{aligned} \tag{6.2}$$

then we say that the retrenchment is a simple simulable retrenchment.

The stepwise simulation property of simple simulable retrenchment is the following.

Theorem 6.2 Let (3.1) describe a simple simulable retrenchment where the set of abstract operation names is Ops^M. Let $\mathcal{T} \equiv [\, v_0 \text{-}(j_0, m_0, p_1)\text{-›}\, v_1 \text{-}(j_1, m_1, p_2)\text{-›}\, v_2 \ldots]$ be an execution sequence of N. Suppose that the sequence of invoked operation names $ms \equiv [\, m_0, m_1 \ldots]$ is an Ops^M sequence. Then there is a stepwise simulation $\mathcal{S} \equiv [\, u_0 \text{-}(i_0, m_0, o_1)\text{-›}\, u_1 \text{-}(i_1, m_1, o_2)\text{-›}\, u_2 \ldots]$ of $\mathcal{T}$.

Proof. This follows standard lines. Let $\mathcal{T} \equiv [\, v_0 \text{-}(j_0, m_0, p_1)\text{-›}\, v_1 \ldots]$ be an execution sequence of N. The $\mathrm{dom}(\mathcal{T}) = \{0\}$ case is trivial because of the retrenchment initialisation PO. Otherwise we go by induction on $\mathrm{dom}^\bullet(\mathcal{T})$.

For $r = 0$, we know that for the given v_0 and j_0 from $\mathcal{T}$, (6.1) holds. So for the m_0 from $\mathcal{T}$ we can find an i_0 such that $G(u_0, v_0) \wedge P_{m_0}(i_0, j_0, u_0, v_0, A_0)$ holds for suitable A_0. Now the initialisation POs for M and N yield $I(u_0)$ and $J(v_0)$. And because $v_0 \text{-}(j_0, m_0, p_1)\text{-›}\, v_1$ is a step of N, $\mathsf{trm}(T_0)$ holds for (v_0, j_0) where T_0 is the body of m_0 in N, so we have the antecedents of the retrenchment operation PO (3.2).

For the inductive step, suppose $\mathcal{S}$ has been constructed as far as the r'th step. Then we have $I(u_r)$, $J(v_r)$, $G(u_r, v_r)$, $\mathsf{trm}(T_r)$ for (v_r, j_r) (from the existence of the r+1'th step of $\mathcal{T}$), and $P_{m_r}(i_r, j_r, u_r, v_r, A_r)$ for suitable i_r and A_r. Applying the retrenchment operation PO (3.2), yields both $\mathsf{trm}(S_r)$ for (u_r, i_r) where S_r is the body of m_r in M, and thence a step of M, $u_r \text{-}(i_r, m_r, o_{r+1})\text{-›}\, u_{r+1}$ such that $G(u_{r+1}, v_{r+1}) \vee C_{m_r}(u_{r+1}, v_{r+1}, o_{r+1}, p_{r+1}, A_r)$ holds. Machine consistency for M and N yields $J(v_{r+1})$ and $J(v_{r+1})$, and from (6.2) we conclude that we can find an i_{r+1} and A_{r+1} such that $G(u_{r+1}, v_{r+1}) \wedge P_{m_{r+1}}(u_{r+1}, v_{r+1},$

o_{r+1}, p_{r+1}, A_{r+1}) holds. We get $\mathsf{trm}(T_{r+1})$ for (v_{r+1}, j_{r+1}) from the existence of the r+2'th step of T which reestablishes the inductive hypothesis. ☺

The above result has a direct counterpart in terms of the automata-theoretic notion of simulation but we do not have space to explore this here. We move directly on to the punctured simulation result.

Theorem 6.3 Let (3.1) describe a simple simulable retrenchment and let $\mathsf{T} \equiv [\ v_0$ -(j_0, m_0, p_0)-› v_1 -(j_1, m_1, p_1)-› v_2 …] be an execution sequence of N . Then T has a large maximal punctured simulation $\mathsf{S} \equiv [\ u_0$ -(i_0, m_0, o_1)-› u_1 -(i_1, m_1, o_2)-› u_2 …] ; and if $\mathsf{S}^{\sim}$ is any large maximal punctured simulation of T , then dom($\mathsf{S}^{\sim}$) = dom(S) .

Proof. We construct a suitable S in the following rather obvious manner, and show that it has the required properties.

```
1.  CHOOSE  a $u_0$ so that (6.1) holds

2.  $r := 0$

3.  REPEAT  steps 4, 5, 6

4.      WHILE  the next step $v_r$ -($j_r$, $m_r$, $p_{r+1}$)-› $v_{r+1}$ of $\mathsf{T}$ is an $\mathsf{Ops}^M$ step
        DO  construct a simulating step $u_r$ -($i_r$, $m_r$, $o_{r+1}$)-› $u_{r+1}$ of $\mathsf{S}$ as in Theorem 6.2
              $r := r + 1$
        END

5.      WHILE  the next step $v_r$ -($j_r$, $m_r$, $p_{r+1}$)-› $v_{r+1}$ of $\mathsf{T}$ is not an $\mathsf{Ops}^M$ step
                      OR the simulation condition (S) fails for $v_{r+1}$
        DO  $r := r + 1$
        END

6.      CHOOSE  a $u_r$ so that the simulation condition (S) holds for $v_r$

7.  UNTIL  all steps of $\mathsf{T}$ have been considered
```

The simulating abstract steps so constructed form S . Clearly this S is a punctured simulation. To see it has the required properties we argue as follows. Each portion of S is maximal since it starts as early as possible by steps 1 and 5, and finishes as late as possible by the WHILE test of step 4. So S is maximal. Also each portion of S is large because an easy induction over the structure of any exception of the punctured simulation demonstrates that no step of such an exception can be simulable. So S is large maximal. Furthermore any other large maximal punctured simulation must have domain within dom(S) , and so its domain must equal dom(S) . ☺

Thus as advertised, we see that the properties of a simple simulable retrenchment ensure the unique domain property for any large maximal punctured simulation of a concrete execution.

7 Memoryless Regular Retrenchment

In this section we define another class of retrenchments with good properties regarding large maximal punctured simulations in particular.

Definition 7.1 Let (3.1) describe a retrenchment. We say the retrenchment is memoryless iff no ramification of an operation of M contains an LVAR clause.

So a memoryless retrenchment can be described using only individual states (and the outputs and inputs pertaining to them), without reference to properties of before-after pairs.

Definition 7.2 Let (3.1) describe a memoryless retrenchment where the set of abstract operation names is Ops^M . We say that M is regular (with respect to the retrenchment) iff the following holds for all $m \in \mathsf{Ops}^M$:

$$\begin{aligned} &G(u, v) \wedge P(i, j, u, v) \wedge \mathsf{stp}(T)(v, j, v', p) \wedge (G(u', v') \vee C(u', v', o, p)) \\ &\qquad \Rightarrow \\ &\qquad \mathsf{stp}(S)(u, i, u', o) \end{aligned} \tag{7.1}$$

where eg. $\mathsf{stp}(S)(u, i, u', o)$ is a predicate that says that u -(i, S, o)-› u' is a step of generalised substitution S , as described at the beginning of section 4.

Now memoryless regularity is not in itself enough to always guarantee a stepwise simulation. However it is enough for a good large maximal punctured simulation result.

Theorem 7.3 Let (3.1) describe a memoryless retrenchment with M regular, and let $\mathcal{T} \equiv [\ v_0$ -(j_0, m_0, p_0)-› v_1 -(j_1, m_1, p_1)-› v_2 …] be an execution sequence of N . Then $\mathcal{T}$ has a large maximal punctured simulation $\mathcal{S} \equiv [\ u_0$ -(i_0, m_0, o_1)-› u_1 -(i_1, m_1, o_2)-› u_2 …] ; and if $\mathcal{S}^{\sim}$ is any large maximal punctured simulation of $\mathcal{T}$, then $\mathrm{dom}(\mathcal{S}^{\sim}) = \mathrm{dom}(\mathcal{S})$.

Proof. We construct a suitable $\mathcal{S}$, and show that it has the required properties. Let $\mathrm{dom}(\mathcal{S})$ be the subset of $\mathrm{dom}^{\bullet}(\mathcal{T})$ for which an abstract state can be found to act as before-state in a simulation step; more precisely:

$$\mathrm{dom}(\mathcal{S}) \ = \ \{r \in \mathrm{dom}^{\bullet}(\mathcal{T}) \mid m_r \in \mathsf{Ops}^M \wedge (\exists\, u, i \bullet I(u_r) \wedge G(u, v_r) \wedge P_{m_r}(i, j_r, u, v_r))\}$$

Since $\mathcal{T}$ consists of execution steps, for every $r \in \mathrm{dom}(\mathcal{S})$, we can build a simulating abstract step for the corresponding step v_r -(j_r, m_r, p_{r+1})-› v_{r+1} of $\mathcal{T}$ because the retrenchment operation PO is satisfied (specifically, the PO gives us the needed $\mathsf{trm}(S_r)$). In particular

$$\mathrm{dom}^{++}(\mathcal{S}) \ = \ \{r + 1 \mid r \in \mathrm{dom}(\mathcal{S})\}$$

is the set of indices of after-states of such steps. Let

$$\begin{aligned} &Befs \ = \ \mathrm{dom}(\mathcal{S}) - \mathrm{dom}^{++}(\mathcal{S}) \\ &Mids \ = \ \mathrm{dom}(\mathcal{S}) \cap \mathrm{dom}^{++}(\mathcal{S}) \\ &Afts \ = \ \mathrm{dom}^{++}(\mathcal{S}) - \mathrm{dom}(\mathcal{S}) \end{aligned}$$

For each $r \in Befs \cup Mids$ choose u_r , i_r such that $(I(u_r) \wedge G(u_r, v_r) \wedge P_{m_r}(i_r, j_r, u_r, v_r))$ holds. For each $r \in Afts$ choose u_r , o_r such that $(I(u_r) \wedge (G(u_r, v_r) \vee C_{m_{r-1}}(u_r, v_r, o_r, p_r)))$ holds. For $Befs \cup Mids$ this is possible by assumption; for $Afts$ this is possible by abstract machine consistency and the retrenchment operation PO. Now for each adjacent pair r , r+1 in $\mathrm{dom}(\mathcal{S}) \cup \mathrm{dom}^{++}(\mathcal{S})$, we choose a step u_r -(i_r, m_r, o_{r+1})-› u_{r+1} . This in turn is possible by the memoryless regularity of M with respect to the retrenchment, (7.1). This gives us a punctured simulation $\mathcal{S}$ with domain $\mathrm{dom}(\mathcal{S})$. The maximality of $\mathcal{S}$ is clear. Large maximality follows because memoryless regularity prevents the existence of overlapping or abutting maximal portions which are not fusible (or not fusible after adjustment of a step or two). We are done. ☺

8 A Power Generation Case Study

We sketch a toy power generation example to illustrate some of the concepts above. A power generation system specification might start with something like the following:

```
MACHINE          GenPower
CONSTANTS        Margin
OPERATIONS
    outpower ⟵ RunGenPower ( powerreq ) ≙
        outpower : ( powerreq – Margin < outpower ∧
                     powerreq + Margin > outpower )
END
```

This is a machine that simply states that when the environment demands power to the tune of *powerreq* , then in the time it takes to execute the *RunGenPower* operation, the generation facility will deliver power within *Margin* units of *powerreq* . This machine lacks vital information about many aspects of a real system (eg. an upper bound for the deliverable power). Nevertheless, embellished with appropriate timing and other data, a similar specification could serve as the definition of “normal service”, required to be available for at least a certain proportion of the time, say 99%.

The “real” specification will consist of a number of generating facilities managed by an overall control strategy. An individual generator might be described by the following state machine. We assume for simplicity that a generator offers little flexibility in its actual power output; it is either running and essentially delivering its full output, or not. Furthermore, a generator cannot go from cold to full power instantaneouly, it needs to go through the preparatory *Init* state first.

```
MACHINE          Generator
SETS             GENSTATES = { Off , Init , Running , Tripped }
VARIABLES        state
INVARIANT        state ⊆ GENSTATES
INITIALISATION   state := Off
OPERATIONS
    StartUp ≙
        PRE  state = Off  THEN  state := Init  END ;
    RunGen ≙
        PRE  state = Init  THEN  state := Running  END ;
    StandBy ≙
        PRE  state = Running  THEN  state := Init  END ;
    TripOut ≙
        PRE  state ∈ { Init , Running }  THEN  state := Tripped  END ;
    ReStart ≙
        PRE  state = Tripped  THEN  state := Init  END
    SwitchOff ≙
        PRE  state ∈ { Init , Tripped }  THEN  state := Off  END
END
```

Now we give the specification of the full system. It incorporates two generators, a gas fired one and an oil fired one which together contain all the state of the system. It contains operations to start the system and to recover after a trip, also a retrenchment of the *RunGenPower* operation. Given the lag between starting a generator and its being able to deliver full power, the signature of the *RunGenPower* operation in the retrenchment has an extra paramenter compared to that of the high level version to allow the environment to warn the control system of upcoming changes in demand. Demands for power that were not adequately anticipated thereby can cause a trip. When a trip occurs the power output is too unstable to be put into the environment, and is immediately switched to a sink, the environment seeing a sudden loss of power. Fig. 3 gives a transition diagram for the non-exceptional part of the *RunGenPower* operation of the *GenPower_Ret* machine. Note the slightly nonstandard I/O labelling.

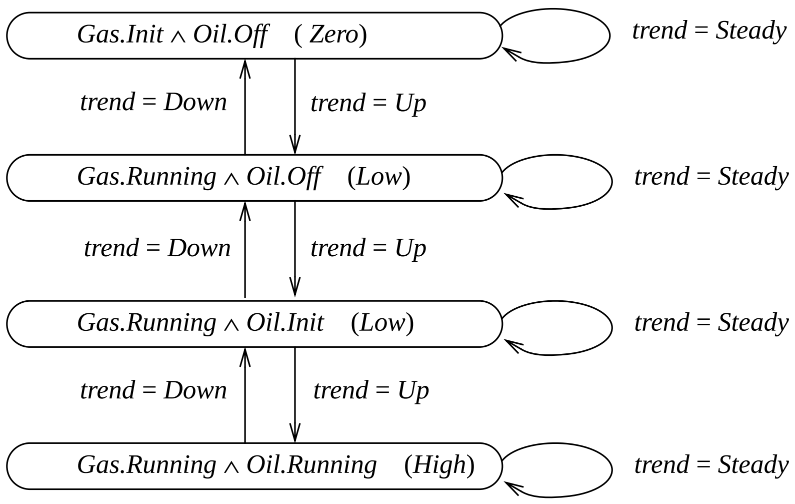

Fig. 3 Transition diagram for *RunGenPower*

```
MACHINE            GenPower_Ret
RETRENCHES         GenPower_Mch
INCLUDES           Gas.Generator , Oil.Generator
CONSTANTS          Threshold
SETS               POWERLEVELS = { Zero , Low , High } ;
                   TRENDS = { Up , Down , Steady }
OPERATIONS
    StartSystem $\triangleq$
        PRE  Gas.state = Off $\wedge$ Oil.state = Off  THEN  Gas.StartUp  END ;
    yield $\longleftarrow$ RunGenPower ( demand , trend ) $\triangleq$
        BEGIN
            PRE
                demand $\in$ POWERLEVELS $\wedge$ trend $\in$ TRENDS
            THEN
                SELECT  Gas.state = Init $\wedge$ Oil.state = Off $\wedge$
                            demand = Zero $\wedge$ trend = Steady
                    THEN  yield := Zero
                WHEN  Gas.state = Init $\wedge$ Oil.state = Off $\wedge$
```

```
                    $demand = Zero \wedge trend = Up$
            THEN  $yield := Zero \parallel Gas.RunGen$
        WHEN  $Gas.state = Running \wedge Oil.state = Off \wedge$
                    $demand = Low \wedge trend = Steady$
            THEN  $yield := Low$
        WHEN  $Gas.state = Running \wedge Oil.state = Off \wedge$
                    $demand = Low \wedge trend = Down$
            THEN  $yield := Low \parallel Gas.StandBy$
        WHEN  $Gas.state = Running \wedge Oil.state = Off \wedge$
                    $demand = Low \wedge trend = Up$
            THEN  $yield := Low \parallel Oil.StartUp$
        WHEN  $Gas.state = Running \wedge Oil.state = Init \wedge$
                    $demand = Low \wedge trend = Steady$
            THEN  $yield := Low$
        WHEN  $Gas.state = Running \wedge Oil.state = Init \wedge$
                    $demand = Low \wedge trend = Down$
            THEN  $yield := Low \parallel Oil.SwitchOff$
        WHEN  $Gas.state = Running \wedge Oil.state = Init \wedge$
                    $demand = Low \wedge trend = Up$
            THEN  $yield := Low \parallel Oil.RunGen$
        WHEN  $Gas.state = Running \wedge Oil.state = Running \wedge$
                    $demand = High \wedge trend = Steady$
            THEN  $yield := High$
        WHEN  $Gas.state = Running \wedge Oil.state = Running \wedge$
                    $demand = High \wedge trend = Down$
            THEN  $yield := High \parallel Oil.StandBy$
        ELSE
            IF  $Oil.state \in \{ Init , Running \}$
            THEN  $Oil.TripOut \parallel Gas.TripOut \parallel yield := Zero$
            ELSE  $Gas.TripOut \parallel yield := Zero$
            END
        END
    WITHIN
        $( powerreq < Margin \Leftrightarrow demand = Zero ) \wedge$
        $( powerreq \geq Margin \wedge powerreq \leq Threshold \Leftrightarrow$
                            $demand = Low ) \wedge$
        $( powerreq > Threshold \Leftrightarrow demand = High )$
    CONCEDES
        $( Gas.state \neq Tripped \wedge Oil.state \neq Tripped ) \Rightarrow$
          $(\ \ ( yield = Zero \Rightarrow outpower < 2 \times Margin ) \wedge$
            $( yield = Low \Rightarrow outpower < Threshold + Margin ) \wedge$
            $( yield = High \Rightarrow outpower > Threshold - Margin )$
          )
    END
  END ;
$RecoverSystem1 \triangleq$
  PRE  $Gas.state = Tripped \wedge Oil.state = Tripped$
  THEN  $Oil.SwitchOff$
  END ;
```

```
    RecoverSystem2  ≙
        PRE  Gas.state = Tripped ∧ Oil.state = Off
        THEN  Gas.ReStart
        END ;
    ShutDownSystem  ≙
        PRE  Gas.state = Init ∧ Oil.state = Off
        THEN  Gas.SwitchOff
        END
END
```

Note that we have built in a constraint between *RecoverSystem1* and *RecoverSystem2* whereby the former must complete (if its precondition applies) before the latter can be applied, alluding to a presumed interactions between oil and gas generators not otherwise modelled.

Execution sequences of *GenPower_Ret* that start properly and end cleanly, can be described by the following regular expression, where an optional operation occurrence is to be included or excluded at a particular point depending on whether or not its precondition holds there:

StartSystem
(; $\underline{RunGenPower\ *}$ [[; *RecoverSystem1*] ; *RecoverSystem2*]) * ;
ShutDownSystem

The underlined $\underline{RunGenPower\ *}$ sections of such an execution sequence are simulable by *GenPower_Mch* . It is easy to check that the given retrenchment is both a simple simulable retrenchment and a memoryless regular one, so Theorems 6.3 and 7.3 apply. In fact any run of concrete *RunGenPower* steps can be simulated by abstract steps though the converse does not hold — we can easily imagine abstract runs with power demands that fluctuate too wildly for the concrete system to be able to keep up. This is a typical characteristic of the transition from a continuous model to a discrete one.

We close this little case study with one further observation. Noting that there is no state at all at the abstract level, there is no RETRIEVE relation either. This reduces the retrenchment PO, $\ldots \Rightarrow [RunGenPower_C]\neg[RunGenPower_A]\neg(G \vee C)$, to a triviality, assuming that the vacuuous G defaults to *true* as we would want it to do in the antecedents of the PO. Of course there is nothing to prevent us from proving the other branch of the disjunction too but we don't have to. In fact in this example the CONCEDES clause acts conjunctively more than disjunctively. Where such distinctions need to be vigorously policed, as in the most highly critical developments, we can separate conjunctive and disjunctive aspects to get a more subtle version of retrenchment (called sharp retrenchment, see [Banach and Poppleton (1999)]). These considerations apply especially to relationships involving abstract and concrete outputs, which certainly *do not* hold *only* when the the retrieve relation fails; i.e. they hold conjunctively.

9 Conclusions

In this paper we briefly reviewed some of the disadvantages of restricting oneself to refinement as the sole means of moving from an abstract high level description of a system to a realistic implementation level one. We find that the refinement POs are often just too demanding to allow the relegation of various kinds of low level considerations to the appropriate level of abstraction, with the result that such aspects of the system end up cluttering the most abstract levels.

Of course it is often the case that people use the word “refinement” much more informally, for development steps where additional detail incompatible with the official refinement POs is indeed introduced, but such steps lack precise semantics, and thereby the potential for machine checkability. Our introduction of retrenchment has as objective the legitimisation of such practices, by offering a syntactic container for the relevant properties, and especially, specific proof obligations.

It is hardly the case that no one else has previously noticed the difficulties imposed by the refinement straitjacket. The complexities arising from being sensitive to the purely finite domains available to real implementations were noted in the work on clean termination (see eg. [Coleman and Hughes (1979), Blikle (1981)]). Another approach to the same subject can be found in Neilson’s thesis [Neilson (1990)], which observes that the infinite idealised domains of textbook examples usually arise as well behaved limits of finite ones, and thus refinement in the idealised case can be understood as the limit of a finite version. Yet another attack can be found in [Owe (1985), Owe (1993)] who proposes dealing with finiteness considerations, and the resulting definedness problems, by using a carefully constructed three-valued logic. The I/O side of the coin has been examined by [Hayes and Sanders (1995)], and more recently by [Boiten and Derrick (1998), Stepney, Cooper and Woodcock (1998), Mikhajlova and Sekerinski (1997)]. Retrenchment provides a relatively simple vessel into which (at least closely related variants of) many of these ideas can be placed.

Perhaps the most obvious related development method to retrenchment is the rely/guarantee method of [Jones (1983)] and its successors. Here too a development step is mediated by an additional pair of predicates per operation, the rely and the guarantee clauses, but the crucial difference with respect to retrenchment is that both act conjunctively, and thus allow no weakening of the retrieve relation. The fact that these methods are mainly designed to assist in the development of concurent programs is what explains the particular form of the POs there.

The notion of retrenchment is thus inspired by very pragmatic considerations. This said, it behoves us to draw out its theoretical consequences, and the majority of this paper was devoted to some aspects of simulation that retrenchment supports. We alighted on the notion of stepwise simulation as the key one. Because of the possibility of weakening the retrieve relation in the after-state of a step, and of strengthening it in the before-state, two consecutive retrenched steps will not always admit sequential composition. Equally, since not all concrete steps are simulable anyway, an attempt to abstractly simulate the whole of a concrete execution sequence is not guaranteed to succeed. Thus the punctured simulation concept arose naturally. The view that it is the concrete sequence that is the one to be simulated, and the abstract sequence that is to do the simulating, derives from the fact that in the end, the concrete machine in a retrenchment development step provides the more accurate model of the real system. The abstract view is likely to be a useful though ultimately oversimplified one.

Punctured simulation is a very general notion, and as such an unpromising place to look for sharp results of a general kind. Nevertheless we were able to identify the unique domain property for large maximal punctured simulations and show that it held for two kinds of statically characterisable systems. Simple simulable retrenchments are very close to being refinements, and thus it is not surprising that good results can be obtained for them. The other kind, memoryless regular retrenchments, also exhibited the unique domain robustness property though additional assumptions would be needed to derive a stepwise simulation property.

Punctured simulation as presented here sets the scene for launching more incisive investigations into the interactions between retrenchment and behavioural approaches to system description. The case study we gave in section 8 merely hinted at these possibilities as we used just about the simplest behavioural description method imaginable, the regular expression. That this was in fact adequate to describe system behaviour comes down to the relatively trivial system structure in this particular case. Obviously, in more complicated situations, where for example non-trivial fairness questions arise, less primitive behavioural description methods will prove useful. However this is again beyond the scope of the present paper.

References

Abadi M., Lamport L. (1991); The Existence of Refinement Mappings. Theor. Comp. Sci. **82**, 253-284.

Abrial J. R. (1996a); The B-Book. Cambridge University Press.

Banach R., Poppleton M. (1998); Retrenchment: An Engineering Variation on Refinement. *in*: Proc. B-98, Bert (ed.), LNCS **1393**, 129-147, Springer.

Banach R., Poppleton M. (1999); Sharp Retrenchment, Modulated Refinement, and Punctured Simulation. *in preparation* .

Blikle A. (1981); The Clean Termination of Iterative Programs. Acta Inf. **16**, 199-217.

Boiten E., Derrick J. (1998); IO-Refinement in Z. *in*: Proc. Third BCS-FACS Northern Formal Methods Workshop. Ilkley, U.K.

Coleman D., Hughes J. W. (1979); The Clean Termination of Pascal Programs. Acta Inf. **11**, 195-210.

Hayes I. J., Sanders J. W. (1995); Specification by Interface Separation. Form. Asp. Comp. **7**, 430-439.

Jones C. B. (1983); Tentative Steps Towards a Development Method for Interfering Programs. ACM Tran. Prog. Lang. Sys. **5**, 596-619.

Lano K., Haughton H. (1996); Specification in B: An Introduction Using the B-Toolkit. Imperial College Press.

Mikhajlova A, Sekerinski E. (1997); Class Refinement and Interface Refinement in Object-Oriented Programs. *in*: Proc. FME-97, Fitzgerald, Jones, Lucas (eds.), LNCS **1313**, 82-101, Springer.

Neilson D. S. (1990); From Z to C: Illustration of a Rigorous Development Method. PhD. Thesis, Oxford University Computing Laboratory Programming Research Group, Technical Monograph PRG-101.

Owe O. (1985); An Approach to Program Reasoning Based on a First Order Logic for Partial Functions. University of Oslo Institute of Informatics Research Report No. 89. ISBN 82-90230-88-5.

Owe O. (1993); Partial Logics Reconsidered: A Conservative Approach. Form. Asp. Comp. **3**, 1-16.

Stepney S., Cooper D., Woodcock J. (1998); More Powerful Z Data Refinement: Pushing the State of the Art in Industrial Refinement. *in*: Proc. ZUM-98, Bowen, Fett, Hinchey (eds.), LNCS **1493**, 284-307, Springer.

Wordsworth J. B. (1996); Software Engineering with B. Addison-Wesley.