Science and Engineering Research Council
Rutherford Appleton Laboratory
Chilton DIDCOT Oxon OX11 0QX

RAL-92-011

On the Role of Read and Write Frames in Algorithm Refinement for States with Invariants

J Bicarregui

January 1992

On the Role of Read and Write Frames in Algorithm Refinement for States with Invariants

Juan Bicarregui
Systems Engineering Division
Rutherford Appleton Laboratory.

January 20, 1992

Abstract

The read and write frames of reference variables used in the VDM style of operation decomposition serve two purposes: semantically, they record information as to what access to the state a valid implementation of the operation can be allowed to make; furthermore, in a more syntactic vein, they serve to bind the variables that occur in the predicates of the operation specification. The use of the frames in these two roles is examined, in particular for the case where there is an invariant on the state, and it is argued that they can be usefully distinguished.

A model for operation specification and refinement is developed that explicitly handles changes in the frames. In order to handle state invariants, it is necessary to define when a part of the state is independent of the rest with respect to the invariant that is in force. A semantic definition and syntactic criterion for independence are given. This idea of partitioning the state into independent parts with respect to the invariant is developed in connection with notions of satisfaction and satisfiability.

1 Introduction

There is little doubt that, if we are to successfully manage the building of ever more complex software systems, the ability to break down the task in hand, separately tackle each sub-task, and then bring these separate solutions together with confidence that the whole will perform successfully, is crucial. Thus when developing a formalism of software specification and refinement the issue of compositionality must be central.

Following the process of top-down design, at any stage we have a specification of a system component and the design task is to develop a component that satisfies that specification. Typically the design will be the composition of several smaller components that, at this stage, will only be specified and which will later be developed independently themselves.

We can state the problem as:

If we require that a specification S is to be refined by a design D, written

$$S \sqsubseteq D$$

then we can build a high level design which states that D should be some composition, C, of components $S_1 \ldots S_n$ which are themselves only specified at this stage,

$$D = C[S_1, \ldots, S_n]$$

such that, if each component is itself refined by a design,

$$S_i \sqsubseteq D_i$$

then the original specification must be satisfied by the composition of those designs.

$$S \sqsubseteq C[D_1, \ldots, D_n]$$

Compositionality states that the development of each S_i should be able to proceed independently of the rest.

This paper considers two issues that arise when we examine the matter of compositionality for the process of algorithm refinement taking as a starting point the VDM style of operation decomposition given, for example, in [Jones90]. The first consideration is that each sub-development, $S_i \sqsubseteq D_i$, should be truly independent of the rest. That is, that sufficient information should be contained in the S_i to characterise its valid implementations, particular attention being paid to the case where there is an invariant that relates the possible values of components of the state. The second issue is the matter of state access: the so called *frame problem* addressed by the "externals" clauses in the operation definition. Typically, each of the smaller components, S_i, involves only part of the state available to S, so in the development of each S_i, we want only to concern ourselves with that part and not have to look outside for contextual information. In VDM, the frame of reference of an operation is given explicitly by the read and write access conditions.

Of course the two issues are related since what is a sensible frame for an operation depends crucially on any invariant that may be in force on the state. The interaction of these two concerns motivates us to distinguish two roles for the read and write frames and the separation of these roles yields a more expressive notation. This extra expressive power comes at the expense of some prolixity – a third frame is introduced into the definition of an operation, however the model proposed is envisaged as the basis for machine based support and it is anticipated that the extra complexity could be handled automatically and only come to the fore when explicitly relevant.

The rest of this section gives an outline of the background to the work presented here and some motivating examples for the model proposed. Section 2 defines a model for state-based operations that explicitly handles frames and invariants and makes a definition of when a part of the state is independent with respect to the invariant which is then used in the examination of the role that the read and write frames play in satisfiability and refinement. The third section briefly considers the present work in a broader perspective, drawing some conclusions and proposing possible future work.

Background

Operation decomposition in VDM

The starting point for the work given here is the treatment of operation decomposition in VDM given in [Jones87, Jones90]. In these works a clear exposition of the underlying principles is given through examples and a set of proof rules for verification of refinements. These rules are proved correct with respect to a denotational semantics. We take on board the arguments given there advocating the convenience of having postconditions relating before and after states and do not reiterate those arguments here. The rules do not, however, explicitly handle the frame of reference variables. An extension of the model is given in [Milne88], however, the rules there are rather cumbersome and no attempt to justify them is made.

[AhKee89] is perhaps the most detailed treatment of operation decomposition in the VDM style. A denotational semantics is given for a language that extends that of [Jones90] with arrays, and procedures. A proof theory for this language is given and is shown to be sound and relatively complete with respect to the denotational semantics. In order to cater for blocks and procedures with static scoping, the usual Hoare triple is augmented by a syntactic environment with static and dynamic components. The static part of the environment may be compared with our set of read variables, and the work does indeed consider extending and contracting this frame, however, it does not distinguish the read-only from the read-write variables.

Other approaches

There is a large body of work stemming from [Dijkstra76] that considers verification of programs in the framework of a weakest precondition semantics. Mostly, this work considers only postconditions of one state which leads to some very elegant mathematics.

Some more recent works add specifications to the language bringing them closer to present concerns. [Morris87] gives an elegant treatment with "prescriptions" which makes use of higher ordinals to handle unbounded non-determinism. [Morgan86] includes a similar "specification statement" and explicitly handles the writable variables and also postconditions of two states. Many previous works are brought together in [Morgan88] and the central ideas are presented in textbook form in [Morgan90] which gives a formalism for algorithm refinement via a set of proof rules defining valid refinements. In this treatment, all state variables are global with respect to read access and a single "frame" is given that denotes those variables that are able to be written.

None of these treatments deal directly with the variables that are in read scope. They assume a global state and carry around the scoping environment only implicitly. Thus the local specification of a part of an operation given during the development is insufficient to determine the valid implementations. For example, one has to look back up the development tree to discover what variables are in scope, what invariants are in force etc.

[Back88] does consider the frame of free variables available for use in expressions and also caters for specification statements through non-deterministic assignment statements. How-

ever, his treatment deals only with single state postconditions and does not distinguish the read-only from the read-write variables. There the frame cannot contract during development. This has the effect that x:=x is not refined by skip, since skip has a smaller read frame. Thus this frame is somewhat different to the read frame considered in this paper.

Motivation

Two example operations are given that motivate the usefulness of a third frame. They both act on the same state, which although very simple, is already complex enough to expose the issues with which we wish to deal:

$$
\begin{aligned}
&\mathit{State} :: a : N \\
&\qquad\quad b : N \\
&\qquad\quad c : N
\end{aligned}
$$

$$
\begin{aligned}
&\mathit{inv\text{-}State}(a, b, c) \triangleq \\
&\quad a \in \{0, 1\} \wedge \\
&\quad b \leq a \wedge \\
&\quad c \in \{1, 2\}
\end{aligned}
$$

A quick syntactic analysis of the clauses in the invariant shows us that the components a and b are in some way "linked" whereas c is independent of a and b.

Example 1, choose_b

Let us consider the specification of an operation that writes b, say choose_b.

choose_b
ext rd b
  wr b
post *true*

Clearly, because of the invariant, the operation is not free to choose any $b: N$, rather it needs to ensure that the invariant is maintained, i.e. $b \leq a$. So perhaps such an operation specification should only be meaningful if it also has read access to a. On the other hand, it is possible to implement the operation by $b := 0$ which is a correct whatever the value of a, so read access to a is not necessarily required by the implementation. In fact, even ignoring the access conditions, there are just two possible implementations of *choose_b* which preserve the invariant, $b := a$ and $b := 0$. Which implementations are valid depends on how we interpret the read frame when an invariant is in force.

Let us consider the first of these alternatives. We will restrict what we consider to be a well-formed operation so that state access is always "sensible" with respect to the invariant.

As a first cut, let us say that an operation must have read access to (at least) all fields that appear in the same conjunct of the invariant that a write field appears in. Thus the

operation reads sufficient of the state to ensure that all the relevant clauses in the invariant are maintained. (There is no need to consider those clauses not mentioning any writes since they will not be affected by the operation.)

This resolves the problems with the example choose_b above which becomes:

choose_b'
ext rd a, b
    wr b
post *true*

This is interpreted as if the relevant clauses, $b \leq a$ and $a \in \{0,1\}$, were added to the pre and post conditions so the satisfiability criterion would then quantify over the read frame and include the appropriate clauses of the invariant:

$$\forall a{:}N,\ \overleftarrow{b}{:}N \cdot a \in \{0,1\} \wedge \overleftarrow{b} \leq a \ \Rightarrow\ \exists b{:}N \cdot a \in \{0,1\} \wedge b \leq a \wedge \mathsf{true}$$

However, by restricting well-formedness in this way, we have lost some of the expressive power in the language. Perhaps we really intended to specify an operation that maintained the invariant but did not read a. That is, we meant that the only valid implementation should be $b := 0$.

Example 2, set_c

Now consider the operation specification:

set_c
ext rd b
    wr c
post $c \in \{b, b+1\}$

This is a well-formed operation by the above criterion and does indeed specify a bona-fide operation with two obvious implementations $c := b$ and $c := b + 1$. But in order to know that it is satisfiable (ie. that $c \in \{1,2\}$) we need the information relating to a that is given in the first clause of the invariant. Furthermore, apart from the two obvious implementations, if we consider the extra information about a given in the first clause of invariant, then $c := 1$ is also a valid implementation.

Thus the "available information" for determining valid implementations must include all the clauses that could give us any information about fields that appear in clauses with fields that could be written. That is, we must take the transitive closure of the "appears in a clause with" relation. This closure partitions the state into independent parts and in order to characterise the possible implementations of the specified operation we may need the information from the invariant about all fields that are "connected" to any field in frame. So for the operation

specification to have a well-defined meaning it must be able to look at all the parts of the state connected to the variables that appear in its definition.

However, there is still good reason to keep the smaller read frame since it can give information as to what implementations are intended to be valid. In this case, for example, although we require access to a in order to ascertain what the valid implementations are, it is not intended that $c := a$ should itself be a valid implementation, which would in fact be the case if the read frame were extended to include a. Thus expanding the read frame to the include all connected fields would change the meaning of the operation specification.

So the approach investigated in this paper, will be to carry around a third frame. This frame plays the "syntactic" role of carrying the declarations of all fields related to the fields in the read and write frames. The read and write frames are thus freed to play their "semantic" role: to give information about the access permitted of valid implementations.

We will also localise those clauses of the invariant that are relevant to this frame and thus make the operation definition independent of the state declaration yielding compositionality as discussed earlier.

We show below that a suitable third frame can, in practice, be deduced syntactically from the other parts of the operation together with the state definition. Thus its manipulation would be expected to go on behind the scenes in a support tool for the process.

2 A Model of Operations with Frames

2.1 Operation Definitions

We have motivated the need for a third frame which I will simply call *the frame*, as opposed to *the reads* and *the writes*. This frame must be an *independent part* of the state in a sense which is defined below. In order to make the operation definition independent of the state, it will also carry those clauses of the invariant that pertain to it. It is also shown below that any invariant can be split into a conjunction of clauses, each conjunct pertaining solely to one independent part of the state.

Thus an operation definition is composed of six parts: the *frame* which carries the declaration of all the variables that are in scope and binds all the free variables that appear in the rest of the operation definition, the *invariant* which contains all the contextual information about these variables[1], the *reads* and *writes* which give information about access to the reference variables that must be maintained by any implementation and the *pre* and *post* which have their usual meaning, that is as if the invariant were conjoined to them[2].

[1] the typing information could be put here instead of in the frame and the whole treatment carried out in an untyped logic

[2] The exposition in this paper does not deal with operations with parameters and results. Their treatment can be considered independently of issues covered here or it can be subsumed within it by considering the parameters as read only variables and the result as a write only variable.

$$
\begin{array}{rcl}
OpDef :: & frame & : \text{map } Id \text{ to } Type \\
& invariant & : Exp \\
& reads & : Id\text{-set} \\
& writes & : Id\text{-set} \\
& pre & : Exp \\
& post & : Exp
\end{array}
$$

where

$$
\begin{array}{l}
inv\text{-}OpDef(mk\text{-}OpDef(F, I, R, W, P, Q)) \triangleq \\
\quad R \subseteq dom(F) \wedge \\
\quad W \subseteq dom(F) \wedge \\
\quad I\colon Exp(dom(F))^3 \wedge \\
\quad P\colon Exp(dom(F)) \wedge \\
\quad Q\colon Exp(dom(F) \cup \overleftarrow{dom(F)})
\end{array}
$$

We do not insist on any relationship between R, W and the free variables of P and Q.

It is also a requirement that a valid frame of an operation definition be an "independent part" of the frame of any operation that it refines. This requirement will later form part of the definition of the satisfaction relation but before we consider it in more detail we introduce some notation.

2.2 Notation

Hooking

If S is any set of identifiers, say

$$S = \{x_a \mid a \in \alpha\}$$

then $\overleftarrow{S}$ is the set with each identifier in S distinguished in some way, with a $\leftarrow$ say. That is,

$$\overleftarrow{S} = \{\overleftarrow{x_a} \mid x_a \in S\}$$

More generally if we want to distinguish just some of the members of S, those in $S_1 \subseteq S$ say, then we write:

$$\overleftarrow{S}^{S_1} \triangleq \{\overleftarrow{x_a} \mid x_a \in S_1\} \cup \{x_a \mid x_a \in S - S_1\}$$

Similarly, if E is an expression with free variables in S, written

[3]This notation is explained in the next section.

$$E\colon Exp(S)$$

and $S_1 \subseteq S$, then

$$\overleftarrow{E}^{S_1} \triangleq E[\overleftarrow{\sigma_i}/\sigma_i]_{\sigma_i \in S_1}$$

Thus:

$$\overleftarrow{E}^{S_1} : Exp(\overleftarrow{S}^{S_1})$$

Identity

We define a shorthand notation for saying that a set of variables are unchanged. Id_S is simply the conjunction of clauses each stating that a variable in S is unchanged.

$$Id_S \triangleq \bigwedge_{x_i \in S} x_i = \overleftarrow{x_i}$$

Quantification

Let F be a composite type

$$\begin{aligned} F :: f_1 &: T_1 \\ &\vdots \\ f_n &: T_n \end{aligned}$$

and let S be a subset of the fields of F

$$S = \{f_{S_1}, \ldots, f_{S_k}\} \subseteq \{f_1, \ldots, f_n\}$$

Let E be an expression with free variables in S

$$E\colon Exp(f_{S_1}, \ldots, f_{S_k})$$

then we use the notation

$$\forall \mathsf{S} \cdot E$$

to stand for the universal quantification of all the free variables from S, that is, as a shorthand for

$$\forall v_1\colon T_{S_1}, \ldots, v_k\colon T_{S_k} \cdot E[v_i/f_{S_i}]_{i=1,\ldots,k}$$

and similarly

$$\forall \overleftarrow{\mathsf{S}} \cdot \overleftarrow{E}$$

for the correspondingly hooked formula.

Invariants

Since we will be analysing the role of invariants on composite types it will be convenient to write them explicitly when we mean the type restricted by the invariant and leave the undecorated type name to denote the "free" type. Thus

$$\forall \mathsf{F} \dagger I \cdot E \triangleq \forall \mathsf{F} \cdot I \Rightarrow E$$

2.3 Independence of a part of the state

We will first give a "semantic" version of when two parts of a state are independent before reverting to a syntactic one.

Definition

Let S be a subset of the fields of a composite type with invariant, $F \dagger I$.

$$S \subseteq \{f_1, \ldots, f_n\}$$

Let T be the rest of the fields.

$$T = \{f_1, \ldots, f_n\}\text{-}S$$

The part S of F is said to be an *independent part*, written $S \overset{\text{ind}}{\subseteq} F$, if and only if, swapping the S parts of two states that each satisfy the invariant maintains the invariant. That is if

$$\forall \mathsf{F}, \overleftarrow{\mathsf{F}} \cdot I \wedge \overleftarrow{I} \Leftrightarrow \overleftarrow{I}^S \wedge \overleftarrow{I}^T$$

This can be shown to follow from the following, more succinct, definition:

$$S \overset{\text{ind}}{\subseteq} F \triangleq \forall \mathsf{F} \dagger I, \overleftarrow{\mathsf{F} \dagger I} \cdot \overleftarrow{I}^S$$

We can, of course, break the state into its independent parts by taking the finest partition that respects independence. As stated earlier, legal frames must be independent parts of the state.

The following theorem states that a part is independent exactly when it is possible to write the invariant in such a way that the independence of that part is syntactically obvious.

Theorem

A part S of composite type $F \dagger I_F$ is an independent part of F, if and only if, there are predicates I_S: $Exp(S)$ and I_T: $Exp(F\text{-}S)$ such that

$$F \dagger I_F = \Sigma \dagger (I_S \wedge I_T)$$

Proof

The proof proceeds by construction of the two required predicates. The key definition, the *restriction* of the invariant to a part of the state, is given below. The proof then follows easily by showing that the required predicates are the obvious restrictions of the invariant. For brevity, details are not given here.

Definition

For a part S of composite type $F \dagger I_F$, define I_S, the invariant *restricted to* S by

$$I_S \triangleq \exists \mathsf{F\text{-}S} \cdot I_F$$

With this definition the following corollary gives a criterion for independence.

Corollary

$$S \overset{\text{ind}}{\subseteq} F \quad \textit{iff} \quad \forall \mathsf{S}, \mathsf{F\text{-}S} \cdot I_S \wedge I_T \Leftrightarrow I_F$$

This criterion for independence is a little reminiscent of the notion of independent events in probability theory: that is, that the probability of two events happening together is the product of the probability of them each happening separately.

The definition of the restriction of an invariant owes something to the notion of hiding in CSP.

It is easy to show that the syntactic condition for independence described in example 2, that is, the closure of the relation "appears in the same clause as", is a sufficient condition for independence. In practical cases, it is this syntactic condition that we are likely to use as it generally yields a sufficiently fine partition of the state for refinement.

2.4 Satisfiability

Given that we have an operation specification with a frame that is an independent part of the overall state and with the restricted invariant also given as part of the specification, it is possible to consider satisfiability (and refinement) using only local information.

The standard satisfiability condition, roughly stated, says that: for any initial state satisfying the precondition there must be a final state that satisfies the post condition. It can be stated formally as follows:

$$\forall \overleftarrow{\sigma} : \Sigma \dagger inv \cdot \exists \sigma : \Sigma \dagger inv \cdot \overleftarrow{pre} \Rightarrow post$$

However this tells us nothing about the fact that the implementation must respect the access conditions given by the read and write frames. How should the condition be generalised to accommodate this information?

Our first attempt might be to simply quantify over the read and write frames, saying something like "for all values of the reads there must exist possible values of the writes such that ...". However this is not correct since the invariant (and predicates) could mention variables outside the read and write frames.

In fact, we require the choice of writes to be made without recourse to the value of things outside the read frame and that this choice should be valid whatever these values actually are. This idea can be captured formally simply by rearranging the order of quantification in the formula.

We get a condition that is "scoped over" the frame but in which the quantifiers for the parts inside and outside the reads have to be interspersed to reflect the fact that values assigned to the writes can only depend on the part inside the reads.

For an operation $mk\text{-}OpDef(F, I, R, W, P, Q)$ we get:

$$\forall \overleftarrow{R} \cdot \exists \mathsf{W} \cdot \forall \overleftarrow{F\text{-}R} \cdot \exists \mathsf{F\text{-}W} \cdot \overleftarrow{P} \wedge \overleftarrow{I} \;\Rightarrow\; Q \wedge I \wedge Id_{F\text{-}W}$$

The presence of $Id_{F\text{-}W}$ reflect the fact that any implementation must respect the write access condition. The innermost quantifier can easily be removed by use of the identities in the last clause yielding[4]

$$\forall \overleftarrow{R} \cdot \exists \mathsf{W} \cdot \forall \overleftarrow{F\text{-}R} \cdot \overleftarrow{P} \wedge \overleftarrow{I} \;\Rightarrow\; \overleftarrow{Q}^{F\text{-}W} \wedge \overleftarrow{I}^{F\text{-}W}$$

The position in this formula of the existential quantification over W captures the fact that the values given to the writes can depend on the reads but not on those fields outside the reads. It brings to light the fact that the write frame is more than just a syntactic sugaring for an addition of the appropriate clauses $x_i = \overleftarrow{x_i}$ to the postcondition.

2.5 Refinement

The generalisation of the definition of refinement[5] to this model does not seem to present any difficulties. Rather than attempt to give a comprehensive proof theory for refinement here, a flavour of the treatment is given by stating a few rules that justify some valid refinements, primarily focusing on those aspects that relate to the frame.

[4]For variables outside the writes, where we know hooked and unhooked values are equal, we have the freedom to use hooked or unhooked variables as we like. Unlike the standard usage, here we choose to use the hooked names so we have hooked variables appearing for the whole frame whereas unhooked variables only appear for the writes.

[5]based on a denotational semantics in the style of [Jones87].

Weaken pre and strengthen post

The rules for weakening the precondition and strengthening the postcondition contain no surprises. They are the obvious extensions of the usual rules.

$$\frac{P_A \wedge I \;\Rightarrow\; P_C}{(F, I, R, W, P_A, Q) \sqsubseteq (F, I, R, W, P_C, Q)}$$

$$\frac{\overleftarrow{P} \wedge \overleftarrow{I} \wedge Q_C \wedge I \wedge Id_{F,W} \;\Rightarrow\; Q_A}{(F, I, R, W, P, Q_A) \sqsubseteq (F, I, R, W, P, Q_C)}$$

Contract reads and writes

An implementation that achieves that specification whilst reading or writing fewer variables than it might is obviously correct. Because the frame is unchanged, we can shrink the reads and writes without worrying about variables becoming unquantified.

$$\frac{R_A \supseteq R_C}{(F, I, R_A, W, P, Q) \sqsubseteq (F, I, R_C, W, P, Q)}$$

$$\frac{W_A \supseteq W_C}{(F, I, R, W_A, P, Q) \sqsubseteq (F, I, R, W_C, P, Q)}$$

Contract frame

We can shrink the frame provided the new frame is an independent part of the old, the invariant is restricted accordingly, and that no variable mentioned falls out of scope. This last condition being captured by insisting that the new OpDef is well-formed.

$$\frac{F_A \overset{\text{ind}}{\supseteq} F_C,\ (F_C, I_{F_C}, R, W, P, Q)\text{: } OpDef}{(F_A, I, R, W, P, Q) \sqsubseteq (F_C, I_{F_C}, R, W, P, Q)}$$

Expand frame

We can expand the frame, that is declare independent local variables. The new variables will not appear in the abstract specification though they can later appear in the implementation. Naturally, since they are to be used as local variables, they are available for reading and writing .

$$\frac{F \cap S = \phi}{(F, I, R, W, P, Q) \sqsubseteq \mathsf{var\ v} : S\ \mathsf{in}\ (F \cup S, I, R \cup S, W \cup S, P, Q)\ \mathsf{end}}$$

Expanding Reads and Writes

Having permitted expansion of the reads and writes by variables from outside the frame, we have a choice as to whether to allow expansion of the writes within the frame provided we

ensure $x = \overleftarrow{x}$ for the new write variables. The choice corresponds to whether the intention is that variables not in the writes should not be changed at any time during execution of the program section under development, or whether they just have to be reset to their original value by the end of the execution.

As we already have the possibility of expanding the reads and writes when the frame is expanded, we will insist that access conditions given by reads and writes are "hard and fast". Namely, that it is not possible to expand the reads and writes by variables that are already in the frame[6].

Assignments

It is in the rule for assignments that the read and write frames are ultimately employed: assignments can only be made to the write variables and the expression evaluated can only refer to read variables. Assuming it is is well-typed[7], the assignment, $x := e$, precisely satisfies the postcondition,

$$x = \overleftarrow{e}^{x} \wedge Id_{F\text{-}\{x\}}$$

thus we get the refinement rule

$$\frac{x \in W, e: Exp(R), Id_{F\text{-}\{x\}} \wedge I \wedge P \;\Rightarrow\; (Q \wedge I)\left[e/x, x/\overleftarrow{x}\right]}{(F, I, R, W, P, Q) \sqsubseteq w := e}$$

3 Discussion

The motivation for the present work came from an intention to develop interactive tools support for operation decomposition in the VDM style. There is much to be gained from mechanical support for this process and there seems to be good reason to develop a formalism with mechanical support specifically in mind since potential difficulties for pencil and paper methods may become unimportant when such support is provided.

Compositionality is of central importance to a design methodology, and the extra complexity of explicitly carrying the necessary contextual information is a small price to pay in the definition of an abstract model of the development process. The syntactic clutter that results is indeed an inconvenience for the presentation of such a model, but this in itself should not be a discouragement since tools based on such a model could go a long way towards handling this cumbersome baggage without distracting the user. Furthermore, localising the state definition to the operation specifications has the potential advantage of allowing more general data reifications than permitted by a retrieve function between global state definitions.

[6]This choice means that the model could include a primitive form of shared-state concurrency though that matter is not gone into here as it not the subject of this paper.

[7]We also assume the definedness of e

As pointed out earlier, the interspersing of the universal and existential quantifiers over read and write frames in the satisfiablity obligation reflects the fact that the read and write frames play more than a purely syntactic role in the decomposition process. The third frame "scopes" the semantics of the operation specification and gives rise to a degree of freedom in the choice of what interpretation to give to the operation outside this frame. Lamport points out that an important decision in the design of his Temporal Logic of Actions, was that the semantics should be that "all can change" unless otherwise stated. This permits more elegant laws concerning the concurrency combinators. Generally the opposite approach has been taken in the development methodologies for sequential programs. There may be some benefit in examining this incongruity further. For the sake of modularity, for instance, there may be some benefit in having a frame, outside of which we assume nothing.

The separation of the syntactic and semantic roles of the two traditional frames can lead to some "interesting" specifications. Why should a specification be constrained to deal with the same variables as the implementation? Although there is a danger of introducing some "surprise" refinements in this way, the specifier is always at liberty to coalesce the two frames if so desired. As well as the obvious implications in the treatment of shared-state concurrency, there may also be the possibility of specifying some forms of "security" requirements in this way.

At the time of writing, only a few examples have been tried in the framework presented here, this is the subject of ongoing work by the author.

The presentation of a set of refinement rules, each of which deals with the manipulation of one part of the operation specification, begs the question as to whether all valid refinements are justifiable through a series of such orthogonal steps or whether sometimes we need to alter more than one part at once. If this were so, then there would be a need for some rules that combine valid refinements to justify others.

Acknowledgements

I would like to acknowledge the considerable contribution of my colleagues to this work: I am grateful to my supervisor professor Cliff Jones for providing the framework upon which this is all based and to my colleagues at RAL, in particular the two Brians, Ritchie and Matthews, and Chris Reade who, together and separately, have helped to clarify these ideas as they are represented both in my mind and on the page.

References

[AhKee89] *Operation Decomposition Proof Obligations for Blocks and Procedures*. J. A. Ah-Kee. Ph.D. Thesis. University of Manchester. 1989.

[Back88] *A Calculus of Refinements for Program Derivations*. R.J.R. Back. Acta Informatica (1988).

[Dijkstra76] *A Discipline of Programming*. E.W.Dijkstra, Prentice-Hall (1976).

[Jones87] *VDM Proof Obligations and their Justification* C.B.Jones, Proceedings of the VDM '87 Symposium, LNCS 252, Springer-Verlag(1987).

[Jones90] C.B.Jones, *Systematic Software Development using VDM* (second edition) Prentice Hall, 1990.

[Milne88] *Proof Rules for VDM Statements.* R. Milne, Proceedings of the VDM '88 Symposium, LNCS 328, Springer-Verlag(1988).

[Morgan86] *The Specification Statement.* C. Morgan. TOPLAS 10, 3 (July 1988).

[Morgan88] *On the Refinement Calculus.* C. Morgan, K. Robinson and P. Gardiner. Oxford University Technical Monograph, PRG-70, 1988.

[Morgan90] *Programming from Specifications* C. Morgan, Prentice Hall, 1990.

[Morris87] *A theoretical basis for stepwise refinement and the programming calculus.* J. Morris, Sci.Comput. Programming, **9** 287-306 (1987).