Data Refinement Refined

H³.M.S⁴ *

Oxford University Computing Laboratory

Programming Research Group

8-11 Keble Road

Oxford OX1 3QD.

DRAFT May 1985

We consider the original work of Hoare and Jones on data refinement in the light of Dijkstra and Smyth's treatment of nondeterminism and of Milner and Park's definition of the simulation of Communicating Systems. A proof method is suggested which we hope is simpler and more general than those in current use, and it is illustrated with examples drawn from programming practice.

* C.A.R.Hoare, I.J.Hayes, Jifeng He, C.C.Morgan, J.W.Sanders, I.H.Sorensen, J.M.Spivey and B.A.Sufrin.

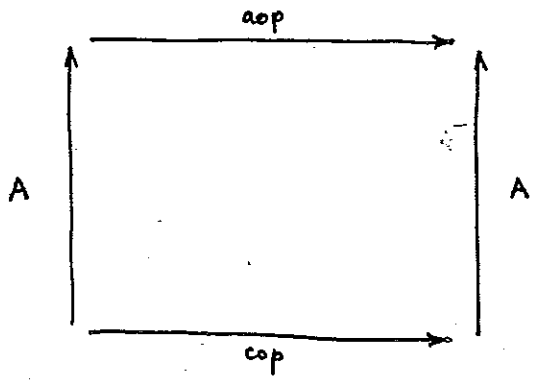
0. Introduction

A data type is generally defined in a manner similar to an algebra as a triple (S,S_0,OP), where S is the set of values of the data, S_0 is a set (usually a singleton) of initial values of the data, and OP is a set of named operations. The operations are usually parametrised by arguments and results drawn from certain other spaces of observable values. One data type C is said to be a **refinement** of a data type A if under all circumstances and for all purposes, the data type C can be validly used in place of A. The practical benefit of this arises when A is a simple, "abstract" data type, such as a partial mapping, which can be specified, understood and used in an applications program but cannot be directly implemented on a computer; whereas C is a complicated, "concrete" data structure, involving a collection of arrays and bitmaps and fileblocks, which can be economically stored and updated on the equipment available. We need to show, using appropriate proof methods, that it is valid to use C in place of A; this is known as "data refinement".

The earliest suggestion for a method of data refinement was given in [Hoare, 1972]. The method was based upon

(1) an **invariant** predicate, which must be proved true after initialistation and, on the assumption that it is true beforehand, true after every operation on the structure.

(2) an **abstraction** function which maps the current value of the concrete data type onto the abstract value which it stands for. Each operation must be proved to update the concrete data type in a manner corresponding to the desired operation on the abstract structure. This obligation is sometimes expressed as a commuting diagram



where A is the abstraction function

aop is the desired operation on the abstract data

cop is the corresponding operation on the concrete data.

This method was adopted and developed in the VDM technique of data refinement [Jones, 1980]. In VDM, certain additional properties of a data type are considered desirable

(1) The abstract data type should be **fully abstract** in the sense of Milner. This means that any two distinct values of the abstract data type can be distinguished by some sequence of operations on the data. In the VDM literature this is called "freedom from implementation bias".

(2) The concrete data type should be **adequate** to represent every value of the abstract data type.

In this paper we attempt simultaneously to generalise and simplify the notion of data refinement in the following ways

(1) Both the abstract and the concrete data types may be nondeterministic.

(2) There is no need for the concepts of full abstraction or adequacy.

(3) The relationship between the concrete and abstract data does not have to be functional.

(4) The invariant and the abstraction relation can be combined into a single relation called a **semisimulation**.

The method of data refinement to be used in the specification and proof of correctness of nontrivial system programs is very significant; we hope that the simplifications described above will make our method easier to teach, to learn, and to apply in practice.

The first programming language to make explicit provision for the concrete representation of abstract data structures was SIMULA 67. In that language, the concrete data is declared as the local workspace of a class, and is initialised in the statements of the class body. The data is updated by invocation of procedures also declared locally within the class. Each invocation uses a normal parameter mechanism to transmit values to the updating procedure, and to obtain results from it. The same ideas have been incorporated in the PASCAL PLUS envelope, which also permits information about the details of the concrete design to be hidden from the subsequent user of the envelope. Such hiding ensures that the concrete data type may be validly substituted for the abstract one. Similar effects can be achieved in FORTRAN 77 by subroutines with multiple entry points. Subroutine calling (or macro substitution) is the preferred method of implementing data types on conventional sequential computers.

An alternative method of implementing a data type is as an independent process, evolving concurrently with its using process and communicating with it by input and output rather than by procedure call. Each operation performed by the data type is triggered by input of the necessary parameters, and completed by output of its results; whenever parameters are unnecessary, a null message (signal) is transmitted. The externally visible behaviour of the data type is fully described by a sequential trace of values of the parameters transmitted on the successive operations, and any mention of internal states of the process is irrelevant. The total observable behaviour of a deterministic process is characterised by the set of all traces of its possible behaviour [Hoare, 1985]. If A and C are deterministic processes which have exactly the same sets of possible traces, it is clearly valid to replace A by C in any context, because there is no way to detect (from the outside) that such a substitution has been made. For this reason, we use sets of traces as the basis of the definition of refinement, even though the data type may be implemented by subroutine calls.

Because of the rigid alternation of input and output, reasoning about traces also applies when A and C are nondeterministic processes. Furthermore, replacement of A by C is valid when the traces of C are just a *subset* of the traces of A. In this case the type C is more deterministic (and therefore more predictable and controllable) than A, and will thus serve every purpose which A can reliably and provably serve. The validity of this extension to nondeterministic processes depends on absence of deadlock in A; this can be dealt with as a seperate condition, or more simply (and without loss of generality) by a construction which replaces deadlock in A by a "broken" process; that is, a process which gives absolutely arbitrary results and therefore has a trace set containing all possible traces.

The description of refinement given above in terms of processes and their traces is equally applicable when the data type is implemented as a collection of subroutines sharing a collection of declared data. We therefore need a method of specifying and describing a data type which is independent of its method of implementation. The method must allow nondeterminism, so the operations must be described as relations rather than functions. We shall adopt the Z method due originally to Abrial (see [Morgan and Sufrin, 1984]) and use predicates to describe both the values and the operations of each data type. When the abstract and concrete data types are described in this way, we need a simple method of proving that the concrete type is a refinement of the abstract one. Reasoning in terms of traces, that is, in terms of arbitrary sequences of operations,

could be complicated; so we introduce a much simpler semisimulation condition about states before and after each single operation, and show that this condition is sufficient for data refinement.

Our semisimulation condition is, as the name implies, slightly weaker than the simulation condition of traditional automata theory. This is because we are content with an asymmetric concept of refinement and with a restricted class of automaton. Nevertheless the semisimulation condition is considerably stronger than is necessary to prove refinement, and there are practical cases of refinement which do not satisfy it. It is important to recognise such cases and take appropriate steps when they arise. Firstly, we define a class of canonical data types in which internal nondeterministic choices are delayed as long as possible. If the abstract data type is canonical, semisimulation is equivalent to refinement and so provides a sufficiently strong proof method. For noncanonical data types the alternatives are

(1) The semisimulation condition may in fact hold, in which case it should be used to prove data refinement since it provides the simplest proof method.

(2) The criterion for refinement in terms of traces may be used.

(3) The abstract data type may be replaced by a canonical data type having the same traces (this can be achieved by the well-known powerset construction of automata theory), and the semisimulation condition applied. Equivalently, a power semisimulation condition in which states are replaced by sets of states may be used to prove refinement. It seems marginally preferable to use the power semisimulation condition and avoid the temptation to respecify the abstract data type simply on the grounds that the proof method involves subsets of abstract states.

1. Abstract Data Types

In this section we define the notion of abstract data type, introduce the schema notation which is used to specify abstract data types throughout this paper, and give some examples both of schemas and of abstract data types. We end the section by showing that no loss of generality is incurred by our restricting attention to certain simplified abstract data types.

1.1 Data type defined.

As has been standard in Computing Science since the early work of Hoare on data structuring (see [Hoare, 1972] and the article by Hoare in [Dahl, Dijkstra and Hoare, 1972]), an abstract data type can be viewed as a data structure on which certain operations are defined. The choice of operations depends on the use of the data type: for example the data structure *stack* might typically be operated on by *push*, *isempty* and *pop* (the first and last operations changing the stack and the other leaving it unchanged and returning a boolean value); the data structure *string* might be operated on by *insertion* and *deletion*, whilst in another context the operations might include sorting and searching.

The common feature of these examples is that the data structure possesses several states (or values), certain of which are initial states, and each operation on the data structure may take input, may change the state of the data structure, and may yield output. By considering, if necessary, a vector of variables, we suppose without loss of generality that each operation on a data structure takes at most one input x? and gives at most one output y! (not necessarily of the same type). Thus we can represent an abstract operation aop which accepts input x? when the data structure is in state a, performs some calculations which are concealed from the environment of the data type, yields output y!, and updates the data structure to state a', as a predicate

aop(a,x?,y!,a').

We do not assume that for each combination of initial state a and input value x? there is an unique combination of final state a' and output y!. That is, the predicate aop need not be functional; we allow an operation to select the next state and output nondeterministically.

The data structure itself can be identified with its states, some of which are singled out as initial states; and then the abstract data type can be identified with a data structure together with certain abstract operators, represented as predicates, on the states of the data structure. This brings us to

Definition 1.1. An (abstract) data type consists of a triple

$A = (AS, AS_0, AOP)$

where

the elements of the set AS are the (abstract) states;

AS_0 is a nonempty subset of AS whose elements are the initial states;

AOP is a nonempty finite set of named predicates of the type defined above, called the (abstract) operations.

Informally, the behaviour of a data type can be described in the following fashion.

(1) The data type starts in one of the states a_0 of AS_0. The choice of initial state is nondeterministic and cannot be controlled by the user; it cannot even be known by the user at the time of initialisation (though it can sometimes be deduced from the subsequent behaviour of the data type).

(2) The user selects some desired value x? of input and transmits it to the data type.

(3) If there is no combination of output y! and next state a' possible for the given combination of current initial state a and input x?, the system breaks in some way which we are not interested in describing further. It is the obligation of the user of the data type to ensure that this does not happen.

(4) If there exist possible values for output y! and next state a', one of these possible combinations is selected nondeterministically; the output y! is supplied to the user and the data type moves to the next state a', ready for the next operation (step (2)).

Before considering an example of a data type (see section 1.3) we need some notation.

1.2 Schema notation.

In this paper we shall use *schemas* to represent both the states of a data type and the operations upon them.

Definition 1.2. A schema consists of a triple (N,S,P) where

N is an identifier called the name of the schema;

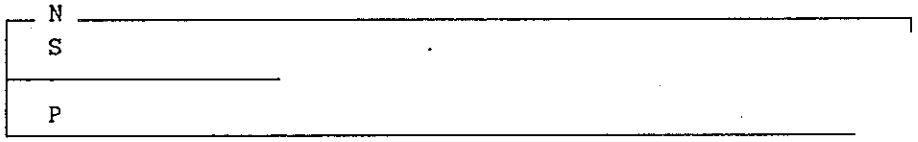
S, the signature of the schema, consists of an alphabet of (identifiers of) schema variables, each of which is assigned a type;

P is a predicate whose bound variables are all typed on quantification and whose free variables, the schema variables, lie in the signature S.

We shall usually follow standard (sloppy) mathematical convention and refer to a schema (N,S,P) as schema N instead of 'schema (N,S,P)' or 'the schema with name N'.

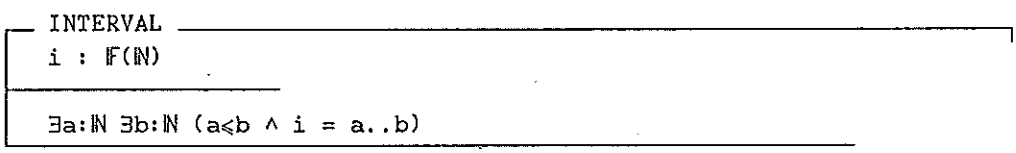
Note. This is a restriction of the usual definition of schema but is sufficient for our needs. In the more general form, a schema may also have free variables which do not appear in the signature of the schema; however they must either be constants of the current theory or be typed by virtue of having appeared in the signature of a previous schema (a workable enough assumption when using a top down methodology for specification). All schemas in the present paper are transparent enough to enable us to avoid this more general notion by using instead the convention, (4) below, on embedded schemas.

We will write a schema (N,S,P) in the form



In this box the name is written at the top, the signature appears above the middle horizontal line and the predicate below it.

X1.1. For example if the states of a data type are to consist of nonempty intervals of natural numbers we can specify this fact by the schema



where

INTERVAL is the name of the schema;

i is the only schema variable and is of type $\mathbb{F}(\mathbb{N})$ (throughout this paper $\mathbb{N}$ denotes the natural numbers and, for any type T, $\mathbb{F}(T)$ denotes all finite subsets of T);

$\exists a:\mathbb{N}\ \exists b:\mathbb{N}\ (a \leq b \wedge i = a..b)$ is the predicate and it describes the strongest (i.e. definitive) invariant property of the data type; in it a..b denotes the (closed) interval $\{c:\mathbb{N} \mid a \leq c \leq b\}$ which is nonempty since $a \leq b$.

//

We have used here, and shall continue to use, a typed version of standard mathematical notation, incorporating the usual conventions (e.g. the predicate $a \leq c \wedge c \leq b$ is contracted to $a \leq c \leq b$).

Before describing an operation by schema, we need some conventions.

(1) If N is (the name of) a schema then N' is (the name of) a schema which is like the first schema except that all occurrences of every schema variable name are dashed. For example INTERVAL' is (the name of) a schema which may be written in full as

```
┌─ INTERVAL' ──────────────────────────────────────────┐
│ i' : 𝔽(ℕ)                                            │
├──────────────────────────────────────────────────────┤
│ ∃a,b : ℕ (a≤b ∧ i' = a..b)                           │
└──────────────────────────────────────────────────────┘
```

Although INTERVAL' specifies the same set of abstract states as INTERVAL, it has quite a different predicate: INTERVAL' has schema variable i' whilst INTERVAL has schema variable i.

(2) A schema name may appear in the signature of another schema. The effect is as if all variables of the embedded schema had been declared within the enclosing schema, and its predicate conjoined to the predicate of the enclosing schema.

We assume that a variable name which ends in a dash (e.g. i') stands for a value on completion of the operation, whereas the corresponding variable without a dash stands for an initial value. Now an operation on a data type which has no input or output, but simply updates the value of the data, can be specified by schema. For instance an operation on the data structure INTERVAL which shifts an interval one unit to the right may be defined

```
┌─ shift ──────────────────────────────────────────────┐
│ INTERVAL                                             │
│ INTERVAL'                                            │
├──────────────────────────────────────────────────────┤
│ minimum i' = 1 + minimum i                           │
│ maximum i' = 1 + maximum i                           │
└──────────────────────────────────────────────────────┘
```

where we have written minimum F and maximum F for the minimum and maximum respectively of a nonempty finite set F of natural numbers. Thus the predicate here guarantees that both endpoints of the interval, and hence all intermediate values, are shifted one place to the right. The enclosing schema is shift and the embedded schemas are INTERVAL and INTERVAL'.

Predicates on successive lines but on the same side of the middle horizontal line of a schema are assumed to be conjoined by $\wedge$.

Written out, the schema for shift becomes

shift

$$
\begin{array}{l}
i : \mathbb{F}(\mathbb{N}) \\
i' : \mathbb{F}(\mathbb{N}) \\
\hline
\exists a:\mathbb{N}\ \exists b:\mathbb{N}\ (a \leqslant b \wedge i = a..b) \\
\wedge \\
\exists a:\mathbb{N}\ \exists b:\mathbb{N}\ (a \leqslant b \wedge i' = a..b) \\
\wedge \\
\text{minimum } i' = 1 + \text{minimum } i \\
\wedge \\
\text{maximum } i' = 1 + \text{maximum } i
\end{array}
$$

To specify a general operation on a data type by schema we need some further conventions.

(3) Declarations of variables having the same type may be combined. For example

$$i : \mathbb{F}(\mathbb{N})$$
$$i' : \mathbb{F}(\mathbb{N})$$

may be shortened to

$$i,i' : \mathbb{F}(\mathbb{N})$$

and

$$\exists a:\mathbb{N}\ \exists b:\mathbb{N}$$

may be shortened to

$$\exists a,b : \mathbb{N}.$$

(4) Two schemas may be conjoined by the $\wedge$ operator. The signature of the result is obtained by taking the union of the alphabets of the two schemas together with the type declaration of each variable. If a variable appears in both the alphabets then its types must coincide; otherwise the conjunction is not defined. The predicate of the conjunction is the conjunction of the predicates of the two components. So, for instance, we can define a schema

$$\Delta INTERVAL \triangleq INTERVAL \wedge INTERVAL'$$

which could have been written out

ΔINTERVAL

$$
\begin{array}{l}
i,i' : \mathbb{N} \\
\hline
\exists a,b : \mathbb{N}\ (a \leqslant b \wedge i = a..b) \\
\exists a,b : \mathbb{N}\ (a \leqslant b \wedge i' = a..b)
\end{array}
$$

In general, if N is (the name of) a schema, we will write ΔN to denote (the name of) a schema constructed in this way. Notice that the effect of conjoining two schemas is similar to that of embedding one in the other (convention (2)), except that embedding has the advantage that the variables of the embedded schema may appear in the predicate of the enclosing schema. For example the operation shift can also be specified

shift

$\Delta INTERVAL$

---

minimum i' = 1 + minimum i
maximum i' = 1 + maximum i

Similarly the identity operation over intervals can be defined

$\equiv INTERVAL$

$\Delta INTERVAL$

---

i' = i

In future we shall adopt the convention that $\equiv N$ stands for the identity operation on states described by the schema (named) N.

(5) If an operation has an input parameter, we observe the convention that its name ends in ? (e.g. x?). Thus we can define an operation which expands an interval by an amount depending on both the interval and on an input parameter x? not exceeding 17

expand

$\Delta INTERVAL$
x? : $\mathbb{N}$

---

x? $\leqslant$ 17
minimum i' = (minimum i) - x?
maximum i' = (maximum i) + x?

The first line of the predicate stipulates that the parameter x? must not exceed 17. If x? is larger than 17, the operation fails in a manner which we do not care to enquire about: it is the responsibility of the agent supplying the value of the parameter x? to avoid such failure. The observant reader will also notice that if the value of x? is larger than (minimum i) then (minimum i) - x? is no longer of type $\mathbb{N}$. Such implicit consequences are a powerful feature of schema notation. They are also dangerous; and the danger must be eliminated by meeting proof obligations such as those described in this paper.

(6) If an operation has an output argument, we observe the convention that its name ends in ! (e.g. y!). Thus we define an operation which, without changing an interval, yields its length

length

$\equiv INTERVAL$
y! : $\mathbb{N}$

---

y! = maximum i - minimum i

(7) Operations which take input and give output as well as changing state can be defined by a combination of all these conventions. For example

whim

$\Delta INTERVAL$
$x?, y! : \mathbb{N}$

$maximum\ i' = x? + maximum\ i$
$x? \leqslant y! \leqslant maximum\ i'$
$y! \leqslant minimum\ i'$

This whimsical operation is highly nondeterministic: its output y! cannot be less than its input x? and cannot exceed x? by more than the value of the right-hand endpoint of the initial interval. The effect of the operation on the interval is also potentially nondeterministic; the final value of the left-hand endpoint may equal the output, or it may be the same as the final value of the right-hand endpoint, or it may lie anywhere between these values. The fact that the left-hand endpoint of the final interval cannot exceed its right-hand endpoint is stated in the predicate of the embedded schema $\Delta INTERVAL$.

(8) If N is (the name of) a schema with schema variable a then N may be transformed by quantification of a in N, $\exists a\ N$. This stands for a schema similar to N, except that a is removed from its alphabet and the quantifier

$\exists a : T$ where T is the type of a

is inserted in front of the predicate of N. Similarly for $\forall a\ N$. This convention enables us to write succinctly a schema specifying the precondition, or domain, of an operation. Suppose that aop is an operation on states a of type AS (a previously defined schema), specified by

aop

ΔAS
$x? : X$
$y! : Y$

$P(a, x?, y!, a')$

where X and Y are the types of x? and y! respectively.

Then the precondition of aop is defined to be the schema obtained from aop by assuming the existence of some a' and y!

$pre\ aop \triangleq \exists a'\ \exists y!\ aop$

or, in full,

pre-aop

AS
$x? : X$

$(\exists a' : AS)\ (\exists y! : Y)\ P(a, x?, y!, a')$

An important but possibly unfamiliar aspect of the schema calculus is the use of decorated names to denote observable attributes of the system being described. Such conventions are characteristic of applied mathematics. If x,y and z denote the position of a particle in a physical system, x' conventially denotes the x-component of its velocity, y" denotes the y-component of its acceleration, and dt denotes an increment of time. The behaviour of the whole physical system is described by predicates (usually equations) which contain these variables. Pure mathematicians traditionally prefer to deal with closed abstract objects such as sets, relations and functions, rather than predicates with free variables. Of course a predicate can be identifed with the class of all its models, and this identification is important in foundational studies. But in examples of realistic complexity, the predicates seem much more convenient for practical use in specifications, calculations and proofs.

In summary, the schema notation is particularly suited to operations involving input, output and change of state. Since a schema might appear to be little more than a predicate, the reader may wonder why we do not simply deal with predicates and avoid schemas entirely. The reason is one of scale. In the specification of a practical system, huge numbers of variables are needed and the relationships between them have to be defined systematically, not only so that the specification is readable, but so that it may be refined in a structured, top-down manner. This requires the nesting of predicates on a scale uncalled for in standard mathematics and so there has been no prior need for a notation which facilitates it. However the embedding of schemas, introduced in convention (4), provides this kind of nesting of predicates; it appears to work well in practice and (together with typing) it distinguishes the calculus of schemas from the calculus of predicates.

1.3 Specification of a data type using schemas.

We have now covered most of the schema notation used in this paper, though a little more will be defined as it is needed. It is time to use it to specify a data type. The approach used in the following example demonstrates how any finite state machine can be viewed as a data type (as defined in Definition 1.1).

X1.2. A finite state machine, FS, having four states, 0,1,2,3 and whose initial states, FS_0, are either 2 or 3, is specified

FS

$a : \mathbb{N}$

$a \leqslant 3$

FS0

FS

$a \geqslant 2$

The only operation is rather strange

fop

ΔFS

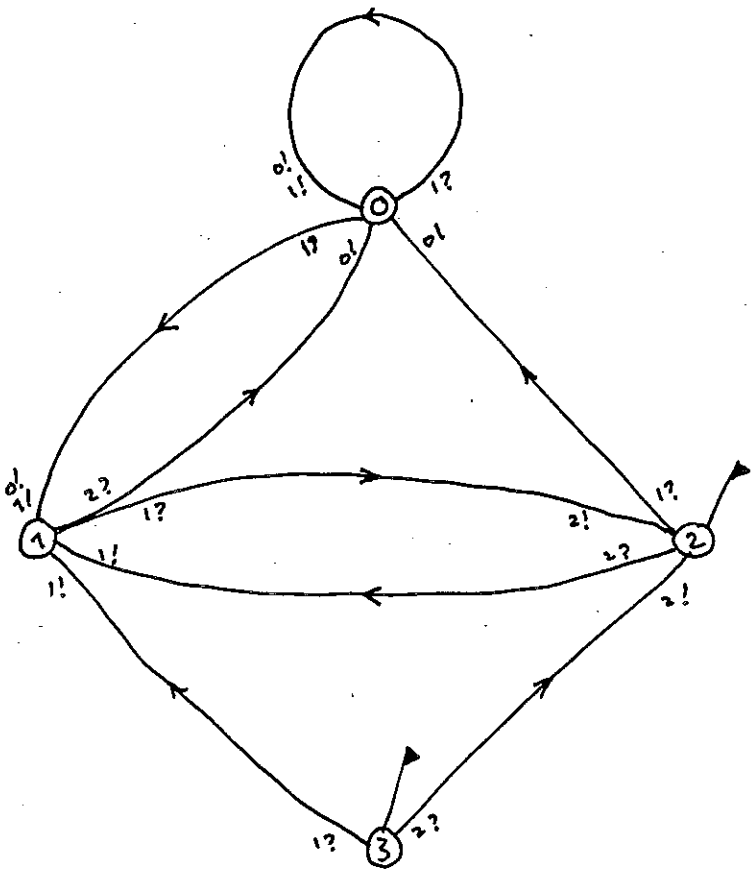
$x?, y! : \mathbb{N}$

$x? \leqslant 2$

$a \neq 0 \Rightarrow x? \neq 0 \wedge y! = a' = (a + x?) \mathit{mod} 3$

$a = 0 \Rightarrow a' \leqslant 1 \wedge y! \leqslant 1 \wedge x? = 1$

For such a small and irregular operation, a picture is clearer than a schema. Using a finite labelled directed graph, as for finite state machines, we can label each node by a state and demark each initial state with a flag. An arrow from a node labelled a to a node labelled a' stands for an operation which transforms state a into state a', and it is also marked with the values of input and output which accompany that transition. The operation is nondeterministic if and only if there is a node from which more than one arrow emerges; for convenience we permit such arrows to share the same tail. Thus the operation fop may be drawn as shown in Figure 1



1.4 Simplifying the schemas used to specify data types.

In the examples later in this paper we shall take full advantage of the generality offered by Definition 1.1 of a data type. However in reasoning about data types it is sometimes convenient to make simplifications; we now demonstrate why certain simplifications incur no loss of generality. The last one, (6), for example, greatly simplifies the treatment of a sufficient condition for data refinement in the next section.

(1) It can be assumed that the schema which specifies the states AS of a data type has exactly one variable a in its alphabet. Where more variables are required the type of a can be given as a cartesian product (e.g. as a record in PASCAL).

(2) It can be assumed that each operation in AOP has at most one input parameter which we name x?. In operations which do not require an input this parameter is just not mentioned in the predicate of the schema, so its value is arbitrary and is ignored. Where more than one parameter is required the cartesian product technique can be employed again.

(3) In exactly the same way it can be assumed that every operation has exactly one output parameter, y!.

(4) It can be assumed that AOP has only one operation in it, which we name aop. Where more than one operation is required, the desired operation can be selected by passing a tag along with the input. The type of the input parameter may have to be specified as a cartesian product whose components consist of the original type of the input and the type of an appropriate tag.

(5) It can be assumed that AS_0 contains only one initial state, which we name a_0. Since the choice of initial state is otherwise nondeterministic (see point (1) after Definition 1.1), in the case where more than one initial state is required, we can introduce a new nondeterministic operation, init, defined by

$$init(a_0,x?,a',y!) \triangleq a' \in AS_0.$$

(For an example in which this construction arises naturally, see X3.3.)

(6) It can be assumed that all operations are total in the sense that, in every state, every value of the correct type is permitted as input parameter. The construction which achieves this simplification replaces type A by a new one, A+, which behaves exactly like A until breakage of A (i.e. until A has been supplied with input outside the domain of its operation aop) when its subsequent behaviour is arbitrary. After A breaks, A+ may output any value (of the correct type); it may even appear to behave correctly for a long time or even forever. The states of A+ have an extra component to record whether type A has broken, so that its operation, aop+, knows whether to behave chaotically

AS+

AS

ok : boolean

Type A+ starts in an initial state for type A and with type A not broken

AS+0

AS0+

---

ok = true

If aop has the standard form

```
aop
ΔAS
x? : X
y! : Y
-------------------
P(a,x?,y!,a')
```

then

```
aop+
ΔAS+
x? : X
y! : Y
-------------------
( ok ∧ pre aop ∧ ok'=ok ∧ aop )
∨
( ok ∧ ¬pre aop ∧ ok'=false )
∨
( ¬ok ∧ ok'=ok )
```

X1.3. For example if

A ≙ (AS,AS_0,AOP), where

AS ≙ {0,1,2,3}

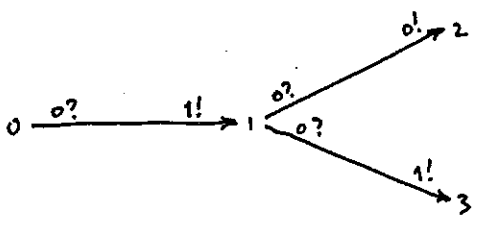
AS_0 ≙ {0}

AOP ≙ {aop}, and aop is defined

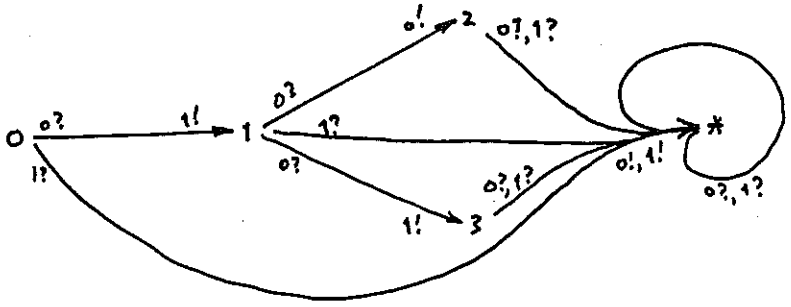
```
aop
ΔAS
x? : {0?,1!}
y! : {0!,1!}
-------------------
 (a=0 ∧ (x?,y!)=(0?,1!) ∧ a'=1)
∨
 (a=1 ∧ (x?,y!)=(0?,0!) ∧ a'=2)
∨
 (a=1 ∧ (x?,y!)=(0?,1!) ∧ a'=3)
```

then A and A+ can be depicted

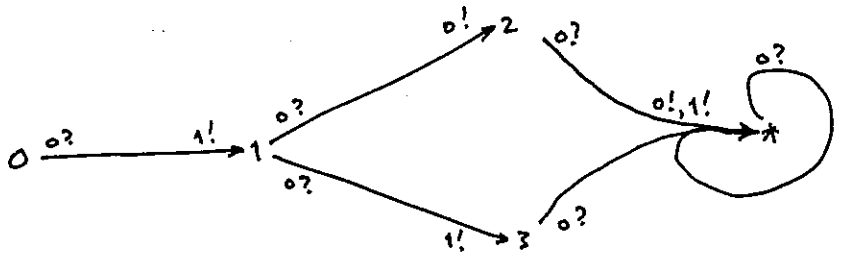
A



A^+



Note that had X been replaced by {0?} then A+ would have been depicted



//

2. Refinement

In this section we define what it means for one data type to refine another in terms of a user interacting with the two types. Because of the impracticality of checking the definition of refinement, we introduce a simple simulation condition which, presented as a checklist, is readily decided in practice and which is sufficient for data refinement. The data types for which the simulation condition is both necessary and sufficient for data refinement are identified and called canonical.

2.1 Refinement Defined.

Suppose that the states of a data type A (for abstract) are simple and that its operations are easy to understand; but that data type C (for concrete) has complicated states and operations which are hard to understand, though designed to be efficient. If C can be proved to refine A then the user can use the efficient implementation C and rely on the specification A as a clear user manual, in the certainty that in all respects (other than efficiency) C will behave like A. That is the objective of data refinement and the reason why it is so useful.

We now wish to define refinement of data types in terms of their interaction with the user. From the description given in section 1, it is clear that the user of a data type communicates with it by providing input and accepting subsequent output; he cannot directly observe the current state of the data type, and when several states are possible, he cannot control or observe the result of choice between them. We stipulate that the user can only observe

> the name of each operation which has been performed on the data type and the values of the input and output on each occasion
>
> whether the data type breaks (in the sense explained in sections 1.1 and 1.4(6)).

We suppose, by convention (6) of the previous section, that the operations on each data type are total. Then the user would deem C to be a correct 'refinement' of A if C's behaviour were specified by that of A, in the sense that

> the user cannot distinguish by name an operation in A from the corresponding operation in C. For this reason we shall suppose that there is a bijection from AOP to COP, but for clarity we shall use distinct names for corresponding operations: usually aop $\leftrightarrow$ cop;
>
> if a trace of alternating input and output can be communicated between the user and C then it can be communicated between the user and A; writing traces(A) for the set of all traces of alternating input and output communicable by data type A, this becomes
>
> $$\text{traces}(C) \subseteq \text{traces}(A).$$

This brings us to the definition we seek.

Definition 2.1. Suppose that A and C are data types, that there is a bijection aop $\leftrightarrow$ cop between AOP and COP and that each operation is total. We say that C refines A (or that A is refined by C) iff, with the definition of traces(A) given above,

$$\text{traces}(C) \subseteq \text{traces}(A).$$

Unfortunately direct use of the definition of refinement is quite difficult because it is phrased in terms of traces and this necessitates reasoning about sequences of operations of arbitrary length. What we need is some simpler proof method which permits the designer to consider the application of only one operation at a time. Such a method was adapted from automata theory many years ago by Milner (see [Milner, 1971]) for the simulation of processes, and reemployed recently for the same purpose by Park, [Park, 1980], and Milner, [Milner, 1984]. A similar technique was also introduced by Hoare, [Hoare, 1972], and used in VDM (see [Jones, 1980]) to verify an implementation of a data type.

In short, the method consists of replacing Definition 2.1, stated in terms of sequences of operations, by a simulation condition involving only a single occurrence of an operation. The price paid in guaranteeing that the simulation condition is sufficient to imply refinement, is that we must define in advance a simulation relation R between the abstract states and the concrete states. This relation will of course remain as invisible to the user as the states themselves, but it can be regarded as the implementer's documentation that his implementation conforms to the abstract specification.

To motivate the simulation condition, we suppose that data type A does not exhibit wanton nondeterminism before breaking, in the following sense:

Definition 2.2. A data type A is canonical iff

A has an unique initial state

for each operation aop, for each pair of state a and input x? in the domain, pre aop, of aop and for each output y!, there is at most one final state a'

$$(\text{pre aop})(a,x?) \wedge \text{aop}(a,x?,y!,a') \wedge \text{aop}(a,x?,y!,a'') \Rightarrow a' = a''.$$

In other words, before breakage each operation of the data type is functional in its fourth argument. This definition is to be contrasted with that of a nondeterministic data type: for each state a an input x? may elicit several outputs y! and states a' satisfying

$$\text{aop}(a,x?,y!,a').$$

The word 'canonical' is used in this definition because, when expressed as a Communicating Sequential Process, such a data type is in canonical form, i.e. nondeterministic choices are made as late as possible (syntactically, the choice operator is pushed as far as possible into the expression for the process). Though we have only defined a data type by its specification, there is a sense in which a data type's being canonical is a property of the particular specification of type A rather than a property of A itself, and the word 'canonical' seems to reflect this. However we have shied away from using the word 'normal' (although it is more common than 'canonical' in the context of CSP) because it seemed too pejorative; although most specifications we have seen are canonical, we do not wish it to be thought that data types which are not canonical are in any sense strange (see, for example, section 5).

The notion of canonical data type is introduced here because for such types we are able to give a very simple motivation of a simulation condition which is sufficient for data refinement. Also for such types the simulation condition is actually equivalent to refinement. However before justifying these claims we consider some simple examples.

X2.1. Of the following two data types A is canonical and C is not. To apply Definition 2.1 in order to decide whether one is a refinement of the other we need to make the operations total; in fact traces(A+) = traces(C+) so each type refines the other.

A is defined as in X1.3.

C ≙ (CS, CS_0, COP), where

$CS \triangleq \{0,1,2,3,4\}$

$CS_0 \triangleq \{0\}$

COP ≙ {cop}, and cop is defined

cop

ΔCS

x? : {0?,1?}

y! : {0!,1!}

(c=0 ∧ (x?,y!)=(0?,1!) ∧ c'=1)

∨

(c=0 ∧ (x?,y!)=(0?,1!) ∧ c'=2)

∨

(c=1 ∧ (x?,y!)=(0?,0!) ∧ c'=3)

∨

(c=2 ∧ (x?,y!)=(0?,1!) ∧ c'=4)

//

X2.2. Of the following two types C refines A but not conversely, since after comunicating trace (0?,0!), A can break on input 0? whilst C cannot.

A ≙ (AS, AS_0, AOP), where

$AS \triangleq \{0,1,2,3\}$

$AS_0 \triangleq \{0\}$

AOP ≙ {aop}, and aop is defined

aop

ΔAS

x? : {0?}

y! : {0!,1!}

(a=0 ∧ (x?,y!)=(0?,0!) ∧ a'=1)

∨

(a=0 ∧ (x?,y!)=(0?,0!) ∧ a'=2)

∨

(a=1 ∧ (x?,y!)=(0?,1!) ∧ a'=3)

C ≙ (CS, CS_0, COP), where

$CS \triangleq \{0,1,2,3\}$

$CS_0 \triangleq \{0\}$

COP ≙ {cop}, and cop is defined

cop

ΔCS

x? : {0?}

y! : {0!,1!}

(c=0 ∧ (x?,y!)=(0?,0!) ∧ c'=1)

∨

(c=1 ∧ (x?,y!)=(0?,1!) ∧ c'=2)

∨

(c=1 ∧ (x?,y!)=(0?,1!) ∧ c'=3)

//

2.2 Semisimulation.

Now let us suppose that A is a canonical data type and that C refines A. In order to explain the role of the simulation relation R we give a scenario in which pairs (a,c) of abstract and concrete states belonging to R are built up during the parallel evolution of the user with each of the two 'processes' A and C.

We imagine a scribe (a Maxwell daemon in the computer age?) whose task it is to prove that C refines A by 'matching' any communication between the user and C with the same communication between the user and A. He does this by moving from his present state along some arrow in the diagram of data type A, to a subsequent state in such a way that the input and output along that arrow account for the communication between the user and C. In this way the scribe will have demonstrated how C's behaviour is specified by that of A, in spite of the fact that A and C may have different sets AS and CS of states and different operations.

We start at the very beginning. If C has been initialised by virtue of being in an initial state c_0 then A too must be capable of being initialised. Thus the scribe, in order to match the empty trace of communications between the user and C (which results in C being in state c_0) need only take an initial position at the initial state, a_0, of A. He records this fact

$$c \in CS_0 \Rightarrow a_0 R c.$$

Now suppose that C is in state c having communicated trace t with the user and that this has been matched by the scribe who has most recently recorded, for some $a \in AS$,

$$aRc.$$

Suppose that the user now selects x? as input, that C outputs value y! and moves to state c'. Since C refines A and A is normal, either there is an unique state a' such that

$$aop(a,x?,y!,a'),$$

or else aop behaves chaotically (i.e. breaks or is already broken) in which case this relationship holds for each state a'. In the former case the scribe can match the user's latest communication with C by moving from state a to state a' and recording

$$a'Rc',$$

and in the second case he simply records this relationship for each state a'.

When the scribe's task of covering is complete and each trace of communications between the user and C has been considered, he will have determined a relation R between the states of A and the states of C, satisfying an initialisation condition and an inductive condition

$$c \in CS_0 \Rightarrow a_0 R c$$

$$aRc \wedge cop(c,x?,y!,a') \Rightarrow \exists a' \; (\; aop(a,x?,y!,a') \wedge a'Rc' \;).$$

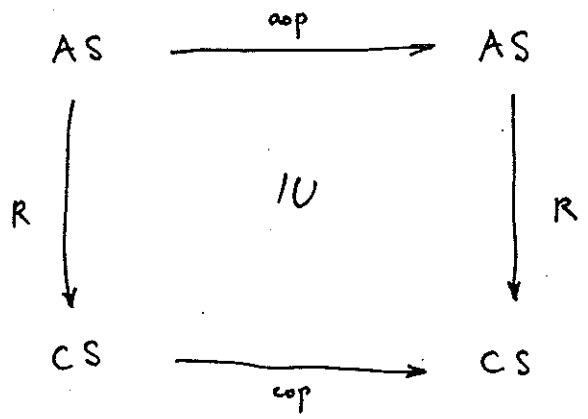
The inductive condition is most succinctly expressed using composition of relations, so it is in that notation that we record the next definition (essentially from [Park,1980]). For simplicity we persist with the assumption that the types each have an unique operation and that A has an unique initial state, though we do not assume that the operations are total.

Definition 2.3. Suppose that A and C are data types with operations aop and cop respectively. A relation $R \subseteq AS \times CS$ is called a simulation (between A and C) iff

$$c \in CS_0 \Rightarrow a_0 R c$$

$$R;cop \subseteq aop;R. \qquad \text{for each } (x?,y!).$$

The inclusion can be depicted by a sub-commuting diagram (that is, a diagram involving composition of relations rather than functions and in which, rather than equality, inclusion holds in the direction of the symbol)



However the notion of simulation is not quite general enough for our needs. Recall that it has arisen in the context of a pair of data types each having an unique operation which has been made total by the construction of 1.4(6), and that A was canonical. All these restrictions have been imposed to simplify the exposition and, as explained in section 1.4, they entail no loss of generality. In order to make the simulation condition roadworthy, for use on examples (in particular those in section 3), we need to remove these restrictions. The result we need in order to undo the + construction is

Lemma 2.1. If A and C are data types with operations aop+ and cop+, respectively, which have been made total by the construction in 1.4(6), and if R is a simulation from A to C then

$$(aRc \wedge (\text{pre } aop)(a,x?)) \Rightarrow (\text{pre } cop)(c,x?)$$

$$(aRc \wedge (\text{pre } aop)(a,x?) \wedge cop(c,x?,y!,c')) \Rightarrow \exists a' (aop(a,x?,y!,a') \wedge a'Rc').$$

Proof. Suppose that aRc, that C is in state c with input x? and that cop breaks or is broken. Then for each y! and each c',

$$cop^+(c,x?,y!,c').$$

So by hypothesis, for each y!,

$$(aRc \wedge \neg(\text{pre } cop)(c,x?)) \Rightarrow \forall c' \exists a' (aop^+(a,x?,y!,a') \wedge a'Rc').$$

But type A is normal so by the construction of R

$$(aRc \wedge \neg(\text{pre } cop)(c,x?)) \Rightarrow \neg(\text{pre } aop)(a,x?)$$

whence

$$(aRc \wedge (\text{pre } aop)(a,x?)) \Rightarrow (\text{pre } cop)(c,x?).$$

Secondly if aRc and neither aop nor cop breaks, the hypothesis shows

$$(aRc \wedge (\text{pre } aop)(a,x?) \wedge cop(c,x?,y!,c')) \Rightarrow \exists a' (aop^+(a,x?,y!,a') \wedge a'Rc')$$

in which aop+ can be replaced by aop on the right-hand side since the antecedent contains the conjunct pre aop. This completes the result. //

Having removed the restriction that the operations in the simulation condition be total, it remains to relax the other two restrictions. The assumption that A have an unique initial state engendered the initial condition

$$c \in CS_0 \Rightarrow a_0 R c$$

which, undoing construction 1.4(5), becomes

$$c \in CS_0 \Rightarrow \exists a \in AS_0 \; a R c.$$

Finally the assumption that each type have an unique operation can be relaxed, as in Definition 2.1, by supposing that there is a bijection aop $\leftrightarrow$ cop from AOP to COP and that the simulation condition works for each pair of corresponding operations.

We are now ready to define the weak simulation condition sufficient for data refinement.

Definition 2.4. Let A and C be data types with a bijection aop $\leftrightarrow$ cop from AOP to COP. A relation $R \subseteq AS \times CS$ is called a **semisimulation** iff, for each aop$\in$AOP,

(R1) $c \in CS_0 \Rightarrow \exists a \in AS_0 \; a R c$

(R2) $(\; a R c \wedge (\text{pre aop})(a, x?) \;) \Rightarrow (\text{pre cop})(c, x?)$

(R3) $(\; a R c \wedge (\text{pre aop})(a, x?) \wedge \text{cop}(c, x?, y!, c') \;) \Rightarrow \exists a' \; (\; \text{aop}(a, x?, y!, a') \wedge a' R c' \;).$

In this case we say C semisimulates A, and write either $A \sqsubseteq C$ or, if we wish to emphasise the relation R, $A \sqsubseteq_R C$.

The conditions for a semisimulation can be re-expressed using schema notation:

(R1') $CS_0 \Rightarrow \exists AS_0 \; R$

(R2') $(\; R \wedge \text{pre aop} \;) \Rightarrow \text{pre cop}$

(R3') $(\; R \wedge \text{pre aop} \wedge \text{cop} \;) \Rightarrow \exists AS' \; (\; \text{aop} \wedge R' \;).$

The suppression of free variables in these formulae and the decoration R' of R might appear strange at first, but they turn out to be very convenient in practice (the examples in section 3 bear this out).

After relaxing the restrictions previously imposed on A and C, the conditions for a simulation become

(S1) = (R1')

(S2) = (R2')

(S3) = $(\; R \wedge \text{cop} \;) \Rightarrow \exists AS' \; (\; \text{aop} \wedge R' \;).$

The reason for the difference between a simulation and a semisimulation is straightforward. Simulation is designed for use with concurrent processes where a data type in a given state can 'refuse' a transaction outside the domain of an operation attempted by one process – or rather delay the operation until a transaction invoked by another concurrent process has brought the data type into a state where the operation is possible. In this case it is vital that the domain of a refinement of an operation coincide with the domain of the operation itself, i.e.

(D) $R \Rightarrow (\; \text{pre aop} \Leftrightarrow \text{pre cop} \;).$

This is a consequence of conditions (S2) and (S3) for a simulation.

On the other hand semisimulation, as it has been defined here, is for use with sequential processes where a data type simply breaks on a transaction with input outside its domain. Clearly such a type can be safely replaced by one which is not as fragile. So it is sufficient that the domain of the concrete type merely includes that of the abstract type, i.e. (R2) holds.

We now consider two vital results about the definition of semisimulation. The first is a consistency result and states that semisimulation implies data refinement.

Theorem 2.1 If C semisimulates A then C refines A:

$$A \sqsubseteq C \Rightarrow traces(C) \subseteq traces(A).$$

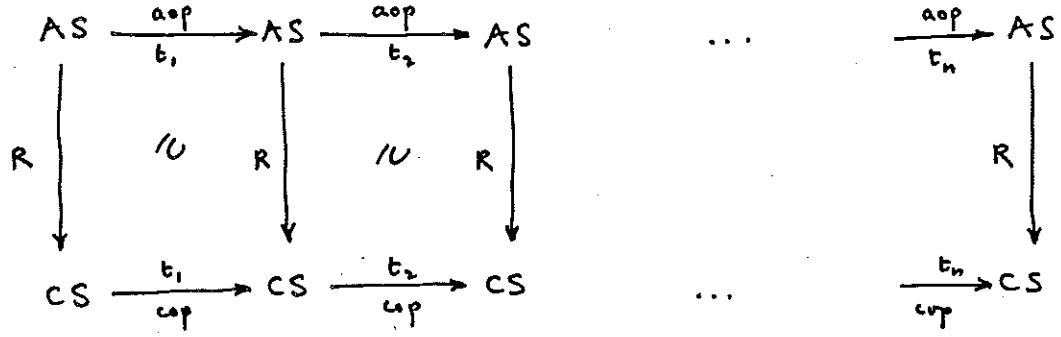
Proof. Suppose that $A \sqsubseteq_R C$ and that $t \in traces(C)$ with

$$t = t_1 \hat{} t_2 \hat{} \ldots \hat{} t_n$$

where $t_j = \langle t_j?, t_j! \rangle$ for $1 \leqslant j \leqslant n$. Then there is a sequence $\langle c_0, c_1, \ldots, c_n \rangle$ of states in C satisfying

$$cop(c_{j-1}, t_j?, t_j!, c_j) \text{ for } 1 \leqslant j \leqslant n.$$

Since $A \sqsubseteq_R C$, there must also be a sequence $\langle a_0, a_1, \ldots, a_n \rangle$ of states in A for which the following squares all sub-commute



hence $t \in traces(A)$ as desired. //

The second result is a completeness one: if A is a canonical data type and C is a refinement of A then the proof obligations embodied in the definition of semisimulation are strong enough for us to be able to prove refinement by meeting these obligations. The reader will be relieved to learn that the proof of this fact has already been covered in our motivation of semisimulation.

Theorem 2.2. If A is a canonical data type and C is a refinement of A then C semisimulates A, i.e.

$$(\ A \text{ canonical} \wedge traces(C) \subseteq traces(A)\ ) \Rightarrow A \sqsubseteq C.$$

Proof. This follows from the scribe construction above. //

2.3 Bisimulation.

The equivalence relation induced by the pre-order relation of refinement between data types is evident:

Definition 2.5. Data Types A and C are said to be inter refinable iff

$$traces(A) = traces(C).$$

The purpose of this section is to consider the equivalence relation induced by the pre-order relation of semisimulation between canonical types. Firstly, semisimulation is indeed a pre-order:

Theorem 2.3. For any data types A, B and C,

$$A \sqsubseteq A$$

$$(A \sqsubseteq B \wedge B \sqsubseteq C) \Longrightarrow A \sqsubseteq C.$$

If $T \subseteq AS \times CS$ enjoys the property that both T and T^{-1} are semisimulations then, for each $aop \in AOP$,

(E1) (a) $CS_0 \Longrightarrow \exists AS_0\ T$

(c) $AS_0 \Longrightarrow \exists CS_0\ T$

(E2) (a) $(T \wedge pre\ aop) \Longrightarrow pre\ cop$

(c) $(T \wedge pre\ cop) \Longrightarrow pre\ aop$

(E3) (a) $(T \wedge pre\ aop \wedge cop) \Longrightarrow \exists AS' (aop \wedge T')$

(c) $(T \wedge pre\ cop \wedge aop) \Longrightarrow \exists CS' (cop \wedge T').$

Since (E2) implies

$$T \Longrightarrow (pre\ aop \Leftrightarrow pre\ cop),$$

the conditions above become those of a bisimulation:

Definition 2.6. If A and C are data types, a bisimulation is a relation $T \subseteq AS \times CS$ satisfying, for each $aop \in AOP$,

(B1) (a) $CS_0 \Longrightarrow \exists AS_0\ T$

(c) $AS_0 \Longrightarrow \exists CS_0\ T$

(B2) (a) $(T \wedge cop) \Longrightarrow \exists AS' (aop \wedge T')$

(c) $(T \wedge aop) \Longrightarrow \exists CS' (cop \wedge T').$

In this case we write $A \simeq_T C$ or just $A \simeq C$ and say that A and C bisimulate each other or are bisimilar.

It seems natural to expect $\simeq$ to be the equivalence induced by $\sqsubseteq$. The corresponding result for simulation would be that if there is a simulation from A to C and a simulation from C to A then there is a bisimulation from A to C - a result which needs qualification (see [Milner, 1984, Lecture 4]). Our interest at present is only in canonical types, in which case we have

Theorem 2.4. If A and C are canonical data types then bisimulation is equivalent to mutual semisimulation

$$A \equiv C \Leftrightarrow (A \sqsubseteq C \wedge C \sqsubseteq A).$$

Proof. If $A \equiv_T C$ then it is routine to check that $A \sqsubseteq_T C$ and $C \sqsubseteq_{T^{-1}} A$.

Conversely if $A \sqsubseteq C$, $C \sqsubseteq A$ and both A and C are canonical, then for any trace (including $\langle\rangle$) of communications between the user and the data type, there is at most one sequence of states in each data type which produces that trace. So the scribe construction, extended to include a scribe in each data type, produces a relation T which is a bisimulation. //

A simple consequence of this result is that, for canonical data types, $\equiv$ is complete in the following sense

Corollary. If A and C are canonical data types then bisimulation is equivalent to inter refinability

$$traces(A) = traces(C) \Leftrightarrow A \equiv C.$$

3. Examples

This section is devoted to examples: section 3.1 contains four small examples and section 3.2 a simplified version of a larger example which arose in practice and motivated the research which led to the semisimulation condition for refinement proposed in this paper.

In X3.1 we consider the familiar example of implementing a bag by a sequence. We observe that, although the two data types are bisimilar in the sense of Definition 2.6, the concrete data type is considered to be an implementation of the abstract one not because it is more deterministic (it is not) but simply because it is closer to that which may be directly implemented on a conventional computer. In X3.2 we modify X3.1 so that the concrete data type refines the abstract one but is not bisimilar to it.

In X3.3 a supermarket wishes to record the smallest number of items bought by any customer. The specification records the number of items purchased by successive customers and outputs the progressive minimum, whilst the refinement records and outputs only the successive minima. Both the specification and its refinement are deterministic but the specification contains extra information not needed by the implementation.

In X3.4 we consider an example in which the implementation is deterministic though the specification is not; yet the two are bisimilar. Again it shows how nondeterminism can arise in practice by the inclusion of extra information at the specification stage.

We see from X3.1 that the relation T in Definition 2.6 (hence also the relation R in Definition 2.5) may be one to many (since each bag is related to all the sequences obtained from it by permuting elements); and from X3.3 we see that T (and hence also R) may be many to one (since many sets have the same minimum value). In general neither T nor T^{-1} (nor R nor R^{-1}) need be functional. We also note that in X3.1, X3.3 and X3.4, A and C are bisimilar types so, by Theorem 2.4, each example actually provides two examples of refinement for the price of one. In practice the designer would only need to prove refinement in one direction by exhibiting a semisimulation from the specification to the implementation.

3.1 Examples Four.

X3.1. Bag and Sequence

In this example we show that the data type 'bags with put and take' can be implemented by a bisimilar data type 'sequences with insert and delete'. For simplicity we omit operations which interrogate the types, so we can ignore output. The abstract data type is represented, as usual,

$$A \triangleq (AS, AS_0, AOP)$$

and its states consist of all (finite) bags of natural numbers with initial state the empty bag $\llbracket\rrbracket$, and its operations are put and take

$$AS \triangleq BAG(\mathbb{N})$$

$$AS_0 \triangleq \{\llbracket\rrbracket\}$$

$$AOP \triangleq \{put, take\},$$

where

put

ΔAS

$x? : \mathbb{N}$

$a' = a \uplus \llbracket x? \rrbracket$

take

ΔAS

$x? : \mathbb{N}$

$a' = a \uplus \{x?\}$

Herein $\uplus$ and $\setminus\!\!\setminus$ denote bag union and difference respectively, and $\{x?\}$ denotes the bag containing only a single occurrence of x?. Note that both operations are total: if an element x does not belong to a bag b then the bag difference $b \setminus\!\!\setminus \{x?\}$ equals b.

We implement a bag by a sequence which records the members of the bag in some order. The initial sequence is the empty sequence. Our idea is thus that a bag b will be represented by a sequence s satisfying

$b = \text{bagrange } s$

where the right-hand side denotes the range of s treated as a bag, that is, including repetitions. In other words we take

$C \triangleq (CS, CS_0, COP)$

whose states consist of finite $\mathbb{N}$-valued sequences, with initial state the empty sequence, and with operations insert and delete

$CS \triangleq SEQ(\mathbb{N})$

$CS_0 \triangleq \{\langle\rangle\},$

$COP \triangleq \{\text{insert}, \text{delete}\},$

where

insert

ΔCS

$x? : \mathbb{N}$

$c' = c \frown \langle x? \rangle$

delete

ΔCS

$x? : \mathbb{N}$

$(\exists s,t)(\ c = s \frown \langle x? \rangle \frown t \wedge c' = s \frown t\ )$

Again the operations insert and delete are total.

In order to demonstrate that A and C are bisimilar we define T by

$aTc \triangleq (\ a = \text{bagrange } c\ ).$

(This definition, suggested above, naturally coincides with that of the scribe construction.)

It remains to verify (B1) and (B2) of Definition 2.6. Since

$\biguplus T\langle\rangle$,

(B1) is obvious. The proofs of (B2) for the cases

(aop,cop) = (put,insert) and

(aop,cop) = (take,delete)

are similar, so we shall leave the latter case for the reader.

Firstly (B2)(a) : if

$(a = \text{bagrange } c) \wedge (a' = a \uplus \{\!| x? |\!\})$

then setting

$c' \triangleq c^\frown\langle x?\rangle$

yields

$(c' = c^\frown\langle x?\rangle) \wedge (a' = \text{bagrange } c')$.

Finally (B2)(c) : if

$(a = \text{bagrange } c) \wedge (c' = c^\frown\langle x?\rangle)$

then setting

$a' \triangleq a \uplus \{\!| x? |\!\}$

yields

$(a' = a \uplus \{\!| x? |\!\}) \wedge (a' = \text{bagrange } c')$.

//

X3.2. Bag and Sequence Revisited.

In certain circumstances it is appropriate to weaken the specification of the operation 'take' in X3.1 by including the precondition 'x? ∈ a' as follows

take

ΔAS

$x? : \mathbb{N}$

$x? \in a$

$a' = a \uplus \{\!| x? |\!\}$

Now the argument of X3.1 shows that $A \sqsubseteq_T C$, although it is no longer true that $A \equiv_T C$ because (E2)(c) fails.

X3.3. The Supermarket.

A supermarket wishes to record, for each day, the smallest number of items bought by any customer. The supermarket has one checkout and is closed on holidays. One specification is obtained by, starting from the empty set, adding to the set the number of items purchased by successive customers and outputting progressive minima. Thus

$$A \triangleq (AS, AS_0, AOP)$$

where

$$AS \triangleq \mathbb{P}(\mathbb{N})$$

$$AS_0 \triangleq \{\}$$

$$AOP \triangleq \{insert\},$$

with

insert

ΔAS

$x?, y! : \mathbb{N}$

$a' = a \cup \{x?\}$

$y! = minimum\ a'$

This specification reflects considerations of how the minimum is to be calculated. In general, such operational specifications are not to be encouraged in Z, however one reason for using them might be that A is one module from a whole system dealing with customers and their purchases, and that the rest of the system uses information about the states a and the non terminal outputs y!. We imagine that the system developer, realising that such non terminal information is unnecessary for the module he is to implement, defines a state to consist of minimum value so far:

$$C \triangleq (CS, CS_0, COP)$$

with

$$CS \triangleq \mathbb{N} \cup \{undefined\}$$

$$CS_0 \triangleq \{undefined\}$$

$$COP \triangleq \{min\},$$

where

min

ΔCS

$x?, y! : \mathbb{N}$

$y! = c' = minimum\ \{c, x?\}$

In other words, no more information is kept than is necessary to determine the successive minima. The element 'undefined' is used to initialise this progressive minimum (in particular it is the final value obtained on holidays). We show that $A \equiv_T C$ where

$$aTc \triangleq (a \neq \{\} \wedge c = \text{minimum } a) \vee (a = \{\} \wedge c = \text{undefined}).$$

Since {}T(undefined), (B1) is obvious. For (B2)(a), if

$$aTc \wedge (y! = c' = \text{minimum } \{c, x?\})$$

then taking

$$a' \triangleq a \cup \{x?\}$$

it follows that

$$\text{insert}(a, x?, y!, a') \wedge a'Tc'.$$

Finally, for (B2)(c), if

$$aTc \wedge (a' = a \cup \{x?\} \wedge y! = \text{minimum } a')$$

then taking

$$c' \triangleq \text{minimum } \{c, x?\}$$

it follows that

$$y! = c' = \text{minimum } \{c, x?\}.$$

//

X3.4. The Garage

A garage, starting with no cars, buys and sells cars without bound on the number stocked at any time. For each model of car, neither the garage nor the buyer cares to distinguish between the cars of that model. Also, the employees of the garage are permitted to use the cars at lunchtime and at night time, without necessarily returning them to their original parking places. In specifying the garage, states are partial functions from the set LOC of all possible locations of cars to the set CAR of models of car, and the initial state is the empty function

$$A \triangleq (AS, AS_0, AOP)$$

with

$$AS \triangleq LOC \nrightarrow CAR$$

$$AS_0 \triangleq \{0\}$$

where $S \nrightarrow T$ denotes the set of partial functions from S to T and where 0 is the function identically equal to zero. Next

$$AOP \triangleq \{abuy, asell\}$$

(named from the point of view of the garage) with, again writing $\uplus$ and $\uplus$ for bag union and difference respectively,

abuy

ΔAS

$x? : CAR$

$bagrange\ a' = (bagrange\ a) \uplus \lbag x? \rbag$

asell

ΔAS

$x? : CAR$

$y! : (trans, notrans)$

$(x? \in bagrange\ a \land bagrange\ a' = (bagrange\ a) \uplus \lbag x? \rbag \land y! = trans)$

$\lor$

$(x? \notin bagrange\ a \land a' = a \land y! = notrans)$

This specification, because it is couched in terms of the locations of the cars, is forced to be highly nondeterministic in order to model the system requirements. Firstly a new car may be stored at *any* empty location. Secondly the condition that the cars may be moved, prohibits the predicate of schema asell from tying cars to locations as in the following (mildly more deterministic) predicate

$$(\exists loc \in LOC \setminus dom\ a)(a' = a \cup \{(loc, x?)\} \land y! = trans).$$

Thirdly during a sale *any* car of a certain model may be deleted from the garage. However this nondeterminism is wholly unapparent to the garage's customers.

The refinement is deterministic. States are functions from CAR to $\mathbb{N}$ which keep count of the number of each make of car on sale and the initial state is the zero function. Again there are two operations, buy and sell

$$C \triangleq (CS, CS_0, COP)$$

with

$$CS \triangleq CAR \rightarrow \mathbb{N}$$

$$CS_0 \triangleq \{0\}$$

$$COP \triangleq \{cbuy, csell\},$$

where

cbuy

ΔCS

$x? : CAR$

$c' = c \oplus \{(x?, c(x?)+1)\}$

csell

ΔCS

x? : CAR

y! : {trans, notrans}

(c(y!) > 0 ∧ c' = c ⊕ {(x?,c(x?)-1)} ∧ y! = trans)

∨

(c(y!) = 0 ∧ c' = c ∧ y! = notrans)

where ⊕ denotes overriding of functions, defined by

$$(f \oplus g)(n) = \begin{cases} f(n) \text{ if } n \in \text{dom } f - \text{dom } g \\ g(n) \text{ if } n \in \text{dom } g \end{cases}$$

Now to see that $A \equiv_T C$, we define

$$aTc \triangleq (\forall z \in CAR)(\#\{loc : (loc,z) \in a\} = c(z))$$

where, if S is a finite set, #(S) equals the number of elements of S.

Clearly {}T0 so condition (B1) holds. Next, (B2)(a) for the pair of operations (abuy,cbuy): if

$$aTc \wedge cbuy(c,x?,y!,c')$$

then, since there is always some $\lambda \in LOC \neg (range\ a)$, it suffices, for such a λ, to take

$$a' \triangleq a \cup \{(\lambda,x?)\}$$

in order to ensure that

$$abuy(a,x?,y!,a') \wedge a'Tc'.$$

Next, (B2)(c) for the same pair of operations: if

$$aTc \wedge abuy(a,x?,y!,a')$$

then it suffices to take

$$c' \triangleq c \oplus \{(x?,c(x?)+1)\}$$

in order to deduce

$$cbuy(c,x?,y!,c') \wedge a'Tc'.$$

We leave as an exercise the proof of (B2) for the pair of operations (asell,csell).

3.2 Example Five.

This example is a simplified version of one being developed by Paul Mundy from IBM Hursley, and for which the extant refinement rules were found to be insufficient. In presenting an abstract data type for storage management and a particular design which refines it, we follow our standard practice for examples of this size and intersperse the description of the design with proofs that it satisfies the specification.

Specification

The abstract data type manages allocation of contiguous blocks of storage. It has one operation, AGet, to get a block of storage of a given size and one operation, AFree, to free a previously allocated block. We write n..m for the interval

$$\{i \in \mathbb{N} : n \leqslant i \leqslant m\}$$

and represent addresses as natural numbers up to some given maximum

$$maxaddr : \mathbb{N},$$

so that addresses lie in the set

$$ADDR \triangleq 0..maxaddr.$$

A block of storage consists of a contiguous range of addresses, i.e. is an element of

$$Block \triangleq \{\ base..top : base, top \in ADDR \ \wedge \ base \leqslant top\ \}$$

and the length of a block of storage, the number of its consecutive addresses, is a non-zero natural number in the interval

$$LEN \triangleq 1..maxaddr+1.$$

The (abstract) state of allocated storage is a set of allocated blocks, no two of which overlap

Allocated

$$alloc : \mathbb{P}(Block)$$

$$\underline{disjoint}\ alloc$$

where, for future use, we find it convenient to have a name for the predicate of this schema so we define

$$\underline{disjoint}\ alloc \triangleq \forall b_1, b_2 \in alloc \ (\ b_1 \neq b_2 \Rightarrow b_1 \cap b_2 = \{\}\ )\ .$$

Initially no storage blocks have been allocated

$Allocated_0$

$$Allocated$$

$$alloc = \{\}$$

Each operation transforms a state before (alloc) into a state after (alloc')

$$\Delta Allocated \triangleq Allocated \wedge Allocated'.$$

Operation AGet accepts as input the size s? of storage requested. If this is available it adds a newly allocated block to the set of existing blocks, which the new block must not overlap, and it outputs an acknowledgement rep! = OK; otherwise it outputs rep! = NOSPACE

```
AGet
ΔAllocated
s?   : LEN
b!   : Block
rep! : {OK,NOSPACE}
-----------------------------
( #b!=s?                  ∧
  ∀b∈alloc ( b∩b!={} ) ∧
  alloc' = alloc∪{b!}  ∧
  rep! = OK )
∨
( alloc' = alloc  ∧
  rep! = NOSPACE )
```

Operation AFree deletes a block b? from the set of allocated blocks

```
AFree
ΔAllocated
b? : Block
-----------------------------
b? ∈ alloc
alloc' = alloc - {b?}
```

Design

We now give a simple design (it is of too high a level to be called an implementation) of the storage management system that allocates storage sequentially without ever reusing blocks that are freed. In spite of its inefficient use of space the design is nontheless a valid refinement of the specification and has the advantage of being time efficient.

We assume that the design has available max units of storage with addresses 0..max-1, where

max : ADDR.

A concrete state is represented by the address, or level, at and above which addresses are free for allocation and below which no such assurance can be given; of course level cannot exceed max

```
Available
level : 0..max
```

Initially the whole of storage is available

```
Available0
Available
-----------------------------
level = 0
```

The semisimulation, R, between an abstract state (alloc) and a concrete state (level) is given by

$$alloc\ R\ level \ \hat{=}\ (\cup alloc) \subseteq 0..level-1$$

where

$$(\cup alloc) \ \hat{=}\ \{\ a \in ADDR : \exists b \in alloc\ (\ a \in b\ )\ \}.$$

To verify (R1) of Definition 2.4 we must show

$$Available_0 \ \Rightarrow\ \exists Allocated_0\ R.$$

That is we must show, given

$$level : 0..max,$$

that

$$level = 0$$
$$\Rightarrow$$
$$\exists alloc \in \mathbb{P}(Block)\ (\ \underline{disjoint}\ alloc \wedge\ alloc = \{\} \wedge\ (\cup alloc) \subseteq 0..level-1\ ).$$

The consequent is obviously satisfied by, and only by,

$$alloc = \{\}$$

and so the implication holds.

The design, or concrete type, also has two operations: CGet and CFree. We first consider CGet. When a block of storage is requested it is allocated from the current level upwards and OK reported, unless this would exceed the maximum level, in which case the level remains unaltered and NOSPACE is reported

CGet

$\Delta Available$
$s? : LEN$
$b! : Block$
$rep! : \{OK, NOSPACE\}$

$(\ level+s? \leqslant max \wedge\ level' = level+s? \wedge\ b! = level..level+s?-1 \wedge\ rep! = OK\ )$
$\vee$
$(\ level+s? > max \wedge\ level' = level \wedge\ rep! = NOSPACE\ )$

To show that CGet refines AGet we prove

$$R \wedge \text{pre AGet} \Rightarrow \text{pre CGet}$$

$$R \wedge \text{pre AGet} \wedge \text{CGet} \Rightarrow \exists \text{Allocated}' \ (\text{AGet} \wedge R').$$

As both AGet and CGet are total operations it suffices to show

$$R \wedge \text{CGet} \Rightarrow \exists \text{Allocated}' \ (\text{AGet} \wedge R').$$

Given

```
alloc  : ℙ(Block)
level  : 0..max
level' : 0..max
s?     : LEN
b!     : Block
rep!   : {OK,NOSPACE}
```

we must show

```
disjoint alloc          ∧
(∪ alloc) ⊆ 0..level-1  ∧
( ( level+s? ⩽ max          ∧
    level' = level + s?     ∧
    b! = level..level+s?-1  ∧
    rep! = OK )
∨ ( level+s? > max  ∧
    level' = level  ∧
    rep! = NOSPACE )
)
⇒
∃alloc' ∈ ℙ(Block) ( ( disjoint alloc'           ∧
                       (∪ alloc') ⊆ 0..level'-1  ∧
                       #b! = s?                  ∧
                       ∀b ∈ alloc ( b∩b! = {} )  ∧
                       alloc' = alloc ∪ {b!}     ∧
                       rep! = OK )
                   ∨ ( alloc' = alloc ∧
                       rep! = NOSPACE )
                   ).
```

Fortunately we can split this into two cases: succesful (rep! = OK) and unsuccessful (rep! ≠ OK).

(a) rep! = OK.

```
disjoint alloc         ∧
(∪ alloc) ⊆ 0..level-1 ∧
level + s? ⩽ max       ∧
level' = level + s?    ∧
b! = level..level+s?-1
⇒
∃alloc' ∈ ℙ(Block) ( disjoint alloc'           ∧
                     (∪ alloc') ⊆ 0..level'-1  ∧
                     #b! = s?                  ∧
                     ∀b ∈ alloc (b∩b! = {})    ∧
                     alloc' = alloc ∪ {b!}
                   )
```

This can be satisfied by choosing

$$alloc' = alloc \cup \{b!\}.$$

We must then show

$$\begin{array}{l} \underline{disjoint}\ alloc \quad \wedge \\ (\cup\ alloc) \subseteq 0..level-1 \quad \wedge \\ level + s? \leqslant max \quad \wedge \\ level' = level + s? \quad \wedge \\ b! = level..level+s?-1 \\ \Rightarrow \\ \underline{disjoint}\ (\ alloc \cup \{b!\}\ ) \quad \wedge \\ \cup(alloc \cup \{b!\}) \subseteq 0..level'-1 \quad \wedge \\ \#b! = s? \ \wedge \\ \forall b \in alloc\ (\ b \cap b! = \{\}\ ). \end{array}$$

From

$$(\cup\ alloc) \subseteq 0..level-1 \quad \text{and} \quad b! = level..level+s?-1$$

we conclude

$$\forall b \in alloc\ (\ b \cap b! = \{\}\ ).$$

From this and disjoint alloc we can deduce

$$\underline{disjoint}\ (alloc \cup \{b!\}).$$

As

$$(\cup\ alloc) \subseteq 0..level-1 \subseteq 0..level'-1$$

and

$$level..level+s?-1 \subseteq 0..level'-1$$

we can thus deduce

$$\cup(alloc \cup \{b!\}) = (\cup\ alloc) \cup \{b!\} \subseteq 0..level'-1.$$

(b) $rep! \neq OK$.

$$\begin{array}{l} \underline{disjoint}\ alloc \quad \wedge \\ (\cup\ alloc) \subseteq 0..level-1 \quad \wedge \\ level + s? > max \quad \wedge \\ level' = level \quad \wedge \\ rep! = NOSPACE \\ \Rightarrow \\ \exists alloc' \in \mathbb{P}(Block)\ (\ \underline{disjoint}\ alloc' \quad \wedge \\ \qquad (\cup\ alloc') \subseteq 0..level'-1 \quad \wedge \\ \qquad alloc' = alloc \quad \wedge \\ \qquad rep! = NOSPACE \\ \qquad). \end{array}$$

Taking $alloc' = alloc$, (b) follows from the fact that $level' = level$.

The second concrete operation, CFree, frees a block by doing nothing; no advantage is taken of being able to reallocate freed storage

```
CFree
ΔAvailable
b? : Block
---
level' = level
```

To show CFree is a refinement of AFree we prove

$$R \wedge \text{pre AFree} \Rightarrow \text{pre CFree}$$

$$R \wedge \text{pre AFree} \wedge \text{CFree} \Rightarrow \exists \text{Allocated}' \bullet (\text{AFree} \wedge R').$$

As CFree is total the first implication is trivial. To prove the second we assume

$$\begin{aligned} &alloc : \mathbb{P}(Block) \\ &level : 0..max \\ &level' : 0..max \\ &b? : Block \end{aligned}$$

and must show

$$\begin{aligned} &\underline{disjoint}\ alloc \wedge \\ &(\cup alloc) \subseteq 0..level-1 \wedge \\ &b? \in alloc \wedge \\ &level' = level \\ \Rightarrow\ &\exists alloc' \in \mathbb{P}(Block) \bullet (\underline{disjoint}\ alloc' \wedge \\ &\qquad (\cup alloc') \subseteq 0..level'-1 \wedge \\ &\qquad b? \in alloc \wedge \\ &\qquad alloc' = alloc - \{b?\}) \end{aligned}$$

Taking alloc' = alloc - {b?} it remains to show

$$\begin{aligned} &\underline{disjoint}\ alloc \wedge \\ &(\cup alloc) \subseteq 0..level-1 \wedge \\ &level' = level \\ \Rightarrow\ &\underline{disjoint}\ (alloc - \{b?\}) \wedge \\ &\cup(alloc - \{b?\}) \subseteq 0..level'-1. \end{aligned}$$

But from $\underline{disjoint}$ alloc we can deduce $\underline{disjoint}$ (alloc - {b?}) and finally

$$\cup(alloc - \{b?\}) \subseteq (\cup alloc) \subseteq 0..level-1 = 0..level'-1.$$

4. Comparison

In this section we compare the semisimulation condition, given in Definition 2.4, with the sufficient condition for data refinement used in VDM and with the "more deterministic than" ordering of Dijkstra and Smyth. We observe that semisimulation is more general than the VDM rule: it always applies when the VDM rule does, and may apply in situations where it fails. We also note that in examples where both rules hold, a semisimulation R is usually simpler than a relation satisfying the VDM rule but that when R satisfies some of the further properties required by the VDM rule, the two coincide.

4.1 The VDM Rule.

In our notation the VDM rule for data refinement (from [Jones, 1985]; this supersedes the version in [Jones, 1980, Chapter 11, Figure 49], by the inclusion of the precondition term in the antecedent of (ra)) is

Definition 4.1. Suppose that A and C are data types with a bijection aop $\leftrightarrow$ bop from AOP to COP. The VDM rule for refinement of A by C comprises a relation $R \subseteq AS \times CS$ satisfying, for each aop $\in$ AOP,

(zz) R is injective : $a_1 R c \wedge a_2 R c \Rightarrow a_1 = a_2$

(aa) R is surjective : $\forall c \in CS\ \exists a \in AS\ aRc$

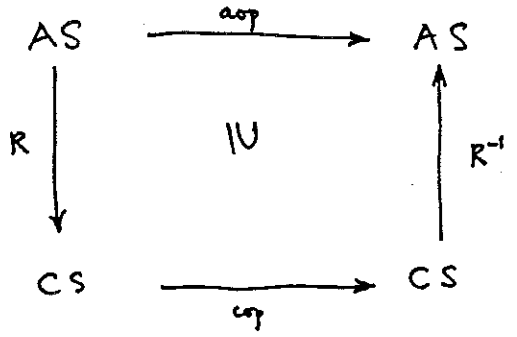
(ab) R is total : $\forall a \in AS\ \exists c \in CS\ aRc$

(da) condition (R2) in Definition 2.4 holds :

$$(aRc \wedge (\text{pre aop})(a,x?)) \Rightarrow (\text{pre cop})(c,x?)$$

(ra) $(aRc \wedge (\text{pre aop})(a,x?) \wedge \text{cop}(c,x?,y!,c') \wedge a'Rc') \Rightarrow \text{aop}(a,x?,y!,a')$.

Condition (ra) contrasts with condition (R3) in Definition 2.4, as the following diagram shows



There is no mention of initial states here: the approach taken in VDM is that each data type has an initialisation operation having precondition *true* and having postcondition expressing membership of the set of desired initial states.

Jones also includes a condition in (aa) guaranteeing that the invariant of data type A is maintained. His condition is, in our notation,

(aa)' $\forall c \in CS\ (\text{inv}_C(c) \Rightarrow \exists a \in AS\ (aRc \wedge \text{inv}_A(a)))$

where inv_A denotes the invariant of type A. However since our semisimulation condition only involves states reachable from the initial states, and since these are included in those states which satisfy the invariant, we have replaced (aa)' by (aa).

How do the two rules compare? Firstly the semisimulation rule applies whenever the VDM one does.

Theorem 4.1. If a relation R satisfies the VDM rule then it is a semisimulation.

Proof. Suppose that R satisfies the VDM rule. Applying (ra) to the initialisation operations we deduce the initialisation condition (R1) of Definition 2.4. It remains to establish (R3). If

$$aRc \wedge (\text{pre } aop)(a,x?) \wedge cop(c,x?,y!,c')$$

then since R is surjective

$$\exists a' \in AS \; a'Rc'.$$

So from (ra)

$$aop(a,x?,y!,a')$$

holds as well as a'Rc', as required. //

Secondly the semisimulation rule is strictly more general. Indeed it is evident by now that the VDM rule for refinement is more restrictive than it need be. Condition (R3) in Definition 2.4 is weaker than condition (ra) and moreover a semisimulation is not assumed to be total, injective or surjective. In particular in both X3.3 and the storage management example of section 3.2, we have already met examples which cannot be verified using the VDM rule. This is the case whenever distinct abstract states correspond to the same concrete state, i.e., in VDM terms, whenever A is biased. The failure of the rules is then regarded as an indication that the bias is a fault in the specification, and should be eliminated by modifying the specification.

Our view is slightly different. For reasons to be explained in X5.1 (but which have already become apparent in the two examples referred to above) we consider biased specifications to be sometimes useful, so it is essential to have a refinement rule strong enough to verify them.

Here is an example which makes this point. Though not the simplest possible (it will be reused in section 5) it is simpler than those mentioned above.

X4.1 . $A \triangleq (AS, AS_0, AOP)$, where

$AS \triangleq \{0,1,2,3,4,5,6,7\}$

$AS_0 \triangleq \{0\}$

$AOP \triangleq \{aop\}$, and aop is defined

aop

```
ΔAS
x? : {0?,1!}
y! : {0!,2!,3!}
-----------------------------------------
 (a=0 ∧ (x?,y!)=(0?,0!) ∧ a'=1)
∨
 (a=0 ∧ (x?,y!)=(0?,0!) ∧ a'=2)
∨
 (a=0 ∧ (x?,y!)=(0?,0!) ∧ a'=3)
∨
 (a=1 ∧ (x?,y!)=(1?,2!) ∧ a'=4)
∨
 (a=2 ∧ (x?,y!)=(1?,2!) ∧ a'=5)
∨
 (a=2 ∧ (x?,y!)=(1?,3!) ∧ a'=6)
∨
 (a=3 ∧ (x?,y!)=(1?,3!) ∧ a'=7)
```

...

C ≙ (CS,CS_0,COP), where

CS ≙ {0,1,2,3,4,5}

CS_0 ≙ {0}

COP ≙ {cop}, and cop is defined

cop

```
ΔCS
x? : {0?,1!}
y! : {0!,2!,3!}
-----------------------------------------
 (c=0 ∧ (x?,y!)=(0?,0!) ∧ c'=1)
∨
 (c=0 ∧ (x?,y!)=(0?,0!) ∧ c'=2)
∨
 (c=1 ∧ (x?,y!)=(1?,2!) ∧ c'=3)
∨
 (c=2 ∧ (x?,y!)=(1?,3!) ∧ c'=4)
```

Since traces(A+) = traces(C+), A and C are inter refinable. A semisimulation from A to C is given by the relation R ⊆ AS×CS

0R0, 1R1, 4R3, 3R2, 7R4.

However the VDM rule obviously does not apply to this example. (Observe that a semisimulation from A to C exists even though type A is not canonical. None exists from C to A but the reader who perseveres to section 5 will be rewarded with a rule more general than semisimulation which enables refinement to be demonstrated in this direction too.) //

it doesn't really follow.

It is interesting to observe that when both rules hold, the smallest semisimulation is smaller, and hence more simple to construct and reason about, than the smallest VDM relation which has to be total and surjective. It also seems to be easier, in general, to find a semisimulation relation R than to satisfy all the VDM conditions. But when a semisimulation R is a total bijection (that is, total, injective and surjective) the two rules coincide. By Theorem 4.1 it suffices to prove

Theorem 4.2. A total, bijective semisimulation satisfies the VDM rule.

Proof. It remains to establish (ra). If

$$aRc \wedge (pre\ aop)(a,x?) \wedge cop(c,x?,y!,c') \wedge a'Rc'$$

then by (R3)

$$\exists a''\ (\ aop(a,x?,y!,a'') \wedge a''Rc'\ )$$

so that by injectivity of R

$$a = a''$$

hence

$$aop(a,x?,y!,a')$$

as desired. //

We omit the simple examples which demonstrate the necessity of each of the hypotheses in Theorem 4.2.

Tobias Nipkow at The University of Manchester has also extended the VDM rule for refinement to obtain a rule which is different from semisimulation but which no longer assumes that R need be injective or surjective. Nipkow's motivation (see [Nipkow, 1985]) lies more in the application to concurrency; thus his rule satisfies, as does simulation, condition (D) in section 2.2.

4.2 The "more deterministic than" Ordering.

The special case of a semisimulation coincident with the identity relation is responsible for much of our intuition about more general semisimulations.

Definition 4.2. If V,W are sets and $P,Q \subseteq V \times W$ then Q is a relational refinement of P iff Q has domain at least as large as that of P and, when restricted to the domain of P, is more deterministic than P

$$dom(P) \subseteq dom(Q)$$

$$Q \upharpoonright dom(P) \subseteq P.$$

We can now compare semisimulation with the "more deterministic than" ordering of [Dijkstra, 1976] and [Smyth,1977]. The proof is trivial.

Theorem 4.3. $A \sqsubseteq_I C$ iff cop is a relational refinement of aop.

5. Completing the Rule

In this section we stretch the semisimulation condition to cover types which are not canonical. The only impediment to the scribe construction of section 2.2 (which led to semisimulation) is that when aop is not functional in a' the scribe has a choice of with which state to match a communication between the user and C. This point is made by X2.2. In matching the communication (0?,0!) the scribe has a choice between states a=1 and a=2, which subsequently lead to quite different behaviour: the former permits the scribe to match the communication (0?,1!) but the latter deadlocks. Until the second communication the scribe has no reason to favour one choice over another. The construction in the present section thus allows him to relate a nonempty set of abstract states to a concrete state. We find it convenient to write $\mathbb{P}^{\cdot}(X)$ for the set of all nonempty subsets of set X.

5.1 Power Semisimulation.

We rework the scribe construction of section 2.2. If C has been initialised to state $c \in CS_0$ then the scribe can match communication of the empty trace between the user and C by starting in any of A's initial states. Thus the scribe relates c to some subset, yet to be fully determined, of AS_0:

$$c \in CS_0 \Rightarrow (\exists \alpha \in \mathbb{P}^{\cdot}(AS_0))\ \alpha R c.$$

Suppose now that C is in state c, having communicated trace t with the user and that the scribe has most recently related

$$\alpha R c.$$

By this we mean that α is a nonempty set of abstract states which the scribe uses to match the user's communication of trace t with C. Suppose that the user selects x? as input, and that C outputs y! and moves to state c'. Since C refines A, there is a nonempty set α' of abstract states which the scribe could use to match the user's communication of $t^{\frown}\langle x?,y!\rangle$ with C. Moreover each state in α' must be able to be reached by moving along an arrow (in the diagram of type A) starting at a state in α

$$(\ \forall a' \in \alpha'\ )(\ \exists a \in \alpha\ )\ aop(a,x?,y!,a')$$

or, more succinctly,

$$aop^{-1}{}_z[\![\alpha']\!] \subseteq \alpha$$

where

$$z = \langle x?,y!\rangle \text{ and}$$

$$aop^{-1}{}_z[\![\alpha']\!] = \{\ a \in \alpha\ :\ (\exists a' \in \alpha')\ aop(a,x?,y!,a')\ \}.$$

The scribe records

$$\alpha' R c',$$

where α' is yet to be fully determined.

When all ways of communicating all possible traces between the user and C have been matched, the scribe will have determined a relation

$$R \subseteq \mathbb{P}^{\cdot}(AS)\times CS$$

satisfying an initial condition and an inductive one

$$c\in CS_0 \Rightarrow (\exists \alpha\in\mathbb{P}^{\cdot}(AS_0)) \ \alpha R c$$

$$(\alpha R c \wedge cop(c,x?,y!,c')) \Rightarrow (\exists \alpha'\in\mathbb{P}^{\cdot}(AS)) (aop^{-1}_{z}(\alpha') \subseteq \alpha \wedge \alpha' R c').$$

The scribe's choice has been left partly open so any choice of α and α' which satisfies these conditions will do.

As in section 2, by removing the assumption that types A and C have a single, total operation, the construction above leads us to a sufficient condition for refinement: this time without the assumption that A be canonical. However it may not be as evident as it was in the case A canonical that these conditions do indeed imply refinement; this is proved in Theorem 5.1.

Definition 5.1. If A and C are data types and there is a bijection aop↔cop from AOP to COP, a relation

$$R \subseteq \mathbb{P}^{\cdot}(AS)\times CS$$

is called a **power semisimulation** (or we say C power semisimulates A) iff, for each $aop\in AOP$,

$$c\in CS_0 \Rightarrow (\exists \alpha\in\mathbb{P}^{\cdot}(AS_0)) \ \alpha R c$$

$$(\alpha R c \wedge (\exists a\in\alpha)\ (pre\ aop)(a,x?)) \Rightarrow (pre\ cop)(c,x?)$$

$$(\alpha R c \wedge (\exists a\in\alpha)\ (pre\ aop)(a,x?) \wedge cop(c,x?,y!,c'))$$

$$\Rightarrow (\exists \alpha'\in\mathbb{P}^{\cdot}(AS)) (aop^{-1}_{z}(\alpha') \subseteq \alpha \wedge \alpha' R c').$$

Evidently when A is canonical each set α can only be a singleton and hence these conditions reduce to those for a semisimulation. However, in general, any attempt to recast power semisimulation in terms of a relation from AS to CS involves the introduction of backtracking. This defeats the purpose of having a simulation-like condition for refinement and explains our preference for the version defined above.

At first sight power semisimulation might appear too general, so the following consistency result is reassuring.

Theorem 5.1. If C power semisimulates A then C refines A.

Proof. We proceed as in Theorem 2.1. Suppose $t \in traces(C)$ and

$$t = t_1{}^\wedge t_2{}^\wedge \ \ldots \ {}^\wedge t_n$$

where $t_j = \langle t_j?, t_j! \rangle$ for $1\leqslant j\leqslant n$. Then there is a sequence $\langle c_0, c_1, \ldots, c_n \rangle$ of states in C satisfying

$$cop(c_{j-1}, t_j?, t_j!, c_j) \text{ for } 1\leqslant j\leqslant n.$$

Since C power semisimulates A, there must also be a sequence

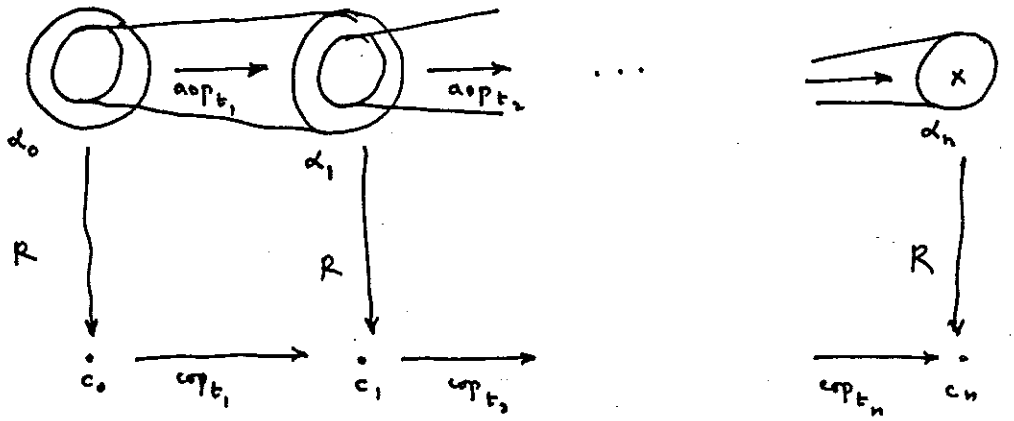
$$\langle \alpha_0, \alpha_1, \ldots, \alpha_n \rangle$$

in $\mathbb{P}^+(AS)$, for which

$$\alpha_0 \subseteq AS_0$$

$$aop^{-1}_{t_j}(\alpha_j) \subseteq \alpha_{j-1} \text{ for } 1 \leq j \leq n$$

$$\alpha_j R c_j$$



Since $\alpha_n \neq \{\}$ we deduce that $t \in traces(A)$ as desired. //

The proof obligation embodied in the definition of power semisimulation is complete, due to

Theorem 5.2. If A is a data type and C refines A then C power semisimulates A.

Proof. It has to be proved that the scribe construction described above, with its undetermined choice of sets of abstract states, is possible. For this it suffices to let

$$\alpha_0 \triangleq AS_0$$

and henceforth insist that the scribe choose the largest possible set of abstract states at each move

$$\alpha' = aop_z(\alpha)$$

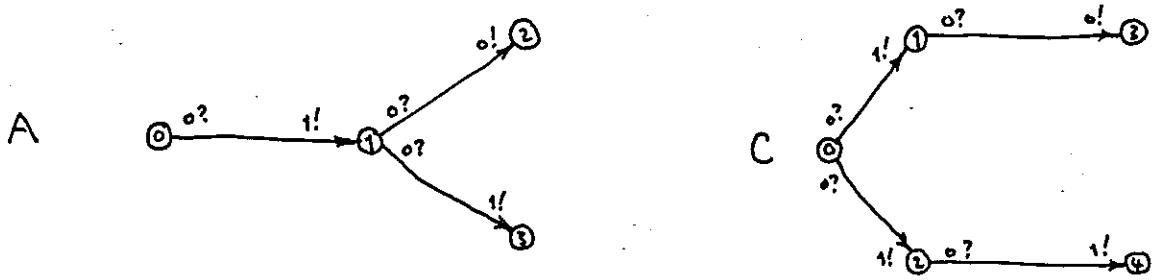
$$\triangleq \{ a' \in AS : (\exists a \in AS)\ aop(a, x?, y!, a') \}.$$

Since C refines A, α' is nonempty. Thus the resulting relation R is a power semisimulation. //

5.2 Examples.

Of the following examples, X5.1 shows how power semisimulation works between the simplest data types for which semisimulation fails, and includes an explanation of why such cases arise in practice. X5.2 illustrates the fact that several power semisimulations may exist between two types; here the smallest one is actually a semisimulation.

X5.1. Let types A and C be defined as in X2.1



Evidently traces(A+) = traces(C+) so A and C are inter refinable. Indeed the relation Q from AS to CS given by

$$0Q0,\ 1Q1,\ 1Q2,\ 2Q3,\ 3Q4$$

is a semisimulation from A to C, which proves that $A \sqsubseteq C$. It is easy to see that there is no semisimulation from C to A. However the relation R from $\mathbb{P}^{+}(CS)$ to AS given by

$$\{0\}R0,\ \{1,2\}R1,\ \{3\}R2,\ \{4\}R3$$

is a power semisimulation from C to A, which completes the proof that A and C are inter refinable.

It is worth pointing out that this example is far from pathological. Let type C describe a car-rental firm which accepts requests

"a Rolls Royce next Tuesday",

then replies

"yes, you've been allocated one" (or "sorry, none left").

On Tuesday it accepts the request

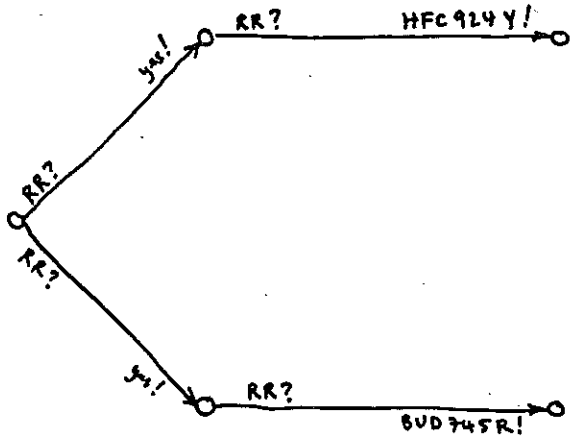
"my Rolls Royce please"

and replies

"HFC924Y, over there".

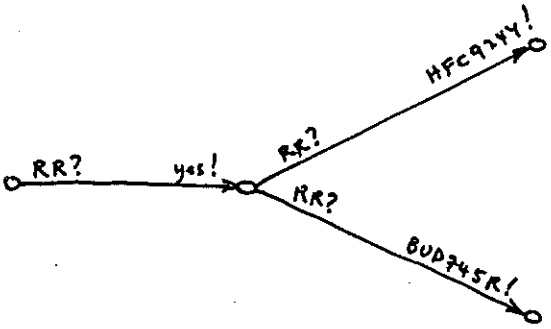
After being allocated a car, the customer may confidently proceed about his other business, contented in the knowledge that he has been assigned a particular Rolls Royce (though this has never been said explicitly)

C



But in fact, on Tuesday, the customer is given the Rolls Royce closest to the garage door

A



and of course he can't tell the difference.

Type C is easier to explain and remember and thus forms a convenient specification; type A is easier to implement. Similar examples can be found in file systems which "choose-ahead" the locations of the next available free blocks: again it is not possible for the user to tell how many blocks have been chosen ahead.

X5.2. Let types A and C be defined as in X4.1, and recall that they are inter refinable. The largest power semisimulation from A to C is the relation R from $\mathbb{P}^{\cdot}(AS)$ to CS given by

$\{0\}R0,\ \{1,2,3\}R1,\ \{1,2,3\}R2,\ \{4,5\}R3,\ \{6,7\}R4.$

The smallest power semisimulation from A to C is the relation S given by

$\{0\}S0,\ \{1\}S1,\ \{4\}S3,\ \{3\}S3,\ \{7\}S4$

which is a semisimulation. The moral of this example is that it may not be necessary to use the power semisimulation rule just because the abstract type under consideration fails to be canonical.

6. Acknowledgements

The authors benefitted from two days of discussion, one at Manchester and one at Oxford, with Toby Nipkow and Cliff Jones. Though Toby Nipkow was working on a closely related problem and obtained similar conclusions, his approach and emphasis turned out to be quite different, and it is a pleasure to record the benefit we gained from deciding with Toby and Cliff exactly what these differences were.

7. References

O.-J.Dahl, E.W.Dijkstra and C.A.R.Hoare, *Structured Programming*, Academic Press, London (1972).

E.W.Dijkstra, *A Discipline of Programming*, Prentice-Hall, Englewood Cliffs, N.J. (1976).

C.A.R.Hoare, Proof of correctness of data representations, Acta Informatica, 1, 271-281 (1972).

C.B.Jones, *Software Development: A Rigorous Approach*, Prentice-Hall International, Englewood Cliffs, N.J. (1980).

C.B.Jones, Private Communication, January (1985).

A.J.R.G.Milner, An algebraic definition of simulation between programs, CS 205 Stanford University, February (1971).

A.J.R.G.Milner, Lectures on a calculus for communicating systems, Lecture notes from the International Summer School on Control Flow and Data Flow, Munich (1984).

C.C.Morgan and B.A.Sufrin, Specification of the UNIX filing system, IEEE Trans. SE-10(2), 128-142 (1984).

T.Nipkow, Nondeterministic Data Types: Models and Implementations, Preprint, March (1985).

D.Park, Concurrency and automata on infinite sequences, in Lecture Notes in Computer Science, 104, Springer-Verlag, 167-183 (1981).

M.B.Smyth, Effectively given domains, Theoretical Computer Science, 5, 257-274 (1977).

B.A.Sufrin, C.C.Morgan, I.Sorensen and I.J.Hayes, Course notes for Mathematics for System Specification, Programming Research Group, Oxford (1983/4).