DATA REFINEMENT REFINED

RESUME

J. He, C.A.R. Hoare and J.W. Sanders

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. Two proof methods are suggested which we hope are simpler and more general than those in current use. They are proved to be individually sufficient for the correctness of refinement and together necessary for it. The proof methods can be employed to derive the weakest specification of an implementation from its abstract specification.

Oxford University Computing Laboratory
Programming Research Group
8-11 Keble Road
Oxford OX1 3QD.

0. Introduction

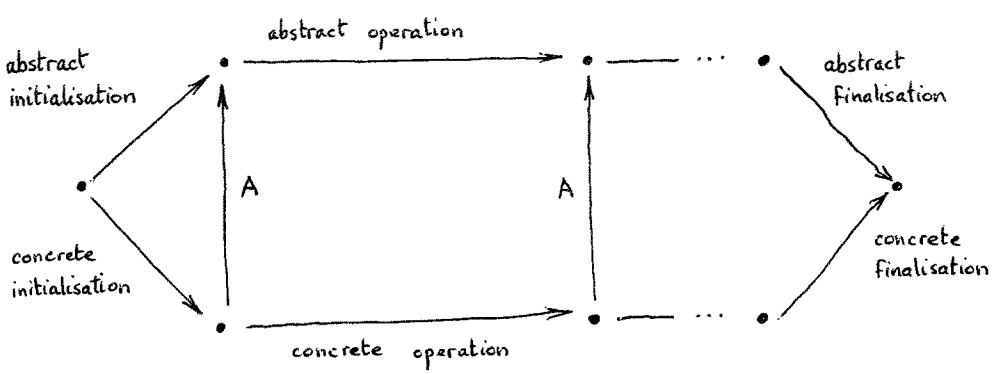
A data type is generally defined, in a manner similar to an algebra, as a set of values together with a family of operations on these values. The operations are indexed by procedure names, usually with parameters for conveying values and results between the data type and the using program. It is only by employing these procedures that the using program can update and interrogate the value of a variable of the given type.

One data type (call it **concrete**) is said to **refine** a data type with the same index set (call it **abstract**) if in all circumstances and for all purposes the concrete type can be validly used in place of the abstract one. The practical benefit of this arises when the abstract data type, such as a partial mapping, can be specified, understood and used in an applications program but cannot be directly or efficiently represented on a computer; whereas the concrete type is some efficient representation of the abstract one involving perhaps a complicated collection of bitmaps, pagetables and fileblocks, which can be economically stored and updated.

An early 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 initialisation and after every operation on the structure, assuming that it was true beforehand.

(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, in which A is the abstraction function



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, that is, the abstraction function should be a surjection.

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 operations 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 types does not have to be functional.

(4) The invariant and the abstraction relation can be combined into a single relation called (following Milner and Park) a **simulation.**

(5) A simulation may be either **upwards** (abstract-to-concrete) or **downwards** (concrete-to-abstract). The two kinds of simulation are sufficient for refinement and together they are necessary. This is a new result for nondeterministic programs.

(6) The downwards simulation rule is given as a formula which enables the weakest specification of each operation on the concrete type to be calculated directly by symbolic simplification from the simulation and the operation on the abstract type.

(7) Our treatment is in terms of the general theory of relations. The results therefore extend beyond data type refinement and include, for instance, proof of correctness of compiler code generation.

The method of data refinement plays a very significant role in the specification and proof of correctness of nontrivial system programs; we hope that the simplifications described above will make the method easier to teach, to learn, and to apply in practice. For examples which illustrate the simplicity and generality of the method we refer to [H^{3}MS^4, 1985] (for verification) and [Hoare and He, 1985] (for derivation).

1. Relations and Programs.

Suppose that S denotes the set of states of a system. We describe an operation on the system by using a binary relation on S: the state of the system before the operation is denoted s and the state after denoted s'. Important relational notation includes

$U \mathrel{\widehat{=}} S \times S$
$I_S \mathrel{\widehat{=}} \{(s, s') : S \times S \mid s' = s\}$
$R^{\smile} \mathrel{\widehat{=}} \{(s', s) : S \times S \mid (s, s') \in R\}$
$R^{\neg} \mathrel{\widehat{=}} \{(s, s') : S \times S \mid (s, s') \notin R\}$
$R \cup T$, $R \cap T$, $R \subseteq T$ and $R;T$ denote the union, intersection, containment and forward relational composition of R and T respectively
$\bigcup W$ and $\bigcap W$ denote the union and intersection respectively of the family W of relations.

Our definitions and proofs will be considerably simplified by confining attention to total relations, in which case $R \subseteq T$ means simply that R is at least as deterministic as T. The justification for this simplification can be found, for example, in [Hoare and He, 1985].

In general a relation b on S is a **condition** iff

$$b = b;U$$

and we interpret b to be true on its domain and false off it. For clarity we use lower case to write conditions and upper case for other relations.

We find it convenient to have notation for the weakest amongst the second of a pair of relations whose composition meets some specification. The dual concept, that of weakest prespecification, can be found in [Hoare and He, 1985].

Definition. If P, Q are relations, the **weakest postspecification** of P with respect to Q, namely Q/P, is the weakest solution X to the inclusion

$$P;X \subseteq Q.$$

Obviously $P;X \subseteq Q$ iff $X \subseteq Q/P$. An explicit characterisation of the weakest postspecification can be given as a predicate, or a relation, relating an initial state s to a subsequent state s'

$$(s, s') \in Q/P \quad \text{iff} \quad \forall t:S \cdot (t, s) \in P \Rightarrow (t, s') \in Q$$

or

$$Q/P = (P^{\smile};Q^{-})^{-}.$$

Programs will be written in an anologue of Dijkstra's language of guarded commands [Dijkstra, 1976]. This is restrictive enough to be implemented efficiently (intersection of relations is excluded) yet powerful enough to include nondeterminism and recursion. If X is a set of operations, $D(X)$ is defined to be the set of all programs whose primitive operations are confined to X. More formally, it is the smallest set satisfying

(1) $X \subseteq D(X)$

(2) If P and Q are in $D(X)$ then so are $P;Q$ and $P \cup Q$

(3) If, furthermore, b is a condition in X then the conditional $P < b > Q$ is in $D(X)$ (it is more commonly written **if** b **then** P **else** Q)

(4) If T is a directed family of relations in $D(X)$ then $\cap T \in D(X)$.

2. Data types.

A data type **X** is defined in a fairly conventional manner to be a quadruple

$$\mathbf{X} \triangleq (XVAL, XI, X, XF)$$

where	XVAL	is the space of values (or states) of the type
	XI	is an initialisation operation
	XF	is a finalisation operation
	X	is a family of (possibly nondeterministic) operations on XVAL.

Each operation in X is a total relation between elements of XVAL: when the relation is a condition we assume that its complement, again a condition, is also in X. XI is a relation from some global data space to XVAL, and XF is a relation from XVAL back to the same global space.

When **X** is a data type we shall write $D(\mathbf{X})$ instead of $D(X)$.

A complete program over a data type **X** is one which begins with initialisation and ends with finalisation. The space of all complete programs over **X** is thus defined to be

$$E(\mathbf{X}) \triangleq \{XI;P;XF \mid P \in D(X)\}.$$

This paper is concerned with various forms of correspondence between one data type and another. We consider abstract and concrete data type respectively

$$\mathbf{A} \triangleq (AVAL, AI, A, AF)$$

$$\mathbf{C} \triangleq (CVAL, CI, C, CF)$$

and we shall assume that these two types are conformal in the sense that

(1) their global data spaces coincide

(2) the indexing sets of A and C coincide

(3) for each condition a_i in A, c_i is also a condition and its negation has the same index as the negation of a_i. For simplicity we shall write these negations as a_i^- and c_i^-.

3. Refinement.

In this section, **A** and **C** are assumed to be conformal data types.

Firstly we define refinement. If P_A belongs to $D(\mathbf{A})$ and Q_A to $E(\mathbf{A})$ then P_C and Q_C denote the members of $D(\mathbf{C})$ and $E(\mathbf{C})$, respectively, having identical program structure and only differing because P_A contains the abstract operation A_i in all places where P_C contains the concrete one C_i, and vice versa.

Definition. Data type **C refines** data type **A** iff for each complete program P_C in $E(\mathbf{C})$,

$$P_C \subseteq P_A.$$

This gives no indication of how to develop the concrete type: it is something which can be verified, with difficulty, when **A** and **C** are both known. We start by giving two simple proof obligations (c.f. [Park, 1981] and [Milner,1984]) which can be readily checked and which prove to be sufficient for refinement.

Definition. A downwards simulation is a relation R from AVAL to CVAL satisfying

$$CI \subseteq AI;R$$
$$R;CF \subseteq AF$$
$$R;C_i \subseteq A_i;R \text{ for each index } i.$$

Similarly an **upwards simulation** is a finitary relation S from CVAL to AVAL satisfying

$$CI;S \subseteq AI$$
$$CF \subseteq S;AF$$
$$C_i;S \subseteq S;A_i \text{ for each index } i.$$

In terms of weakest specification the final inclusions in these definitions become

$$C_i \subseteq (A_i;R)/R$$

$$C_i \subseteq S\backslash(S;A_i)$$

which provide methods for calculating the specification of C_i from the abstract operation A_i and the simulations using relational algebra.

Our next concern is with the correctness and completeness of these definitions of simulation as methods for proving refinement. Firstly the correctness result.

Theorem. If there is a downwards simulation from **A** to **C** or an upwards simulation from **C** to **A** then **C** refines **A**.

Proof outline. If R is a downwards simulation from **A** to **C** then, for each index i,

$$CI;C_i;CF \subseteq AI;A_i;AF.$$

So by structural induction, for each P in $E(\mathbf{C})$,

$$P_C \subseteq P_A.$$

The proof for upwards simulation is similar.

Next the completeness result.

Theorem. If **C** refines **A** then there is an upwards simulation S on **A** and a downwards simulation R from $S(\mathbf{A})$ to **C**.

Proof outline. If **A** is a data type then a power domain construction establishes the existence of a deterministic type $\mathbf{A}^*$ and an upwards simulation S from **A** to $\mathbf{A}^*$ satisfying, for each P in $E(\mathbf{A})$,

$$P_A = P_{A*} .$$

If **C** refines **A**, **C** thus refines $\mathbf{A}^*$ and, because $\mathbf{A}^*$ is deterministic, there can be shown to be a downwards simulation R from $\mathbf{A}^*$ to **C**. This completes the proof.

Corollary. When **A** is canonical (see [H^3MS^4, 1985]), downwards simulation alone is necessary for refinement. This enables the weakest specification of **C** to be determined in the manner already observed.

4. Related Work.

We have introduced two simulation conditions which guarantee that a concrete data type refines an abstract one. These simulation conditions are more general than the rule used in VDM: the downwards rule always applies if the VDM rule does, but there are situations to which the downwards rule applies though the VDM rule does not. In cases where both rules apply, the VDM relation is total and surjective though the downwards simulation need not be; when the downwards simulation is a total bijection, the two rules coincide.

Tobias Nipkow at the University of Manchester has also extended the VDM rule for refinement to obtain a rule which is different from ours but which also no longer assumes that R is injective or surjective ([Nipkow, 1985]). Nipkow's motivation lies more in the application to concurrency; thus his rule preserves, as does an untotalised simulation, the domains of the abstract and concrete operations.

5. References.

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

H^3MS^4 Data Refinement Refined, Draft 1, PRG preprint, May, 1985.

He, Jifeng, C.A.R. Hoare and J.W. Sanders Data Refinement Refined, to appear.

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

C.A.R. Hoare and He, Jifeng The weakest prespecification, Technical Monograph, PRG-44, June, 1985.

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

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.

T. Nipkow Nondeterministic data types: models and implementations, University of Manchester preprint, March, 1985.

D. Park Concurrency and automata on infinite sequences, in LNCS, **104**, 167 - 183, 1981.

M.B. Smyth Effectively given domains, TCS, **5**, 257 - 274, 1977.