The Rigorous Development of a System Version Control Database

by

Ian D. Cottam

Technical Monograph PRG-31
October 1982

Oxford University Computing Laboratory,
Programming Research Group,
45, Banbury Road,
OXFORD, OX2 6PE

ABSTRACT

The "Rigorous Approach" to software development espoused by C.B. Jones in [1] is followed in developing a special-purpose database designed to control the various versions of a system. The version control program developed is firmly based upon the Gandalf [2] System Version Control Environment [3] in use at Carnegie-Mellon University.

Most projects involve the production of more than one source document. The documents (after processing) may be combined in various ways to compose systems. As systems evolve through different versions it becomes increasingly difficult to keep track of the source documents and their interdependencies; the consistency of systems put together from collections of documents is consequently hard to verify. The program developed supports the most common forms of document interdependency relations, and methods of system evolution. The entire top-level specification is presented. Crucial design decisions taken during a prototype development are also included and justified. The paper documents the author's experiences with the method on a small, yet non-trivial, problem.

Index Terms:

Formal specification, correctness, program proving, version-control, special-purpose database.

INTRODUCTION

Many people in the computing community believe that to transform programming into a professional engineering discipline requires the mastery and application (by programmers) of development methods which are rooted in the formality of mathematics and logic. This paper attempts to demonstrate the power of formal methods via an example case-study development of a System Version Control Database (SVCD). The "rigorous approach" to software development [1] promulgated by C.B. Jones is the method employed. Our version control system is a variant of the Gandalf System Version Control Environment [3] produced by A.N. Habermann and co-workers at Carnegie- Mellon University.

Below we briefly introduce both the development method and our philosophy regarding the control of system versions. The paper continues with, firstly, the formal functional specification of SVCD (together with informal commentary); then we present successive refinement stages of our specification until executable (Pascal) code is reached; finally, problems encountered during the development are presented and their consequences assessed.

The rigorous approach is also known as the Vienna Development Method (VDM [4]),and was first used by Jones and co-workers at the IBM Laboratory in Vienna. It is based on a denotational semantics [5] framework. The focal point is a specification notation ("Meta-IV") oriented towards the philosophy of modelling abstract data types. Abstract functional specifications are reified ("made more concrete") by successive refinement steps, resulting in a traditional program in some desired programming language. Each stage is shown to be correct with respect to the previous stage. Since a notation change occurs between the penultimate refinement

and the implementation (say, from Meta-IV to Pascal), assertions may be added at appropriate points in the program code (as comments) for verification purposes. The rigorous approach may be thought of as following a formal method but, often, the correctness of theorems is shown via informal arguments. Such arguments are, however, developed hand-in-hand with the specification and design. If a dispute regarding the correctness of a development arises, the appropriate theorems may be subjected to formal proof. In this case-study we concentrate on data type refinement concerns at the expense of operation de-composition (or code proofs).

The customer's statement of requirements was:

> "Many projects involve the cooperation of several people and the production of more than one 'source' document. The documents (after processing) may be combined in various ways to compose 'systems'. As systems evolve through different versions it becomes increasingly difficult to keep track of the source documents and their interdependencies; the consistency of systems put together from collections of documents is consequently hard to verify.
>
> Specify a database (and operations on it) to support the most common forms of document interdependency relations, and methods of 'system evolution'. The relationships between components should be specified in a language-independent fashion. The database must be able to describe (software) systems written in a variety of (programming) languages. Specified components must be 'implementable' either as (atomic) modules or as (composed) (sub)systems. The main purpose of the database would be to keep track of source documents etc. Implement (in Pascal) enough of the specified system to enable a judgement to

be made about the utility of a complete implementation.

The 'systems' may not necessarily be software systems, nor need all the documents be 'program texts'."

The proposed solution to our customer's requirements is based on facilities provided by the Gandalf 'Integrated Software Development Environment'. The Gandalf project is a development of the Computer Science Department at Carnegie-Mellon University (CMU); implemented as modifications and extensions to the Unix(Tm.) operating system [6]. It consists of three major components: (a) an Integrated Program Construction Facility, (b) a System Composition and Generation Facility, and (c) a Project Management Facility. Only (b), and only the composition part, is considered further. For brevity it is (at CMU and herein) referred to as SVCE (for System Version Control Environment).

The central philosophy is the belief that a system component descriptive approach is more suitable for handling version control activities than the more common imperative or procedural approach. To quote from [3]:

"System component descriptions provide a means of representing functional specifications that convey sufficient information to potential users of those components. These descriptions include compositions and implementations to delineate specific versions, to guarantee that versions are consistent, and to automatically generate particular system versions."

SVCE is essentially a practical realisation of the ideas presented in two recent CMU Doctoral theses [7][8], complemented by a theory of 'well-formed systems' [9]. Again, we quote from [3]:

"SVCE provides a simple Ada-like language that enables the system designer to describe the various system components, their versions, and their interconnections. The form of the system component descriptions is based on the structure of the Ada Package; there is a specification part and an implementation part. The contents of the two parts, however, are related to system version control instead of matters of general programming.

There are two type of components from which to construct systems: modules and systems. The visible part of a module displays the facilities provided by the module (the provide list is essentially the contents of the visible part of an Ada module) and the various implementations of the module. The visible part of a system displays the facilities provided by the system and the various compositions of the system. A module body describes how its implementations are derived. A systems body lists all of its components and how its compositions are constructed."

Habermann and Perry go on to describe several basic properties of correct (well-formed) system descriptions. Since these properties all have counterparts in our SVCD we delay discussion of them until they are formally introduced. Clearly, with minor exceptions, the SVCE model satisfies our requirements.

Figure 1 is an example of SVCD input which, hopefully, will provide the reader with a feeling for system description.

```
spec DiscScheduler provides schedule, service;
     FCFS;
 std SSTF
end {spec DiscScheduler}

spec ds provides schedule, service;
      FCFS requires rcb, dequeue, queue, enqueue;
  std SSTF requires rcb, list, remove, inspect, insert
end {spec ds}

spec rcbqueue provides dequeue, queue, enqueue;
 std staticq requires rcb;
     dynamicq requires rcb
end {spec rcbqueue}

spec rcblist provides list, remove, inspect, insert, position;
 std slist requires rcb
end {spec rcblist}

{ MODULES }

mod ds;
 FCFS impl "/usr/ian/svcd/fcfs.ada.make";
 SSTF impl "/usr/ian/svcd/sstf.ada.make"
end {mod ds}

mod rcbqueue;
 staticq impl "/usr/ian/svcd/sq.ada.make";
 dynamicq impl "/usr/ian/svcd/dq.ada.make"
end {mod rcbqueue}

mod rcblist;
 slist impl "/usr/ian/svcd/sl.ada.make"
end {mod rcblist}


{ SYSTEMS }

sys DiscScheduler
 use ds, rcbqueue, rcblist;
  FCFS = ds.FCFS, rcbqueue;
  SSTF = ds, rcblist
end {sys DiscScheduler}
```

Fig. 1. An Example Version Control Description for a Disc Scheduler Subsystem (based on the Gandalf example given in [3]).

THE FORMAL SPECIFICATION

The specification philosophy of VDM is constructive or model-oriented. The notion of a state is explicitly recognized and the specification primitives are the well-understood mathematical objects: sets, lists (often called sequences or tuples), maps, abstract syntax, and (recursive) functions.

The specification is divided into three sections: the state with associated invariant, the operations which 'change the database' (state transformers), and the operations which return values but have no effect on the database (state interrogators). Extensive commentary follow each of the three sections. The dialect of the specification notation (Meta-IV) is influenced by that used in [10]. Minor changes (notably in the area of exception conditions) are explained as and whenever necessary. For example, we use the conditional logical conjunction (&) and disjunction (\/) connectives. The 'standard' Meta-IV operators and functions are defined in [1].

```
/* SVCD STATE SPECIFICATION */


1.Ø  Db ::
 .1        prov : Cn -m-> Fac-set
 .2        req  : Cn -m-> (Vn -ml-> Fac-set)
 .3        std  : Cn -m-> Vn
 .4        mod  : Cn -m-> (Vn -ml-> Imp)
 .5        sys  : Cn -m-> (Vn -ml-> (Cn -ml-> Vn))
1.6        uses : Cn -m-> Cn-set


/* ABSTRACT INVARIANT ON Db */
2.Ø  inv dom prov = dom req = dom std &
 .1       (∀c ∈ dom req; (∀v ∈ dom req(c);
 .2        isdisj(prov(c), req(c)(v)) &
 .3         std(c) ∈ dom req(c) )) &
 .4       (dom mod ∪ dom sys) ⊆ dom prov &
 .5       isdisj(dom mod, dom sys) &
 .6       (∀c ∈ dom mod; dom mod(c) = dom req(c)) &
 .7       (∀c ∈ dom sys; dom sys(c) = dom req(c)) &
 .8       dom uses = dom sys &
 .9       union rng uses ⊆ dom prov &
 .10      iscomplete(db) &
 .11      ~iscircular(db) &
 .12      is_min_well_formed(db)
 .13   where db ≙ mk-Db(prov, req, std, mod, sys, uses)

 .14   and
 .15    iscomplete(db) ≙
```

```
.16     (∀s ∈ dom sys;
.17       prov(s) ⊆ union{prov(c)|c ∈ uses(s)})

.18   and
.19    iscircular(db) ≙
.20     (∃s ∈ dom sys; iscomp(s, s, db))
.21      where
.22       iscomp(s1, s2, db) ≙
.23         s2 ∈ dom sys &
.24          s1 ∈ uses(s2) \/
.25          (∃ cn ∈ uses(s2); iscomp(s1, cn, db))

.26   and
.27    is_min_well_formed(db) ≙
.28     (∀s ∈ dom sys;
.29      (∀v ∈ dom sys(s);
.30       (∀c1,c2 ∈ dom sys(s)(v);
.31      ~isdisj(prov(c1), prov(s) ∪ r) &
.32      c1≠c2 => isdisj(prov(c1), prov(c2)) &
.33      c1 ∈ uses(s) & sys(s)(v)(c1) ∈ dom req(c1)&
.34      prov(s) ⊆ p &
.35      (r-rs)  ⊆ p )))
.36     where
.37       p ≙ union{prov(c)|c ∈ dom sys(s)(v)}
.38     and
.39       r ≙
.40        union{needs(c, sys(s)(v)(c))|c ∈ dom sys(s)(v)}
.41
.42     and
.43       rs ≙
```

```
.44        union{needs(c, v)|c ∈ uses(s) & v ∈ dom req(c)}
.45
.46      where
.47        needs : C:Cn x V:Vn --> Fac-set
.48        pre      c ∈ dom prov & v ∈ dom req(c)
.49        needs(c,v) ≙
.50         if c ∉ dom sys then req(c)(v)
.51         else
2.52          union{needs(cx, sys(c)(v)(cx))|cx ∈ dom sys(c)(v)}
```

```
3.0  init  prov=req=std=mod=sys=uses=[]
```

Annotation

1.0 Four sets used in the state definition are not explicitly defined: Cn - a set of component names; Vn - a set of variant (or version) names; Fac - a set of abstract facility (or resource) names; and, Imp - a set of 'implementations'. The specification may be thought of as being gereric over these sets. The Db state is defined by six maps (-m->) from component names to the ranges explained below.

1.1 The prov map takes a component name and maps it to the set of facility names which it provides or 'exports' for (potential) use by other database components.

1.2 Each component has one or more versions, each of which may require any number of facilities (from other components). This is specified by the req map.

1.3 One particular version of a component is distinguished as the standard version. This is the intent of the std map.

1.4 A component may be implemented as a module. Each version of such a component must have an implementation. The mod map fulfills this requirement.

1.5 Similarly, a component may be defined as a system. Each version of such a component must have its composition (of modules and other system components) defined. This is specified by the sys map. Note how the set of component/version pairs which constitute the composition for a particular system version is also defined by a map (from component names to version names).

1.6 The uses map defines which components each system may choose from for forming particular compositions.

Before going on to discuss our database invariant (2.0 - 2.52), some justification of the chosen abstraction is called for. Maps are a common and powerful abstraction for information retrieval. They capture exactly the properties that are of interest. (Note: maps which exclude the empty map are denoted by -m1->.) Questions of database size and the structures and media needed for storage and access are of no concern at this level. The most debatable decision concerning the abstraction is the use of six separate maps (ignoring the nested maps) where one is sufficient. The domains of the maps are either equal (i.e. prov, req, std) or are subsets of the domain of prov (i.e. mod, sys, uses). There are (at least) two important heuristics which may be used to guide the choice of an abstract model. The more important of the two is that the specification should be free from implementation bias (see page 261 of [1]). A model-oriented abstract data type specification is said to be biased towards certain implementations if equality of the underlying objects cannot be tested using operators of the data type. In other words, the abstraction should not contain any irrelevant

information. We state, without giving a proof, that our specification is without bias. It is often said that reality is too ragged to be totally mirrored by neat mathematical models. Hence the necessity that the set of all possible states, defined by Db, be restricted by a data type invariant. The second heuristic is that the trade-off between state and invariant should be made so as to minimize the invariant. In this case the heuristic has not been followed since, clearly, the aforementioned relationships between the domains of the maps must be included in our invariant. The trade-off made here is one of readability for mathematical elegance. The single-map abstraction requires an extremely complicated range definition. The notation required to specify the operations on such a type would tend to be obscuring. This is, of course, a very subjective view. (Students at Manchester have re-written the specification with a minimal invariant and, whilst concluding that their version is better, agree that several auxiliary functions are required in order to manipulate the revised state.) In the description of the invariant, below, much of the nomenclature and concepts are taken from [3] and [9].

2.Ø The prov, req, and std maps always return information on the same set of component names.

2.1-2 A component version may not require a facility that the component is specified to provide. This fundamental property simplifies the rules for well-formed system composition.

2.1-3 Each component has a standard (default) version.

2.4 This rule states that components which have been <u>defined</u> as either (a family of) modules or (a family of) systems must also be <u>specified</u>. By specified is meant that the component must have its provisions and version requirements present in the database.

2.5 A component may be defined either by implementation of its versions as modules, or as system compositions, but not both.

2.6-7 These two rules simply state that the versions of defined components must be the same as the names of the versions specified in the req map.

2.8 The uses and sys maps always return information on the same set of component names.

2.9 All components that may be used in system compositions must be present in the database.

2.10 This is the basic sufficiency rule for system descriptions. Systems are 'complete' in that their advertised provisions are satisfiable by other database components. The definition of the iscomplete predicate is given on lines 2.15-17.

2.11 The 'not circular' rule for system compositions prevents a system version from including itself, directly or indirectly, in its own composition. The definition of the iscircular predicate is given on lines 2.19-25.

2.12 System compositions must be minimally well-formed. The properties of valid compositions are grouped together under the predicate is_min_well_formed. They are named and, briefly, described below.

2.31 We insist that well-formed compositions are also minimal. That is, no component in a composition is superfluous to requirements. A component is superfluous if it neither provides facilities which the system advertises nor provides facilities required by other components of the composition. In the definition, 'r' is the set of requirements of a given composition.

2.32 Components in a composition must not conflict, i.e. provide the same facility. This is similar to the situation in programming languages where declarations in the same scope must be unique.

2.33 This term is referred to as the self-supporting property of valid system compositions. A self-supporting composition is made from component versions which are already specified in the database. (N.B. They need not be defined as modules or systems.)

2.34 Valid compositions must be self-sufficient. A self-sufficient composition provides, at least, those facilities advertised as being provided by the system. 'p' is the set of facilities provided by a composition.

2.35 Finally, valid system compositions must be self-contained. A self-contained composition has no outstanding requirements that could be satisfied by, one or more, (useable) components not included in the composition. Note that the uses map defines which subset of database components may be used in each system definition. 'rs' is the set of outstanding requirements of a system.

Observe that system composition requirements are not stored in the database. They are derived whenever necessary by the function 'needs' (2.47-52).

To end this section line 3.Ø specifies the initial state of the abstract type Db. As might be expected the database is initially empty (the maps are all equated to the empty map []).

Db Transformers

Ten state transformers are specified implicitly by relating

their input and output conditions. In other words, the required properties of an operation are defined without dictating exactly how they are to be computed. Operations are, in general, specified by four clauses according to the following schema:

1. OP(A1:Type1, A2:Type2, ..., An:Typen)Result:Type

Both the arguments and result fields may be empty. They are known types other than the class of states over which the operations are defined. Such states are given in a second clause

2. globals G1:(rd|wr)Type1,...,Gn:(rd|wr)Typen

State components which the operation may access are listed in the globals clause. It defines a minimal environment that the operation may be invoked in. State components labelled rd are read-only. It is illegal for the operation to specify a change to such objects. Components labelled wr may be updated. The effect of the operation on such variables must be defined (see 4. below). If a group of operations all have the same globals the clause may be factored out and appears before any of the operations.

3. pre pred

Pred is a predicate over the state and arguments. The pre-condition characterises the standard domain of an operation. Should an operation be invoked in a state where the predicate, given by the pre clause, is false then the effect of the operation is undefined. An alternative is to define the exceptional domain of an operation. The convention is adopted that the result of an operation invoked in its exceptional domain is an identity on the state. Additionally, an exception signal is raised in some implementation dependent manner. This is often exactly what is required for 'user-level'

operations where robustness and well-defined error handling are paramount. For a comprehensive introduction to the area of robust data types the reader should consult [11]. An exception domain may be conveniently partitioned to permit precise error reporting.

ex-Ex1 pred1
ex-Ex2 ~Ex1 & pred2
ex-Exn predn

Any of the predicates Ex1 to Exn which are true cause the corresponding exception to be raised. No ordering is defined over the partitioned exception domains. However, in practice (i.e. an implementation) it will be sensible to check some exceptions before others. This is hinted at above where Ex2 has the negation of Ex1 guarding pred2 from being undefined. The pre-condition of an operation is equivalent to

(~Ex1 & ~Ex2 ... & ~Exn)

4. post pred

Pred is a predicate of the initial state, the final state, and any arguments and results of the operation. The post-condition defines the required relationship between the state prior to invocation of the operation and immediately following invocation. Values of the final state are indicated by priming. For example, assume x is a global integer which may be updated (wr) then

post x'-1 = x

specifies that (in some entirely unspecified way) x is to be incremented.

```
/* SVCD TRANSFORMER OPERATION SPECIFICATIONS */

    globals D : wr Db


4.0  ADD-SPEC(C:Cn, P:Fac-set, A:Vn -ml-> Fac-set, STAND:Vn)
 .1  ex-AlreadySpec      c ∈ dom d.prov
 .2  ex-NotUnique        ~isdisj(p, union rng a)
 .3  ex-CantBeStd        stand ∉ dom a
 .4  post  d' = mk-Db(d.prov U [c->p], d.req U [c->a],
4.5                     d.std U [c->stand], d.mod, d.sys, d.uses)


5.0  ADD-MOD(C:Cn, I:Vn -ml-> Imp)
 .1  ex-NoSuchComp       c ∉ dom d.prov
 .2  ex-VersionsDiffer   ~NoSuchComp & dom i ≠ dom d.req(c)
 .3  ex-AlreadyDef       c ∈ (dom d.mod U dom d.sys)
5.4  post  d' = mk-Db(d.proc, d.req, d.std, d.mod U [c->i], d.sys, d.uses)


6.0  ADD-SYS(C:Cn, USE:Cn-set,
 .1          COMPOS:Vn -ml-> (Cn -ml-> [Vn]))
 .2  ex-NoSuchComp       c ∉ dom d.prov
 .3  ex-VersionsDiffer   ~NoSuchComp & dom compos ≠ dom d.req(c)
 .4  ex-CantBeSys        ~NoSuchComp & union rng d.req(c) ≠ {}
 .5  ex-AlreadyDef       c ∈ (dom d.mod U dom d.sys)
 .6  ex-UnknownComps     use ∉ dom d.prov
 .7  ex-Incomplete       ~iscomplete(dd)
 .8  ex-Circular         iscircular(dd)
 .9  ex-Illformed        ~is_min_well_formed(dd)
```

```
 .10 post  d' = dd
 .11  where dd ≙ mk-Db(d.prov, d.req, d.std, d.mod,
 .12                    d.sys  U [c->findstd(compos, d)],
6.13                    d.uses U [c->use])


7.0  DEL-SPEC(C:Cn)
 .1  ex-NoSuchComp      c ∉ dom d.prov
 .2  ex-InUse           c ∈ union rng d.uses
 .3  post  d' = mk-Db(d.prov\{c}, d.req\{c}, d.std\{c},
7.4                   d.mod\{c}, d.sys\{c}, d.uses\{c})


8.0  DEL-MOD(C:Cn)
 .1  ex-NoSuchMod       c ∉ dom d.mod
8.2  post  d' = mk-Db(d.prov, d.req, d.std, d.mod\{c}, d.sys, d.uses)


9.0  DEL-SYS(C:Cn)
 .1  ex-NoSuchSys       c ∉ dom d.sys
9.2  post  d' = mk-Db(d.prov, d.req, d.std, d.mod, d.sys\{c}, d.uses\{c})


10.0 ADD-MODVAR(C:Cn, V:Vn, R:Fac-set, I:Imp)
  .1 ex-NoSuchMod       c ∉ dom d.mod
  .2 ex-AlreadyExists   ~NoSuchMod & v ∈ dom d.req(c)
  .3 ex-NotUnique       ~NoSuchMod & ~isdisj(r, d.prov(c))
  .4 post  d' = mk-Db(d.prov, d.req†[c->d.req(c) U [v->r]], d.std,
10.5                  d.mod†[c->d.mod(c) U [v->i]], d.sys, d.uses)
```

```
11.0 ADD-SYSVAR(C:Cn, V:Vn, CMPS:Cn -ml-> [Vn])
  .1 ex-NoSuchSys        c ∉ dom d.sys
  .2 ex-AlreadyExists    ~NoSuchSys & v ∈ dom d.req(c)
  .3 ex-Circular         iscircular(dd)
  .4 ex-Illformed        ~is_min_well_formed(dd)
  .5 post  d' = dd
  .6  where dd ≙ mk-Db(d.prov, d.req†[c->d.req(c) U [v->{}]],
  .7                       d.std, d.mod,
  .8                       d.sys†[c->findstd(d.sys(c) U [v->cmps], d)},
11.9                       d.uses)


12.0 DEL-MODVAR(C:Cn, V:Vn)
  .1 ex-NoSuchMod        c ∉ dom d.mod
  .2 ex-NoSuchVersion    ~NoSuchMod & v ∉ dom d.req(c)
  .3 ex-CantDelStd       ~NoSuchMod & v = d.std(c)
  .4 ex-InUse            [c->v] ⊆ union applys(rng d.sys, rng)
  .5 post  d' = mk-Db(d.prov, d.req†[c->d.req(c)\{v}], d.std,
12.6                        d.mod†[c->d.mod(c)\{v}], d.sys, d.uses)


13.0 DEL-SYSVAR(C:Cn, V:Vn)
  .1 ex-NoSuchSys        c ∉ dom d.sys
  .2 ex-NoSuchVersion    ~NoSuchSys & v ∉ dom d.req(c)
  .3 ex-CantDelStd       ~NoSuchSys & v = d.std(c)
  .4 ex-InUse            [c->v] ⊆ union applys(rng d.sys, rng)
  .5 post  d' = mk-Db(d.prov, d.req†[c->d.req(c)\{v}], d.std, d.mod,
13.6                        d.sys†[c->d.sys(c)\{v}], d.uses)
```

```
14.0 where  findstd : Vn -m1-> Cn -m1-> [Vn]  x  Db
  .1                  -->  Vn -m-> Cn -m-> Vn
  .2        findstd(m, d) ≙
  .3             [v->[c->if m(v)(c)=NIL then std(c) else m(v)(c)
  .4                     | c ∈ dom m(v)]
14.5               | v ∈ dom m]
```

Annotation

4.0 Add, or insert, a component specification to the database. Arguments are the component name, the names of facilities which it provides, the names of versions together with their respective requirements, and the name of the version which is to be regarded as the default or standard (in some sense) version. The pre-condition guarantees that the operation does not override any existing component with the same name. The Db invariant is safeguarded by predicates which assert that the standard version is present and that provisions and requirements are disjoint.

5.0 Add a module definition to the database. The component name must be specified but not already defined. All the version names must have a corresponding implementation and the names must match those in their specification.

6.0 Add a system definition to the database. Arguments are a (previously specified) component name, the set of components which may be used in the compositions, and the version compositions. The pre-condition of ADD-MOD applies plus the following extra constraints. The versions must have been specified as having no requirements. (Module requirements are

given, and assumed correct by SVCD; system requirements are computed when necessary.) The Db invariant is protected by predicates which assert that the composable components exist, and the completeness, non-circularity, and minimal well-formedness of system compositions are maintained.

7.Ø Delete a component specification (and definition, if present) from the database. The pre-condition prevents the deletion of components which are used in system compositions.

8.Ø Delete a module definition.

9.Ø Delete a system definition.

1Ø.Ø Add a new module version definition to the database.

11.Ø Add a new system version definition.

12.Ø Delete a version from a module component definition. The pre-condition prevents both the deletion of the standard version and the deletion of a version which is used in a system composition.

13.Ø As per 12.Ø, but for a system version.

14.Ø The auxiliary function, findstd. The user may omit the version names in system compositions. Findstd 'looks-up' the standard versions.

Correctness

The next step is to document the validity conditions of the initialization and transformer operations. For initialization to be valid the invariant must be true for the 'empty' database.

```
invDb( mk-Db([], [], [], [], [], []) )
```

This is immediately obvious by inspection of the constituent terms of the invariant. Remember that predicates universally quantified

over the empty set are trivially true. The general 'preservation of validity' rule is

```
(∀d∈Db;
  invDb(d) & pre-OP(d,args) & post-OP(d,args,d')
    => invDb(d'))
```

The correctness of the ten transformer operations with respect to the above rule must be demonstrated. Here we reproduce the correctness argument for the ADD-MODVAR operation to illustrate the level of rigorous documentation typically produced for validity preservation proofs. The instance of the rule for ADD-MODVAR is

```
(∀d∈DB, c∈Cn, v∈Vn, r∈Fac-set, i∈Imp;
  invDb(d) &
  pre-ADD-MODVAR(d,c,v,r,i) &
  post-ADD-MODVAR(d,c,v,r,i,d')  =>  invDb(d'))
```

Substituting for the pre/post conditions

```
  invDb(d) &
  (c∈dom d.mod&(v∉dom d.req(c) & isdisj(r,d.prov(c))))
    => invDb( mk-Db(d.prov,
                    d.req†[c->d.req(c)U[v->r]],
                    d.std,
                    d.mod†[c->d.mod(c)U[v->i]],
                    d.sys,
                    d.uses) ))
```

The proof is broken down into a series of sub-arguments or lemmas.

Lemma (Ai): d.req≠d'.req and d.mod≠d'.mod, and the remaining four maps are identities across ADD-MODVAR. [Proof: obvious by inspection of post-condition, pre-condition term (v∉d.req(c)), and

term (2.6) of invariant.]

Lemma (Aii): the terms of invDB concerned with system composition correctness (2.8, 2.9, 2.10, 2.11, 2.12) are preserved. [Proof: follows from lemma (Ai), and the 'self-supporting' term (2.33) of the invariant.]

Lemma (Aiii): dom d.req=dom d'.req and dom d.mod=dom d'.mod. [Proof: obvious by inspection of post-condition, pre-condition term (cedom d.mod), and terms (2.0, 2.4) of the invariant.]

Lemma (Aiv): the database well-formedness conditions (2.0, 2.3, 2.4, 2.5, 2.7) are preserved. [Proof: follows immediately from lemmas (Ai) and (Aiii).]

Lemma (Av): the database well-formedness conditions (2.2, 2.6) are preserved. [Proof: obvious by inspection of post-condition, and pre-condition term (isdisj(r, d.prov(c)))).]

It is conjectured that the above five lemmas are sufficient to demonstrate the validity of the ADD-MODVAR operation.

Db Interrogators

The SVCD user will naturally require to interrogate the database in order to answer such queries as "what components are present?", "which components provide at least facilities X, Y, and Z?", and "where is component C used?". A fairly minimal set of such query operations is specified below.

```
/* SVCD INTERROGATION OPERATION SPECIFICATIONS */

     globals D : rd Db

15.0 SUPPLIES(C:Cn)Facilities:Fac-set
  .1 ex-NoSuchComp        c ∉ dom d.prov
15.2 post  facilities' = d.prov(c)


16.0 NEEDS(C:Cn, V:Vn)Facilities:Fac-set
  .1 ex-NoSuchComp        c ∉ dom d.prov
  .2 ex-NoSuchVersion     ~NoSuchComp & v ∉ dom d.req(c)
16.3 post  facilities' =  needs(d, c, v)


17.0 CHOICE(F:Fac-set)Cmps:(Cn, Vn)-set
  .1 post  cmps' =
17.2  {(c,v)|c ∈ dom d.prov& v ∈ dom d.req(c)& f ⊆ d.prov(c)}


18.0 STANDARD(C:Cn)Version:Vn
  .1 ex-NoSuchComp        c ∉ dom d.prov
18.2 post version' = d.std(c)


19.0 UNDEFINED()Sc:Cn-set
19.1 post  sc' = dom d.prov - (dom d.mod U dom d.sys)


20.0 UNUSED()Sc:Cn-set
```

```
20.1 post  sc' = dom d.mod - union rng d.uses
```

```
21.0 WHEREUSED(C:Cn)Sc:Cn-set
  .1 ex-NoSuchComp        c ∉ dom d.prov
  .2 post  sc' = part({c}) - {c}
  .3  where part : Cn-set --> Cn-set
  .4   part(cs) ≙
  .5    if cs={} then {}
21.6    else cs U part({s ∈ dom d.sys | ~isdisj(cs, d.uses(s))})
```

Annotation

15.0 SUPPLIES returns the set of facility names provided by the database component C.

16.0 NEEDS takes a component and one of its versions and returns the names of facilities which it requires. For brevity the needs function defined in the invariant is used with the obvious extension to a third argument (a particular database).

17.0 CHOICE returns the set of component-version pairs which provide at least the argument facilities. The user may list all the components, and their versions, present in the database by invoking CHOICE with the empty set as argument.

18.0 STANDARD identifies the default version for a given component.

19.0 The set of components in the database which have still to be defined (i.e. as modules or systems).

20.0 The set of module components which are not available for use in system compositions.

21.0 WHEREUSED returns the set of system components which include the argument in one or more of their compositions.

FIRST-LEVEL REFINEMENT

In this section the specification of SVCD is refined to a less abstract or 'more concrete' representation. This involves moving from pure specification to the world of design specification. In other words, the remaining specifications will all be biased towards the implementation of choice. Abstract maps cannot be implemented directly in our given programming language (Pascal). In this first refinement step we model the maps of the Db state with mathematical lists. Sets of complex objects are still used freely even though Pascal has no direct equivalent (cf. the Pascal type constructor set of).

```
/* SVCD FIRST-LEVEL REFINEMENT OF STATE */

22.0 Dbl ::  c : Cn-list   h : Def-list
  .1  Def :: prov : Fac-set  vns : Vn-listl req : Fac-set-listl
  .2           std : Vn  defn : [Module|System]
  .3      Module = Imp-listl
  .4      System :: uses : Cn-set sys : Compo-listl-listl
22.5        Compo :: sc : Cn sv : Vn


23.0 inv isuniquel(c) &
  .1      len c = len h &
  .2      (∀ i ∈ inds h;
  .3      isuniquel( h(i).vns ) &
  .4      len h(i).vns = len h(i).req &
  .5      cases h(i).defn:
  .6       mk-Module(m) -> len m       = len h(i).vns
  .7       mk-System(s) -> len sys(s) = len h(i).vns &
```

```
.8                    (∀ j ∈ inds s.sys;
.9                      isuniquel( applyl(s.sys(j), sc) ))
.10             NIL  -> TRUE) &
23.11  invDb( retrDb( mk-Dbl(c, h) ) )


24.0  init c = h = <>
```

Annotation

22.0 The outer-level Db maps are now represented by the two lists 'c' and 'h'. The c list of component names corresponds to the, union of the, domains of the maps. The h list of Defs corresponds to the ranges of the maps.

22.1 A 'Def' object defines a set of provisions (prov), a non-empty list of version names (vns), a non-empty list of sets of requirements (req), a standard version (std), and the implementation definition (defn) for a component.

22.3 The Db map from versions to implementations is modelled by a list.

22.4 As per 22.3 but for system compositions.

23.0 To understand formally the relationships between the objects of the Dbl state we need to examine the invariant. Observe that it is split into two parts. One part being 'refinement' predicates (i.e. modelling the Db maps), and the other (23.11) the original Db invariant formed by viewing the Dbl state through a function which 'retrieves the abstraction' (see below). Consider the former: the first predicate asserts that all the elements of the c list are unique. Secondly, the two lists are always of equal length. The idea of the

representation is that the specification part of a component (cn), present in c, may be accessed by indexing the h list by cn's index in c. The index may be obtained in a pleasingly abstract manner via the inverse of c. How this look-up function is achieved (operationally) is left to a further refinement stage. Finally (22.2-10), we specify the invariant which holds for each Def object. The list of version names (vns) for each component is also defined to be unique. Each version name has a corresponding set of requirements and, therefore, the lengths of req and vns are invariantly equal. The two implications assert that for modules (resp. systems) there exists a corresponding implementation (composition), and that system composition names are unique. A few words concerning the inherited (Db) invariant are called for. As is pointed out in [10], this technique may aid the correctness proof but has the disadvantage that a design cannot be understood at a single level. This is particularly important when several people are involved in the different stages of development. During the actual refinement the inherited invariant was expanded (i.e. re-formulated to operate on the Db1 state). Only the inherited form is used here for brevity.

Correctness of Db1

The key to demonstrating refinement correctness is in showing the existence of a total function from values in the refinement state to their corresponding abstract values. Such an object is known as a retrieve function.

```
/* SVCD DB RETRIEVE FUNCTION */

25.0  retrDb : Dbl --> Db
  .1  retrDb(mk-Dbl(c, h)) ≜
  .2    mk-Db( [cn->prov(h(i))| cn ∈ elems c],
  .3
  .4           [c->[vns(h(i))(j)->req(h(i))(j)| j ∈ inds vns(h(i))]
  .5               | cn ∈ elems c],
  .6
  .7           [cn->std(h(i))| cn ∈ elems c],
  .8
  .9           [cn->[vns(h(i))(j)->defn(h(i))(j)| j ∈ inds vns(h(i))]
  .10              | cn ∈ elems c & is-Module(defn(h(i)))],
  .11
  .12          [cn->[vns(h(i))(j)->[sc(x)->sv(x)
  .13                                  | x ∈ elems sys(defn(h(i)))(j)]
  .14                  | j ∈ inds vns(h(i))]
  .15              | cn ∈ elems c & is-System(defn(h(i)))],
  .16
  .17          [cn->uses(defn(h(i)))(j)
  .18              | j ∈ inds vns(h(i)) &
  .19                cn ∈ elems c & is-System(defn(h(i)))] )
  .20
  .21   where i ≜ inverse(c)(cn)
  .22    where inverse : L:Cn-list --> (Cn --> Nat)
  .23          pre       isuniquel(l)
25.24          inverse(l) ≜ [c->i | i ∈ inds l & l(i)=c]
```

Retrieve functions greatly aid our intuitive feeling for the

adequacy of representations. The conjecture is that all objects in the abstract state Db have at least one representation in the refinement state Dbl.

(∀a∈Db; invDb(a)=>(∃r∈Dbl; invDbl(r) & a=retrDb(r)))

The rule that the function must be total may be stated:

(∀r∈Dbl; invDbl(r)=>(∃a∈Db; invDb(retrDb(r))) & a=retrDb(r)))

A (reasonably formal) proof of the above rules is long and tedious. In practice the adequacy of a representation can be seen or justified using informal arguments.

Consider the (Db) prov, req, and std maps. These maps are adequately represented as their domain is the set of elements of the 'c' list (by the Dbl invariant, all the elements of the list are unique), and their ranges are taken from the corresponding positions in the 'h' list (i.e. at index inverse(c)(cn), for any element cn in c). Observe that h is invariantly the same length as c; thus h(inverse(c)(cn)) is always defined. In the case of the req map, a range element (also a map) is well-defined by a similar argument. That is, that the elements of vns are unique and that h(i).vns and h(i).req are invariantly of equal length for any i in the indices of h. For the remaining maps the only modification to the previous argument is that the disjoint domains, of mod and sys/uses, are restricted to those elements of c whose corresponding h elements are defined to be Modules and Systems respectively.

Cost-effectiveness dictates that many rules must be used as check-lists, and only subject to formal proof if and when an error is suspected in the design.

Two Sample Dbl Transformer Operations

The specification of the state transformer operations is fairly simple following closely their Db equivalents. Update and delete operations are now, largely, in terms of lists rather than sets. Only a couple of operations are included here to give the reader a feeling for what Dbl operations look like. Note that "^" is the list concatenation operator.

```
/* SVCD FIRST-LEVEL REFINEMENT OF TRANSFORMER OPERATIONS */

   globals D : wr Dbl

26.0 ADD-SPEC1(CN:Cn, PR:Fac-set, VERS:Vn-listl, RS:Fac-set-listl, STAND:Vn)
  .1 ex-AlreadySpec      cn ∈ elems d.c
  .2 ex-NotUnique        ~isdisj(pr, union elems rs)
  .3 ex-IllStand         stand ∉ elems vers
  .4 ex-DupVersions      ~isuniquel(vers)
  .5 ex-Inconsistent     len vers ≠ len rs
26.6 post d' = mk-Dbl(d.c^<cn>, d.h^<mk-Def(pr, vers, rs, stand, NIL)>)


27.0 ADD-MOD1(CN:Cn, VERS:Vn-listl, IMPLS:Imp-listl)
  .1 ex-NoSuchComp       cn ∉ elems d.c
  .2 ex-AlreadyDef       ~NoSuchComp &  defn(d.h(i))=NIL
  .3 ex-VersionsDiffer   ~NoSuchComp & vers ≠  vns(d.h(i))
  .4 ex-Inconsistent     len vers ≠ len impls
  .5 post d' =
  .6  mk-Dbl(d.c, modl(d.h, i, mk-Def(d.h(i).prov, d.h(i).vns, d.h(i).reqs,
  .7                                  d.h(i).std, impls)))
27.8   where i ≙ inverse(d.c)(cn)
```

The question now is: do the new operations successfully model the corresponding abstract operations?. By model is meant that each operation has a sufficiently wide domain and yields appropriate results. This is stated formally below.

For all Db operations OP, and Db1 operations OP1:

```
(∀r∈Db1; invDb(r) &
  pre-OP(retrDb(r), args) => pre-OP1(r, args) )

(∀r∈Db1; invDb(r) &
  pre-OP(retrDb(r), args) & post-OP1(r, args, r', result)
   =>
       post-OP(retrDb(r), args, retrDB(r'), result) )
```

Formal proofs, of the above were not performed during the development. Instead, appropriate instatiations were included in the documentation labelled Assumed. Additionally, as with the original Db transformer operations, the 'preservation of validity' rule is also applicable to the Db2 transformers.

SECOND-LEVEL REFINEMENT

We now refine the Db1 state, and associated operators, in order to arrive at even more implementation-oriented types and objects. The primary objective of this refinement stage is to replace the sets of complex objects by some combination of data types directly (or almost directly) available to us in Pascal. Firstly, however, let us look at the expected relative frequency of use of the various types of database operators that we have defined. In our given application area we expect the database to be interrogated frequently, and also added to fairly often; deletions should be much rarer. The important design decision is therefore taken not to delete information physically. Instead, information to be deleted is marked as being no longer valid. This makes implementation of the delete operations trivial. The small penalty is that the other operations must check an objects 'liveness' flag.

Below we present the new state to demonstrate how machine-oriented the design has become. Of course during the actual development a Db2 data type invariant was documented, a retrieve function formulated, and the operations re-specified in terms of Db2. At this stage the operations are complex and require numerous auxiliary functions.

```
/* SVCD SECOND-LEVEL REFINEMENT OF STATE */

28.Ø  Db2 ::  c : Cn-list
  .1          h : Def-list
  .2          v : Vn-list
  .3          f : Fac-list
  .4          m : Imp-list
```

```
.5          s : Compo-listl-list
.6          u : Nat-set-list

.7      Def ::  live : Bool
.8              type : { UNDEF, MOD, SYS }
.9              prov : Nat-set
.10              std : Nat
.11              vns : Nat-listl
.12            vlive : Bool-listl
.13              req : Nat-set-listl
.14             defn : Nat-list

28.15       Compo :: sc : Nat  sv : Nat
```

Annotation

All members of the generic sets are replaced by positive integers which index an appropriate list of the original type. For example, the list of version names per component is refined into a list of integers per component plus a list of all version names in use. Similarly, the sets of facility names become sets of integers indexing a list of all facility names in use. Note how each 'Def' has a type field (28.8). The interpretation of a component's definition (28.14) depends on the value of type. If type is MOD the elements of defn index the list of implementations (28.4). If type is SYS the elements index the list of system compositions (28.5). Finally, if type is UNDEF the defn list is undefined.

THIRD-LEVEL REFINEMENT

This is the final refinement, and consists of a program written in Pascal. Critical areas of the code contain assertions which relate to the second-level refinement. The vast majority of program annotation is, however, the usual informal commentary. In this section final implementation decisions and constraints are defined. A concrete syntax is also given.

The unbounded lists of the second-level refinement are replaced by fixed length arrays (the prototype allows, for example, a maximum of 53 component names). The Pascal set type is used to implement the mathematical sets in a very direct manner. In an actual production implementation, with a large number of components, this would be either impossible or extremely expensive in terms of store depending upon the particular Pascal system. This is because the usual method for implementing Pascal sets is via bit-strings (i.e. one bit per possible set member). The invariant predicate asserting non-circularity of system compositions is implemented via S. Warshall's transitive closure algorthm [12]. Component, version, and facility name look-up is performed by the method known as 'Open hash with quadratic retry'. This routine was taken almost verbatim from [13].

Figure 2 is a concrete syntax for SVCD. The syntax is the familiar multi-character keyword type. Input is free-format with 'white space' used to separate basic tokens. This input method was chosen over a special single key per operator, or a template approach, for two reasons. Firstly, the program code to recognise such input is derivable in a mechanistic way directly from the Extended Backus-Naur Form of syntax. This was done, followed by the stepwise enrichment of the syntax skeleton in the fashion familiar

from compiler development [13]. Secondly, for simplicity it was decided to hold the database on disk in source format; thereby allowing input to be prepared off-line via a standard editor, and, indeed, allowing the disk-based database to be manipulated as an ordinary text file.

```
SVCD-OP = add-spec | add-mod | add-sys |
          del-spec | del-mod | del-sys |
          add-modvar | add-sysvar |
          del-modvar | del-sysvar |
          "supplies" cn | "needs" cn"."vn |
          "choice" [facs] "end" | "standard" cn |
          "undefined" | "unused" | "whereused" cn |
          "in" string | "out" string | "quit".

add-spec = "spec" cn "provides" facs ";"
           ["std"]vn["requires" ("all-of" cns) | facs]
           {";" ["std"]vn ["requires"("all-of" cns)|facs]}
           "end".

add-mod = "mod" cn ";"
          vn "impl" imp {";" vn "impl" imp}
          "end".

add-sys = "sys" cn "use" cns ";"
          vn "=" compos {";" vn "=" compos}
          "end".

del-spec = "delspec" cn.

del-mod = "delmod" cn.

del-sys = "delsys" cn.

add-modvar = "modvar" cn"."vn
             ["requires"("all-of" cns)|facs]
             "impl" imp.

add-sysvar = "sysvar" cn"."vn "=" compos "end".

del-modvar = "delmodvar" cn"."vn.

del-sysvar = "delsysvar" cn"."vn.

facs = fac {"," fac}.

cns  = cn {"," cn}.

compos = cn"."vn {"," cn"."vn}.

fac = ident.

cn = ident.

vn = ident.

imp = string.

ident = (letter|digit){letter|digit|"-"|"_"}.

string = """{character}""".
```

Fig. 2. SVCD Concrete Syntax.

Component, version, and facility names are written as letters and digits possibly with embedded underscores and hyphens. Implementations are written as quoted character strings. Although their semantics are not defined, in this paper, typically they would identify a disk file containing the information on how to regenerate a component. Comments are allowed in the SVCD input. They are enclosed in the braces "{" and "}", may be placed between any two input tokens, and are completely ignored. Two methods of indicating requirements are allowed. The facilities may be listed explicitly, or the "requires all-of" option may be used. In the latter case, a list of previously specified component names is given. The requirements are then taken as being the union of the provisions of the stated components.

Three extra commands, not formally specified, are included for initialising and saving the (in-store) database. To initialise from a disk file the user enters:

in "<file-spec>"

The input file should end with the parameterless command "quit". If quit is typed at the keyboard control is returned to the top-level command interpreter. Before a quit, the current in-store database may be saved on disk by entering:

out "<file-spec>"

Exceptions in the formal specification are translated into explicit error checks in the prototype SVCD. Upon discovering an error in the input SVCD issues an appropriate message and abandons the current operation. This is rather poor man/machine interaction, but suffices for our prototype implementation.

CONCLUSIONS AND EVALUATION

A simple, yet useful, system version control database was developed in a pleasingly rigorous fashion. The majority of the development was performed whilst the author was a student at the Programming Research Group of Oxford University. Recently, at Manchester University, the specification language was changed to the presented dialect of Meta-IV, and a couple of features extended or improved. This was the author's first experience with a rigorous development method. It may, therefore, be of interest to note the estimated versus actual time taken for specification and implementation. Eight weeks were originally scheduled for specification and data type refinement. In reality about ten weeks were spent on this activity. Numerous abstractions were tried and discarded over this period (mainly in the interests of simplicity). The choice of the various development stages was crucial and changed several times. Four weeks were scheduled for coding in Pascal and implementation on a Digital Equipment LSI-11/2 single-user workstation. The actual time spent on implementation was two weeks.

During the implementation stage six errors were discovered in the specification and/or refinements. Of these, five were trivial, and were corrected quickly. The sixth was more worrying. In the original specification it was possible for the ADD-SPEC operator to break the self-containment invariant of existing system compositions. When the Gandalf system was studied it was deemed a simplification to remove the scope-restricting feature from system definitions. This was a mistake. An even bigger mistake was the decision to postpone proving the validity rule for ADD-SPEC, and ADD-SYS. Two important lessons can be learnt from this experience.

Firstly, that attempting to simplify a system before it is fully comprehended is foolhardy. Secondly, that operation validity should only be assumed for trivial cases (not those with complex pre-conditions).

Sixteen final-year undergraduate students at Manchester, working in four groups of four, have studied and extended the SVCD specification. For example, one group added the specification of project management facilities. During this period only one additional error was shown to exist in the SVCD specification. Under certain, obsure, conditions the NEEDS interrogation operation produced erroneous output. The correction of this problem actually simplified the specification.

Perhaps the most important lesson learnt during this development was the need for machine support of formal specifications and their refinements. Frequently, pages and pages of handwritten formulae were discarded and re-drafted, often to effect only trivial modifications. The majority of the simple, but annoying, errors could have been detected by a syntax and type checker. Decisions to change, or invent new, notation were not taken lightly because of the amount of paper work involved. The repeated instantiation of verification conditions was tedious. It is worth stressing that the aforementioned examples of the sort of tools desired by the author are, in many ways, typical. Automatic theorem provers, proof checkers, or program verifiers were missed far less.

ACKNOWLEDGEMENTS

I am indebted to many people at the Oxford Programming Research Group and also my present colleagues at Manchester. In particular, I am especially grateful to Jean-Raymond Abrial, A. Nico Habermann, Tony Hoare, Cliff Jones, Bernard Sufrin, and the University of Manchester students in the 1982 CS301 project class.

REFERENCES

[1] C.B. Jones, "Software Development: A Rigorous Approach", Prentice-Hall International, 1980.

[2] A.N. Habermann, "The Gandalf Research Project", Dept. of Computer Science, Carnegie-Mellon University, Research Review 1978-1979, 1980.

[3] A.N. Habermann and D.E. Perry, "System Composition and Version Control for Ada", in Software Engineering Environments, ed. H. Huenke, North-Holland, 1981.

[4] D. Bjorner and C.B. Jones, "The Vienna Definition Method: The Meta-Language", Lecture Notes in Computer Science, Vol. 61, Springer-Verlag, 1978.

[5] J.E. Stoy, "Denotational Semantics - the Scott-Strachey Approach to Programming Language Theory", MIT Press, 1977.

[6] D.M. Ritchie and K. Thompson, "The Unix Time-Sharing System", Bell System Technical Journal, pp 1905-1930, July-August, 1978.

[7] L.W. Cooprider, "The Representation of Families of Software Systems", Ph.D Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1979.

[8] W.F. Tichy, "Software Development Control Based on System Structure Description", Ph.D Thesis, Dept. of Computer Science, Carnegie-Mellon University, 1980.

[9] A.N. Habermann and D.E. Perry, "Well-formed System Compositions", Technical Report CMU-CS-80-117, Dept. of

Computer Science, Carnegie-Mellon University, 1980.

[10] C.B. Jones, "Development Methods for Computer Programs Including a Notion of Interference", Oxford University Computing Laboratory, Technical Monograph PRG-25, June 1981.

[11] F. Cristian, "Robust Data Types", in Program Specification, Proceedings of a Workshop, Aarhus, Denmark, August 1981, Lecture Notes in Computer Science, Vol. 134, Springer-Verlag, 1982.

[12] S. Warshall, "A Theorem on Boolean Matrices", J.ACM, Vol. 9, No. 1, pp 11-12, 1962.

[13] N. Wirth, "Algorithms + Data Structures = Programs", Prentice-Hall, 1976.

Affiliation of Author

This work was performed at the University of Oxford Programming Research Group and the University of Manchester Department of Computer Science. At Oxford the author was supported by a U.K. Science and Engineering Research Council Advanced Course Studentship.

I.D. Cottam is with the University of Manchester Department of Computer Science, Manchester, England.

OXFORD UNIVERSITY COMPUTING LABORATORY
PROGRAMMING RESEARCH GROUP TECHNICAL MONOGRAPHS

JULY 1982

This is a series of technical monographs on topics in the field of computation. Copies may be obtained from the Programming Research Group, (Technical Monographs), 45 Banbury Road, Oxford, OX2 6PE, England.

PRG-2 Dana Scott
Outline of a Mathematical Theory of Computation

PRG-3 Dana Scott
The Lattice of Flow Diagrams

PRG-5 Dana Scott
Data Types as Lattices

PRG-6 Dana Scott and Christopher Strachey
Toward a Mathematical Semantics for Computer Languages

PRG-7 Dana Scott
Continuous Lattices

PRG-8 Joseph Stoy and Christopher Strachey
OS6 - an Experimental Operating System for a Small Computer

PRG-9 Christopher Strachey and Joseph Stoy
The Text of OSPub

PRG-10 Christopher Strachey
The Varieties of Programming Language

PRG-11 Christopher Strachey and Christopher P. Wadsworth
Continuations: A Mathematical Semantics for Handling Full Jumps

PRG-12 Peter Mosses
The Mathematical Semantics of Algol 60

PRG-13 Robert Milne
The Formal Semantics of Computer Languages and their Implementations

PRG-14 Shan S. Kuo, Michael H. Linck and Sohrab Seadat
A Guide to Communicating Sequential Processes

PRG-15 Joseph Stoy
The Congruence of Two Programming Language Definitions

PRG-16 C. A. R. Hoare, S. D. Brookes and A. W. Roscoe
A Theory of Communicating Sequential Processes

PRG-17 Andrew P. Black
Report on the Programming Notation SR

PRG-18 Elizabeth Fielding
The Specification of Abstract Mappings and their Implementation as B$^+$-trees

PRG-19 Dana Scott
Lectures on a Mathematical Theory of Computation

PRG-20 Zhou Chao Chen and C. A. R. Hoare
Partial Correctness of Communicating Processes and Protocols

PRG-21 Bernard Sufrin
Formal Specification of a Display Editor

PRG-22 C. A. R. Hoare
A Model for Communicating Sequential Processes

PRG-23 C. A. R. Hoare
A Calculus of Total Correctness for Communicating Processes

PRG-24 Bernard Sufrin
Reading Formal Specifications

PRG-25 C. B. Jones
Development Methods for Computer Programs including a Notion of Interference

PRG-26 Zhou Chao Chen
The Consistency of the Calculus of Total Correctness for Communicating Processes

PRG-27 C. A. R. Hoare
Programming is an Engineering Profession

PRG-28 John Hughes
Graph Reduction with Super-Combinators

PRG-29 C. A. R. Hoare
Specifications, Programs and Implementations

PRG-30 Alejandro Teruel
Case Studies in Specification. Four Games