University of Glasgow
Department of Computing Science
Lilybank Gardens
Glasgow G12 8QQ

University of St Andrews
Department of Computational Science
North Haugh
St Andrews KY16 9SS

A Requirements Modelling Tool Built in PS-algol

Persistent Programming
Research Report 54
December 1987

Preface

The two papers included here describe aspects of an implementation of a Requirements Modelling Language in the language PS-algol. The first paper describes the motivation behind the work and the functionality of the system. This paper was given at the Second International Workshop on Persistent Object Systems held at Appin during August 1987 and will also appear in the proceedings of that workshop.

The second paper describes the implementation of a type hierarchy in which objects are allowed to change type and has been submitted to the European Workshop on Extending Database Systems. This paper was written as a response to a paper given by Professor Stanley Zdonik at the International Workshop on Database Programming Languages held at Roscoff in France during September 1987. However, the ideas in the paper essentially describe the type system used in the Requirements Modelling program, with a few minor extensions. Therefore the two papers have been bound together to form this report.

Requirements Modelling in a Persistent Object Store

Richard Cooper and Malcolm Atkinson

Abstract

The history of Computer Science may be viewed as the development of methodologies which enable human beings to specify complex commands to machines with increasing informality and greater levels of abstraction. In the field of Software Engineering, the **Requirements Model** has been described as the first in the series of models with which an informal specification of requirements is transformed into a machine executable form. A desirable development, therefore, would be the production of software which enables Requirements Models to be specified and then executed. This has proved difficult to do in conventional programming systems, but with the appearance of Persistent Programming Languages, many of the difficulties, which are arbitrary in nature, disappear.

This paper describes the first steps in an attempt to produce an executable form of Greenspan's Requirements Modeling Language (RML) in PS-algol. The program, PSRML, provides a graphical interface to a system in which Entities and Activities can be specified. It is planned to extend the program in a variety of ways.

Introduction.

In his thesis, Greenspan stated that what was needed for Requirements Modelling was a formal language for specifying "a model that reflects the content and structure of the application world" [GREE84]. He further stated a number of principles to be adopted when designing such a language:

- it should be **object-centred** - that is, the elements of the language should be objects representing the concepts and entities of the modelled world;
- it should provide mechanisms of **abstraction** to assist the management of complex data;
- of particular usefulness are the well-known abstraction mechanisms -

  aggregation - the creation of entities out of their component parts;

  classification - the collection of similar objects into sets;

  and **generalisation** - the organisation of classes of objects into ISA hierarchies;
- the language should be capable of expressing **assertions, entities** and **activities;**
- there should be **uniformity** in the way in which objects within the language are treated;
- the language should be **formally** expressed;
- stress is also put on the value of "box-and-arrow" languages to provide a **visual** bridge between the mental model and the formal one.

To implement such a language has, in the past, proved to be of considerable difficulty using conventional programming languages. Most programming languages are either lacking in simplicity or power or flexibility or coherence or all of these at once. That is, languages which are simple and elegant such as the object-oriented languages lack sufficient flexibility in the structures which they may model. Many languages gain power by "tacking on" constructs until they become baroque. Most languages which provide the flexibility to tackle the guidelines given above do so by reference to external components. That is they make use of file managers to handle data management and graphics packages to handle the visual aspects of the program. The complexity of these systems has proved a substantial barrier to the implementation of any kind of Conceptual Modelling schemes. The introduction of Persistent Programming Languages takes a significant step in reducing these difficulties.

The PS-algol Programming Language.

PS-algol [PSAL] is the first of a series of languages being designed by the Persistent Programming Research Group, based at the Universities of Glasgow and St. Andrews [ATKI87]. It is a member of the algol family of languages with the following features:

Orthogonal Persistence: A data object is treated in the same way whether or not it is expected to persist beyond the end of the current program run. In essence this means that the low-level data management is handled by the system automatically.

Data Type Completeness: No arbitrary restrictions are placed upon the way in which different data types may be manipulated. All data types may be stored and retrieved, passed as the arguments and results of procedures or be the fields of structures.

First-class Procedures: In particular, procedures may be so manipulated [ATKI85]. This provides particularly flexible mechanisms for the modular development of programs, increasing software re-use and providing a method for supplying data in the form of Abstract Data Types. Of particular importance here, however, is the power which first-class procedures provide in modelling "active data" [COOP87a].

A structuring mechanism using an **Extensible Union Type:** PS-algol provides a mechanism for introducing new classes of objects each consisting of a set of sub-objects specified as its fields. The powerful feature of this mechanism is that all objects constructed to be an instance of any such class have the same type - **pntr**. Thus within a strictly type-checked language, there is a facility for manipulating objects of different types and deferring the type checking until the last possible moment [ABM88].

The provision of **Graphics Facilities** within the language: PS-algol has two graphical data types: one for the manipulation of line drawings in the Cartesian plane and one for rectangular bit-maps. This enables the user interface to an applications program to be developed in the same language as the rest of the program, with a consequent reduction in system complexity. It also provides an ability to model graphical data in a uniform way with textual, numerical and, as mentioned above, active data.

The provision of **a compiler, callable at run-time:** A particular consequence of the availability of first-class procedures is that it has been possible to provide a call to the compiler as a system function. That is, the program can build the source code of a procedure as a string, pass it to the compiler and then manipulate the resulting compiled code as if it had been put into the program in the normal way. The implications for the development of powerful polymorphic programs in heterogeneous environments which are structured for efficiency are discussed in [DEAR87] and [COOP87b].

It should immediately be noted how these facilities correspond to Greenspan's principles. If not an "object-oriented" language, PS-algol is certainly a language which makes the programmer oriented towards handling objects. The data-type completeness, orthogonal persistence and presence of graphical types naturally leads to a uniformity of approach. The graphical types also facilitate the provision of a "box-and-arrow" language alongside the formal language. Several experiments at the University of Glasgow are in progress, for instance, for the construction of graphical interfaces to standard database systems. Further, the abstraction mechanisms are quite simply implemented in PS-algol by using the flexible structuring mechanism, which will now be described.

A new data class is introduced, for instance by -

structure *Address*(**int** *HouseNo*; **string** *street, city*)

which introduces a new class, *Address*, consisting of three fields, an integer for the house number and two strings for the street and city names, and a constructor for that class. This enables instances of the class to be constructed, for instance,

let *CS* = *Address*(17, "Lilybank Gdns", "Glasgow")

Two points are to be emphasised. Firstly the object named *CS* is of type **pntr** as is every object constructed by any constructor introduced by **structure** command. Secondly, although the fields of this particular structure were merely integers and strings, a field can take any of PS-algol's legal types. Thus fields can be graphical objects, procedures or pointers to other objects.

Returning to the three abstraction mechanisms, clearly the **structure** command itself is an implementation of **aggregation** - the *Address* structure is an aggregation of *houseNo, street* and *city*. The **classification** of a set of objects can be implemented in a number of ways in PS-algol. Two system provided features are: the **vector**, which groups together an ordered set of elements of the same type (but they may be any type at all); and the **table**, which is an unordered set of key, value pairs. Alternatively, using **pntr** fields, a set may be represented as a list, a tree or any complex structure that is appropriate. (In fact, tables themselves are implemented as a mixture of hash-coded trees ending in linear lists to resolve clashes).

The notion of ISA-hierarchies and the abstraction mechanisms of **generalisation** and its inverse **specialisation** are simply implemented using pointer fields to represent the edges of the graph representing the hierarchy. Here we present two different ways of representing such a hierarchy.

A general model for representing a complete hierarchy might use:

structure *ISAnode*(**string***Class*; **pntr***superClass*, *Values*)

so in the case of modelling the three classes:

- PERSON - with attributes Christian name and surname;
- STUDENT - subtype of PERSON, with added attribute, matriculation number;
- and STAFF - also a subtype of PERSON, with added attribute, staff number.

we would create the structures

structure *PERSON*(**string** *cname*, *sname*)
structure *STUDENT*(**int** *matricno*)
structure *STAFF*(**int** *staffno*)

and represent a staff member by two objects:

let *RCP* = *ISAnode*("PERSON", **nil**, *PERSON*("Richard", "Cooper"))
and **let** *RCS* = *ISAnode*("STAFF", *RCP*, *STAFF*(1234)).

The advantage of this scheme is that it allows the hierarchy to be traversed polymorphically in a fairly obvious kind of way. The availability of the compiler at run-time allows us to make a more efficient representation of the same data, viz:

structure *PERSON*(**pntr** *PERSONsupertype*; **string** *cname*, *sname*)
structure *STUDENT*(**pntr** *STUDENTsupertype*; **int** *matricno*)
structure *STAFF*(**pntr** *STAFFsupertype*; **int** *staffno*)

and so lose the level of indirection above, as in:

let *RCP* = *PERSON*(**nil**, "Richard", "Coóper")
and **let** *RCS* = *STAFF*(*RCP*, 1234).

Note that we have chosen to represent an object by a set of type instances, rather than by duplicating the substructures in each type definition. Note also that although the single pointer field, *supertype*, has been used here in a mechanism which only supports single inheritance, there is no intrinsic reason why multiple inheritance could not be specified by use of a list or a vector of pointers.

The Language RML and Some Descendants.

Requirements Modeling Language (RML) was developed as part of the TAXIS project at the University of Toronto by Sol Greenspan. It is fully specified in his thesis report [GREE84], but its essentials are now described.

RML attempts to model three kinds of object: the **entity**, the **activity** and the **assertion**. Classes of each of these are defined by a syntax which makes the three abstraction mechanisms immediately apparent. For instance a new class of entity is introduced by a specification in which the class it is in, the aggregated properties and the classes supertype(s) are explicitly stated. For instance:

CHILD in PERSON_CLASS isa PERSON with
part reading_age: AGE_VALUE
association guardian: PERSON
invariant guardian•age > 21

which means that a child in the set of all people, is a subtype of PERSON, has reading_age as an intrinsic part, has an associated guardian entity and the constraint that its guardian is over 21 years of age.

Similarly, activities have components, supertypes and are grouped together, as in for instance:

ADMIT_PATIENT in ACTIVITY_CLASS with
input p: PERSON
control w: HOSPITAL_WARD
phys: PHYSICIAN
output pt: PATIENT
precondition arrvl: ARRIVAL(who:p)
postcondition admitted?: IN_HOSPITAL(who:p)
part check-id: CHECK_ID(p)
put: CHOOSE_WARD(w, phys)

in which, the "arguments", "local variables" and "results" of the activity are specified, followed by assertions representing pre- and post-conditions. Finally, the body of the activity is described as an **unordered** set of "calls" to other activities, with values of their parameters being supplied.

Finally, assertions may be specified, an example being:

IN_HOSPITAL in ASSERTION_CLASS with
argument p: PERSON
part patient?: IN(p, PATIENT)
present?: PHYSICALLY_PRESENT(who:p)

in which, we see the assertion presented as a list of arguments followed by an unordered set of sub-specifications.

In RML, a **model** consists of a set of definitions of these objects and Greenspan's thesis describes how models are developed by use of the diagrammatic language, SADT [SOFTEC]. Greenspan lays out plans to provide a consistency checker for models, but nowhere does he set out any thoughts on how to provide an "executable" version. This would be useful in that the behaviour of a model could be examined under various input conditions.

As part of the Alvey project to develop a semi-intelligent IPSE, attention was focussed on Greenspan's ideas, which was extended to be a process moel, by adding a new object class, the **role**, which may be viewed as an entity whose subparts include activities. Again, this was tied to a diagrammatic language, this time via the RAD - "Role Activity Diagram".

In order to create a version they could use, the me-too project then created a language which is essentially an executable subset of RML, called Teeny [BENN86]. Teeny is an interactive interpreter, which understands definitions like:

```
( CHILD
      IN (PERSON_CLASS)
      ISA (PERSON)
      WITH( ( part ( (reading_age AGE_VALUE) ) )
            ( assoc ( (guardian PERSON ) ) ) ) )
```

and

```
( ADMIT_PATIENT
      IN (ACTIVITY_CLASS)
      ISA (ACTIVITY)
      WITH ( ( input ( ( p PERSON ) )
            ( control ((w HOSPITAL_WARD)
                  (phys PHYSICIAN) ) )
            ( output ( (pt PATIENT ) ) )
            ( precond ( (arrvl ARRIVAL( p to 'who ) ) ) )
            ( postcond ( (admitted? IN_HOSPITAL( 'p who ) ) ) )
            ( apart ( (check-id (exec 'CHECK_ID(makeff( 'p who ) )
                  ( put ( exec 'CHOOSE_WARD( makeff(w ward) )
                                    (makeff ('phys doc)) ) ) ) ) ) ) ) ) )
```

Entities can then be instantiated, for instance by:

```
(inst 'peter (makeset 'CHILD) 12 nil)
```

which would create a child object, called peter, whose reading age was 12. Activities can then be executed by for instance:

```
(exec 'ADMIT_PATIENT (makeff( 'peter 'who )
                        ( 'hisward 'ward )
                        ('hisdoc 'doc ) ) )
```

The sub-activities are then queued and executed as their pre-conditions become satisfied. Clearly the syntax of this language leaves a lot to be desired, although this is understandable given the speed with which the language was implemented. The form of the interface - typed in commands - can also be improved upon.

PSRML: The Goals.

PSRML is a version of RML written in PS-algol. It aims to provide a fully working environment within which models may be developed. It will provide:

- a complete well-defined language for the specification of models;
- a graphical interface for the construction of models;
- commands to examine the behaviour of models by instantiating entity classes and executing activities;
- the graphical display of models as they are executed.

It is envisaged that a system with these features would be a powerful tool for the development of software systems. Given the description of PS-algol above, there seems no intrinsic reason why an elegant version of such a system could not be written. We now present a description of a first pass in the development of such a system, describing, first of all, the PSRML language, then the user interface and finally some implementation details.

PSRML: The Language.

At present, PSRML supports only activities and entities. A full description of PSRML Syntax is given as Appendix A. An entity type is defined, for instance, by:

```
entity CHILD isa PERSON with
      required guardian : PERSON
      optional readingAge : int
      modifier changeReadingAge;
```

in which the attributes of an entity are separated by whether they are mandatory or not, rather than whether they are intrinsic parts or associated entities. The types of the attributes can either be one of the PS-algol scalar types - **bool, int** or **string** - or an already defined entity type. Note that throughout PSRML, all elements encountered must already have been defined, although recursive definitions are permitted. This is a weakness of the language that we hope will eventually by removed.

An activity would be defined by:

```
activity ADMITPATIENT with
      input p: PERSON
      control w: HOSPITAL_WARD,
            phys: PHYSICIAN
      output pt: PATIENT
      body checkID: CHECK_ID( p ),
            assignDoc: pick( "DOCTOR" -> phys ),
            assignWard: CHOOSEWARD( phys -> w ),
            makePatient: make( "PATIENT", p, phys, w -> pt );
```

in which the specification of "variables" has been left essentially as in RML, but the syntax for specifying the body of the activity (corresponds to *part* in RML) is slightly changed. Each element of the body is a partially specified activation of some other activity. That is each element of the body consists of:

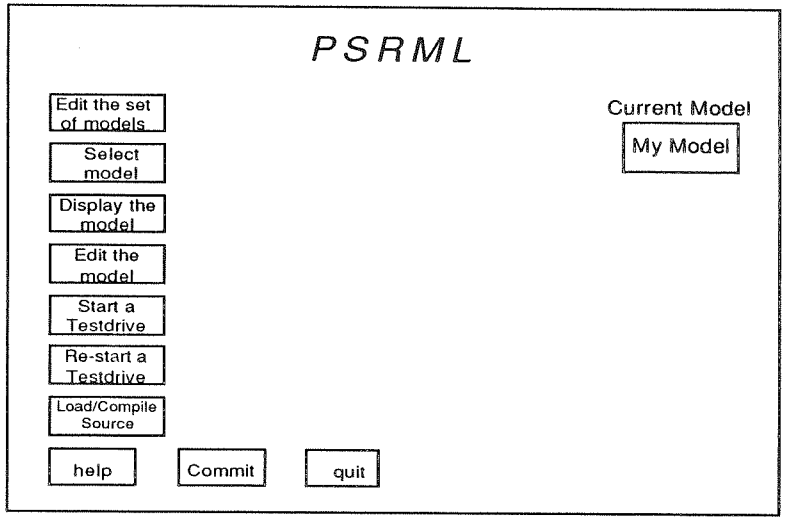
- an identifier;
- the name of some activity which is either a **base activity** or has been already specified;
- an opening bracket;
- an optional list of parameter values, which are separated by commas and are either literal values or parameter names, from which a value will be extracted at "run-time";
- also optionally, an arrow followed by a list of parameter names, which are to receive the output values from the called activity;
- a closing bracket.

Two points need to be noted. Firstly, the parameters are passed in the specified order for the activity. Secondly, a set of base activities had to be provided in order to give the executing system a bottom. In the example, the two base activities *pick* and *make* are used. The former, given the name of an entity type, provides a menu of all instances of the type for the user to choose between. *make* also takes in a class name together with a list of values for the attributes of the object and returns an instantiation of the class, with the given attribute values. Two other base activities which have so far been provided are: *makenull*, which given the name of a class returns an instance of the class, with its field values set to null; and *changeField*, which, given an object, the name of an attribute of such an object and a new value for that attribute, sets the value of that field to that provided.

Clearly this language is still too limited for serious work. The conclusions list some of the extensions which we have in mind to create a more powerful language.

PSRML: The User Interface.

PSRML is provided as a forms-based system, using many of the same modules as the Bibliographic Reference Database System described in [COOP87c]. The programs starts of with a menu display which looks like:



The seven boxes on the left-hand side of the screen are light buttons which provide the following functions:

Edit the set of models: activates a sub-menu in which the user is allowed to view, add and delete from the list of models, and to display or edit a particular one.

Select model: select, from the set of models, one to be currently active.

Display the model: display the currently selected model.

Edit the model: edit the currently selected model.

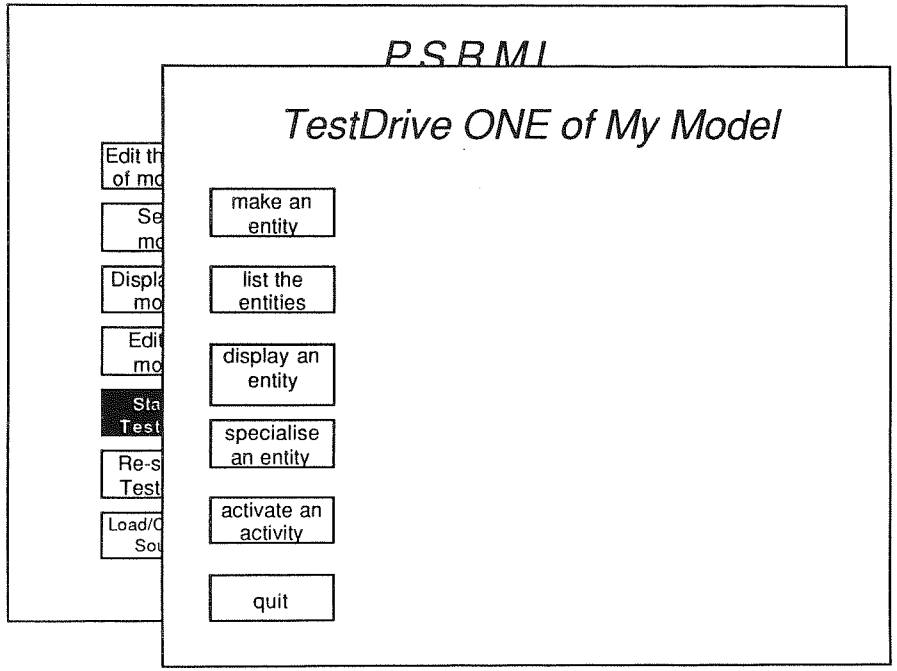
Start a Testdrive: initiate an instantiation of the current model.

Re-start a Testdrive: re-enter an instantiation of the current model.

Load/compile source: load a file from backing store and submit it to the compiler, storing compiled objects in the currently active model.

The display module shows a list of all the entity types, activities and "testdrives" which have been specified in the displayed model. Further work will implement the ability to click on elements in these lists to view further details. The editor module is also unimplemented, but will allow syntax-directed editing of parts of the model in a format compatible with the display module. Again the report [COOP87c] describes how such a feature has been implemented in the BRDS system. The source loader is a compromise to a non-persistent world in that it allows bulk-loading of a model from the file store.

The concept of the testdrive is that of a mixture of instantiation and execution of the current model. Starting or restarting a testdrive brings up the following window, which largely obscures the initial display:

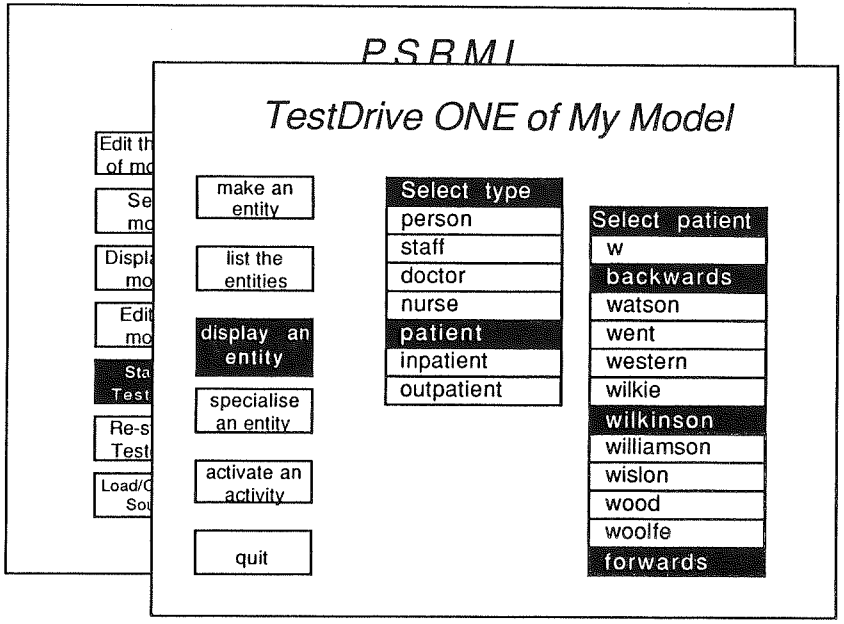


Within the test drive, five more options become available:

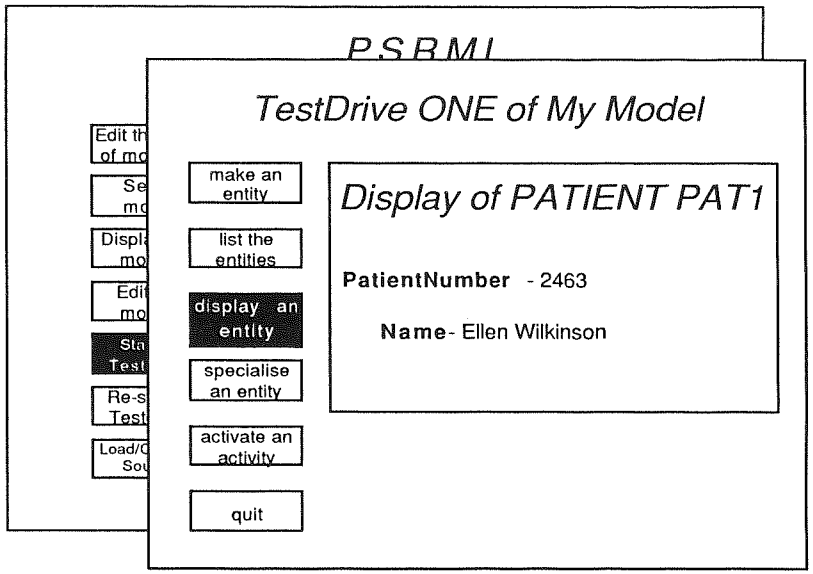
make an entity: The program requests by menu the type of entity which is to be instantiated and then requests, via a simple string editor, an identifier for the new object. Then by recursively calling a system supplied set of input procedures, the system requests values for the attributes of the new object. The new object is inserted into the set of objects of the given type. If the entity has supertypes, then values for the inherited properties will also be requested and appropriate structures will inserted to the sets of objects of those supertypes.

list the entities: A new window is opened which lists all the types of entities together with all instances of them.

display an entity: Again a type of entity is requested followed by the identifier for the object to be displayed. Both are requested by menu, following the general principle of never giving the user a chance to make a syntactic error (in this case by having to type in the identifier again) when this is unnecessary. The menus appear as follows:



The object is shown as in the diagram, with all the information for a patient, together with all the specific information inherited from any supertype. Note that if the object, *PAT1*, had been requested from the *PERSON* class, its properties which are relevent to the *PATIENT* class would **not** have been displayed. The information is displayed as follows:



specialise an entity: Having selected an object of a given type, a menu of all entity types which are sub-types of the current one is given The user selects one of these sub-types and all of the property values required to specialise the current object to the new class are requested. Note that the converse, generalising an entity has not been implemented. We have chosen to implement our inheritance hierarchy in only one direction. This again might be seen as a restriction which can later be removed if desirable. There should be no significant difficulty in achieving this.

activate an activity: Values for the input parameters are sought and a symbol table is generated with just these objects on it. Then all of the sub-activities are queued and each is examined in turn to see if all of its input parameters can be evaluated (these would either be literals or values already on the symbol table). If they can, it is called and all of its sub-activities are queued. Sub-activities are recursively queued until a base activity is encountered, whereupon it is executed and its results returned to its parent. The parent then updates its symbol table from the values returned as results from its sub-activities. Non-base activities are completed when all of their sub-activities complete. Eventually, either all of the original activity's sub-activities are completed or no more can be initiated. The latter leads to an error message and the former results in the user being asked for identifiers for each of the activity's results, which are then stored in their appropriate sets. This whole process should eventually be animated for the user's benefit.

PSRML: Implementation Issues.

PS-algol structures are used to hold entities and their types. The former are tailored to be an efficient representation of the entity we are modelling. For instance, a child would be represented by

```
structure ECHILD( int CHILD.readingAge; pntrCHILDguardian )
```

The entity type structure has a number of fields, including:

- the type name;
- the definition in PSRML source language - for later editing;
- vectors of pointers to sub- and super-types;
- pointers to lists of required, optional and modifier fields;
- and a group of four procedures which are automatically built to be efficient for the given entity type.

These four procedures are:

makenull - returns an empty instance of the class;
makeuser - returns an instance of the class, requesting from the user values for all the fields (including inherited fields);
printer - prints the values of the fields (including inherited fields);
changeField - changes the field of an entity to a given value.

For instance, in the entity class, *CHILD*, the makenull procedure would look like:

```
let makenull = proc( pntr TD, IP -> pntr )
    begin
        structure ECHILD( int CHILD.readingAge; pntrCHILD.guardian )
        ECHILD( 0, nil )
    end
```

while the *changeField* procedure looks like:

```
let changeField = proc( pntr ENT; stringFname; pntr Fvalue )
    begin
        structure ECHILD( int CHILD.readingAge; pntrCHILDguardian )
        structure intBox( int intV )  !  ditto for string and bool
        case Fname of
            "readingAge": ENT( CHILD.readingAge ) := Fvalue( intV )
            "guardian": ENT( CHILD.guardian ) := Fvalue
            default: print "SHOULDN"T GET HERE!"
    end
```

Similarly the *makeuser* and *printer* procedures are constructed to make use of the tailored *ECHILD* structure. They also make calls to system procedures which respectively get values form the user and print objects of the base types.

An activity is also implemented as a PS-algol structure, whose fields include:

- the activity name;
- the definition in PSRML source language;
- vectors of pointers to sub- and super-types;
- pointers to lists of formal input, control and output parameters (these lists are simply specified as a structure with two string fields containing the field name and type and two pointer fields to the value and the next parameter on the list);
- a pointer to a list of "partial" activations of other activities.

A partial activation is a version of an activation structure. This contains four pointer fields, which point to the activity, actual lists of the input and output parameters and to a progress record. In the partial activation, the progress record is always **nil**, while the value fields of the input list will point to either a packaged literal value or a packaged name for the parameter which is to be supplied.

When the activity is activated, each of the partial activations are checked to see if the input parameters have been sufficiently specified as yet - that is are they all literals or objects in the symbol table. If they are, parameter names are replaced by their current values and the progress record is set to "started". The sub-activity is then initiated. If it is a PSRML activity, its input parameters are put into its symbol table and the activity is activated in the same way as the parent activity. If the sub-activity is a base activity, some PS-algol code associated with the sub-activity is executed. Upon completion of either of these types of sub-activity, the progress record is set to "finished" and the output values are assigned to the parameter variables found in the output list. When the activity originally initiated by the user is finished, any outputs of the activity are stored in the database, having been associated with identifiers requested from the user.

Conclusions.

It has been possible to implement a minimal Requirements Modelling System in PS-algol. The programming effort for this task was less than one man-fortnight as the implementation language was so suited to the task in hand. There has been insufficient experimentation to see to what degree inconsistent models can be created in the system, but it would seem that the system can be made proof against such misuse. It is clear that the system is reasonably powerful and easy to use. PSRML code compiles much more quickly than we would have expected. However, many extensions need to be made.

Firstly, the syntax must be considerably expanded. Some expected developments in the near future should be:

- a return to Greenspan's *part* and *association* separation of entity properties - upon reflection this does model a real distinction;
- the ability to specify the **constancy** of properties;
- the ability to specify **sets** of objects;

- the inclusion of the whole PS-algol type system as base types, thus allowing the use of pictures as properties, for instance;
- the removal of the need for the definitions to be ordered with only backwards references - this is a real restriction in a requirements modelling language, much more so than in a programming language.

Secondly, **assertions** should be added, together with some notion of processes. We are unconvinced of the value of free standing assertions, but as *condition* fields of activities and entities, they are very useful. They can be simply implemented in PS-algol as procedures which return a boolean result.

Finally, an animated display of the activation of an activity would prove informative and helpful.

Acknowledgements.

The work reported here was funded partly by an ICL URC grant and partly by Alvey/SERC grant GR/D43259. We would also like to thank our colleagues on the IPSE2.5 Alvey Project for their collaboration in this work.

Bibliography

ATKI85 Atkinson, M.P. and Morrison R. - "Procedures as Persistent Data Objects", *ACM TOPLAS*, 7, 4, 539-559, Oct 1985.

ATKI86 Atkinson, M.P., Morrison R. and Pratten, G.D. - "Designing a Persistent Information Space Architecture", *proc. Information Processing 1986*, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

BENN86 Bennett, S. and Rowles, J. - "Teeny: an Executable Language Based on RML", me-too Internal Report SETC/IN/215, STC Technology Ltd., Newcastle-under-Lyme, 1986.

COOP87a Cooper, R.L. - "Applications Programming in PS-algol", *Persistent Programming Research Report*, 25, Universities of Glasgow and St. Andrews, 1987.

COOP87b Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D. - "Constructing Database Systems in a Persistent Environment", to be presented at the 13th Conference on Very Large Databases, Brighton, September 1987.

COOP87c Cooper, R.L., Atkinson, M.P. and Blott, S.M., "Using a Persistent Environment to Maintain a Bibliographic Database", *Persistent Programming Research Report*, 24, Universities of Glasgow and St. Andrews, 1987.

DEAR87 Dearle, A. and Brown, A.L. - "Safe Programming in a Strongly Typed Persistent Environment", Persistent Programming Research Report 33, Universities of Glasgow and St. Andrews, 1987.

GREE84 Greenspan, S.J. - "Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition", *Technical Report* CSRG-155, University of Toronto, March 1984.

PSAL87 "The PS-algol Reference Manual, Fourth Edition", *Persistent Programming Research Report*, 12, Universities of Glasgow and St. Andrews, 1987.

SOFTEC "An Introduction to SADT", Softech Inc., Waltham, Mass, USA.

Appendix A. The PSRML Syntax.

< *object-decl* > ::= < *entity-decl* > | < *activity-decl* >

< *entity-decl* > ::= **entity** < *entity-name* > [**isa** < *entity-name-list* >]
[**in** < *entity-extent-list* >] **with** < *entity-group-list* >

< *entity-name-list* > ::= < *entity-name* > { ,< *entity-name* > }*

< *entity-extent-list* > ::= < *entity-name* > { , < *entity-name* > }*

< *entity-group-list* > ::= < *entity-group-item* > { , < *entity-group-item* > }*

< *entity-group-item* > ::= < *optional-group* > | < *required-group* > |
< *invariant-group* > | < *modifier-group* >

< *optional-group* > ::= **optional** < *field-name* > : < *ref-type* >

< *required-group* > ::= **required** < *field-list* >

< *invariant-group* > ::= **invariant** < *predicate-list* >

< *modifier-group* > ::= **modifier** < *activity-spec-list* >

< *field-list* > ::= < *field-name* > : { **constant** } < *field-type* >
< *field-type* > ::= < *PSalgol-type* > | < *entity-name* > | **set of** < *entity-name* >

< *ref-type* > ::= < *entity-name* > | **set of** < *entity-name* >

< *predicate-list* > ::= < *predicate* > { ,< *predicate* > }*
< *predicate* > ::= < *predicate-name* > < *predicate-rule* >
< *predicate-rule* > ::= < *PSalgol proc(pntr -> bool)* > | < *rml-rule* >

< *activity-decl* > ::= **activity** < *activity-name* > [**isa** < *activity-name-list* >]
[**in** < *activity-extent-list* >] **with** < *activity-group-list* >

< *activity-name-list* > ::= < *activity-name* > { ,< *activity-name* > }*

< *activity-extent-list* > ::= < *activity-name* > { , < *activity-name* > }*

< *activity-group-list* > ::= < *activity-group-item* > { , < *activity-group-item* > }*

< *activity-group-item* > ::= < *uses-group* > | < *precond-group* > |
< *postcond-group* > | < *body-group* >

< *uses-group* > ::= **uses** < *field-list* >

< *precond-group* > ::= **precondition** < *predicate-list* >

< *postcond-group* > ::= **postcondition** < *predicate-list* >

< *predicate-list* > ::= ..... ! as yet incompletely specified

< *body-group* > ::= **body** < *activity-spec-list* >

< *activity-spec-list* > ::= < *activity-spec* > { ,< *activity-spec* > }*
< *activity-spec* > ::= < *activity-item-name* > : < *action* >
< *action* > ::= < *base-activity-application* > | < *activity-application* >
< *activity-application* > ::= < *activity-name* > (< *params-pack* >)
< *base-activity-application* > ::= < *base-activity-name* > (< *params-pack* >)
< *params-pack* > ::= {< *input-field-value-list* >} {-> < *output-field-value-list*>}
< *input-field-value-list* > ::= < *input-field-value* > {, < *input-field-value* > }
< *input-field-value* > ::= < *literal* > | < *entity-name* >
< *output-field-value-list* > ::= < *entity-name* > {, < *entity-name* > }

< *literal* > ::= < *integer* > | " < *string* > | true | false

Type Hierarchies and Type Evolution Using Deferred Binding in Building Database Application Programs

Cooper R.L. and Atkinson M.P.

Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ

Abstract.

The provision within the persistent language, PS-algol, of a extensible union type and a compiler which is callable at run-time, permits a range of times at which the binding of program objects together may be performed. Using these facilities, it becomes possible to implement database models fairly rapidly. This paper describes how this freedom may be exploited to support a type system which can evolve dynamically. The extensible union type is used to facilitate data structure extension, while run-time compilation is used to create software which at once dynamically responds to novel data and yet avoids the inefficiencies of interpretation. We conclude that our approach is of general value in database research in that novel methodologies can be more easily investigated.

Introduction.

In many languages, [MILN79, HARP86, LISK81, ALBA85], the time at which a program is bound to its data is restricted to the time of its compilation. It has been argued [MORR87, ATKI88] that such restriction causes severe problems in the production of large, portable application programs. In particular, the following requirements are not met:

- the need to distribute programs to run against a number of independent databases;
- the need to select which database is to be used at run-time;
- the need to combine data from a number of databases; and
- the need to update both the data definition and the program.

The language PS-algol has a more flexible approach to the time at which binding takes place. This paper describes the mechanisms provided and how this flexibility is used to create a type system within which objects can change type.

PS-algol [PSAL87] is a block-structured, strongly typed language with orthogonal persistence. Every object in the language consists of a quadruple of attributes: its name; its type; its value; and its constancy. Of these, only the value may change and then only if it has not been declared as a constant object. Objects in the program can be made to persist between program runs by the simple mechanism of making it reachable from a root node and executing a **commit** command. The type system is simple, but rich enough to include types to represent graphical data and to include procedures as first-class objects (the language is data-type complete). Thus numerical, textual and graphical data are all handled in the same way as each other and in the same way as "programs" (i.e. procedures).

Although the language is strongly typed, with all objects being bound to a type at compile-time, two mechanisms are provided which allow a program to retain flexibility over the nature of such bindings. These are the extensible union type and the function which calls the compiler at run-time.

To illustrate the value of the extensible union type, consider the PS-algol command:

```
structure address( int house; string street, city )
```

which introduces a new class of objects, named *address*, with one integer field and two string fields. The name, *address*, is introduced as a constructor for instances of the class, so:

```
let CS = address( "17", "Lilybank Gdns", "Glasgow" )
```

creates a new object which is in the class, *address*. The type of a reference to this object is the extensible union type denoted by **pntr** and the data type completeness of

PS-algol means that the fields of a structure may be of any type. In particular they themselves may be of type **pntr**. This provides a powerful modelling tool since complex structures may be represented in PS-algol. For instance the nodes of a binary tree of strings could be represented by objects of the class:

```
structure SBTnode( string Svalue; pntr left, right)
```

However, a further feature is that all objects created as instances of any of these class constructors are of the same type: **pntr**. This means that programs can be written which manipulate objects without knowing which class they are in. Thus we may generalise our binary trees to contain any type of value at the nodes, for instance using the structure:

```
structure GBTnode( pntr Gvalue, left, right)
```

where the *Gvalue* field now points to some (unidentified) structure containing the value. We could encompass the previous example by making it point to a structure containing just one string field. We can then provide generalised tree manipulation procedures, such as:

```
let add = proc( pntr newval, tree; proc( pntr, pntr -> bool ) lessthan )
    if lessthan( tree( Gvalue ), newval )
        then if tree(right) = nil
            then tree( right ) := GBTnode( newval, nil, nil )
            else add( newval, tree( right ), lessthan )
        else if tree(left) = nil
            then tree( left ) := GBTnode( newval, nil, nil )
            else add( newval, tree( left ), lessthan )
```

This procedure when provided with a new value, a tree of values of the same class and a procedure for deciding the order of any two members of the class, will insert the new value in the tree at the appropriate point. The procedure, *add*, does not need to know what class of object it is handling. It has left to the calling program the decision on what actual data class it is to be bound into the tree. Thus the procedure is truly polymorphic, but our style of writing polymorphic procedures explicitly identifies the token for the values, **pntr**, which is used in most implementations of parametric polymorphism [CARD85]. The advantage of exposing this is that we also have a structure with polymorphic fields, which we exploit extensively.

The provision of first-class procedures in the language has meant that it has also been possible to provide a function which calls the compiler. A procedure may be built as a string and then passed to the compiler function, which returns the compiled procedure. This may then be used in the same way as any of the statically written procedures of the program. Thus

```
structure procHolder( proc() theProc )
let Pstring = "proc(); write 12345 "
let emptyHolder = procHolder( proc(); nullproc )
let compHolder = compile( Pstring , emptyHolder )
let Pcompiled = compHolder( theProc )
Pcompiled()
```


will have the same effect as

```
let Pcompiled = proc(); write 12345
Pcompiled()
```

In the example of using the callable compiler, note that we have had to provide an empty structure in which the compiler can put the compiled procedure. This is because it has been written to be a completely general function which can compile procedures of any type. The mechanism which has been used to defer the choice of the type of the procedure is to force the caller to specify this by providing a package of the correct type. This means that although the compiler does not know the type of procedure it is to compile until it is actually called, the type of the resulting procedure is known at compile-time and so can be statically type-checked. To put it another way, the choice of type has been deferred when writing the compiler function, but not when writing the calling program.

To use the compiler as just illustrated would clearly be of little value. The mechanism comes into its own when writing programs designed to run against an unbounded set of different classes of data. We then can write programs which discover the class of data they are expected to deal with this time and, using string manipulation, merge the class information with the algorithm and compile the resulting procedure. For instance, a generalised procedure to add data to a binary tree, when it discovers that it has been given a string might **build** the following procedure:

```
let add = proc( string newval; pntr tree )
  begin
      structure stringBTnode( string value; pntr left, right )
      if newval < tree( value )
          then if tree(right) = nil
              then tree( right ) := stringBTnode( newval, nil, nil )
              else add( newval, tree( right ) )
          else if tree(left) = nil
              then tree( left ) := stringBTnode( newval, nil, nil )
              else add( newval, tree( left ) )
  end
```

where the parts which are underlined depend upon the particular type (in this case 'string') and have been embedded into a template representing the insertion algorithm.

These two mechanisms have given us ways of deferring the binding of program to data. In the case of **pntr** objects, programs can be written which, as they ignore the class-specific information, are not constrained by the class of the object. Only those programs specific to data of that class need be explicitly bound to it. The callable compiler gives us even more flexibility in that the program can bind itself to any novel data class that comes along at run-time - and yet the binding is done once and for all and is consequently safe. Note that the use of the callable compiler not only allows the type checks to be done once, but also allows other aspects of the computation to be factored out, such as choice of algorithm, constant folding and other optimisations.

Data Structure Extension.

One of the problems faced by strongly typed database programming languages is allowing a program to continue running after the data has been extended in some way. For instance, let us consider a program which doubles all the house numbers in a set of addresses. Our procedure to do this takes as its parameter a vector of pointers to the addresses and might be written as follows:

```
structure address( int house; string street, city )
let wreck = proc( *pntr V )
        for i = lwb( V ) to upb( V ) do V(i)( house ) := 2*V(i)(house)
```

Our problem comes when we wish decide to increase the information we know about addresses to include a postcode. To do this, the data structure is changed to:

```
structure address( int house; string street, city , postcode)
```

We put in new data with this structure and find that our previous procedure no longer works on this new data - we must provide a new version of *wreck* for the new data. Note that this version will not work on the original data - we need to keep the old version around too.

All this chaos could have been prevented by using an extra **pntr** field to defer the decision on whether there will ever be an extension to the *address* structure or not. We should bind *wreck* to the following structure:

```
structure address( int house; string street, city ; pntr extra )
```

Now when we add a field for the postcode, we provide a new structure, such as:

```
structure address2( string postcode; pntr extra2 )
```

and create subsequent instances of address, such as:

```
address( 17, "Lilybank Gdns", "Glasgow", address2( "G12 8QQ", nil ) )
```

Note that *wreck* (bound to the second definition of *address*) still functions correctly. It has no need of the postcode and so never attempts to dereference the *extra* field. Note also that we have created the *address2* structure with foresight - providing a second extra field in case we make further extensions. We could also then easily provide a small program which scans all our old addresses and put in their postcodes by changing the value of the *extra* field to point to an *address2* structure.

There are several reasons, however, why this approach is a poor one. Firstly, we require the programmer to have forethought that the structure may someday be extended. Secondly, the program now accesses the *postcode* field in a way which is both inefficient (requiring an extra level of indirection) and inconsistent with the way the other fields are accessed. Finally it does not provide a satisfactory means of withdrawing fields from a type and so is insufficient for the general problem of type evolution.

Objects Changing Type.

Zdonik [ZDON87] has considered the problems of the incremental development of data in the context of an object changing its type within a hierarchical type system. His type system has the following properties:

- an object X consists of instances of a set of types;
- the types of this set which have no subtypes are called *most immediate subtypes*;
- two types one step away from each other in the inheritance hierarchy are called the *direct subtype* and the *direct supertype* of each other;
- two operations are defined on type instances -
  - *addType* specialises an object by creating an instance of a subtype of one of its most immediate subtypes;
  - *deleteType* generalises an object by removing one of it's most immediate subtypes;
- some types are defined to be **essential**, that is an object may never stop being of this type (a *person* type is given as an example);
- types may also be defined to be **exclusionary**, that is objects can only be of that type if they are created to be (a *child* type is given as an example, since a person having stopped being a child, can never revert to being one);
- changes of type may then be considered to be a sequence of *addType* and *deleteType* operations.

Using PS-algol's extensible unions, we now show how to implement Zdonik's type evolution. We require two general structures for types and instances. For types, we specify a structure of the following form:

```
structure type( string typeName;
                bool essential, exclusionary;
                proc(->pntr) make,      ! procedure to create an instance
                proc( pntr ) display,   ! procedure to display an instance
                *pntr subtypes,         ! point to other
                      supertypes,       !   type structures
                      instances )       ! points to a set of instances
```

in which we support the inheritance hierarchy via pointer fields and provide two procedures for each type to create and display instances (we leave to a later stage how this may be done automatically).

For instances of types, the following structure suffices:

```
structure instance( pntr theType,           ! points to the type structure
                          theValues,        ! points to a specific structure for this type
                    *pntr subinstances, superinstances )
```

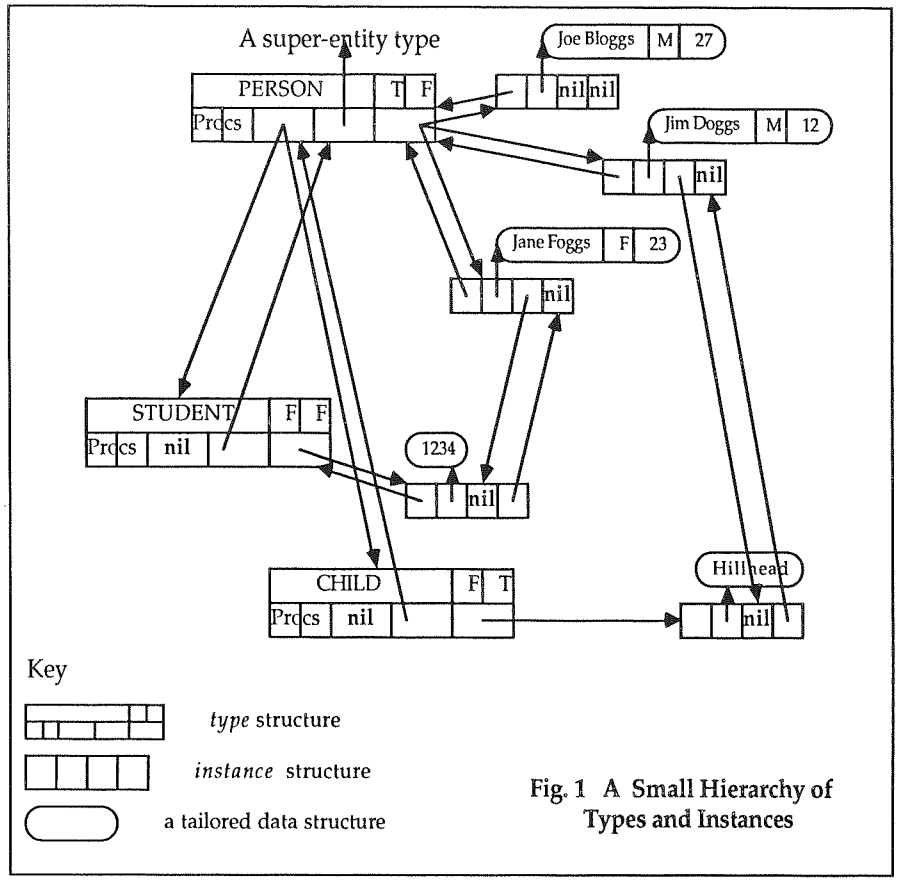


Fig. 1 A Small Hierarchy of Types and Instances

Here, *theValues* field is another example of the use of deferred binding in that it points to a structure like

structure *TypePERSON*(**string** *PERSONname*, *PERSONsex*; **int** *PERSONage*)

which is unknown to us as we write our *addType* and *deleteType* operations. Note also that once again these structures can be generated automatically by the program which adds types to the type hierarchy.

Fig. 1 shows a diagram showing a hierarchy of 3 types - *PERSON*, *STUDENT* and *CHILD* - and three instances of type *PERSON*, one of which is also a *STUDENT* and one of which is also a *CHILD*. Note that for any instance, all the information known about it is reachable by following *subinstances* and *superinstances* fields. Thus we can find the matriculation number of a student or the school of a child.

The two generic operations *addType* and *deleteType* can now be written:


```
let addType = proc( pntr X, t )          ! specialise object X to type t
  if t(exclusionary) then error( "can"t specialise to an exclusionary type" ) else
      begin
        let T = X(theType)
        if notIn( t, T(subtypes) )
            then error( t(typeName) ++ " is not a subtype of " ++ T (typeName) )
            else
              begin
                  let newX=instance(t, t(make)(), nullVector, @ 1 of pntr [X] )
                  addTOset( newX, t(instances) )
                  addTOset( newX, X(subinstances) )
              end
      end
```

New information about *X* is obtained by calling the *make* procedure for new type *t* and creating a new instance object. This new object has no subinstances as yet (indicating by setting its *subinstances* field to a null vector and has *X* as its only superinstance. After it has been created it is added to the set of instances of *t* and to the set of subinstances of *X*. A little more work would be required to handle multiple inheritance.

Removing information is similar:

```
let deleteType = proc( pntr X, T )       ! generalise object X to type T
  begin
      let t = X(theType)
      if t(essential) then error( "can"t generalise an essential type" ) else
        if notIn( t, T(subtypes) )
            then error( t(typeName) ++ " is not a subtype of " ++ T (typeName) )
            else
              begin
                  removeFROMset( X, t(instances) )
                  for i = lwb( X(superinstances) ) to upb( X(superinstances) ) do
                        removeFROMset( X, X(superinstances)(i) )
              end
  end
```

In this procedure, generalisation is achieved by removing all references to one of the nodes of *X*'s representation.

Using these two procedures as primitives it is possible to build programs with the ability to move objects about the type hierarchy within the restriction of essential and exclusive types. It would, for instance, be relatively simple to provide a procedure which, given an object and a type, "moved" the object to be of that type.

Polymorphic Tailored Procedures.

The availability of the compiler as a function permits another range of techniques for deciding at what point to bind program and data together. [DEAR87] and [COOP87a] describe a technique for tailoring a procedure to classes of data as yet unspecified at the time of writing the program. Briefly, the technique to write a procedure to implement an operation on a range of object classes proceeds as follows:

- the procedure discovers the class of the object as a string;
- by string-manipulation, it combines the class and the algorithm for the operation into a string containing a syntactically correct PS-algol procedure;
- the procedure is submitted to the compiler and the compiled procedure is run against the object.

[DEAR87] describes how this technique was used to write a safe browser, which traverses a PS-algol database and displays the values of objects which are encountered. [COOP87a] describes a relational database system in which relations were created as Abstract Data Types whose operations were tailored to be efficient for the particular relation.

Here we will describe how the same sort of technique may be used to create the type structures of our evolutionary type system described earlier. We will present our types to be added to the system in some Conceptual Modelling Language, like RML [GREE84], for instance:

```
essential type PERSON isa ENTITY
with attributes     name: string
                    sex: string
                    age: int;
```

We will present that string to a procedure *addNewType*, a sketch of which is given as Fig 2, to add it to our type hierarchy and to a lookup table of all the types.

The values of the fields of the *type* structure are elicited from the input *description*. This is done in fairly simple manner for the *typeName, essential* and *exclusionary* fields. The *subtypes* and *instances* fields are empty at the time of creation. The *supertypes* field is created by picking up supertype names from the *isa* clause in the *description* and then looking up the types in the type table (we are providing for multiple inheritance in this procedure). At the end of the procedure, the *subtypes* field of each of this type's supertypes has a pointer to the new type inserted.

The main complexity of *addNewType* resides in the automatically generated *Make* and *Display* procedures, which are built as strings, shown underscored. The first requirement for these procedures is a string containing the structure class to be used for storing objects of the class. This is built as *Structure*, which in the case of *PERSON* will be:

```
structure TypePERSON( string PERSONcname; string PERSONsex int PERSONage)
```

which is somewhat redundant, but should ensure no name clashes with objects of other types. Strings containing the sources of the procedures are then built up from *Structure*, a set of standard procedures for inputing or displaying simple data objects and a command creating the object from calls to these procedures. Access to the input/display primitives is facilitated by making their names be in a canonical form - "get" ++ the type name in the case of input.


```
let addNewType = proc( string description )
  begin
      let TypeName =            ! derived by string-manipulation
      let Essential =           !   from description
      let Exclusionary =
      let Supertypenames =      ! vector derived by string-manipulation
      let Supertypes = vector 1::upb( Supertypenames ) of nil
      for i = 1 to upb(Supertypes) do
            Supertypes(i) = s.lookup( Supertypenames(i), TypeTable )

      let AttribNames =         ! vectors of strings again
      let AttribTypes =         !   derived by string-manipulation
      let StructureName = " Type"  ++TypeName
      let Structure := "structure" ++ StructureName ++ "("
      for i = 1 to upb ( AttribNames ) do
            Structure := Structure ++ AttribTypes(i) ++ "."  ++
                         TypeName  ++ AttribNames(i) ++ ","
      Structure := Structure ++ ")"

      let MakeString = "proc( -> pntr )
                        begin
                  " ++ Structure++ "
                  let getint = proc(->int)
                  .....    ! standard proc to get an integer from the user
                  .....    ! ditto for string, bool, real, etc
                  " ++ StructureName ++ "("
      for i = 1 to upb ( AttribNames ) do
            MakeString := MakeString ++ "get" ++ AttribTypes (i)++ "(),"
      MakeString := MakeString ++ ")
            end
            "
      structure Makepack( proc(-> pntr) makeproc)
      let MakeEmpty = MakePack( proc(-> pntr); nullproc )
      let MakeProc = compile(MakeString, MakeEmpty )( makeproc)

      let DisplayString = "proc( pntr X ) ..... ! exactly analagous to Make
      let DisplayProc = compile( ...

      let NewType = type( TypeName, Essential, Exclusionary, MakeProc, DisplayProc,
                          nullvector, Supertypes, nullvector )
      s.enter( TypeName, TypeTable, NewType)
      for i = 1 to upb (Supertypes) do insert( NewType, Supertypes(i)(subtypes))
  end
```

Fig. 2 A Procedure for Automatically Generating Type Structures

The *make* procedure for type *PERSON* looks like:

```
begin
    structure TypePERSON( string PERSONcname; string PERSONsex int PERSONage)
    let getint = proc( -> int ) .....
    .....
    TypePERSON( getstring(), getstring(), getint() )
end
```

A simple extension of the above procedure permits attributes of one type to be instances of another. For instance, the *PERSON* type could have another field to contain an address, by adding the line

address: *ADDRESS*

to the list of attributes, where *ADDRESS* refers to another, previously defined, type. In this case, the field in the type structure for *address* would be of type **pntr**, and the *display* procedure would recursively call the equivalent procedures for each of the fields of the *ADDRESS* type. The *make* procedure might also be implemented to recursively call the *make* procedures of the fields of the *ADDRESS* type, but we might prefer, as PSRML does [COOP87b], to implement it to present a menu of all instances of type *ADDRESS* to choose from.

Using a procedure like this, we have been able to create a type-secure world in which objects are efficiently represented and yet no restriction is made on the kind of objects we can describe. PSRML [COOP87b] contains an implementation of types very similar to that described here. In fact, the types are slightly richer in that they also include automatically produced procedure fields to create null instances of objects and to change the values of attribute fields. It is designed to embody a Requirements Modelling Language, similar to that described by [GREE84], which includes descriptions of entity types and activities on those entities. The whole system allows models to be created; new entity types and activities to be added to a model; and models to be instantiated. The instantiation of a model includes the explicit creation of instances of the entity types, the specialisation and generalisation of entity instances and the activation of activities (which create or modify type instances as a by-product). It proved to be relatively easy to implement, given the facilities described in this paper - the whole system, including a good quality interface, taking about two weeks to program.

Deferred Binding and Persistence.

We have shown how the callable compiler allows us to write polymorphic procedures which use efficient tailored storage structures. The only apparent source of delay is at type creation time as this now includes some compilation. However, slight delays at type creation can probably be tolerated, both on psychological grounds in that this might seem to the user to be an expensive operation, and on the grounds that type creation is rare compared with operations on instances and so the delay is more than offset by having increased efficient access to instances.

Moreover, the number of delays can be reduced by storing types in some canonical form, based on the types of the fields of the object, and re-using the same automatically generated procedures for different types. These would be memo-ised by using the persistent store to hold an associative table between the canonical description and procedures. At type-creation time, the table would be searched to see if an equivalent type had already been created and, if so, the appropriate procedures would be retrieved instead of being freshly generated and compiled. In our evolutionary type system, for example, the new type:

type *ADDRESS* isa *ENTITY*
with attributes *house*: int
street string
city string;

would use the same *make* and *display* procedures as *PERSON* since both have two string fields and one integer field.

We could also exploit deferred binding within a persistent store to provide access to a range of good interface tools. As described above, information is sought from the user by primitives like *getint*, which invites the user to type in an integer. Clearly there could be a whole range of such primitives which provide input via menus, scroll-bars, radio-buttons, etc. The notion of getting the referend of a **pntr** field via an automatically generated menu, which has already been mentioned, is one such example. As the system stands, the input primitive for each type is hardwired in. Using deferred binding it would be possible to tailor the input style (and, of course, the output style) appropriately for different applications.

In this paper, we have tended to keep the use of deferred binding to a minimum in order to present the ideas as simply as possible. For instance, we have provided polymorphic structures *instance*, with a pointer to specific information. In fact, we could tailor our structures and procedures to embed the specific information in the *instance* structure and thus save an extra access. In general, there are many places in a representation where there is a choice between placing a pointer from a generalised structure to a specific one or making the general structure into a set of specific ones. The decision as to which of these to do is particular to each point of the design.

Discussion.

PS-algol provides two mechanisms for controlling the time at which type checking and binding are performed. The type **pntr** constitutes an extensible union type which enables programs to be written which know as little about the type of an object as they need to know - for instance the tree insertion procedure knows only that the objects it handles are to be put into a tree. The callable compiler allows programs to be written which bind in new types of data at run-time.

Using these mechanisms, we have described a method of providing a type system, within which objects can change type easily by movement about a type lattice. The representation of the objects is efficient as each is represented by a PS-algol structure which is tailored to the specific components of the object. The detail of this tailoring is abstracted out by providing a few core modules to create types and instances of types and to move objects about the type lattice. To implement a given data model, such as PSRML, it remains to provide parsing facilities for the model description language and a user interface. In PS-algol, the former has also been abstracted in the form of a set of generators of lexical analysers, parsers and syntax-directed editors described in [BLOT87]. The production of the interface is also facilitated in a language with good graphics commands [MORR86].

The approach taken here has a number of beneficial effects. It is possible, using the **pntr** type, to partition the operations on a given class of objects into different levels of abstraction. Thus, in our example, the primitives to change the type of an object take no account of the object's type - all the complexity of making such type changes to any object at all has been factored out. Any operations specific to a particular type can be provided at a separate level. This method has therefore the effect of supporting a modular approach to software development and in increasing the clarity of the resulting code.

The facilities described in [BLOT87] show another use made of this technique. Programs have been produced which produce lexical analysers and parsers. The former takes in a lexical specification and produces a procedure which turns a string of characters into a string of lexeme/token pairs. The latter takes in a syntactic specification and a lexical analyser and produces a procedure which turns a string of characters into a parsed syntax tree. Thus programs which require to perform some parsing of character strings can get a parser by providing the grammar. Parsers thus generated are stored in a lookup table and so a given program may switch from one language to another by looking up a different parser - the choice of which language to use in a given program has thus been deferred until the program has run.

An example of using the generators illustrates a point at which the parser generator has deferred the choice of what type of objects it is handling. The syntax tree generated by the parsers contained a **pntr** field called *value*, whose referend is determined by the calling program. In general, a parser will be used in conjunction with a compilation system and, in this case, the *value* field points to a node in the Abstract Syntax Tree of the expression being parsed. However, a parser has been created for the p-string grammar described in [GONN87]. This grammar describes entries in a bibliography and a program has been written which parses such an entry and produces a tree diagram for it. In this case, the *value* field has been used to point to a container for the string parsed at this node - the author node contains just the author name, the top-level node contains the string for the whole entry. The field has been used to give fast access to a derived expression. The point being that this was not envisaged when the parser generator program was written, but the **pntr** type permitted the simple extension of its use, without rewriting the parser generator.

A separate area for investigation is that of version control. Recent work on version control for CAD objects [ECKL87, KATZ87] and office systems [AHLS84] provide models which we are starting to implement. Deferring the binding of the program to its data seem to facilitate our work here. The control of software libraries, both in methods of version control and module retrieval, is another related area in which we seek to put these methods to work for us by implementing a system similar to that proposed by [HULL87]. We hope to report on this work and its implications for software development in the near future.

Another main beneficial effect is that we can separate data representation from the algorithm, which uses it by using the callable compiler. The algorithm is provided as one set of strings and the representation can be provided either as a formal description (as in the CML-like description shown above) or it can be derived from a pointer to the object as described in [COOP87a]. This paper describes a

program which is designed to provide the bottom level of a multi-interface relational system. In the system, relations are represented as Abstract Data Types and the operations of the ADTs are tailored to the specific relation in a similar way to that used for the *make* and *display* procedures in our type evolution example.

A further benefit deriving from this is that algorithms provided in such a way can be built to cater for classes of data as yet unspecified and yet use an efficient representation. We feel that this goes a considerable way to countering the claims of [DONA87], particularly when the run-time compilation costs are ameliorated as described above.

To sum up, we have introduced an approach to the provision of data models which relies on the following facilities:

- an extensible union type to allow some parts of the program to ignore the type of some parts of the data;
- a mechanism for deferring the binding of program to data and the type-checking of the data as long as possible;
- a callable compiler which can produce procedures, which are only compiled at run-time but whose types can be statically determined.

These facilities permit the development of programs in a modular way, which are efficient, type-secure and yet extensible to data whose type is not yet known.

We believe that the combination of deferred binding with run-time compilation in a persistent language provides a potentially efficient means of implementing by well-structured software many of the specific models being discussed by the database community. We suggest the database research would benefit from the wider use of these rapid implementation techniques.

Acknowledgements.

The work reported here was funded partly by ICL URC grant and partly by Alvey/SERC grant GR/D43259. We would also like to thank our colleagues in the PISA group at Glasgow and St. Andrews and Dr. David Harper for many useful comments.

Bibliography.

[AHLS84] Ahlsén M, Björnstedt A, Britts S, Hultén C and Söderlund L, "Making Type Changes Transparent", *SYSLAB Report*, 22, University of Stockholm, February 1984.

[ALBA85] Albano A, Cardelli L and Orsini R, "Galileo: A Strongly Typed, Interactive, Conceptual Language", *ACM TODS*, 10, 2, June 1985.

[ATKI85] Atkinson MP and Morrison R, "Procedures as Persistent Data Objects", *ACM TOPLAS*, 7, 4, 539-559, October 1985.

[ATKI86a] Atkinson MP and Morrison R, "Integrated Persistent Programming Systems", in *proc 19th Annual Hawaii International Conference on Systems Sciences, Jan 7-10, 1986 (ed BD Shriver), vol IIA, Software*, 842-854.

[ATKI86b] Atkinson MP, Morrison R and Pratten GD, "Designing A Persistent Information Space Architecture", in *proc Information Processing 1986*, North Holland Press, 115 - 119, September 1986.

[ATKI88] Atkinson MP, Buneman OP and Morrison R, "Binding and Type Checking in Database Programming Languages", to appear in *Computer Journal*, 1988.

[BLOT87] Blott SM and Campin J, "Lgen, Pgen and Sgen: Language Development Tools", *Persistent Programming Research Report*, 49, Universities of Glasgow and St. Andrews, 1987.

[COOP87a] Cooper RL, Atkinson MP and Abderrahmane D, "Constructing Database Systems in a Persistent Environment", in *proc 13th International Conference on Very Large Databases, Brighton, England*, 117-126, September 1987.

[COOP87b] Cooper, RL and Atkinson, MP "Requirements Modelling in a Persistent Object Store", in *proc 2nd International Workshop on Persistent Object Systems, Appin*, August 1987.

[DEAR87] Dearle, A and Brown, AL, "Safe Browsing in a Strongly Typed Persistent Environment", *Persistent Programming Research Report*, 33, Universities of Glasgow and St. Andrews, 1987.

[DONA87] Donahue J, "What is a Database?", in *proc 2nd International Workshop on Persistent Object Systems, Appin*, August 1987.

[ECKL87] Ecklund D, Ecklund E, Eifrig R and Tonge F, "DVSS: A Distributed Version Storage Server for CAD", in *proc 13th International Conference on Very Large Databases, Brighton, England*, 443-454, September 1987.

[GONN87] Gonnet, G and Tompa, FW, "Mind Your Grammar: a New Approach to Modelling Text", in *proc 13th International Conference on Very Large Databases, Brighton, England*, 339-346, September 1987.

[GREE84] Greenspan, SJ - "Requirements Modeling: A Knowledge Representation Approach to Software Requirements Definition", *CSRG Technical Report*, CSRG-155, University of Toronto, March 1984.

[HARP86] Harper R, MacQueen D and Milner R, "Standard ML", *LFCS Report*, ECS-LFCS-86-2, University of Edinburgh, 1986.

[HULL87] Hull R and Jacobs D, "Towards a Formalism for Module Interconnection and System Evolution", in *proc International Workshop on Database Programming Languages, Roscoff, France*, 313-336, September 1987.

[KATZ87] Katz RH and Chang E, "Managing Change in a Computer-aided Design Database", in *proc 13th International Conference on Very Large Databases, Brighton, England*, 339-346, September 1987.

[LISK81] Liskov B, "The CLU Reference Manual", *Lecture Notes in Computer Science*, 114, Springer-Verlag, Berlin, 1981.

[MILN79] Milner R, "A Theory of Type Polymorphism in Programming", *JACM*, 26, 4, 792-818.

[MORR86] Morrison R, Dearle A, Brown AL and Atkinson MP, "An Integrated Graphics Programming Environment", *Computer Graphics Forum*, 5, 2, 147-157, June 1986.

[MORR87] Morrison R, Atkinson MP and Dearle A, "Flexible Incremental Bindings in a Persistent Object Store", *Persistent Programming Research Report*, 38, Universities of Glasgow and St Andrews, 1987.

[PSAL87] "The Ps-algol Reference Manual - Fourth Edition", *Persistent Programming Research Report*, 12, Universities of Glasgow and St Andrews, 1987.

[ZDON87] Zdonik S, "Can Objects Change Type? Can Type Objects Change?", in *proc International Workshop on Database Programming Languages, Roscoff, France*, 193-200, September 1987.

Bibliography

Copies of documents in this list may be obtained by writing to:

The Secretary,
Persistent Programming Research Group,
Department of Computing Science,
University of Glasgow,
Glasgow G12 8QQ
Scotland.

or

The Secretary,
Persistent Programming Research Group,
Department of Computational Science,
University of St. Andrews,
North Haugh,
St. Andrews KY16 9SS
Scotland.

Books

Davie, A.J.T. & Morrison, R.
"Recursive Descent Compiling", Ellis-Horwood Press (1981).

Atkinson, M.P. (ed.)
"Databases", *Pergammon Infotech State of the Art Report*, Series 9, No.8, January 1982. (535 pages).

Cole, A.J. & Morrison, R.
"An introduction to programming with S-algol", Cambridge University Press, Cambridge, England, 1982.

Stocker, P.M., Atkinson, M.P. & Gray, P.M.D. (eds.)
"Databases - Role and Structure", Cambridge University Press, Cambridge, England, 1984.

Published Papers

Morrison, R.
"A method of implementing procedure entry and exit in block structured high level languages", *Software, Practice and Experience*, 7, 5, 535-537, July 1977.

Morrison, R. & Podolski, Z.
"The Graffiti graphics system", *Proceedings of the DECUS conference*, Bath, 5-10, April 1978.

Atkinson, M.P.
"A note on the application of differential files to computer aided design", *ACM SIGDA newsletter*, Summer 1978.

Atkinson, M.P.
"Programming Languages and Databases", *Proceedings of the 4th International Conference on Very Large Data Bases*, Berlin, (Ed. S.P. Yao), IEEE, Sept. 78, 408-419. (A revised version of this is available from the University of Edinburgh Department of Computer Science (EUCS) as CSR-26-78).

Atkinson, M.P.
"Progress in documentation: Database management systems in library automation and information retrieval", *Journal of Documentation*, 35, 1, 49-51, March 1979. Available as EUCS departmental report CSR-43-79.

Gunn, H.I.E. & Morrison, R.
"On the implementation of constants", *Information Processing Letters*, 9, 1, 1-4, July 1979.

Atkinson, M.P.
"Data management for interactive graphics", *Proceedings of the Infotech State of the Art Conference*, October 1979. Available as EUCS departmental report CSR-51-80.

Atkinson, M.P. (ed.)
"Data design", *Infotech State of the Art Report*, Series 7, No.4, May 1980.

Morrison, R.
"Low cost computer graphics for micro computers", *Software Practice and Experience*, 12, 767-776, 1981.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", *ACM SIGPLAN Notices*, 17, 7, 24-31, July 1981. Also as EUCS Departmental Report CSR-94-81.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Nepal - the New Edinburgh Persistent Algorithmic Language", in *Database, Pergammon Infotech State of the Art Report*, Series 9, No.8, 299-318, January 1982 - also as EUCS Departmental Report CSR-90-81.

Morrison, R.
"S-algol: a simple algol", *Computer Bulletin* II/31, March 1982.

Morrison, R.
"The string as a simple data type", *ACM Sigplan Notices*, 17, 3, 46-52, 1982.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Progress with Persistent Programming", presented at CREST course UEA, September 1982, revised in "Databases - Role and Structure".

Morrison, R.
"Towards simpler programming languages: S-algol", *IUCC Bulletin*, 4, 3, 130-133, October 1982.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Problems with persistent programming languages", presented at the *Workshop on programming languages and database systems*, University of Pennsylvania. October 1982. Circulated (revised) in the Workshop proceedings 1983.

Atkinson, M.P.
"Data management", in *Encyclopedia of Computer Science and Engineering 2nd Edition*, Ralston & Meek (editors) January 1983. van Nostrand Reinhold.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", *Software Practice and Experience*, 13, 3, 259-272, March 1983. Also as EUCS Departmental Report CSR-109-82.

Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", *Software Practice and Experience*, 13, 3, 273-285, March 1983. Also as EUCS Departmental Report CSR-110-82.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"Current progress with persistent programming", presented at the *DEC workshop on Programming Languages and Databases*, Boston, April 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", *The Computer Journal*, 26, 4, 360-365, 1983.

Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"PS-algol a language for persistent programming", *10th Australian Computer Conference, Melbourne*, 70-79, September 1983.

Morrison, R., Weatherill, M., Podolski, Z. & Bailey, P.J.
"High level language support for 3-dimension graphics", *Eurographics Conference Zagreb*, North Holland, 7-17, September 1983. (ed. P.J.W. ten Hagen).

Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", *Software Practice and Exerience*, 14, 1, 49-71, January 1984.

Kulkarni, K.G. & Atkinson, M.P.
"Experimenting with the Functional Data Model", in *Databases - Role and Structure*, Cambridge University Press, Cambridge, England, 1984.

Atkinson, M.P. & Morrison, R.
"Persistent First Class Procedures are Enough", *Foundations of Software Technology and Theoretical Computer Science* (ed. M. Joseph & R. Shyamasundar) Lecture Notes in Computer Science 181, Springer Verlag, Berlin, 1984.

Atkinson, M.P., Bocca, J.B., Elsey, T.J., Fiddian, N.J., Flower, M., Gray, P.M.D. Gray, W.A., Hepp, P.E., Johnson, R.G., Milne, W., Norrie, M.C., Omololu, A.O., Oxborrow, E.A., Shave, M.J.R., Smith, A.M., Stocker, P.M. & Walker, J.
"The Proteus distributed database system", *Proceedings of the third British National Conference on Databases*, (ed. J. Longstaff), BCS Workshop Series, Cambridge University Press, Cambridge, England, July 1984.

Atkinson, M.P. & Morrison, R.
"Procedures as persistent data objects", *ACM TOPLAS*, 7, 4, 539-559, Oct. 1985.

Morrison, R.,Bailey, P.J., Dearle, A., Brown, P. & Atkinson, M.P.
"The persistent store as an enabling technology for integrated support environments", *8th International Conference on Software Engineering*, Imperial College, London, 166-172, August 1985.

Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988

Davie, A.J.T.
"Conditional declarations and pattern matching", *Proceedings of Data Types and Persistence Workshop*, Appin, 278-283, August 1985.

Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988

Buneman, O.P.
"Data types for data base programming", published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988.

Cockshott, W.P.
"Addressing mechanisms and persistent programming", published in *Data Types and Persistence*, Atkinson, Buneman & Morrison (eds), Springer-Verlag, 1988.

Norrie, M.C.
"PS-algol: A user perspective", *Proceedings of Data Types and Persistence Workshop*, Appin, 399-410, August 1985.

Owoso, G.O.
"On the need for a Flexible Type System in Persistent Programming Languages", *Proceedings of Data Types and Persistence Workshop*, Appin, August 1985, 423-438.

Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. & Dearle, A.
"A persistent graphics facility for the ICL PERQ", *Software Practice and Experience*, 14, 3, 1986.

Atkinson, M.P. and Morrison R.
"Integrated Persistent Programming Systems", *Proceedings of the 19th Annual Hawaii International Conference on System Sciences*, January 7-10, 1986 (ed. B. D. Shriver), vol IIA, Software, 842-854, Western Periodicals Co., 1300 Rayman St., North Hollywood, Calif. 91605, USA.

Atkinson, M.P., Morrison, R. and Pratten, G.D.
"A Persistent Information Space Architecture", *Proceedings of the 9th Australian Computing Science Conference*, January, 1986.

Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", *The Computer Journal*, 29, 1, 38-45, 1986.

Buneman, O.P. & Atkinson, M.P.
"Inheritance and Persistence in Database Programming Languages", *Proceedings ACM SIGMOD Conference 1986*, Washington, USA, May 1986.

Morrison R., Dearle, A., Brown, A. & Atkinson M.P.; "An integrated graphics programming environment", *Computer Graphics Forum*, 5, 2, 147-157, June 1986.

Atkinson, M.G., Morrison, R. & Pratten G.D.
"Designing a Persistent Information Space Architecture", *Proceedings of Information Processing 1986*, Dublin, September 1986, (ed. H.J. Kugler), 115-119, North Holland Press.

Brown, A.L. & Dearle, A.
"Implementation Issuses in Persistent Graphics", *University Computing*, 8, 2, Summer 1986.

Kulkarni, K.G. & Atkinson, M. P.
"Implementing an Extended Functional Data Model Using PS-algol", *Software - Practise and Experience*, 17, 3, 171-185, March 1987.

Cooper, R.L. & Atkinson, M.P.
"The Advantages of a Unified Treatment of Data", *Software Tool 87: Improving Tools, Advance Computing Series*, 8, 89-96, Online Publications, June 1987.

Atkinson, M.P, Morrison, R. & Dearle, A.
"A strongly typed persistent object store", *Presnted at the 1986 International Workshop on Object-Oriented Database Systems*, Pacific Grove, California, September 1986.

Atkinson, M.P., Morrison, R. & Pratten G.D.
"PISA : A persistent information space architecture", *ICL Technical Journal*, 5, 3,477-491, May 1987.

Atkinson, M.P. & Morrison, R.
"Polymorphic Names, Types, Constancy and Magic in a Type Secure Persistent Object Store", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Cooper, R. & Atkinson, M.P.
"Requirements Modelling in a Persistent Object Store", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Wai, F.
"Distribution and Persistence", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Philbrow, P.
"Associative Storage and Retrieval: Some Language Design Issues", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Guy, M.R.
"Persistent Store - Successor to Virtual Store", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Dearle, A.
"Constructing Compilers in a Persistent Environment", *Presented at the 2nd Internaional Workshop on Persistent Object Stores*, Appin, August 1987.

Carrick, R. & Munro, D.
"Execution Strategies in Persistent Systems", *Presented at the 2nd International Workshop on Persistent Object Stores*, Appin, August 1987.

Brown, A.L.
"A Distributed Stable Store", *Presented at the 2nd International Workshop on Persistent object Stores*, Appin, August 1987.

Cooper, R.L., Atkinson, M.P., Dearle, A. & Abderrahmane, D.
"Constructing Database Systems in a Persistent Environment", *Proceedings of the Thirteenth Internaional Conference on Very Large Databases*, Brighton, 117-126, September 1987.

Atkinson, M.P. & Morrison, M.
"Polymorphic Names and Iterations", *Presented at the Workshop on Database Programming Languages, Roscoff*, September 1987.

Brown, A.L., Dearle, A. & Morrison, R.
"An Architecture for a Strongly Typed Persistent Object Store", *Proceedings of Workshop on Object Oriented Programming Systems, Languages and Applications at the OOPSLA Conference in Orlando*, Florida, 1987.

Brown, A.L.
"The Implementation of a Distributed Persistent Object Store", *Presented at the Workshop on Object-Oriented Databases*, Semantic Aspects, at the OOPSLA Conference in Orlando, Florida, 1987.

Morrison, R., Brown, A.L., Carrick, R., Connor, R.C., Dearle, A. & Atkinson, M.P.
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object - Oriented Environment", *IEE & BCS Journal on Software Engineering*, Dec 1987.

Atkinson, M.P., Buneman, O.P. & Morrison, R.
"Binding and Type-Checking in Database Programming Languages", *Computer Journal*, 31, 2, 99-109, 1988.

Dearle, A. & Brown, A.L.
"Safe Browsing in a Strongly Typed Persistent Environment", *Computer Journal*, 31, 2, 1988.

Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P.
"Flexible Incremental Bindings in a Persistent Object Store", *ACM Sigplan Notices*, 1988.

Philbrow P., Armour I., Atkinson, M.P. and Livingstone J.
"PS-algol's Device-Independent Output Statement", *Computer Journal*, 31, 1988.

Internal Reports

Morrison, R.
"S-Algol language reference manual", University of St Andrews CS-79-1, 1979.

Bailey, P.J., Maritz, P. & Morrison, R.
"The S-algol abstract machine", University of St Andrews CS-80-2, 1980.

Atkinson, M.P., Hepp, P.E., Ivanov, H., McDuff, A., Proctor, R. & Wilson, A.G.
"EDQUSE reference manual", Department of Computer Science, University of Edinburgh, September 1981.

Hepp, P.E. and Norrie, M.C.
"RAQUEL: User Manual", Department of Computer Science Report CSR-188-85, University of Edinburgh.

Norrie, M.C.
"The Edinburgh Node of the Proteus Distributed Database System", Department of Computer Science Report CSR-191-85, University of Edinburgh.

Theses

The following theses, for the degree of Ph. D. unless otherwise stated, have been produced by members of the group and are available from the address already given,

W.P. Cockshott
Orthogonal Persistence, University of Edinburgh, February 1983.

K.G. Kulkarni
Evaluation of Functional Data Models for Database Design and Use, University of Edinburgh, 1983.

P.E. Hepp
A DBS Architecture Supporting Coexisting Query Languages and Data Models, University of Edinburgh, 1983.

G.D.M. Ross
Virtual Files: A Framework for Experimental Design, University of Edinburgh, 1983.

G.O. Owoso
Data Description and Manipulation in Persistent Programming Languages, University of Edinburgh, 1984.

J. Livingstone
Graphical Manipulation in Programming Languages: Some Experiments, M.Sc., University of Glasgow, 1987.

Persistent Programming Research Reports

This series was started in May 1983. The following list gives those which have been produced at 6th May 1988. Copies of documents in this list may be obtained by writing to the addresses already given.

PPRR-1-83	The Persistent Object Management System - Atkinson,M.P., Bailey, P., Chisholm, K.J., Cockshott, W.P. and Morrison, R.	£1.00
PPRR-2-83	PS-algol Papers: a collection of related papers on PS-algol - Atkinson, M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-5-83	Experimenting with the Functional Data Model - Atkinson, M.P. and Kulkarni, K.G.	£1.00
PPRR-6-83	A DBS Architecture supporting coexisting user interfaces: Description and Examples - Hepp, P.E.	£1.00
PPRR-7-83	EFDM - User Manual - K.G.Kulkarni	£1.00
PPRR-8-84	Progress with Persistent Programming - Atkinson,M.P., Bailey, P., Cockshott, W.P., Chisholm, K.J. and Morrison, R.	£2.00
PPRR-9-84	Procedures as Persistent Data Objects - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-10-84	A Persistent Graphics Facility for the ICL PERQ - Morrison, R., Brown, A.L., Bailey, P.J., Davie, A.J.T. and Dearle, A.	£1.00
PPRR-11-85	PS-algol Abstract Machine Manual	£1.00
PPRR-12-88	PS-algol Reference Manual - fourth edition	£2.00
PPRR-13-85	CPOMS - A Revised Version of The Persistent Object Management System in C - Brown, A.L. and Cockshott, W.P.	£2.00
PPRR-14-86	An Integrated Graphics Programming Environment - 2nd edition - Morrison, R., Brown, A.L., Dearle, A. and Atkinson, M.P.	£1.00
PPRR-15-85	The Persistent Store as an Enabling Technology for an Integrated Project Support Environment - Morrison, R., Dearle, A, Bailey, P.J., Brown, A.L. and Atkinson, M.P.	£1.00
PPRR-16-85	Proceedings of the Persistence and Data Types Workshop, Appin, August 1985 - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£15.00
PPRR-17-85	Database Programming Language Design - Atkinson, M.P. and Buneman, O.P.	£3.00
PPRR-18-85	The Persistent Store Machine - Cockshott, W.P.	£2.00
PPRR-19-85	Integrated Persistent Programming Systems - Atkinson, M.P. and Morrison, R.	£1.00
PPRR-20-85	Building a Microcomputer with Associative Virtual Memory - Cockshott, W.P.	£1.00
PPRR-21-85	A Persistent Information Space Architecture - Atkinson, M.P., Morrison, R. and Pratten, G.D.	£1.00
PPRR-22-86	Inheritance and Persistence in Database Programming Languages - Buneman, O.P. and Atkinson, M.P.	£1.00
PPRR-23-86	Implementation Issues in Persistent Graphics - Brown, A.L. and Dearle, A.	£1.00
PPRR-24-86	Using a Persistent Environment to Maintain a Bibliographic Database - Cooper, R.L., Atkinson, M.P. & Blott, S.M.	£1.00
PPRR-25-87	Applications Programming in PS-algol - Cooper, R.L.	£1.00
PPRR-26-86	Exception Handling in a Persistent Programming Language - Philbrow, P & Atkinson M.P.	£1.00
PPRR-27-87	A Context Sensitive Addressing Model - Hurst, A.J.	£1.00
PPRR-28-86b	A Domain Theoretic Approach to Higher-Order Relations - Buneman, O.P. & Ochari, A.	£1.00
PPRR-29-86	A Persistent Store Garbage Collector with Statistical Facilities - Campin, J. & Atkinson, M.P	£1.00
PPRR-30-86	Data Types for Data Base Programming - Buneman, O.P.	£1.00
PPRR-31-87	An Introduction to PS-algol Programming (third edition) - Carrick, R., Cole, A.J. & Morrison, R.	£1.00
PPRR-32-87	Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment - Morrison, R, Brown, A, Connor, R and Dearle, A	£1.00
PPRR-33-87	Safe Browsing in a Strongly Typed Persistent Environment - Dearle, A and Brown, A.L.	£1.00
PPRR-34-87	Constructing Database Systems in a Persistent Environment - Cooper, R.L., Atkinson, M.P., Dearle, A. and Abderrahmane, D.	£1.00
PPRR-35-87	A Persistent Architecture Intermediate Language - Dearle, A.	£1.00
PPRR-36-87	Persistent Information Architectures - Atkinson, M.P., Morrison R. & Pratten, G.D.	£1.00

PPRR-37-87	PS-algol Machine Monitoring - Loboz, Z.	£1.00
PPRR-38-87	Flexible Incremental Bindings in a Persistent Object Store - Morrison, R., Atkinson, M.P. and Dearle, A.	£1.00
PPRR-39-87	Polymorphic Persistent Processes - Morrison, R., Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J.and Livesey, M.J.	£1.00
PPRR-40-87	Andrew, Unix and Educational Computing - Hansen, W. J.	£1.00
PPRR-41-87	Factors that Affect Reading and Writing with Personal Computers and Workstations - Hansen, W. J. and Haas, C.	£1.00
PPRR-42-87	A Practical Algebra for Substring Expressions - Hansen, W. J.	£1.00
PPRR-43-87	The NESS Reference Manual - Hansen, W. J.	£1.00
PPRR-44-87	Persistent Object Systems: their design, implementation and use. (proceedings of the Appin workshop August 1987) - ed. Atkinson, M.P., Buneman, O.P. and Morrison, R.	£20.00
PPRR-45-87	Delayed Binding and Type Checking in Database Programming Languages - Atkinson, M.P., Buneman, O.P. & Morrison, R.	£1.00
PPRR-46-87	Transactions and Concurrency - Krablin, G.L.	£1.00
PPRR-47-87	Persistent Information Space Architecture - PISA Club Rules - Atkinson, M.P., Lucking, J.R., Morrison, R. and Pratten, G.D.	£1.00
PPRR-48-87	An Event-Driven Software Architecture - Cutts, Q. and Kirby, G.	£1.00
PPRR-49-87	An Implementation of Multiple Inheritance in a Persistent Environment - Benson, P.J., D'Souza, E.B., Rennie, I.S., Waddell, S.J.	£1.00
PPRR-50-87	A Distributed Stable Store - Brown, A.L.	£1.00
PPRR-51-87	Constructing Compilers in a Persistent Environment - Dearle, A.	£1.00
PPRR-52-87	Lgen, Pgen and Sgen - Language Development Tools for a Persistent Programming Environment - Blott, S.M. and Campin, J.	£1.00
PPRR-53-87	Polymorphic Names and Iterations - Atkinson, M.P. and Morrison, R	£1.00
PPRR-54-87	A Requirements Modelling Tool Built in PS-algol - Cooper, R.L. and Atkinson, M.P.	£1.00