This paper should be referenced as:

Atkinson, M.P., Morrison, R. & Pratten, G.D. “Designing a Persistent Information Space Architecture”. In Proc. 10th IFIP World Congress, Dublin (1986) pp 115-120.

DESIGNING A PERSISTENT INFORMATION SPACE ARCHITECTURE

Malcolm P. Atkinson, Ronald Morrison and Graham D. Pratten

University of Glasgow, Glasgow, Scotland G12 8QQ.
University of St Andrews, St Andrews, Scotland KY16 9SX.
and
International Computers Ltd., Kidsgrove, England. ST7 1TL.

Abstract

We contend that the complexity of modern computer systems is caused by a dependency on a plethora of mechanisms that by their lack of coherence increase the cost of developing applications. By integrating these mechanisms we can reduce the complexity of the system and thereby achieve savings throughout product life cycles. From our experience in operating systems, programming languages and databases we identify five major areas of system design for modern systems: controlling complexity, orthogonal persistence, controlled system evolution, protection of data and concurrent computation. We are currently engaged in building a system which will achieve the simplicity above and here we describe the design issues involved in the construction of a persistent information space architecture (PISA) capable of integrating all these activities.

1. INTRODUCTION

When we examine existing computer systems we find many dichotomies and discontinuities in their design and in the design of their user interfaces. Some of these were deliberately introduced during the 1960's and 1970's to satisfy performance constraints. Others reflect the order in which the basic design concepts of computer systems have emerged over a period of twenty to thirty years. We contend that the present dependence on a plethora of inconsistent mechanisms such as command languages, editors, file systems, compilers and interpreters, linkage editors and binders, debuggers, DBMS sublanguages, graphics libraries increases the cost of understanding and maintaining software, and training programmers, for even the simplest of activities. It is important to remove this incoherence now since it is placing a considerable overhead on the users and developers of computer systems. The removal of this incoherence is now possible because of the dramatic shift in the cost of hardware relative to software. In this paper we report on our current research on programming languages and environments and discuss the problems in designing a persistent information space architecture (PISA) capable of integrating the above programming activities.

Inherent in our approach is that any program must be integrated with its environment. We have identified the following problems faced by users and developers of current systems which when negated become requirements of modern systems in order to achieve simplicity and integration. We do not regard this list as exhaustive. Indeed we would expect that having achieved these requirements new ones will emerge in the pursuit of better programming systems. They are

a. Controlling complexity: The complexity of the system must be kept under control, so that developers and users can concentrate on the application rather than the complexity of the system. This depends on establishing consistent rules which apply throughout the design and being parsimonious in the introduction of new concepts into these designs.

b. Orthogonal persistence: The discontinuity between the method of using data that is short term and manipulated by program and long term data that is manipulated by the file system or DBMS causes unnecessary complexity. We have defined the persistence of data to be the length of time for which the data exists and is

useable[2]. We aspire to systems where the use of data is independent of its persistence.

c. Controlled system evolution: The uses of data (including program) are neither limited nor predictable. It is necessary to support the construction of unanticipated software systems or databases which make use of pre-existing data (or program) even when the data and program were defined independently of one another. For large scale, widely used or continuously used systems any alteration to part of the system should not require total rebuilding. We require a mechanism which will allow the programmer to control the units of reconstruction.

d. Protection of data: Large bodies of data are inherently valuable. It is necessary to protect them from misuse and from hardware and software failure. This implies a sophisticated type and protection system and recovery mechanisms to limit the losses due to component failure.

e. Concurrent computation: A large body of data requires a community effort for its construction and maintenance. Any useful body of data is likely to be of concurrent interest to many users, probably in dispersed geographic locations. Different models of concurrency and transactions may have to be accommodated by the underlying mechanism.

.We have designed and implemented the language PS-algol [3] as a testbed for experiments on the above requirements. We report here on some results and propose further experiments in the search for better programming systems.

2. CONTROLLING COMPLEXITY

In many ways the main task of computer science is to invent and implement appropriate languages to suit applications at any level, both hardware and software, in order that programming may become easier and cheaper. We may therefore use the experience of programming language designers in controlling complexity with the aim of building a total system that will integrate all the naming, binding and other mechanisms of programming languages, database systems and operating systems.

Central to our method of language design is that the language itself should be simple. Merely adding features to languages without integrating them into some overall design simply increases the complexity of the language and can often lead to it being beyond the intellectual capacity of the programmer and sometimes even the implementor. This complexity is often due, in part at least, to being too restrictive but can be avoided by the language designer formulating the fundamental concepts behind the language and generalising these wherever possible. Simplicity is achieved by having no exceptions to these general rules.

Our languages are designed using three principles due to Landin[16] and Strachey[22] and further developed by Tennent[23]. They are

1. The principle of correspondence: the use of names should be consistent within a language. In particular there should be a one to one correspondence between the method of introducing names in declarations and in parameter lists.

2. The principle of abstraction: all the major syntactic categories should have abstractions defined over them. For example, functions are abstractions over expressions.

3. The principle of data type completeness: all data types should be first class without arbitrary restriction on their use.

The astute reader will notice that the principle of abstraction can be seen as a special case of the principle of data type completeness if we insist that every syntactic category has a type. However for the present we will leave them as separate.

We have described our design methodology fully elsewhere[19] and will not labour it here. However in applying these rules no exceptions are allowed since having exceptions makes the language more complex in terms of defining rules and usually less powerful since there are restrictions. Designing languages using the above rules yields languages that are free from inconsistencies and idiosyncrasies in this manner. Thus the rules can be used as a metric for comparing languages. It is part of our research into programming language design to find further rules, if there are any, to aid the design process. We have applied this general philosophy to the design of programming languages and programming environments in the PS-algol system.

3. ORTHOGONAL PERSISTENCE

We seek to eliminate the differences between programming language models of data and the environment models of data. This can be done by separating the issue of data modelling for a particular problem from the issue of identifying and preserving the data. This second property we call persistence. A spectrum of persistence exists and is categorised by

1. transient results in expression evaluation.

2. local variables in procedure activations.

3. own variables, global variables and heap items whose extent is different from their scope.

4. data that exists between executions of a program.

5. data that exists between various versions of a program.

6. data that outlives the program.

The first three persistence categories are usually supported by programming languages and the second three categories by a DBMS, whereas filing systems are predominantly used for categories 4 and 5. Persistence is therefore an abstraction over the programming language and file system or DBMS views of data.

There are three major advantages in having only one model of data. Firstly in any traditional program there is usually a considerable amount of code, typically 30% of the total, concerned with transferring data to and from files or a DBMS. Much space and time is taken up by code to perform translations between the program's form of data and the form used for the long term storage medium. For example, we normally have to explicitly flatten and rebuild graphs or trees modelled in the programming language in order to write them out or read them back in from the file store. If this activity is performed by the underlying system or if the relevant utilities are parametrically polymorphic then every application program would be 30% smaller in terms of code, with all the attendant benefits that entails throughout the life cycle of the product.

The second major advantage is that it is conceptually simpler to have only one program view of data. Again in traditional systems there are three models of data, one in the real world, one in the database system and one in the program's runtime system. This is unsatisfactory since the programmer or system designer has to visualise and maintain all of these models correctly. This leads to an intellectual overhead in using these models and preventing them from becoming mutually inconsistent.

The third major advantage is that the data type protection offered by the programming language on its data is now available for all data. Thus the main protection mechanism is not lost across an unnecessary mapping.

We regard the provision of persistence as orthogonal to all other properties of data. For this we define the Principle of Persistence Independence as

> The persistence of a data object is independent of how the program manipulates that data object.

That is, all code should be written so that it will work with the same interpretation independent of the persistence of the data on which it operates. This reduces the number of conceptual mappings from three to one - the one between the real world and the program model and thus simplifies the task of building an application system.

We also identify an extension to the Principle of Data Type Completeness to handle persistence. It is

> All data objects should be allowed the full range of persistence.

Elsewhere we have described the advantages of including procedures[4] and graphics[20] as data objects in a persistent environment. By integrating other data objects such as sound with persistence we expect to achieve the same simplification by uniform application.

Large systems need long lived data since they all need to store their code and data. Computing systems that involve humans must support long term activity. Since the attention span of the human is short in computing terms then the storage of the present state of an attempted solution to a problem is essential. To achieve this in an integrated manner the system should provide persistence designed by the above principles.

4. CONTROLLED SYSTEM EVOLUTION

In traditional programming languages, operating systems and file systems there are a number of binding mechanisms which are often not easy to comprehend or use. A binding mechanism has four components by which it can be categorised. They are

a. what does the name bind to?

b. when is the binding performed?

c. what scoping is involved?

d. when is the type checking performed?

Names of variables bind to locations whose value may change without altering the binding. Constant names bind to values. The binding of both variables and constants is usually performed when the location is created or the value calculated. That is, dynamically unless of course the location or value is manifest (a literal) when the binding may be performed statically. Manifest constants can be seen in BCPL[21] or Pascal[25] and manifest locations are the variables of Fortran. In block structured languages it is usual for variables to bind to locations created dynamically. Constants whose values are created dynamically can be seen in S-algol[12], the simple values of Ada[15]and most applicative languages.

Names may be scoped statically in their compile time environment or dynamically in their run time environment. The traditional algol scoping rule is static whereas dynamic scoping can be seen in Lisp[18] or the segment binding mechanism of Multics[11].

Type checking may be performed statically by the compiler or dynamically by the run time system. Dynamic type checking occurs when the run time system executes code to ensure that the data is of the correct type. This typically occurs in read statements and in projections out of a union. Some languages such as SASL[24] deliberately choose run time type checking to facilitate polymorphism. For example it is difficult to write down or deduce the type of the self apply function

$$\lambda x.\ x\ x$$

but simple to check it at run time.

There are potentially 16 different methods of binding commonly in use in modern computer systems based on the four binding choices given above. The most static form of binding is where only manifest constants are allowed with static scoping and static type checking. The most dynamic form allows variables with non manifest values, dynamic scoping and dynamic type checking. We coin the term flexible incremental binding (FIB) to describe this mixture of bindings, which we expect to obey the Principle of Correspondence. In practice most languages have more than one binding mechanism.

The example of Pascal is interesting here where we observe that variable names may be bound to locations but not manifest locations and constant names may be bound to manifest values but not values created dynamically. The scoping rule is static except for file names where it is evaluated in the environment and is therefore dynamic. Type checking is static except in read statements and also on the projection out of a union of variants both of which require a dynamic check.

4.1. Name Spaces

The advantage of statically bound systems is that errors may be detected early in the software life cycle and that some checks for correct program execution may be factored out by the compiler. It is impossible to have a completely static system since we require some mechanism, e.g. the compiler, to create the bindings in the first place. Thus if the system contains that binding mechanism it already has a modicum of dynamic binding in it. This really should be obvious since to create dynamic change we need some dynamic mechanism. The modern trend in programming languages is to make as much of the binding as static as possible. The degree to which that can be achieved depends on the application.

We assert that the total spectrum of binding mechanisms is necessary for large system construction and evolution but we are unsure as to whether all of the intermediate binding mechanisms are useful. We suggest therefore that the programmer should have control over the binding mechanism.

To accommodate flexible incremental bindings we have invented the name space. This is an environment mechanism that permits the following:

a) the storage of bindings in a name space;

b) the dynamic use of names from a name space;

c) the static use of names from a name space;

d) the evolution of the names available in a name space;

e) safe exchange of arbitrary data between parts of the system - especially between the permanent store and programs.

Name spaces are data objects which act like abstract data types to yield the values of the name space. These values are bindings between name and object and the name may be used in the current environment. To statically bind to a name space we use

with name-space-expression **compile** statement

which provides an environment for the statement containing all the names in the name space. The name space may be an object in the persistent store and the statement is bound in the compile time environment. The name-space-expression is a manifest constant, statically scoped and statically type checked. The bindings in the name space provided for the qualified statement are variable or constant, manifest, statically scoped and statically type checked.

To bind a name space dynamically we use

using name-space-expression **with** signature **do** statement

In this the name space yielded by the expression must have the properties (name-type pairs) described by the signature in order to match correctly. It is a partial match and is the same mechanism that we use to match actual to formal type parameters in procedures and abstract data types to achieve parametric dependent polymorphism as first used in Russell[10]. In this case the name-space-expression, which may be regarded as an environment name, is constant but not manifest, dynamically scoped and dynamically type checked using its signature. The objects yielded from this name space are variable or constant, not manifest, and statically scoped and statically type checked.

We have chosen this combination of binding mechanisms to be under the programmer's control because we believe it will yield the right mix for building and maintaining large systems. For safety, most bindings will be as static as possible with some dynamic element to accommodate evolution. It is for the application designer to decide which mechanism is appropriate for a particular application.

To accommodate evolution, name spaces may grow and contract. For example

extend name-space-expression **with** identifier-list **from** statement

adds the new definitions of the identifier names to the name space. To contract the name space we use

drop identifier-list **from** name-space-expression

which will remove the identifier names from the name space. As a name space is itself an object in the language, a name in a name space may be bound to a name space. Hence hierarchical naming structures similar to directory hierarchies are possible.

4.2. Outstanding problems

When file directories are used as an environment mechanism it is possible to write general utilities which scan a directory, often interactively, revealing the directory's contents and possibly allowing user action - e.g. delete - on each file. Some equivalent operations on name spaces may be useful, however, it is difficult to identify a primitive in terms of which they may be written. Current practice is to step outside the type system or to depend on calling the compiler. Our desire to support more of the total programming activity within the strictly typed language leaves us confronting this problem.

5. TYPE SYSTEMS FOR PERSISTENT DATA

Ideally we would like a simple set of types, and a type algebra, so that by a succession of operations of the algebra, and provision of parameters, we could define a data type equivalent to any data model or conceptual data model. Parameterisation of such types yields schemata, i.e. the types of databases in the model, and each database is then an instance of such a type. We call this the "type alchemist's dream" as we do not yet know how to do it. In consequence the language designer has to opt for a particular model. Contrary to our original beliefs, this has to contain types motivated by long term data, because it tends to become large, and hence bulk operations and describing its regularity become important.

Based on an extensive survey[5] of existing, research and proposed languages together with experience of PS-algol[4], we have come up with the following general advice with regard to programming environments and persistent data.

i) the data types should include a mechanism to capture user communication including graphical and audio facilities;

ii) the data types should include program, as procedures and abstract data types;

iii) the data types should include a general purpose indexing mechanism, both polymorphic and variadic;

iv) the data types should include a bulk data type with operations on collections of data;

v) the data types should include some form of inheritance to model specialisation;

vi) the types and bindings used should allow system evolution on an incremental and localised basis.

Our own preferences for a particular type system are defined elsewhere[6].

6. CONCURRENCY AND TRANSACTIONS

In the context of concurrency and transactions in the persistent information space architecture we have identified the following as outstanding problems for which solutions will need to be found. They are

i) there should be mechanisms to permit and control shared and concurrent usage;

ii) it should be possible to encapsulate any sequence of operations on the store into a transaction, which behaves as a single operation;

iii) there should be privacy and access control mechanisms;

We have experimented with several versions of persistence[2,8]. Each one has a notion of transaction which allows a group of operations on data to appear atomic. The effects of the transaction are only visible to other transactions that start after the transaction has committed. Thus they are a mechanism for viewing and controlling change.

The concept of a transaction is clearly recursive with the fixed point being the machine's atomic store update. Nested transactions may be formed by grouping more than one store update into a single transaction. The semantics of these nested transactions are reasonably well understood[1] for a sequential process. At present we know of no satisfactory semantics for nested transactions in a concurrent environment.

Often associated with a transaction mechanism are methods for making the store reliable and for recovering from an aborted transaction. The stable store mechanism ensures that information is always kept (or copied) on non-volatile storage devices. Thus in the event of a system crash no information will be lost. Any stable store mechanism only gives relative protection since it may not be able to recover from some types of disaster, such as an atomic explosion. Most systems however provide an adequate level of service using a disk for the stable store. It is beyond the scope of this paper to discuss reliability except to say that our information space will be built on such a system.

Recovering from aborted transactions is much more controversial, especially in a concurrent environment. The problem is essentially one of undoing an action that other processes may have acted upon. For example having printed a document it is difficult to see how it may be unprinted. Most problems in this area involve interacting with the outside world. More about this will be said in the next section.

For expressing concurrency there is no shortage of models. Programming languages have provided semaphores[9], conditional critical regions[13], monitors[14], path expressions[7] and the rendezvous[15] among others. There is very little agreement on which are appropriate mechanisms for all programming activity. It would appear therefore to be sensible not to build any of these into a system but to provide a method by which they may be built using a primitive and some abstraction mechanism.

7. A PERSISTENT INFORMATION SPACE ARCHITECTURE

Specifying a computer architecture is equivalent to specifying a programming language that defines the architecture. In recognising this we can use our rules for controlling complexity when defining the language.

Central to our aim of building a total system capable of providing for all programming activity in an integrated manner is our persistent information space.

This space is made up of objects which may be simple or highly structured and are part of the universe of discourse defined by the type system of the PISA architecture. The type system must therefore be rich enough to satisfy all our requirements. This we recognise as a research challenge.

The information space is persistent. That is, the programmer has no knowledge of where the data resides. This may be done locally in main store or disk or remotely on non-local processors. The programmer is relieved of the burden of organising the physical storage of data in the system and presented with a conceptually simple model of data.

The mechanisms for binding in the persistent information space are those of the name space together with those used for introducing names in the architecture language.

At present we feel that it is premature to build in mechanisms and protocols for concurrency and transactions. We subscribe to the view that it is more sensible to build in a primitive for synchronization and a mechanism for specifying non-deterministic parallel computation and building the required protocols out of them by layers of abstraction[17]. For example it is possible to build monitors out of a primitive semaphore and a cobegin statement.

At a lower level the persistent information space is supported by a stable store mechanism for reliability. The stable store may be distributed over many storage devices and processors and it is currently the focus of some research to build such a system.

Given the above components we envisage an architecture where users of the persistent information space request a level of service for the data they are going to use. For example,

two Ada tasks may request a rendezvous protocol over certain data. If this can be granted, the protocol is given to the user to manipulate the data. The type of the protocol, be it a simple lock or a complicated transaction mechanism with recovery depends on the application. The persistent information space should be regarded as being inhabited by many processes concurrently. To use data simultaneously two processes must agree on a protocol. Since the protocols contain the data any two processes that are independent in their use of data may utilise competing protocols simultaneously.

8. CONCLUSIONS

A cause of difficulty in constructing suites of application programs out of existing languages, libraries, operating systems and databases has been identified. It may be decomposed into two parts

1. an overcomplex relationship between programs and their environment

2. inconsistency among the services provided by the environment

The approach proposed to overcome this is to design the environment and the language as a coherent whole. The initial steps outlined in this paper are sketches of the way certain facilities, such as flexible binding, reliable storage, long term storage etc. - currently provided by the operating system may be profitably specified as part of a language. Elsewhere we publish work where we demonstrate this approach is feasible.

9. ACKNOWLEDGEMENTS

Our dependence on a working implementation of PS-algol, from which we have learnt much, cannot be overstated. It was built largely by the following members of our team: Pete Bailey, Fred Brown, Paul Cockshott and Al Dearle. We also benefited from suggestions and ideas arising in discussions with Peter Buneman and Tony Davie. We are grateful to the referees for their comments and to Robert Milne for his notes on the paper. Alvey grant IKBS 104 supports this research at both Universities and at ICL. The work is also supported at Glasgow by SERC grants GRC 21977 and GRC 21960 and at St Andrews by SERC grant GRC 15907 and at both Universities by grants from ICL.

10. REFERENCES

[1] Atkinson, M.P., Cockshott, W.P. & Chisholm, K.J. NEPAL - The New Edinburgh Persistent Algorithmic Language. DATABASE Infotech State of the Art Report 9, 8 (1982), 299-318.

[2] Atkinson, M.P., Bailey, P.J., Cockshott, W.P., Chisholm, K.J. & Morrison, R. An approach to persistent programming. Computer Journal 26, 4 (1983), 360-365.

[3] Atkinson, M.P., Bailey, P.J., Cockshott, W.P. & Morrison, R. PS-algol reference manual. Universities of Glasgow and St Andrews PPRR-12 (1984).

[4] Atkinson, M.P. & Morrison, R. Procedures as persistent data objects. ACM.TOPLAS 7, 4 (1985).

[5] Atkinson, M.P & Buneman, O.P. Database programming language design. in preparation.

[6] Atkinson, M.P. & Morrison, R. Types, bindings and parameters in a persistent environment. Proc. Appin workshop. Universities of Glasgow and St Andrews PPRR-16 (1985), 1-24.

[7] Campbell, R.H. & Habermann, A.N. The specification of process synchronization by path expressions. Lecture Notes in Computer Science, 16 Springer-Verlag (1974).

[8] Cockshott, W.P., Atkinson, M.P., Bailey, P.J., Chisholm, K.J. & Morrison, R. The persistent object management system. Software, Practice & Experience 14 (1984).

[9] Dijkstra, E.W. THE multiprogramming system. Comm.ACM 11,5 (1968),341-346.

[10] Demers, A. & Donahue, J. Revised report on Russell. Technical report TR79-389, (1979), Cornell University.

[11] Dennis, J.R. Segmentation and the design of multiprogrammed computer systems. J.ACM 12,4 (1965), 589-602.

[12] Gunn H.I.E.,Morrison R., On the implementation of constants Information Processing Letters 9,1 (1979), 1-4.

[13] Hoare, C.A.R. Towards a theory of parallel programming. In Operating System Techniques (Ed Hoare & Perrot). Academic Press, London (1972), 61-71.

[14] Hoare, C.A.R. Monitors : an operating system structuring concept. Comm.ACM 17, 10 (1974), 549-557.

[15] Ichbiah et al., The Programming Language Ada Reference Manual. ANSI/MIL-STD-1815A-1983. (1983).

[16] Landin, P.J. The next 700 programming languages. Comm.ACM 9, 3 (1966), 157-164.

[17] Krablin, G.L. Building flexible multilevel transactions in a distributed persistent environment. Proc. of the Appin workshop. Universities of Glasgow and St Andrews PPRR-16 (1985), 83-106

[18] McCarthy, J. et al. Lisp 1.5 Programmers manual. M.I.T. Press Cambridge Mass. (1962).

[19] Morrison, R. S-algol language reference manual. University of St Andrews CS/79/1 (1979).

[20] Morrison, R., Brown, A.L., Dearle, A. & Atkinson, M.P. An Integrated Graphics Programming Environment. 4th UK Eurographics Conference, Glasgow (March 1986)

[21] Richards M. BCPL, a tool for compiler writing and systems programming AFIPS SJCC 1969

[22] Strachey, C. Fundamental concepts in programming languages. Oxford University Press, Oxford (1967).

[23] Tennent, R.D. Language design methods based on semantic principles. Acta Informatica 8 (1977), 97-112.

[24] Turner, D.A. SASL language manual. University of St.Andrews CS/79/3 (1979).

[25] Wirth, N. The programming language Pascal. Acta Informatica 1, 1 (1971), 35-63.