Persistent Information Architectures

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

University of Glasgow, 14 Lilybank Gardens, Glasgow, Scotland G12 8QQ.,
University of St. Andrews, North Haugh, St Andrews, Scotland KY16 9SS.
and
STC Technology Ltd., Copthall House, Nelson Place, Newcastle-under-Lyme,
England ST5 1EZ.

Abstract

The work of the PISA group is briefly reported to present our views on the directions that future research and development in persistent architectures will need to take. The long term and large scale issues lead us to consider software architecture as important and to propose four domains of influence for this architecture research.

1. Introduction

New architectures are required since the existing options are the consequence of outmoded constraints of previous technologies. The old architectures have become over complex as they have attempted to assimilate new methods of working enabled by modern technology. This complexity prevents the construction of the large and sophisticated information systems we would now like to build, by diverting the intellectual efforts of software and knowledge engineers. New architectures will replace this complexity with simple consistent rules and may also permit higher performance execution and storage engines to be built.

The concept of persistence was introduced to simplify the programmer's world. It allows any data structure of whatever type to exist for any length of time without explicit coding effort by the programmer [1,2,5]. It has proved to have far reaching consequences; for example data and program may be treated in precisely the same way in our abstract store, with very significant conceptual and implementation savings [6]. Persistence facilitates the reuse of data and program over long periods, but, as longer term usage is considered, new techniques have to be developed [9].

Ultimately the support of program and data is divided between hardware and software on the basis of cost, performance and available technology. In terms of current trade offs we envisage a substantial proportion of the applications system environment being provided by software. This software has to conform to an architecture if the programming environment is to remain under control. We take this further. The division between systems software and applications software is an inappropriate artifact of past habits. The architectural discipline and techniques that apply at one level apply at each level, above and below it, as improved software composition technology allows us to consider one person's applications suite as another person's supporting software. Thus the software architectures being developed will influence all applications programming [26,28].

An important goal of a software architecture is to facilitate the reuse of program and data. It requires a sufficiently good component description system to allow components to be found, and to check on their composition, and a sufficiently powerful binding system to allow choice of the rebuilding necessary to replace components with improved components. As it is not only the components themselves, but also their

definition which sometimes requires evolution, the binding mechanisms must incorporate support for meta data access and evolution [11].

Many of the approaches to software, data description and organisation which work quite well for small collections of data and program become intractable when the system is large or long lived. For example, appealing to a universal name space or a single name allocation authority. These issues become more significant as distribution is attempted. Their solution depends on a recognition of the need for independence and an architecture based on a federation of independent agents interacting via negotiated protocols and descriptions [8].

The basic principals of persistence and their implications for type systems and bindings are well understood and are ripe for development. The effort needed here is the investment in good implementation technology including support for object addressing and type checking based on structural equivalence. Significant development is now possible in building systems using persistence: operating systems, data models, data base systems, AI systems, object stores, generic tool sets and a wide range of application suites [26,28]. It is becoming widely agreed that persistence should be used as the basis for building operating systems and knowledge bases [13]. The longer term research will address the issues of system evolution, of heterogeneity and of algorithms applicable in distributed stores.

We divide the total architecture into four mutually supporting domains, and our expansion of the above introduction is based on these domains.

2. The Four Architectural Domains

2.1 The Application Domain

Most software effort is expended writing application software, and we consider it important to improve the organisation of this work and to support it with suitable tools and components; this combination of organisation, methods, tools and components we call the application's level architecture. It includes generic tool sets which can work over a range of types and sizes of data. It provides adaptive components that modify their behaviour to suit new types of data, types that had not been declared when the component was coded and compiled. A class of adaptive components we have worked

with, use the compiler at execution time to produce new subcomponents which match specific given data - gaining both flexibility and efficiency [9,11]. An application of this technique which is worthy of exploitation and further research has allowed us to rapidly implement a variety of data models and data manipulation languages. This will have benefits in the construction of new data base models, the verification of the utility of existing proposals, and the production of data and object oriented application systems.

The adaptive components can allow high performance algorithms for large scale data to be encapsulated in a manner similar to numerical algorithms. Developing such algorithms is a continuing line of research. It has become clear that no one definition of transaction suits all applications, so the applications architecture must provide the means for defining appropriate transactions [14,32,48]. Without the discipline of a well defined architecture work at this application's level soon deteriorates into a morass of complexity.

As the application's programmer has to interact with the environment, a consistent model, independent of machine and installation must be available [15,37,39,40,43].

2.2 The Language Domain

The language domain describes the facilities required in a programming language to support persistent programming in the applications domain. We expect the language to cater for all programming activity including the control of the programmer's environment. In such an environment the language must support the easy construction of data models, generic tool sets, object stores and plug in componentry for system building. At present we have two demonstrator languages PS-algol and Napier in which we have based our various experiments. The major research issues of such a language are:

2.2.1 Finding a type system rich enough to capture all our uses of data.

When the environment is included in our remit this includes types to describe the data models and the data bases we use. Ideally we would like a set of types and a type algebra so that by a succession of operations of the algebra and the provision of parameters we can define a data type equivalent to any data model or conceptual data model [7,12,17,18,19,26,33,41,42].

In traditional terms we wish to merge the concepts of type systems with data base schemata to provide a type secure environment. One result we have already achieved is that the type equivalence rule for persistent systems is structural equivalence. This allows independently prepared types or schema to be matched [9]. This is only one step away from the facilities required of the schema to support a distributed persistent store. That is, the schema should be flexible enough to be held in parts, in distributed dictionaries rather than in a centralised one.

When describing and using large scale and long term data, one method of system modelling is to encapsulate the data in a system of co-operating concurrent processes with an access protocol. Support for concurrency must be included in the type system to allow for this kind of modelling [14,48].

With long lived data we cannot predict its use, we require data type and a schema that adapt to changed requirements. For example, it is often necessary to add attributes to the data without destroying the old data or bindings to it.

Our persistent language Napier contains a type system based on types as sets [7,21,41]. This allows polymorphic procedures of universally quantified types [30,34,35], abstract data types of existentially quantified types [36] and extensible data types [7]. The schema may be distributed with structural equivalence as the matching rule for types. Such a type system may be used to develop data models, generic tool sets and plug in component system building.

2.2.2 A binding mechanisms suitable for coping with adaptive data.

We have identified the inconsistencies of the binding and naming mechanisms present in current programming languages, operating systems, file systems and database management systems as a major source of distracting complexity in system construction. A binding consists of a name, a value, a type and an indication of whether the object is mutable or not and we have identified 16 different categories that are common in programming languages, file systems and operating systems and database management systems.

With static binding there can be no change in the system. That is, since everything is static we cannot accommodate any alteration to the data. To allow system evolution in a controlled manner we must develop binding mechanisms within the language that accommodate change. Such binding mechanisms are dynamic or at least incremental.

We expect some bindings to be static for safety and others to be dynamic for flexibility and we have proposed the notion of flexible incremental binding sets in persistent systems to deal with this [10]. Some of our success in building software systems out of components will be in deciding which of the components are statically bound for safety and which are dynamically bound for flexibility. Our mixture of binding mechanisms allows for this, and our experience shows it can be understood and exploited by application's programmers.

2.2.3 A contextual naming scheme.

With persistent systems the naming of objects in a conceptually flat store becomes a major problem. If the name space were also flat then name clashes and identifying components would become a major problem to the user. Programming languages (block structure), file systems (file directories) and operating systems (segments) have proposed solutions to this problem by imposing a contextual naming system in the form of a tree. In the most general case, the persistent information will form a graph and we have developed a scheme based on extensible environment constructors that allow contextual naming on the graph. In general, the environment constructors allow us to dynamically construct contexts in a manner that models both block structure and directories.

Names also have an important role to play in the description of data and in

bindings. For this reason we have invented a method of abstracting over the names in a set of bindings in a type secure manner. This allows us to discover the names in bindings and to write algorithms which depend upon the meta-data without loosing the normal type and value system of the language [11].

2.2.4 Iterators

As a consequence of persistence we have invented a rich polymorphic type system with extensible environment constructors and first class names. To harness the power of the environments we require an iterator that works over the abstract form of the name and the type of the bindings in the environment [11]. With such an iterator, we can define an abstract form of interaction with the program's environment. This allows, for example, the building of the equivalent of file store operations in a type secure manner.

We consider that only by basing the overall architecture on the well defined semantics of a language can the architecture be kept consistent and understandable.

2.3 The System Building Domain

The system building domain supports the construction of the persistent language and the persistent environment. The major components of system construction considered here are

(a) compiler componentry
(b) support for compilation and execution merging
(c) abstract program graphs
(d) abstract machine design
(e) demand driven optimisation

2.3.1 Compiler componentry

_____Plug in tool sets are a very important method of system construction and the early UNIX systems were a good example of this. The essence of the technique is provision of the framework for storing individual components together with a method of using them in the construction of larger systems.

In order to support the rapid prototyping of compiler systems we have developed a

technology based on plug in componentry. The compilation system is composed of separately compiled modules which are plugged in to the compiler itself as it is dynamically loaded. This technique has been used to develop compilers for PS-algol [23,45], Napier [7], SASL [47], STAPLE [27] and Hope$^+$. Each compiler module when run, plugs itself in. When the compiler runs, it constructs itself from the latest versions of these components in the persistent store.

We expect to use the plug in componentry architecture to both aid the construction of systems from generic tool sets and also to replace the generic tools set method when it is not appropriate. These methods of system construction are also applicable in the applications domain.

2.3.2 Support for the merging of compilation and execution

In persistent system the boundary between compilation time and execution time is blurred. It is further blurred when we consider interactive compilers which may work against objects in the persistent store. To support this we require a compiler that is callable from within the system, in a type secure manner. This raises issues of where the type security is controlled in the system that are the same as the issues of capability control in machine architectures.

Other support facilities, such as browsers [28,38] and debuggers, must also be provided without endangering the type security of the persistent store.

2.3.3 Abstract program graphs

The traditional method of producing files from compilers, often losing information for compactness, is not necessary in a persistent system. The tables and graphs produced by the compiler can be kept until execution without explicitly altering their form. This allows better system diagnostics to be generated including browsers and debuggers.

An abstract description is also required of the abstract graphs. We have invented such a description called PAIL (Persistent Architecture Intermediate Language) [29] which we are using as the standard for code generators to work against. We are at present discussing with other groups the canonical form of such graphs.

2.3.4 Abstract machine design

As with any architecture there is an abstract model of a machine on which it will best execute. This method of compiler design has been common in defining an abstract machine to execute the language. Both PS-algol and Napier have well defined abstract machines [22,44]. These are block retention abstract machines to support both higher order procedures and abstract data types. We are discussing with other persistent language implementors (Albano at Pisa) on the ideal abstract machine.

2.3.5 Demand driven optimisation

With abstract descriptions of program graphs, abstract machine descriptions and program source available throughout the lifetime of the program and data it is possible to optimise the object according to its use. The optimisation is merely another view of the object and we can use it for greater efficiency.

We would expect lazy demand driven optimizers, based on usage statistics, to be developed in persistent systems.

2.4 The Store Domain

One of the principal uses of computers is as a storage device - much system building is concerned with constructing repositories for data with a variety of data capture, data retrieval and organisation mechanisms. A large proportion of programmers therefore need to use the architecture and languages to visualise a store and specialise it to their needs. The PISA approach has developed a uniform view of an arbitrarily large reliable and distributed store populated by strongly typed objects. The programmer defines new types of store by defining storage types and instances of those types become objects. Thus the power of the type system to define the range of stores required is of paramount importance, and the integrity of the store is achieved via type checking. Type is again important as a means of communicating to the store information about objects, essential when data is to be transmitted between hetrageneous, distributed components while preserving semantics. The type information can also be used to activate special treatment of some objects, for example: the use of special purpose hardware, data compression, encryption and protection. The overall performance of the store is critical

to the total computational process, and this requires further development of adaptive algorithms and the introduction of hardware support for this kind of storage architecture [3,4,16,20,24,25,31,46].

The reliability of the store is also an issue for persistent systems and we have developed several, often complementary, mechanisms for supporting stable stores.

3. Acknowledgement

We acknowledge the support of ICL, Alvey and the S.E.R.C. in enabling us to have brought the research to its present state.

4. Bibliography

1. 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.

2. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"PS-algol: An Algol with a Persistent Heap", ACM SIGPLAN Notices Vol.17, No. 7, (July 1981) 24-31.

3. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"Algorithms for a Persistent Heap", Software Practice and Experience, Vol.13, No.3, 259-272 (March 1983).

4. Atkinson, M.P., Chisholm, K.J. & Cockshott, W.P.
"CMS - A chunk management system", Software Practice and Experience, Vol.13, No.3

5. Atkinson, M.P., Bailey, P.J., Chisholm, K.J., Cockshott, W.P. & Morrison, R.
"An approach to persistent programming", The Computer Journal, 1983, Vol.26, No.4, 360-365.

6. Atkinson, M.P. & Morrison, R.

"Procedures as persistent data objects", ACM TOPLAS 7, 4, 539-559, (Oct. 1985).

7. Atkinson, M.P. & Morrison, R.
"Types, bindings and parameters in a persistent environment", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 1-24.

8. 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.

9. Atkinson, M.P., Buneman, O.P. & Morrison, R.
"Delayed Binding and Type Checking in Database Programming Languages", in preparation.

10. Atkinson, M.P. & Morrison, R.
"Flexible Incremental Bindings", in preparation.

11. Atkinson, M.P. & Morrison, R.
"Polymorphic Names for Persistent Object Systems", in preparation.

12. Atkinson, M.P., & Buneman, O.P.
"Database Programming Language Design", Universities of Glasgow and St Andrews PPRR-17-85.

13. Balzer, R.
"Living with the Next Generation of Operating Systems", proceedings of Information Processing 1986, Dublin, September 1986, (ed. H.J. Kugler), 283-292, North Holland Press.

14. Barter, C.J., Brown, A.L., Carrick, R., Connor, R., Dearle, A., Hurst, A.J., Livesey, M.J. & Morrison, R.
"Polymorphic Persistent Processes", in preparation.

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

(Summer 1986).

16. Brown, A.L. and Cockshott, W.P.
"CPOMS - A Revised Version of The Persistent Object Management System in C". Universities of Glasgow and St Andrews PPRR-13-85.

17. Buneman, O.P.
"Data types for data base programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 291-303.

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

19. Buneman, O.P. & Ochari, A
"A Domain Theoretic Approach to Higher-Order Relations", Universities of Glasgow and St Andrews PPRR-28-86b.

20. Campin, J. & Atkinson, M.P.
"A Persistent Store Garbage Collector with Statistical Facilities", Universities of Glasgow and St Andrews PPRR-29-86.

21. Cardelli, L. & Wegner, P.
"On understanding types, data abstraction and polymorphism", ACM.Computing Surveys 17, 4 (December 1985), 471-523.

22. Cardelli, L.
"The Functional Abstract Machine", Polymorphism, VOL 1, NO 1 (1983).

23. Carrick, R., Cole, A.J. & Morrison, R.
"An Introduction to PS-algol Programming", Universities of Glasgow and St Andrews PPRR-31-86.

24. Cockshott, W.P., Atkinson, M.P., Chisholm, K.J., Bailey, P.J. & Morrison, R.
"POMS : a persistent object management system", Software Practice and Exerience, Vol.14, No.1, 49-71, January 1984.

25. Cockshott, W.P.
"Addressing mechanisms and persistent programming", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 363-383.

26. Cooper, R.L., Adberrahmane, D., Atkinson, M.P. & Dearle A.
"Applications Programming in PS-algol", Proc VLDB Brighton 1987.

27. Davie, A.J.T., Munro, D. & Mc Nally, D.J.
"An Informal Description of the STAPLE Language Version 1.4", University of St Andrews CS/86/3.

28. Dearle, A and Brown, A.L.
"Safe Browsing in a Strongly Typed Persistent Environment", Universities of Glasgow and St Andrews PPRR-33-87.

29. Dearle, A.
"A Persistent Architecture Intermediate Language", Universities of Glasgow and St Andrews PPRR-35-87.

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

31. Hurst, A.J.
"A Context Sensitive Addressing Model", Universities of Glasgow and St Andrews PPRR-27-87.

32. Krablin, G.L.
"Building flexible multilevel transactions in a distributed persistent environment, proceedings of Data Types and Persistence Workshop, Appin, August 1985, 86-117.

33. Kulkarni, K.G. & Atkinson, M.P.
"EFDM : Extended Functional Data Model", The Computer Journal, Vol.29, No.1, (1986) 38-45.

34. Matthews, D.
"An Overview of the Poly Programming Language", proceedings of Data Types and Persistence Workshop, Appin, August 1985, 265-274.

35. Milner, R.
"A theory of type polymorphism in programming", JACM 26(4), 792-818.

36. Mitchell, J.C. & Plotkin, G.D
"Abstract types have existential type", Proc POPL 1985.

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

38. 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 (August 1985), 166-172.

39. 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, Vol.14, No.3, (1986).

40. Morrison R., Dearle, A., Brown, A. & Atkinson M.P.
"An integrated graphics programming environment", Computer Graphics Forum, Vol. 5, No. 2, June 1986, 147-157.

41. Morrison, R, Brown, A, Connor, R and Dearle, A
"Polymorphism, Persistence and Software Reuse in a Strongly Typed Object Oriented Environment", Universities of Glasgow and St Andrews PPRR-32-87.

42. 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.

43. Philbrow, P & Atkinson M.P.
"Machine Independent print Facilities for Bitmapped Screens", in preparation.

44. PS-algol Abstract Machine Manual
Universities of Glasgow and St Andrews PPRR-11

45. PS-algol Reference Manual - 4th edition
Universities of Glasgow and St Andrews PPRR-12

46. Rosenberg, J. & Abramson, D.A.
"The Monads Architecture: Motivation and Implementation", Proc First Pan Pacific Conference, Melbourne 1985, 4/10-4/23.

47. Turner, D.
"SASL Language Reference Manual", University of St Andrews, CS79/3 (1979).

48. Wai, F.

"Parallelism and Distribution in Persistent Programming Languages", in preparation.