South African
Computer
Journal
Number 11
May 1994

Suid-Afrikaanse
Rekenaar-
tydskrif
Nommer 11
Mei 1994

Computer Science and Information Systems

Rekenaarwetenskap en Inligtingstelsels

The South African Computer Journal
An official publication of the Computer Society of South Africa and the South African Institute of Computer Scientists

Die Suid-Afrikaanse Rekenaartydskrif
'n Amptelike publikasie van die Rekenaarverenging van Suid-Afrika en die Suid-Afrikaanse Instituut vir Rekenaarwetenskaplikes

Editor
Professor Derrick G Kourie
Department of Computer Science
University of Pretoria
Hatfield 0083
Email: dkourie@dos-lan.cs.up.ac.za

Subeditor: Information Systems
Prof John Shochot
University of the Witwatersrand
Private Bag 3
WITS 2050
Email: 035ebrs@witsvma.wits.ac.za

Production Editor
Dr Riël Smit
Mosaic Software (Pty) Ltd
P.O.Box 16650
Vlaeberg 8018
Email: gds@cs.uct.ac.za

Subscriptions

	Annual	Single copy
Southern Africa:	R45,00	R15,00
Elsewhere:	$45,00	$15,00

to be sent to:
Computer Society of South Africa
Box 1714 Halfway House 1685

Phone: +27 (11) 315-1319 Fax: +27 (11) 315-2276

Metadata and Security Management in a Persistent Store

Sonia Berman
Department of Computer Science, University of Cape Town, Rondebosch, 7700

Abstract

Since its emergence in the early eighties, persistence has become an important branch of Computer Science. Many persistent systems have now been developed and a wide variety of related issues have been well researched. Two areas which have received little attention are metadata management and security enforcement. This paper investigates the incorporation of these features into an existing persistence machine. The CPOMS (Persistent Object Management System in C) was used as a vehicle for the study, because of its popularity and reliability. We discuss design alternatives, motivate those selected for this experiment and illustrate how they have been implemented. Our results are evaluated, highlighting some areas for future work.

Keywords: *Design, Experimentation, Performance, Metadata, Security*
Computing Review Categories: *H.2.0*

Received: March 1993, Accepted: September 1993, Final version: February 1994.

1 Introduction

Persistent programming languages extend general purpose languages by providing persistence for data of any type; they were proposed as an alternative to conventional database systems in [2]. As a result of its programming language origins, the large body of research that has arisen in this area has failed to adequately examine data-oriented issues [7], particularly as regards metadata and security management. This paper describes an attempt to rectify this situation by including these facilities in the CPOMS (Persistent Object Management System in C) [4]. This machine was chosen as the basis for our experiment because it is well established and flexible, having been used by a variety of persistent languages [1, 3, 6, 9–12]. Our aim was to improve this environment and at the same time extend its applicability, so that it is capable of supporting a wider range of programming language features. The paper begins by introducing the concept of a persistence machine. The differences between a persistence environment and conventional DBMS scenarios are highlighted, and the implications for type management and data protection are discussed. We examine alternative approaches to security and metadata handling in this context, and explain the options selected for the extended CPOMS. These are described from the user viewpoint and at the implementation level. The paper concludes with an evaluation of the work and suggestions for future research.

2 Background

This section introduces the notion of a persistent language and describes its advantages over a conventional database environment. We then outline the CPOMS system in terms of its database-oriented instructions and its management of the data store. In conclusion, the effects of this architecture on metadata and security subsystems are presented.

Persistent Languages

A persistent programming language extends a general purpose language by allowing data of any type to remain on disk after execution terminates. This data is kept on a database, but is manipulated identically to conventional, transient data. In other words, it is impossible to tell whether a program statement is handling conventional heap objects or database items, because the same structures and operators are used in both contexts. This environment offers several advantages; experience with PS-Algol [12] showed that productivity increased, program code was shortened and maintenance simplified, compared with conventional database applications [2]. In the first place, programmers do not need to learn a separate database language with its own types and operators. A program fragment is expressed independently of the persistence of data it uses. Secondly, there is no need for code that translates objects from database format into programming language format and vice versa; nor for statements that move data between disk and main memory every time a database is accessed. Studies have shown that such code typically takes up 30 percent of an application program [2]. Furthermore, the database is able to support the complex data types and data structures of the programming language, and is not restricted to flat files or first normal form relations.

The Object Management System

The CPOMS [4] was originally developed for the persistent language PS-Algol. It manages the persistent store containing all database data, and will automatically retrieve or save objects there when necessary. It provides for flexibility through simplicity, and has been used to support a variety of languages [1, 6, 9–12]. This paper describes an extension of this system to incorporate metadata and security management, without changing its interface or introducing

any restrictions on the kinds of language supported. In this way, the new features are available to a wide range of persistent languages. Before giving details of this extension, some description of the workings of the CPOMS is in order.

The persistent store is divided into a number of databases which form the unit of locking. That is, each database may be concurrently opened by one writer or many readers. Inter-database references are permitted however, so databases are not independent collections. Any type of data may be added to the store at any time; there is no schema restricting database contents or curtailing data organisation.

To minimise database store and retrieve commands, only so-called "primary" objects are explicitly written to (using "Enter") or fetched from (using "Lookup") a database. The majority of items are automatically inserted on the store using the principle of reachability, whereby the system automatically stores on disk all objects referenced by persistent data. In other words, the transitive closure of pointer references from a primary becomes persistent. For example, if a program creates a large tree structure and Enters its root, the entire tree persists. When the machine detects a pointer dereference which is a disk address, it automatically copies the item off the store into memory. The system keeps a record of objects that have already been placed in memory, to prevent repeated accesses. Changes are only written back to a database when a "Commit" statement is executed. This enables a program to apply a transaction atomically: that is, either all its changes are applied to the store, or none. Commit also writes new objects to the store, placing these on the same page as the data that references them wherever possible. As there can be several pointer paths to an item, it cannot in general be determined on which database a non-primary object resides.

Implications for Data Management

The CPOMS environment has two fundamental properties, neither of which applies to conventional database systems, that affect metadata and security management. Firstly, it adheres to the Persistence Principle [2] that underlies all persistent languages; and furthermore it must be sufficiently flexible to support a wide range of programming languages. In this section we consider the effects of each of these characteristics in turn.

The persistence principle [2] states that there should be no distinction between transient and persistent objects. Programs should be able to freely and implicitly add new types of data to the store; there must be no schema defining "persistent types", and no database-related specifications in programs. Reading and writing to a database is largely transparent in program text; and it is impossible to distinguish database objects in the code.

The absence of a schema makes it extremely difficult to find out about the types of objects on a database and so there is a real need for some metadata query facility. There cannot be some program section declaring the database types to be used, or its authorisation to access information. Special handling of database objects, e.g. to check their security clearance, is not possible at compile time because persistent objects are indistinguishable from other data. Thus, in particular, attempts to illegally update an item cannot be detected until runtime. Nor can one tell by examining the program code at which points data is fetched from a database, although type- and security checking may be required. Since insertions are not explicit, it is only when executing Commit that unauthorised creation of objects can be detected. Deletions are only evident on garbage collection, which occurs after all programs have ceased using the persistent store.

The metadata system must permit any kind of item to be stored, along with is complete type specification; not just normalised relations or object-oriented objects. Without this flexibility, we would have to limit the languages that can be supported. Thus the internal metadata representation must in no way restrict the types allowed on the store. Type checking must be performed by any persistent system, to ensure that the type of a database object conforms to that expected by the program wishing to use it. It is possible to do all this type checking statically – but only if the compiler is written in a persistent language (and is thus able to access the metadata) and only if the language has no dynamic typing. In order to support other persistent environments as well, it is necessary to provide for runtime type checking of database data.

The majority of persistent languages which are currently supported by the system have no notion of sets or classes. Instead, because of the reachability principle, most databases consist mainly of networks of objects connected through pointers. Graph structures do not lend themselves to security enforcement as easily as simpler structures like sets, sequences or bags. Access predicate protection (eg all employees where salary > 9999) and statistical control (whereby individual values are inaccessible but averages, minima, etc. are disclosed) are hard to provide for in an environment where data is available through reachability. It becomes difficult to ensure that data which a program is entitled to use is accessible to it along some path; and also to enforce the protection of an item which is reachable via many routes. As an example, consider the case of a write-protected pointer. This can prevent unauthorised alteration of the pointer value, but cannot prevent the contents of the target object from being changed. Write-protection along a pointer path is easily circumvented, since a program can always construct a new database item having a pointer to the target object and then edit it through this new path, as shown in Figure 1.

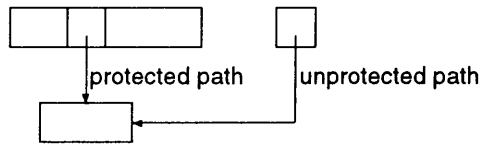

Figure 1. Write-protection of pointers cannot prevent target object alteration.

3 Extending the System

The extended CPOMS [3] has four new features: metadata management, security, set types and complex objects; only the first two are discussed in this paper. Metadata is automatically maintained to allow type checking and security enforcement, and to permit querying of type information. Thus users are able to learn about a database, recall type specifications or study the mix of data across databases, etc. This metadata is also available to system software like browsers and diagnostic aids.

The type system of a programming language is but one way of protecting data from misuse. Preventing unauthorised disclosure or alteration of certain information is equally important in most database applications. Our security subsystem protects database items from being examined, changed, deleted or created without appropriate authorisation.

For the sake of completeness, we note the new types supported by the extended system: a set type was developed to cater for bulk data collections, and nested structures weré introduced to directly support languages with complex data [3]. Previously, all values had to be atomic; any structured component was implemented as a pointer to a separate object.

4 Metadata Management

Since metadata handling is affected by the introduction of security control, we first consider a system without database protection. Looking at type maintenance alone therefore, we outline our requirements for the metadata management system. Its implementation is described in the two subsections that follow.

Using Type Information

There are two reasons for keeping metadata: to permit the types on a database to be queried; and for type checking purposes.

Interrogation by the User

Metadata incorporates type definitions, security, ownership, instance counts, placement information and cross references. The last of these is optional – if there are a large number of inter-related types, the additional space required for storing inverse relationships may not be warranted. All types are treated identically, including Standard types and component types having a definition but no name. For example, if a component is declared to be an array with elements of type X, this unnamed array type will be included in cross-references to X.

Since inter-object references can span several databases, it cannot always be determined on which database a persistent object will be placed. Therefore type names must be unique across the entire store and type-related access privileges must be the same for all databases. A program can then have a new object stored on any database without conflicts arising, and can make copies of an object even if some of these end up being stored on a different database.

Type information can be obtained interactively using a metadata query utility. This allows programmers to learn about a database prior to writing code that will access it. Within a program, standard metadata access functions are available; for example, one of these returns the number of occurrences of a given type on a given database. Metadata can be accessed in the same way as data. Since type information is organised in sets, and the set operations available are relationally complete, this form of metadata querying is particularly easy. The sets are indexed in such a way that queries relating to a specific type, field or person are efficiently handled (information on a person identifies the types for which they are responsible). Queries are phrased with respect to a particular database, all databases currently open, the persistent store as a whole, or all currently reachable data (i.e. including non-database items created by the program).

Type checking

Program declarations must be checked against database types to prevent persistent data from being wrongly interpreted or manipulated. For languages where all typing is static, it is sufficient to type check each primary on a Lookup and to add to metadata when new primaries are Entered. This requires that every primary be associated with a type object on the database, which is some encoded representation of its type declaration. Naturally all non-standard types referenced must also be fully specified. The compiler includes encoded type declarations in the object code so that the runtime system can add new types to the metadata.

An important feature of the CPOMS is its ability to support languages with dynamic typing. For example, PS-Algol has untyped pointers, so pointer dereferences require dynamic type checking. As dynamic typing may be necessary for objects retrieved from a database, all persistent objects include a pointer to their type. If a pointer dereference causes an object to be fetched from the store, the system converts the database representation of its type into the required heap format, to permit the dynamic type checking that follows. When a newly persistent item is written to a database, the system ensures that its type pointer references the correct Persistent Type. Before explaining how these tasks have been implemented, the internal storage of type information is outlined.

Internal Metadata Representation

This section briefly describes how type information is stored; firstly on the persistent store and secondly on the local heap. For a more detailed description of the data structures used and the reasons behind their choice, the reader is referred to [3].

Metadata comprises a set of Persistent Type objects, each of which encapsulates the definition of one data type referenced on a database. There is a separate copy of every Persistent Type on each database where it is used, so as to reduce metadata access conflicts. A central metadata repos-

itory in the form of a system-owned database METADB is used to maintain consistency across databases.

All data objects include a pointer to their type in their header, as this is needed for languages with dynamic typing. It should be pointed out that type pointers are useful in any persistent system for facilities such as type-related security and physical clustering by type [3]. The links between persistent data and metadata are illustrated in Figure 2.

A type declaration was initially encoded as a string, but this was replaced by a scheme using type graphs. A single node in this graph represents one data type T; if T references another type U (e.g. U is the type of a field or element of T), then there is a pointer to the node for U. Thus structured types take the form of a graph. This representation speeds up type checking, simplifies the interface to metadata, saves space [5], facilitates garbage collection of types and avoids keeping in memory information which is not actually needed during execution [3]. The only potential problem with a graph format is that it might limit the type constructors that can be supported, and we do not wish to place any restrictions on languages that may use the system. To illustrate this, suppose we were supporting a single language with say M type constructors. We could have $M + 1$ kinds of node in the type graphs: M kinds to store the information for the M constructors and an additional one for standard types. Since we wish to make the representation completely general however, a combination of strings and graphs is used. Each node in the graph is a string, and so can be a compacted representation of any type declaration whatsoever. Within this string, referenced types are represented by the special symbol "%%", along with a pointer to the node for that type.

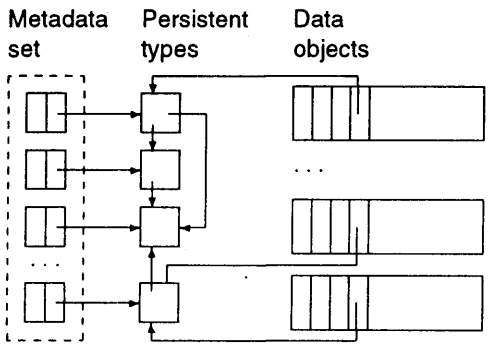

Figure 2. Metadata on the persistent store.

In addition to metadata recorded on the persistent store, transient types must also be kept in memory so that they can be compared against their persistent counterparts, or added to the metadata if necessary. Object code thus includes encoded type declarations. Each of these is associated with a type number, which is allocated according to the ordering of declarations in the program text. Instructions with types as parameters identify these by type number. At runtime, the encoded declarations are used to create "Local Type" objects, which are kept in an array indexed on type number. These Local Types contain type name, the encoded type definition, security data and the disk addresses of the corresponding type on different databases. After the type has been checked against its database counterpart, the definition is replaced by a flag indicating if type checking succeeded.

Implementing Type Management

To manage type information in the persistent store, the system needs to ensure that type checking is done where necessary, that the type pointer in every object's header is correct, and that new information is added to metadata on the persistent store when appropriate.

Type checking

It would appear that checking program types against their definition on the persistent store must be done as each object is retrieved from a database. Since it cannot be determined statically which pointer traversals access persistent data, the interpreter would have to be informed of the type expected with every pointer dereference and would carry the overhead of always testing if type checking is required. Fortunately a more efficient approach is possible however: when we encounter a primary object Lookup, deep type checking is performed. This ensures that all data reachable from there can subsequently be retrieved without checking.

A compiler which is itself written in a persistent language can access the store and type check primary structures statically. Should this not be the case, the extended CPOMS will do so at runtime when executing a Lookup; the compiler simply has to supply the type number of the primary type. To check type compatibility, declarations in the Local Type array are compared with type graphs on the persistent store. Deep type checking stops when a Local Type is encountered that is already flagged. If type checking succeeds, the database address of the Persistent Type is recorded so that any newly persistent objects of this type can have their type pointers set accordingly. Type graphs on the heap are disposed of after type checking.

Newly Persistent Objects

Programs create objects by means of a New instruction. The object code for a New statement includes a parameter identifying the object's type. When the instruction is executed, this type number is used to index to the appropriate Local Type, and the type pointer in the header of the new object is made to point there. Thus it is always possible to determine data type by following this pointer.

The Commit instruction creates new items on the persistent store. It must first examine such objects to check their type and to add new types to the metadata. Commit also has the task of identifying these newly persistent objects and updating the type pointer in their headers to reference the corresponding Persistent Type. Now an object X can only be created on the store if some persistent object points to X, or if X is Entered. If X is referenced by a persistent object, then its type will already have been checked before its parent was accessed. Therefore to handle newly persistent objects, the only additional type checking required is ensuring that new primaries conform with the types expected on a database. This can be done by deep type checking new primaries, either on Commit or on

Enter.

The checking is performed on Commit to cater for situations where a program chooses not to Commit a transaction in which an illegal Enter occurred. It is in any case more efficient to check then, because the Local Type needs to be located during Commit in any case, to set type pointers in object headers. Commit follows the type pointer from a new object's header to the Local Type, and checks the flags there. If this indicates a type mismatch, then the Commit aborts. Otherwise, the address of the corresponding Persistent Type on the database is obtained from the Local Type and the type pointer in the object header is overwritten with this value. The type is added to the metadata of a database if it is new there; if the type is totally new to the persistent store, it is added to the METADB database as well.

5 The Security Subsystem

Security Policy

When designing a security system, there are several aspects to decide, including granularity, privileges, authorisation method, access right specification and the points at which control is enforced. This section discusses the choices made in our experiment.

Granularity

In most database systems, security is a property of data type. In the extended CPOMS, one can protect not only whole databases or entire types, but also individual objects. The objects that can be protected independently are the primary objects. Security can be enforced at the component level, and read- and write-privileges are distinguished. We have not distinguished between insertion, update and deletion rights because these are not explicit operations in a persistent environment.

Authorisation Method

Three common authorisation methods are lock-and-key schemes, classification mechanisms and the use of privilege lists (see e.g. [8]). The lock-and-key approach typically associates a secret string (the lock) with a protected item; access to the affected data is then restricted to those who are able to supply this string (the key). The last of these methods associates with a type (or object) a list of userid–authorisation pairs. Only users appearing in this list may access the type (or object); the authorisation part indicates which operations that user is entitled to perform on instances of that type (or on that single object). Classification methods are not considered further since they are too inflexible for most enterprises: each user and each object is assigned a security level, and users may only access items at their level or below. For example, if the levels are ordered Unprotected, Confidential, Secret and Top-Secret then a user rated "Secret" can work with items classified Secret, Confidential or Unprotected, but not Top-Secret ones. One difference between the remaining two methods is that privilege lists associate rights with users; but with lock-and-key systems, authorisation is a property of a program or procedure. It is more logical that security clearance should be determined by the task being performed, rather than according to the person executing this task. To restrict the users who may run a program, should this be necessary, is a separate issue which can be handled by the program itself. Another advantage of the lock-and-key approach is that it requires less space on the store for recording authorisations.

Since database-related specifications are out of place in a persistent environment, a problem arises as to how security clearance can be established. Privilege list authorisation takes the form of statements like "Grant Read Access on StudentType to Bloggs" or "Grant Write Access on WinnerObject to Bloggs". Such security-related statements are clearly database-oriented and violate the persistence principle. Privilege list schemes thus require a separate utility for granting access rights, and newly persistent types and objects are completely inaccessible to all but their creator until this utility is run. On the other hand, lock-and-key schemes are easily and neatly incorporated within programs by using passwords as parameters to Lookup, Enter and type declarations. Instead of statements like "Readkey for StudentType is XYZ", passwords can be given in declarations e.g. "Type StudentType (XYZ) = ... " or as parameters e.g. Lookup(WinnerObject, MyDB, "XXY"). A consequence of this approach is that type protection is then dependant on scope. This raises a problem if the language has dynamic typing or structural equivalence: an object can be accessible in different scopes which have different security clearance for that type. This scoping problem does not apply to privilege list systems simply because declarations are made in separate utilities rather than within application programs.

The lock-and-key method was selected for the extended CPOMS. The one difficulty it presents (i.e. scoping) is limited to languages with structural equivalence or dynamic typing, and a policy was devised to cope with this problem. Protection is always determined by the scope in which data is retrieved from the store. An advantage of this policy is that library routines can force the calling program to fetch an object; its security clearance then reflects the privileges afforded the caller, and its use within the routine will be restricted accordingly. Authorisation to insert a new object on the store is determined by the privileges held at the time the item was created. Thus new data can be Committed in any scope; even if its type is no longer in scope at all! In any case, to enforce security according to current scope is far too costly – it means re-establishing authorisation on every access to every value, persistent or transient.

Passwords are given with type declarations and, for primaries, they are supplied as parameters of Lookup and Enter. Keys may be given as variables instead of literals, so security clearance can be tailored to the program's end-user. Complex objects can comprise any number of protected components. These keys are specified as a set of strings, since a set parameter is a convenient way of passing an indeterminate number of values.

Access Privileges

To access an object requires authorisation to use both its type and its "access path". For a primary, access path means the right to Lookup that object; for an object reached via a pointer dereference it means authorisation to read that pointer. As a result, having clearance for a type, its instances can be accessed via any authorised path, irrespective of rights to other paths. If some component of an object (i.e. a field or element) is read-protected, then neither this value nor the object as a whole may be referenced; only its other components are accessible. This prevents it being copied to a variable where that component is unprotected. Similarly, if unauthorised to write to a component, other components may be assigned new values – but not that component, nor the object in its entirety. In particular, write-protection of a pointer component implies that the referenced object cannot be destroyed – that is, the pointer cannot be altered to NIL to make the target object unreachable.

In the case of a read-only pointer component, the protection does not prevent changing the contents of the referenced object (see Figure 1). For example, suppose Winner is a write-protected pointer to a Person instance. It cannot be changed without the necessary write-authorisation to Winner; but the contents of that Person (e.g. Name, Address, etc.) can be updated. However, the change will cause all relationships of the affected object to refer to the new individual, and hence is unlikely to go undetected. In a relational system, such a Winner attribute would contain the key of some Person tuple. Changing the Winner would involve associating this identifier with a different person. If user-visible keys serve as identifiers the change is not possible, since e.g. a person's Social Security Number cannot be changed. Otherwise, the situation is analogous to that of the extended CPOMS.

Enforcement Points

It is not possible to enforce security at compile-time because persistent objects cannot be distinguished from transient ones; and the points at which objects move between disk and memory cannot be determined until runtime. Since any persistent object can contain a pointer to a primary, it is also impossible to statically enforce security on primaries. Thus protection of persistent data takes place at runtime.

A read-violation occurs as soon as an illegal attempt is made to access a persistent value. Write permission could similarly be checked on every input, assignment statement and parameter pass. The only benefit in immediately checking write permission is that it simplifies debugging. However this is very costly, particularly as it will be done for transient objects as well and will be repeated every time a value is changed. There is also the possibility that a program may be terminated because of an invalid write when in fact it had not intended to Commit the transaction. As it is only on Commit that newly persistent objects become apparent, database insertions cannot be checked for write permission until this stage. Therefore it is more consistent to detect all write-violations on Commit. This is a logical point at which to do so, and is a far more efficient approach.

Representing Security Information

Before discussing security enforcement, the storage of access privileges is outlined in this section. We first describe how Local Types record security clearance, and then identify the access rights attached to individual data items.

The type descriptions in the object code include the security strings associated with a type. With each Local Type, the system initially keeps pointers to these passwords. As with type checking, a Lookup causes security clearance to be established for the types involved, disposing of password strings and recording authorisation in security flags associated with the Local Type. To store authorisations for individual components of a structured type, $2N$ security bits are stored with the Local Type (where N is the size of its instances in words). There is one read- and one write-bit for each word to cater for nested structures. To cover the case where a component is itself complex, with individually protected components, access privileges are kept at the word level rather than the component level.

To attach access rights to data, four bits of the object header are used: the read-only, partly-readable, partly-writable and is-primary flags. Partly-readable/writable applies only to complex data with protected components. Whenever a persistent object is fetched from disk, these flags are set according to the clearance associated with its Local Type. An extra N authorisation bits are attached to structured objects; where a bit is set if the corresponding word is inaccessible. Although they are associated with individual objects rather than (Local) types, these flags require a negligible amount of extra memory in return for rapid checking of read permission, which must be performed on every component reference. Write-protection of fields is not carried with data in this way: the information is obtained by following the object's type pointer to the associated Local Type. The deferring of write-checks until Commit means this extra access to the Local Type does not occur too frequently.

Security Enforcement

This section illustrates how the data structures above are utilised to ensure that objects imported from the store are properly protected, and that new items are not written to a database unless a program has the right to do so.

Determining Privileges

Authorisation to manipulate database data cannot be determined at compile time because it is not known statically which pointer dereferences cause objects to be fetched from the persistent store. Therefore every database retrieval needs to fetch not only the desired data object but also its type, and record its security clearance accordingly. The alternative is to include type number as part of a pointer dereference instruction; this was not implemented because only a small minority of pointer traversals will actually require database access.

With lock-and-key protection, the "locks" associated with the database type need to be checked against the "keys" provided in its program declaration. The security bits of a Local Type are set during type checking: deep checking

ensures the correct protection of embedded structures. On a database retrieval, the type pointer in the object's header identifies its Persistent Type. A map from Persistent Type address to type name is kept in memory. The type name is used to identify the corresponding Local Type, and security flags in the data header are then set according to the security bits attached to this Local Type.

Primary protection is applied whenever an object retrieved from the persistent store has the is-primary bit set (not only on a Lookup). The Offset instruction, which advances to some field of a structure, checks component authorisation by examining the component-read-flags attached to the object. This instruction includes a parameter giving the size of the value being accessed, so that the corresponding number of bits may be checked.

The Effect of Program Scope

If a language uses structural equivalence or dynamic typing, a database object fetched onto the heap during a particular procedure call C, can be accessed outside this scope as well. It will then be associated with a different type declaration, and so its protection may differ in different scopes. As explained earlier, the protection of an object is determined by the scope in which the program retrieved it from the store: the data can only be used accordingly in other scopes. Since security clearance is a property of objects on the program heap, not of databases, this information is maintained by the runtime system (the persistent store is not affected) in a manner described below.

To keep track of scope changes, we can either maintain a single set of current types which is updated on every function call and exit, or we can keep a list of sets (similar to the way scoping is done with a static chain). Because of extensive support for indexing [3], the former approach has been used; indexing a single collection is faster than searching a chain. A CurrentMeta index on typename points to types currently in scope. Function calls adjust CurrentMeta by deleting types no longer in scope and inserting new ones. When a new type is encountered on scope entry, any type in CurrentMeta with the same name must be replaced; the situation is reversed when the routine terminates. The activation record of each routine includes two sets, New-types and Old-types. On routine exit, CurrentMeta has all New-types removed and all Old-types restored.

Write Protection

To provide atomicity, the Commit routine has two phases. The first validates all database changes made by the transaction and writes copies of their original state to a Before Looks File. The second phase writes new data to the store. The first phase enforces all write protection: newly persistent data and changed database items must be validated. Before Committing a new object to the store, its type pointer is always traversed to obtain the appropriate Local Type. From this the program's right to create such data is determined.

With updated items, the first step is to consult the object's read-only flag, and abort immediately if it is set. If the part-writable flag is set, the read-only fields are identified by examining the associated Local Type. Component write-protection can be applied at little extra cost by comparing current values against original values (obtained for the Before Look) to detect alteration of read-only fields.

6 Conclusion

We have shown how to maintain information on the types in use on a persistent store, without limiting the persistent languages it is able to support. We illustrated how this information can be employed to type check database usage and provide for flexible metadata querying. The implementation of a security system was described and its use by languages with dynamic typing and structural equivalence was outlined. We conclude by discussing the benefits and disadvantages offered by the resulting system, and identify some areas requiring futher work.

Evaluation

The introduction of data protection requires negligible space; but extra local heap dereferences on persistent store retrievals, to check authorisation, are unavoidable.

The CPOMS permits one writer or many readers to access a database concurrently. It is for this reason that the types in use on a database are described by Persistent Type objects on that same database. Several databases can have existing persistent types added to their metadata concurrently, as this requires at worst only read access to METADB. However, once a program introduces a new type to the persistent store, no other run-unit can simultaneously copy a type to a new database, since METADB is open for writing. Metadata handling is thus complicated because inter-object references are not limited by the unit of locking. We conclude that until concurrency is properly supported, inter-database references should not be allowed.

The use of pointers to reference related objects, rather than keys (as in relational systems for example), permits a more flexible security system. There can be multiple paths to an object, enabling a program authorised to traverse any one of these paths to use the data concerned. In addition, confidential parts of a complex object can be stored separately, and kept private by read-protecting the pointer to these values. The existence of named primary structures enables these objects to have individual protection. They may still be referenced by other persistent objects, so data structuring is unaffected by this. It is more efficient to check for write-violations when a database is updated, rather than on every instruction that changes some value. The Commit statement that demarkates the end of a transaction means that all database changes made by that transaction can be validated together at a logical point in the program execution.

In the original CPOMS all the components of an object had to be atomic. This restriction has been removed by allowing for nested structures [3], in order to support languages with complex data types. Such complex objects were seen to require more space for read-flags and to be more cumbersome to protect (eg the Offset instruction re-

quires the size of the value to subsequently be extracted). The bulk data type we introduced proved useful for non-procedural access to metadata, and facilitated the handling of cross-references, variants, local metadata and persistent types.

Compilers for persistent languages should themselves be written in a persistent language so that they are able to access metadata and type check database structures. This would remove the burden of type checking from the runtime system. Security cannot be enforced statically because persistent store retrievals are not distinguishable in program code. Data flow analysis could be used to detect pointer dereferences involving disk addresses and to flag the symbol table entry for the corresponding object according to its security clearance. However this analysis is limited by end-user inputs and database values, which are not known statically. Hence the runtime system would still need to authorise accesses; the only benefit of static checking would be to raise some errors at compile time – but execution time would not improve.

Future Work

Several extensions to the existing metadata management system are possible. If part of a program type definition is incompatible with that of the store, the security subsystem could be used to treat the object as partly-readable, instead of terminating execution. We suggest that the entire item be write-protected, as it seems unreasonable to allow a program to alter an object if its type is not fully known. More sophisticated protection of set types should be possible, such as access predicate and statistical control. Furthermore, persistent functions create opportunities for protection of behavioural information, which has not been investigated. Instead of dynamically keeping track of scope for type-related security, it appears in retrospect that it might have been better to do this at compile time, by making the pointer dereference instruction include the type number of the target object. Although this would be used only infrequently – when a database retrieval occurred – this overhead may be justified by the resulting simplication of function entry and exit.

References

1. A Albano, L Cardelli, and R Orsini. 'Galileo: A strongly-typed, interactive conceptual language'. *ACM Trans. on Database Systems*, **10**(2):230–260, (1985).
2. M Atkinson, P Bailey, K Chisholm, W Cockshott, and R Morrison. 'An approach to persistent programming'. *Computer Journal*, **26**(4), (1983).
3. S Berman. *P-Pascal: A Data-Oriented Persistent Programming Language*. PhD thesis, University of Cape Town, 1991.
4. A Brown and W Cockshott. 'The CPOMS persistent object management system'. Persistent Programming Research Report PPRR 13, Universities of Glasgow and St Andrews, (1985).
5. R Connor, A Brown, Q Cutts, A Dearle, R Morrison, and J Rosenberg. 'Type equivalence checking in persistent object systems'. In A Dearle and G Shaw, eds., *Proc. 4th Int. Workshop on Persistent Object Systems*, pp. 151–164, (1990).
6. A Davie and D McNally. 'STAPLE User's Manual'. Research Report CS/90/12, Department of Computational Science, University of St. Andrews, (1990).
7. A Dearle. *On the Construction of Persistent Programming Environments*. PhD thesis, Department of Computational Science, University of St Andrews, 1988.
8. E Fernandez, R Summers, and C Wood. *Database Security and Integrity*. Addison-Wesley, Reading, Mass., 1981.
9. P Gray, D Moffat, and J Du Boulay. 'Persistent Prolog: A searching storage manager for Prolog'. In M Atkinson, O Buneman, and R Morrison, eds., *Data Types and Persistence*, pp. 353–368. Springer-Verlag, (1988).
10. R Morrison, A Brown, R Connor, and A Dearle. 'The Napier88 reference manual'. Persistent Programming Research Report PPRR 77, Universities of Glasgow and St Andrews, (1989).
11. N Perry. 'Hope+'. report IC/FER/LANG/2.5.1/7I, Imperial College.
12. Persistent Programming Research Report PPRR 12, Universities of Glasgow and St Andrews. *PS-Algol Reference Manual (4th ed.)*, 1988.

Notes for Contributors

The prime purpose of the journal is to publish original research papers in the fields of Computer Science and Information Systems, as well as shorter technical research papers. However, non-refereed review and exploratory articles of interest to the journal's readers will be considered for publication under sections marked as Communications or Viewpoints. While English is the preferred language of the journal, papers in Afrikaans will also be accepted. Typed manuscripts for review should be submitted in triplicate to the editor.

Form of Manuscript

Manuscripts for *review* should be prepared according to the following guidelines.

- Use wide margins and $1\frac{1}{2}$ or double spacing.
- The first page should include:
  - title (as brief as possible);
  - author's initials and surname;
  - author's affiliation and address;
  - an abstract of less than 200 words;
  - an appropriate keyword list;
  - a list of relevant Computing Review Categories.
- Tables and figures should be numbered and titled. Figures should be submitted as original line drawings/printouts, and not photocopies.
- References should be listed at the end of the text in alphabetic order of the (first) author's surname, and should be cited in the text in square brackets [1–3]. References should take the form shown at the end of these notes.

Manuscripts accepted for publication should comply with the above guidelines (except for the spacing requirements), and may be provided in one of the following formats (listed in order of preference):

1. As (a) LaTeX file(s), either on a diskette, or via e-mail/ftp – a LaTeX style file is available from the production editor;
2. As an ASCII file accompanied by a hard-copy showing formatting intentions:
   - Tables and figures should be on separate sheets of paper, clearly numbered on the back and ready for cutting and pasting. Figure titles should appear in the text where the figures are to be placed.
   - Mathematical and other symbols may be either handwritten or typed. Greek letters and unusual symbols should be identified in the margin, if they are not clear in the text.

   Further instructions on how to reduce page charges can be obtained from the production editor.
3. In camera-ready format – a detailed page specification is available from the production editor;
4. In a typed form, suitable for scanning.

Charges

Charges per final page will be levied on papers accepted for publication. They will be scaled to reflect scanning, typesetting, reproduction and other costs. Currently, the minimum rate is R30-00 per final page for LaTeX or camera-ready contributions and the maximum is R120-00 per page for contributions in typed format (charges include VAT).

These charges may be waived upon request of the author and at the discretion of the editor.

Proofs

Proofs of accepted papers in categories 2 and 4 above will be sent to the author to ensure that typesetting is correct, and not for addition of new material or major amendments to the text. Corrected proofs should be returned to the production editor within three days.

Note that, in the case of camera-ready submissions, it is the author's responsibility to ensure that such submissions are error-free. However, the editor may recommend minor typesetting changes to be made before publication.

Letters and Communications

Letters to the editor are welcomed. They should be signed, and should be limited to less than about 500 words.

Announcements and communications of interest to the readership will be considered for publication in a separate section of the journal. Communications may also reflect minor research contributions. However, such communications will not be refereed and will not be deemed as fully-fledged publications for state subsidy purposes.

Book reviews

Contributions in this regard will be welcomed. Views and opinions expressed in such reviews should, however, be regarded as those of the reviewer alone.

Advertisement

Placement of advertisements at R1000-00 per full page per issue and R500-00 per half page per issue will be considered. These charges exclude specialized production costs which will be borne by the advertiser. Enquiries should be directed to the editor.

References

1. E Ashcroft and Z Manna. 'The translation of 'goto' programs to 'while' programs'. In *Proceedings of IFIP Congress 71*, pp. 250–255, Amsterdam, (1972). North-Holland.
2. C Bohm and G Jacopini. 'Flow diagrams, turing machines and languages with only two formation rules'. *Communications of the ACM*, **9**:366–371, (1966).
3. S Ginsburg. *Mathematical theory of context free languages*. McGraw Hill, New York, 1966.

South African Computer Journal

Number 11, May 1994
ISSN 1015-7999

Suid-Afrikaanse Rekenaar-tydskrif

Nommer 11, Mei 1994
ISSN 1015-7999

Contents

GUEST CONTRIBUTIONS

Ideologies of Information Systems and Technology
LD Introna 1

What is Information Systems?
TD Crossman 7

RESEARCH ARTICLES

Intelligent Production Scheduling: A Survey of Current Techniques and An Application in The Footwear Industry
V Ram 11

Effect of System and Team Size on 4GL Software Development Productivity
GR Finnie and GE Wittig 18

EDI in South Africa: An Assessment of the Costs and Benefits
G Harrington 26

Metadata and Security Management in a Persistent Store
S Berman 39

Markovian Analysis of DQDB MAC Protocol
F Bause, P Kritzinger and M Sczittnick 47

TECHNICAL NOTE

An evaluation of substring algorithms that determine similarity between surnames
GdeV de Kock and C du Plessis 58

COMMUNICATIONS AND REPORTS

Ensuring Successful IT Utilisation in Developing Countries
BR Gardner 63

Information Technology Training in Organisations: A Replication
R Roets 68

The Object-Oriented Paradigm: Uncertainties and Insecurities
SR Schach 77

A Survey of Information Authentication Techniques
WB Smuts 84

Parallel Execution Strategies for Conventional Logic Programs: A Review
PEN Lutu 91

The FRD Special Programme on Collaborative Software Research and Development: Draft Call for Proposals . . . 99

Book review 102