Conch: Experimenting with Enhanced Name Management for Persistent Object Systems

Alan Kaplan
Jack C. Wileden
{kaplan,wileden}@cs.umass.edu

CMPSCI Technical Report 94–70
September 1994

Computer Science Department
University of Massachusetts
Amherst, Massachusetts 01003

Appeared in Proceedings of the Sixth International Workshop on Persistent Object Systems Tarascon, France, September, 1994

This material is based upon work sponsored by the Advanced Research Projects Agency under grant MDA972-91-J-1009 and by Texas Instruments under grant SRA-2837024. The content does not necessarily reflect the position or policy of the U.S. Government or Texas Instruments and no official endorsement should be inferred.

Abstract

The name management capabilities currently provided by most existing persistent object systems (POSs) are rather limited. In particular, existing POSs tend to lack powerful and general mechanisms for forming, manipulating, controlling and reasoning about contexts. As a result, these POSs offer only weak or awkward support for large-scale data storage, multi-user computing, code reuse, interoperation of independently developed object stores and other similarly important classes of applications. As part of our work on improved name management for convergent computing systems, we have developed a framework for uniform treatment of the context and interface control facets of name management. In this paper we describe a realization of that framework, in the form of a shell-style user interface to a POS, that we are using to experiment both with the framework itself and with enhanced context control capabilities for POSs.

1 Introduction

A *convergent computing system* is any system in which two or more distinct computing domains or paradigms are combined into what is intended to be a synergistic whole. Naturally, various issues and complications arise in constructing such systems.

We are particularly interested in name management for convergent computing systems. By *name management* we mean the way in which a computing system allows names to be established for objects, permits objects to be accessed using names, and controls the availability and meaning of names at any point in time. Problems that arise in convergent computing systems often result from complexities or shortcomings in the name management mechanisms of their underlying components or from incompatibilities among those mechanisms. An important goal of our research is improved mechanisms for name management in convergent computing systems.

Persistent object systems (POSs) are a particularly interesting class of convergent computing systems. POSs promise to break down barriers between capabilities traditionally found in programming languages, database systems and operating systems and to combine all (or at least most) of the best features of each. In recent years, POS research has led to significant progress at overcoming barriers and synergistically combining capabilities related to various aspects of such systems, including type models (e.g., [1]), persistence mechanisms (e.g., [2]), optimization techniques (e.g., [3]) and concurrency control mechanisms (e.g., [4]).

A notable exception to this trend is the limited attention that has been given to name management for POSs. There have been a few instances of POSs offering some improvements in name management (e.g., [5, 6, 7, 8, 9]). By and large, however, the relatively weak name management mechanisms found in the ancestors of POSs, i.e., programming languages, database systems and operating systems, have tended to endure in POSs, being neither improved nor even effectively integrated. Without better name management, POSs will likely prove cumbersome to use and prone to error, will provide inadequate support for large-scale data storage, multi-user computing, code reuse and interoperation of independently developed object stores, and will therefore fail to realize their potential for beneficial employment in a variety of important application areas.

We believe that our work on name management for convergent computing systems has particular relevance for POSs. In fact, our interest in the general topic of name management grew out of earlier work on PGraphite [2, 10] and R&R [11], two complementary prototype systems that together implemented a rudimentary persistent object capability as an extension to Ada. Experimental use of these systems highlighted a number of name management problems, arising primarily from the need to simultaneously manage names of both transient and persistent objects. Among these, the two most fundamental were the problems of:

Uniformity: the lack of integration between names for transient (programming language-internal) and persistent (programming language-external) data objects, and

Context Control: the incommensurate, and inadequate, mechanisms for controlling exactly which names (of both transient and persistent objects) are available for use (at any given point) within a program.

Having observed these problems while experimenting with PGraphite and R&R, we soon discovered that similar problems existed in most other POSs[1] and, more generally, in most convergent computing systems. Both problems are particularly prevalent in "persistent[X]" POSs, those created by extending some existing programming language with support for persistence (and possibly some additional database and operating systems capabilities), but otherwise leaving the underlying language unchanged. The other major class of POSs, namely the *de novo* POSs, i.e., those that have been created by defining new languages specifically designed to provide persistence (and possibly some additional database and operating system capabilities), have sometimes avoided the uniformity problem but generally not the context control problem.

Initially, our research on name management for convergent computing systems centered on defining a taxonomy of problems and potential solutions, and on developing and experimenting with some simple, but uniform, naming mechanisms [13], as a basis for addressing the uniformity problem. More recently, we have focused on context control issues. In particular, we have defined a framework, called PICCOLO, for describing context control problems and mechanisms in convergent computing systems [14] and begun to experiment with its use.

In this paper, we describe an ongoing experiment in improving the context control facet of name management in POSs. We begin by sketching some representative context control problems arising in POSs and outlining the PICCOLO framework. We then describe a prototype realization of the framework in the form of CONCH, a shell-style user interface to a persistent[C++] POS, and show how it can be used to address the representative problems. The paper concludes with an assessment of the current status of the experiment, a discussion

[1]In fact, in their list of requirements for persistence [12], Atkinson and Buneman appear to refer to both of these problems (requirements (4) and (9)).

of related work, and a summary of ongoing and future efforts aimed at further improvements in name management for POSs.

2 Context Control Problems in POSs

As noted in the previous section, POSs tend to provide inadequate approaches to name management, especially with respect to controlling context. While orthogonal persistence should obviate the need for traditional persistence mechanisms, such as file systems and databases (at least from the programmer's perspective), users of POSs are typically still faced with using primitive name management mechanisms, based on those found in operating systems or database management systems, for controlling the meanings of names for persistent objects. As a result, existing POSs tend to lack powerful and general mechanisms for forming, manipulating, controlling and reasoning about contexts.

As an illustration of the shortcomings of context control for POSs, consider the situation facing a hypothetical application developer wishing to build a system that accesses various electronic mail-related objects from a persistent store. Some name management problems that the developer must resolve include:

1. The developer wants to be able to organize objects and their names in the persistent store into logical, meaningful collections.

2. The developer wants to be able to flexibly form contexts giving particular meanings to names used in the application. The resulting contexts may not correspond exactly to any single collection in the set of collections from (1) above. Furthermore, the developer wants to be able to specify different contexts for the application without necessarily having to re-compile (or possibly even re-link) the application.

3. The developer wants to be able to reason about how contexts are formed and used by the application. For example, given an application, the developer should be able to determine what names the application refers to, whether objects corresponding to those names exist, whether the objects can be accessed and/or modified by others, and whether the names used have unambiguous meanings.

As an example of the first point, our hypothetical developer might wish to impose the conceptual organization depicted in Figure 1 on a relevant subset of objects in the persistent store. In this figure, persistent objects are represented by both ellipses and rectangles. Ellipses denote collections of named objects, where names are attached to arcs emanating from an ellipse to a corresponding object, represented by either a rectangle or another ellipse. The only exception to this is the name ROOT, which identifies the root collection of the structure and is not contained in any other collection. For the purposes of this example, assume that the objects named **mailbox** contain electronic mail messages, that the objects named **broadcast** and **lab** represent distribution lists, and that the collections named **ABC** and **XYZ**

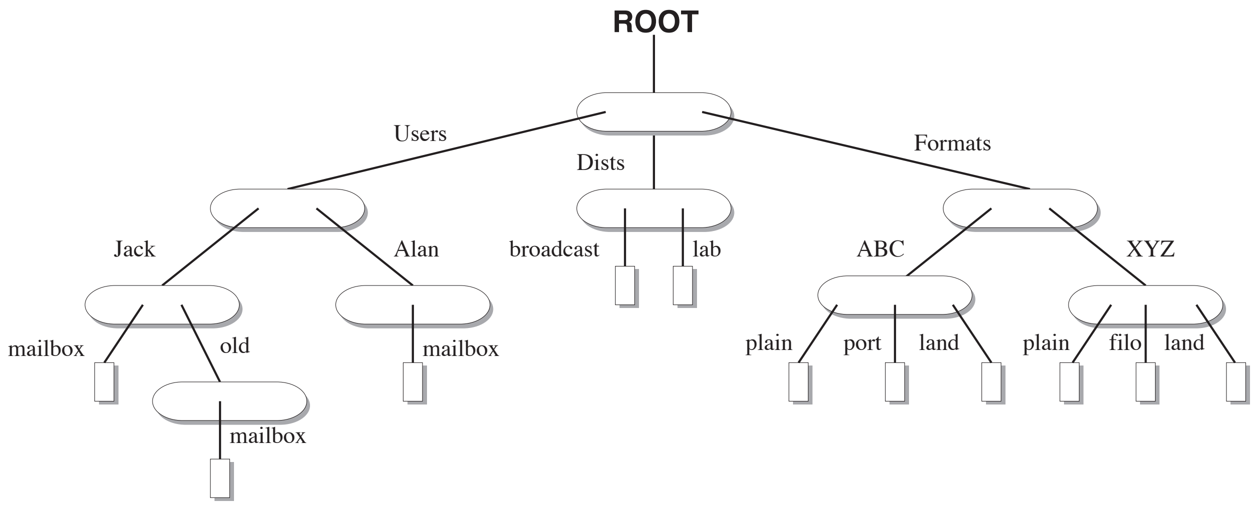

Figure 1: **Conceptual Organization of the Persistent Store**

contain a variety of message-printing formats provided by two different vendors. Note that these collections happen to have common names for some objects, a problematic, though not uncommon, situation in the commercial arena [15]. In addition, assume that multiple users and/or applications may be accessing and modifying the persistent store.

Given this logical organization of the persistent store, our developer's goal is an application that can manipulate a user's mailbox (e.g., compose, send, read, print and archive messages), making use of all the distribution lists and some subset of the message formats. Correct execution of this application will depend upon the availability of an appropriate context, connecting names used by the application with specific objects in the persistent store. Thus the application developer, or user, needs some means of controlling context formation. For example, the developer should be able to specify what names are needed in the desired context, what objects are bound to those names, at what point in the application's lifetime name-object associations are formed, and whether names are associated with existing objects or copies of objects in the persistent store.[2] For instance, one possible set of context requirements (and clearly not the only set) for this hypothetical application might include the following:

1. It should be possible to use the application with *any* user's mailbox without either having to re-compile (or re-link) the application or having to interactively interrogate the user for the location of the mailbox. This means that the context for the application must include an association between the name **mailbox** and a specific object in the persistent

[2]For example, linkers in traditional programming environments typically create copies of object code modules from libraries and make links to the copies, rather than to the library modules themselves, when constructing an executable program.

store that is formed at the time when the mailbox object is actually accessed in the application (as opposed to at compile-time, link-time or load-time).

2. The application will need to access both the **broadcast** and the **lab** distribution lists, but the contents of these lists should be constant during any single execution of the application. Hence the context associating specific (and subsequently immutable) objects with those names should be formed at the time the application is loaded into the run-time system.

3. Finally, our developer is satisfied with the current versions of the message-printing formats named **plain** and **port** contained in the **ABC** collection, and the formats named **filo** and **land** contained in the **XYZ** collection. So, it would be appropriate to form a context associating those names with copies of those specific objects at compile-time. This means that, once compiled, the application will be shielded from any changes to the objects through the persistent store (e.g., name changes, message format changes).

One goal, then, for a context control mechanism is to give developers and users explicit control over context specification for persistent objects [16]. Another important goal is to give subsequent developers or maintainers access to this information in a uniform manner. In other words, instead of having the tedious and error-prone task of manually analyzing source code and/or configuration scripts, developers and maintainers should be able to query an object directly for its relevant context formation information. Similarly, both humans and automated tools should be able to reason about this context information, thus facilitating the detection of potential name management-related inconsistencies and errors.

Most existing POSs support few, if any, of these capabilities. In the next two sections, we outline a framework and a prototype realization of that framework with the potential to support all these capabilities and more in a uniform and powerful way.

3 Piccolo

As a step toward enhanced name management for convergent computing systems, and specifically for POSs, we wish to provide powerful and general mechanisms for forming, manipulating, controlling and reasoning about contexts. By the very nature of a convergent computing system, such as a POS, the context mechanism we seek must be uniformly applicable across a broad range of computing paradigms. In the case of POSs, in particular, it should be applicable to context-related aspects of programming languages, databases and operating systems, and should apply to both persistent and transient objects.

We believe that this goal is best achieved by basing the mechanism on an abstract model or framework. We have therefore developed the PICCOLO model, a framework for Precise Interface and Context Control in Object Libraries and Objects. PICCOLO is inspired in part by the PIC framework for precise interface control in programming languages [17]. In the remainder of this section we briefly outline the PICCOLO framework. The next section describes one realization of the framework that we are currently using to experiment both with the framework itself and with enhanced context control capabilities for POSs.

The PICCOLO framework is based on the following set of fundamental name management concepts:

object: An item of interest in a given setting.

name: An identifier used to reference, access or manipulate an object.

binding: In its simplest, most basic form a (name,object) pair. The availability of binding (n, o) makes it possible to use name n to reference, access or manipulate object o. Bindings may also include additional information, such as type and mutability information (as, for example, in Napier [18]).

binding space: A set of bindings that serves as a collection of definitions for names.

context: A set of bindings that is available for use in referencing, accessing or manipulating objects. A context may consist of, or be formed from, one or more binding spaces, or parts of binding spaces, or other contexts. PICCOLO's explicit distinction between binding spaces, which are primarily a means for organizing collections of bindings, typically for user convenience, and contexts, which a system uses in interpreting names during its operation, significantly facilitates modeling of many name management approaches.

closure: In its simplest, most basic form an (object, context) pair. Given the existence of the closure (o, c), it is possible for object o to use context c in referencing, accessing or manipulating other objects.

resolution: The action of returning an object when given a name and a context.

Given these fundamental concepts, the PICCOLO framework defines specific name management approaches and mechanisms using the following two kinds of components:

context formation template (CFT): A collection of directives governing the formation of contexts. In a typical approach to name management we can further distinguish between two kinds of CFTs:

Requestor: A CFT Requestor (or CFTR) describes the context requirements for an object. In particular, it specifies a collection of names referenced by the object. It can additionally include directives indicating such things as preferred sources of definitions (i.e., binding spaces) for the referenced names and preferred times (i.e., epochs) for context formation or modification steps.

Provider: A CFT Provider (or CFTP) describes the context definitions potentially available from an object. Analogously to CFTRs, it can additionally specify such things as preferred targets (i.e., other objects) for these definitions and preferred times for context formation or modification steps.

context formation process (CFP): A procedure that produces a context from an initial context and one or more CFTs.

As suggested by the above definitions of CFTR and CFTP, the PICCOLO framework provides a means for explicitly representing distinct times at which context manipulation activities may occur:

epoch: An epoch denotes a particular time period during which context manipulation activities may take place. The set of all epochs is described by an enumeration such as $E = \{e_1, e_2, \ldots\}$.

Applying the PICCOLO framework involves defining a specific set of epochs appropriate to the host system's context formation and manipulation needs, specific kinds of directives that can appear in CFTs and one or more specific CFPs. It also implies a suitable generalization of the concept of closure, such as the following:

closure $\equiv (o, (C_i, \{CFT*, CFP*\}))$ where C_i represents an initial context, $CFT*$ represents a (possibly empty) set of context formation templates for directing the incremental formation of future contexts, and $CFP*$ represents a corresponding set of context formation processes that will be directed by those CFTs during incremental formation of future contexts.

One example application of the framework is the experimental context control shell, CONCH, which is described in Section 4.

We envision two major categories of use for the PICCOLO framework. First, it can serve as a basis for analyzing context control mechanisms and problems, especially in convergent computing systems but in more traditional settings as well. Careful description of a particular context control mechanism in terms of PICCOLO's CFT and CFP constructs permits rigorous reasoning about and comparison of various existing, proposed or possible approaches to context control.

Second, the PICCOLO framework can serve as a foundation for defining and implementing context control mechanisms. We are, of course, especially interested in applying it to the definition and implementation of such mechanisms for convergent computing systems, such as POSs. By defining a mechanism using the framework, we can avoid *ad hoc* solutions, reason about properties of a mechanism before implementing it, and achieve uniformity in the functioning of the mechanism. This latter feature is particularly important in the setting of convergent computing systems, where the mechanism may well have several different realizations corresponding to different aspects, or ancestral constituents, of the convergent system.

In the next section, we describe an experiment in which we have implemented a context control mechanism based on the PICCOLO framework in the form of a shell-style user interface to a POS. While there are several other forms in which this mechanism might, and probably should, be realized in a POS, we have found this one convenient for experimentation and also rather easy and natural to use for certain kinds of context control operations. We will provide further assessment of this experiment in the paper's final section.

4 Conch: A Context Control Shell

The idea of a "shell" acting as an intermediary between applications and the underlying persistent store is by no means a new one. In traditional systems, such as UNIX,[3] a variety of shells have been implemented to provide an interface between programs and the UNIX file system. Similarly, relational database systems often provide an interactive SQL that allows users to interact with the database. In recent years, graphical browsers for POSs have begun to emerge [19, 20]. Neither previously existing shells nor graphical browsers, however, provide sufficiently powerful and general context control capabilities for use with convergent computing systems like POSs. In this section, we report on CONCH, a prototype CONtext-Controlling sHell for POSs that we have implemented as a user interface for Open OODB, a persistent[C++] POS [21]. We then illustrate how CONCH can be applied to the scenario presented earlier in Section 2.

CONCH is a realization of the PICCOLO framework. It facilitates experimentation with that framework as well as with enhanced context control capabilities for POSs. More specifically, the present version of the CONCH prototype represents an experiment with a particular method of providing a particular set of name management capabilities and an attempt to unify these capabilities from a "shell" perspective.

In its plain, vanilla form, Open OODB[4] provides relatively limited support for name management for persistent objects. There exists only a single, flat space of names; names for persistent objects must always be resolved at run-time and always return a reference to an object (instead of, perhaps, a copy of that object); and reuse of a name always overrides the existing binding.

The CONCH prototype addresses many of these shortcomings and has resulted in improved support for name management in Open OODB. The current set of commands defined by the CONCH interface is listed in Table 1. (Since Open OODB ordinarily runs under UNIX, the CONCH command names and syntax are intentionally UNIX-like.) First, CONCH directly

[3]UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/OPEN corporation.

[4]Alpha Release 0.2

Command	Description
lbs	List contents of the *active* binding space
abs <name>	Binding space with name <name> becomes *active* binding space
cbs <name>	Create a new binding space
rtbs	ROOT binding space becomes active
rmb <name>	Remove binding for name <name>
bind <name1> <name2>	Bind name <name2> to object bound to name <name1>
cftr <name>	Create a CFTR with name <name>
cftp <name>	Create a CFTP with name <name>
cftq <name>	Query the CFT with name <name>
inclos <cft-name> <obj-name>	Insert CFT into object's closure
prclos <name>	Print contents of object's closure

Table 1: Conch Commands

supports the PICCOLO concepts of *name*, *object*, *binding* and *binding space*, thus, allowing persistent stores to be organized similarly to the one depicted in Figure 1. Furthermore, the commands **lbs**, **abs**, **cbs**, **rtbs**, **rmb**, and **bind** permit developers and users to easily traverse, modify and query the Open OODB persistent store. Similarly to other shell-style approaches, CONCH supports the notion of an *active* (or current) binding space. Note that the only way to create new bindings from the shell is through use of the **bind** command. In addition, CONCH provides a collection of C++ class interfaces that allow applications to interact with the persistent store in a similar manner.

Second, CONCH allows developers to precisely and explicitly express context formation requirements for applications accessing the persistent store by providing specific kinds of CFTRs, CFTPs, and CFPs. The commands **cftr**, **cftp** and **cftq** allow users to create and query CFTs. More specifically, a CFTR in CONCH consists of one or more *request* clauses, where a request clause consists of the following fields:

Names: the names whose corresponding bindings should be contained in the context.

Bind: an indication of whether the bindings should be to existing objects (REF) or to copies of objects (COPY).

Epoch: when the bindings in the context should be formed. (CONCH supports the epochs {COMPILE, LOAD, RUN}.)

Sources: the source binding spaces for the desired bindings.

Similarly to a CFTR, a CFTP in CONCH consists of one or more *provide* clauses, where a provide clause consists of analogous fields. Finally, CONCH defines a rudimentary CFP for forming

```
Context CFP (Context current,
             CFTR c,
             Epoch current)
begin
 Context new = current
 foreach clause in c do
   if current = clause.Epoch then
     foreach name in clause.Names do
       object = find (name) in
            clause.Sources
       if not found then
         raise ContextError
       if clause.Value = copied then
         new.Insert (name, object.copy)
       else
         new.Insert (name, object)

 return new
end CFP
```

Figure 2: Default CFP

```
request plain, port
copy
compile
sources Root.Formats.ABC;

request filo, land
copy
compile
sources Root.Formats.XYZ;

request broadcast, lab
copy
load
sources Root.Dists;

request mailbox
ref
run
sources *active*;
```

Figure 3: Example CFTR

contexts. In the current prototype, this CFP is automatically included in the closure for all objects, thus ensuring that all objects utilize the same method for context formation. The pseudo-code for the CFP employed by CONCH is shown in Figure 2. This simple CFP creates a new context by augmenting the contents of a given context with the bindings as directed by a CFTR and the current epoch. It also checks to ensure that the resulting context is valid and provides warning or error information in the event the checking should fail. Note that the CFP is not an explicit command in CONCH. Instead, in the prototype the CFP is invoked by various system-level tools, such as the compiler, linker and run-time system.

To illustrate the use of CONCH, we return to the scenario outlined in Section 2, in which a developer is faced with the problem of specifying and constructing a context that associates appropriate objects in the persistent store with names used in a mail system application. The developer's task is complicated by the fact that the different bindings in the resulting context must satisfy different requirements. With CONCH, one way to solve the problem in the scenario is to use the **cftr** and **inclos** commands to create the CFTR shown in Figure 3 and form a closure associating that CFTR and the application code.

The CFTR directs a context to be created as follows:

- At compile-time, the context must contain the names **plain**, **port**, **filo** and **land**, each bound to a copy of the appropriate persistent object. The preferred source for the objects named **plain** and **port** is the binding space named **ABC**, while the preferred source for the objects named **filo** and **land** is the binding space named **XYZ**.

- When the program is first loaded into the run-time system, the context must additionally contain the names **broadcast** and **lab** bound to copies of the appropriate persistent objects, whose preferred source is the **Dists** binding space.
- Each time the mailbox object is accessed, the context should be updated to contain a binding pairing the name **mailbox** with the then-current object associated with the name **mailbox** in the *active* binding space.

Once this CFTR has been created and inserted into the applications' closure, the next step is to compile, link and run the application. The entire process is depicted in Figure 4, where the arrows denote and identify epoch boundaries. The starting point is a closure consisting of

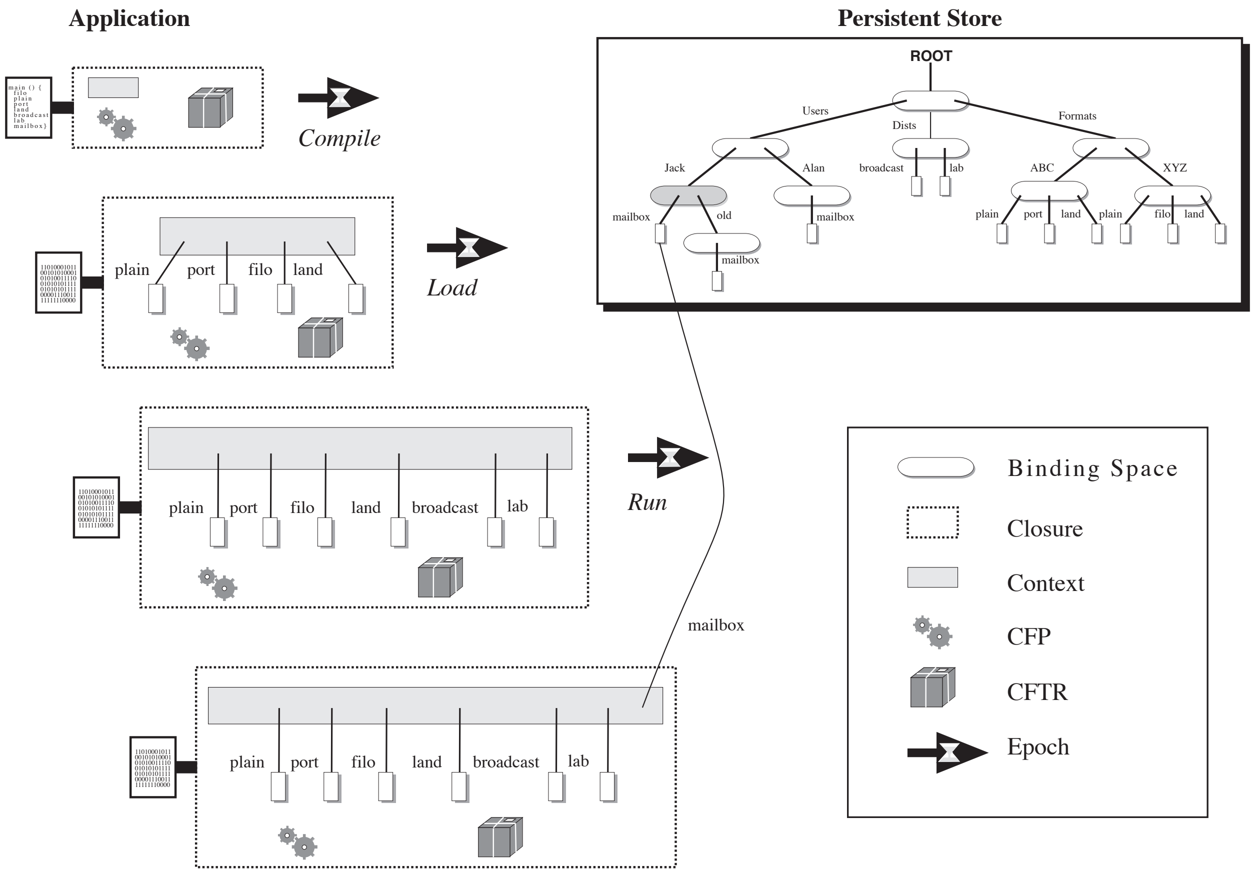

Figure 4: **Context Formation Process**

the application source code, an (initially) empty context, the CFP from Figure 2, and the CFTR from Figure 3. Invocation of the compiler initiates the **compile** epoch. The CFP, CFTR and context are retrieved from the source code's closure and a new closure consisting of a binary executable, a (partially formed) context, the CFP and the CFTR is produced. Then, using the **cbs** command, the user sets the *active* binding space to be **Jack**.[5] In Figure 4, the shaded ellipse denotes the *active* binding space. Next the program is loaded into the run-time system. This signals initiation of the **load** epoch, so the CFTR directs the context to be augmented appropriately and a correspondingly updated closure is formed. Finally, execution of the application begins, initiating the **run** epoch, and the mailbox object is accessed. As directed by the CFTR, the context is now augmented with a binding pairing the name **mailbox** with the object associated with the name **mailbox** in the *active* binding space, i.e., the object named **mailbox** in the binding space named **Jack**.

Note that the closure mechanism in CONCH associates a CFP and a CFTR as well as a context with each object. Although not illustrated in this simple example, maintaining this information allows application developers and maintainers to easily determine the context formation requirements for an application using the **prclos** command.

5 Summary

In this paper we have described CONCH, a prototype context-controlling shell implemented as a user interface to Open OODB, and briefly outlined the PICCOLO framework on which it is based. Initial experimentation with CONCH appears to confirm our expectation that a context control mechanism based on PICCOLO would provide more general, flexible and powerful context control than that available in existing POSs. In particular, the approach facilitates the sharing of context information among objects, while at the same time permitting individual objects to define their own context formation requirements. Furthermore, since context data is never discarded, application developers and maintainers can access this valuable information in a uniform and efficient manner. We therefore believe that the approach embodied in CONCH and PICCOLO can contribute to making POSs easier to use, less prone to error and better suited for use in a wide range of applications. Experimentation has, however, also pointed up a number of possible improvements and extensions that would make the approach even more beneficial.

We believe that existing approaches to context control in POSs are not as powerful and general as the approach described in this paper. While Napier [22, 18], for example, certainly allows for flexible organizations of persistent stores, developers must describe specific persistent store navigations in each application [23]. Moreover, Napier does not provide adequate means for querying objects regarding their contextual formation information. This can be problematic in Napier programs that create bindings in closures local to a procedure, since there is no means for determining whether other Napier programs may be able to access the

[5]In fact, this step could be taken either earlier or later than this, so long as it has occurred before the beginning of the **run** epoch.

objects in those bindings. The approach taken by Farkas et al. [9] ameliorates these problems to some degree, but it is unclear how well a graphical browsing paradigm is suited for managing contexts in large and complex applications. We view our work on the PICCOLO framework and the CONCH shell as an approach that could complement these and other approaches in existing and future POSs. Indeed, in the near future we hope to explore the incorporation of CONCH-like capabilities into a *de novo* POS such as Napier in order to further assess the generality and the efficacy of both the PICCOLO framework and the CONCH constructs.

As noted above, experiments with CONCH have suggested a number of improvements and extensions that could be made. Some of these involve additional capabilities for the shell, such as richer mechanisms for binding specification (e.g., allowing CFTs to specify local renaming of objects), for context specification (e.g., incorporating CFTPs into closure definitions or allowing CFTRs to define sources as combinations of binding spaces [13]) and for creating and querying CFPs. Others would make the approach more user-friendly; the existing, relatively low level and explicit, shell commands are suitable for the fine-grained control needed in experimentation, but much of the effort involved in creating and manipulating CFTs could, and probably should, be automated or hidden to benefit POS users. Still others involve enhancements to the underlying PICCOLO framework. For example, the relationship of context manipulation to such traditional mechanisms as versioning and transactions needs further exploration; either descriptions of such mechanisms in terms of PICCOLO need to be formulated or else the PICCOLO framework should be extended to account for them. We are already pursuing some of these directions, and we expect that continued refinement and further experimentation with both CONCH and PICCOLO will contribute to significant enhancements in name management capabilities for POSs and, more generally, for convergent computing systems.

REFERENCES

[1] R.C.H. Connor, A.L. Brown, Q.I. Cutts, A. Dearle, R. Morrison, and J. Rosenberg. Type equivalence checking in persistent object systems. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 154–167, Martha's Vineyard, MA, August 1990.

[2] J.C. Wileden, A.L. Wolf, C.D. Fisher, and P.L. Tarr. PGRAPHITE: An experiment in persistent typed object management. In *Proceedings of the Third Symposium of Software Development Environments*, pages 130–142, September 1988.

[3] V. Benzaken and C. Delobel. Enhancing performance in a persistent object store: Clustering strategies in O_2. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 403–412, Martha's Vineyard, MA, August 1990.

[4] M.H. Nodine, A.H. Skarra, and S.B. Zdonik. Synchronization and recovery in cooperative transactions. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 329–344, Martha's Vineyard, MA, August 1990.

[5] M.P. Atkinson and R. Morrison. Types, bindings and parameters in a persistent environment. In *Data Types and Persistence*, pages 3–20. Springer-Verlag, 1988. (*Proceedings of the First International Workshop on Persistent Object Systems*, Appin, Scotland, August, 1985).

[6] M.P. Atkinson and R. Morrison. Polymorphic names, types, constancy and magic in a type secure persistent object store. In *Proceedings of the Second International Workshop on Persistent Object Systems*, pages 1–12, Appin, Scotland, August 1987.

[7] P.A. Buhr and C.R. Zarnke. Persistence in an environment for a statically-typed programming language. In *Proceedings of the Second International Workshop on Persistent Object Systems*, pages 317–336, Appin, Scotland, August 1987.

[8] J.W. Schmidt and F. Matthes. Naming schemes and name space management in the DBPL persistent storage system. In *Proceedings of the Fourth International Workshop on Persisent Object Systems*, pages 39–58, September 1990.

[9] A.M. Farkas, A. Dearle, G.N.C. Kirby, Q.I. Cutts, R. Morrison, and R.C.H. Connor. Persistent program construction through browsing and user gesture with some typing. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 375–394, San Miniato, Italy, 1992.

[10] P.L. Tarr, J.C. Wileden, and A.L. Wolf. A different tack to providing persistence in a language. In Richard Hull, Ronald Morrison, and David Stemple, editors, *Second International Workshop on Database Programming Languages*, pages 41–60, June 1989.

[11] P.L. Tarr, J.C. Wileden, and L.A. Clarke. Extending and limiting PGRAPHITE-style persistence. In *Proceedings of the Fourth International Workshop on Persistent Object Systems*, pages 74–86, Martha's Vineyard, MA, August 1990.

[12] M.P. Atkinson and P. Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[13] A. Kaplan and J.C. Wileden. Name management and object technology for advanced software. In *International Symposium on Object Technologies for Advanced Software*, number 742 in Lecture Notes in Computer Science, pages 371–392, Kanazawa, Japan, November 1993.

[14] A. Kaplan and J.C. Wileden. More precise name management for object-oriented methods, systems and databases. In preparation.

[15] T. Andrews. Designing linguistic interfaces to an object database or what do C++, SQL and Hell have in common? In *Fourth International Workshop on Database Programming Languages*, New York, NY, Aug–Sep 1993. (Invited Talk).

[16] M.P. Atkinson, P. Buneman, and R. Morrison. Binding and type checking in database programming languages. *The Computer Journal*, 31(2):99–109, February 1988.

[17] A.L. Wolf, L.A. Clarke, and J.C. Wileden. The AdaPIC Tool Set: Supporting interface control and analysis throughout the software development process. *IEEE Transactions on Software Engineering*, 15(3):250–263, March 1989.

[18] R. Morrison, F. Brown, R. Connor, Q. Cutts, A. Dearle, G. Kirby, and D. Munro. The Napier88 reference manual (release 2.0). Technical Report CS/93/150, University of St. Andrews, St. Andrews, U.K., 1993.

[19] A. Dearle and A.L. Brown. Safe browsing in a strongly typed persistent environment. *The Computer Journal*, 31(6):540–544, April 1988.

[20] A. Dearle, Q.I. Cutts, and G.N.C. Kirby. Browsing, grazing and nibbling persistent data structures. In John Rosenberg and David Koch, editors, *Proceedings of the Third International Workshop on Persisent Object Systems*, pages 56–69, Newcastle, Australia, January 1989.

[21] D.L. Wells, J.A. Blakely, and C.W. Thompson. Architecture of an open object-oriented database management system. *IEEE Computer*, 25(10):74–82, October 1992.

[22] A. Dearle. Environments: A flexible binding mechanism to support system evolution. In *22nd Hawaii International Conference on System Sciences*, pages 46–55, Hawaii, January 1989.

[23] M.P. Atkinson. Persistent programming practices. In *Proceedings of the Fifth International Workshop on Persistent Object Systems*, pages 352–353, San Miniato, Italy, 1992.