Implementing Persistency in Common LISP

Heiko Kirschke*

May 21, 1999

This paper presents an overview of PLOB, a system for *P*ersistent *L*isp *Ob*jects. Besides describing the architecture of the system, the paper focuses on the practical issues of developing a heterogenous and portable system using LISP, CLOS and C. The major features of PLOB are explained.

Keywords Persistency, LISP, Common LISP object system (CLOS), Metaobject Protocol (MOP), persistent objects, schema evolution, foreign function interface (FFI), Allegro Common LISP, LispWorks Common LISP

1 INTRODUCTION

This paper provides a survey of PLOB, a system for persistent LISP objects. An overview of the architecture of PLOB is given. The main topics of this paper focus on the experience of developing a heterogenous system in Common LISP and C, which is portable among the major LISP systems from Harlequin and Franz Inc. and a number of operating systems. A closer look is taken at the Metaobject Protocol (MOP) as implemented in both above mentioned LISP systems. The development of LISP software is compared against software development using other programming languages.

More details about using PLOB and its design can be found in [Kirschke 1997].

2 DESIGN GOALS

The design goal of PLOB was to develop a system for making built in, LISP and CLOS objects persistent, plus providing sound standard database functionality as already established for relational databases, like transactions, locking, indexes and schema evolution. Using this functionality should be as simple and transparent as possible, without too many losses in efficiency or safety. Another major point was that persistency becomes an orthogonal feature of an object, independent from other features like belonging to some certain class or having a special representation. A direct consequence of this goal had been that all LISP objects can become persistent, independent of their concrete (transient) type.

The dynamic typing of LISP influenced the design of the data representation used: except for the very first layer, PLOB represents its data as dynamic typed persistent objects, also in the layers not coded in LISP. An additional reason for using dynamic typed objects in the non-LISP layers is that this allows movement of much of the database functionality into lower layers without losing the data representation paradigm of LISP, especially for the lower layers which are used at the server side. This means that code can be moved into the server layers, thus needing less coding to be done for the client; and the server can use the advantages of this data representation for its internal purposes.

In the lowest layer, PLOB's persistent objects are allocated from a persistent heap. Systems using a similar approach are WOOD, AllegroStore and Itasca. Some other systems use a relational database as a persistent backend (SQL/ODBC interface by Paul Meurer, Allegro ODBC, the Common SQL interface for LispWorks [Harlequin 1997, chapter 17] and PCLOS [Paepke 1991]). This has the advantage of passing the rich relational database data manipulation functionality for persistent objects across to the LISP system using them, but often results in trapping the programmer into the *impedance mismatch* between relational and object-oriented systems. For LISP, this nor-

*mailto:Heiko.Kirschke@acm.org

mally means restricting persistent objects to be statically typed objects, or emulating dynamic typing which in turn cannot be stored in an index table (for example by using the evaluation of an s-expression as an object's slot value). These systems often have limited capabilities for doing schema evolution (promoting the effects of a changed class definition into the database).

3 CONSIDERATIONS FOR IMPLEMENTATION

Except the highest layer containing the LISP API, all lower layers have been coded in C. One of the reasons for this approach was that when work started on PLOB, a third-party tool coded already in C for representing persistent objects was chosen (the POSTORE library developed by the Persistent Programming Research Group at St. Andrews University). To bypass the relatively bad performance of foreign function calls from LISP to C, more and more functionality has been added to the layer beyond LISP (the C layers below). Another reason for this approach was that in 1993, no middleware was available to build simply a distributed network of heterogenous subsystems which exchange data by some communication channel. This situation seems still to persist, as some newer LISP projects illustrate [Cramer 1998].

The first version of PLOB was implemented as a single foreign function library loaded into the current LISP image, with the idea in mind to subdivide it into a client/server system communicating by standard Remote Procedure Calls (RPCs) at a later time. This was actually done; now, PLOB can be used in one of two modes of operation, one a classical client/server system, the other one a serverless system. The first mode allows multiple user access to a database, whereas the second mode is for fast single user access.

4 ARCHITECTURE

Depending on whether the client/server or the serverless mode is chosen, different layers are used. Figure 1 shows the layers of PLOB. In client/server mode, all layers are used. In serverless mode, the caching and communication layers shaded gray are replaced by a single layer which connects the LISP API layer directly to the database layer. The mode can be selected dynamically at runtime when opening the database.

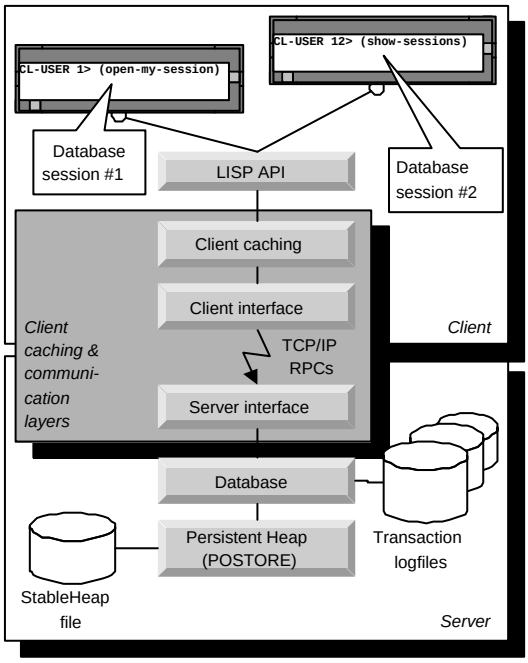

Figure 1: PLOB layers

The following sections explain the functionality of each layer: each is supplemented by experience gained when developing that layer.

4.1 LAYER 1: PERSISTENT HEAP

The persistent heap layer contains the POSTORE library coded in C. This layer provides a heap from which persistent records can be allocated; a persistent record can be locked using a semaphore. Garbage collection based on reachability is also provided within the POSTORE library and is used by PLOB. In this sense, an object is reachable if it is referenced by the transitive closure of a designated root object. This is equivalent to the definition of reachability as implemented in most transient Common LISP systems, where the 'root object' can be assumed to reference the container which holds all packages:

root → package* → symbol* → object*

Besides allocating persistent records and simple semaphores, no additional database functionality is pro-

vided by this layer. This lowest layer is the only one which has been used as a ready-to-use third-party product.

Since the POSTORE library has been maintained now for at least 7 years, the code is rather stable and almost bug-free. Although the library is using the virtual memory functions of the host operating system, ports from the original SunOS version to Solaris, Linux and Windows/NT have been possible.

Because the POSTORE library has a tendency towards causing swapping when handling sufficient large databases, a disadvantage showed up for operating systems with limited disk performance (like Windows/NT).

4.2 LAYER 2: DATABASE

The database layer adds all standard database functionality, like locking, two-phased transactions and utilities for index administration (btrees). Like the layer below, this layer is coded in C. It defines all persistent data types in terms of the low-level persistent heap records. The data types actually implemented are close to the type definitions as specified in [CLtLII 1990], with database-specific extensions. For example, each persistent object in this layer has an additional slot containing some locking information. Locking differs from using semaphores (see layer 1) by recognizing and trying to resolve lock conflicts arising from multiple concurrent accesses to an object's state. The locking algorithm used is *hierarchical locking* at the levels of database, object, and object slot with the sharing lock mode 'read' and 'read-only' and the exclusive lock mode 'write' (figure 2) [Gray & Reuter 1993, p. 406]. In hierarchical locking, a lock set at a certain level implies a lock also is set at all levels below; this allows adapting of the lock granularity to the degree needed. For example, when a lock is set on database level and assuming this is the highest lock level, this implies that all objects in the database are locked too.

Granted→ / Requested↓	**Read-Only**	**Read**	**Write**
Read-Only	✔	✔	✔
Read	✔	✔	✘
Write	✘	✘	✘

✔ ≡ Compatible locks ✘ ≡ Conflicting locks

Figure 2: Lock conflictmatrix

This layer is also responsible for maintaining the locks on persistent objects which are accessed by the higher layers. Each access is categorized as needing either a read lock, a write lock or no lock at all. When the actual access is requested from the higher layer, this layer tries to achieve the corresponding lock necessary for the operation; if the lock is not granted, the operation is not done at all and the higher layer is informed of a lock conflict. It is the responsibility of the higher layer to decide what to do with a lock conflict; the normal strategy is to retry the offending operation anew after some time, assuming that the lock which caused the conflict has been released in the meantime. For the serverless version, it is possible to suspend the LISP process instead of running a busy wait until the lock (which caused the conflict) is released. Locks can only be set within an active transaction; it is the caller's responsibility to ensure that there is an active transaction on calling this layer.

The database layer fulfills the ACID [Gray & Reuter 1993, p. 166] properties: changes to an object's state are *A*tomic, *C*onsistent, *I*solated and *D*urable. For performance reasons, it is possible to configure PLOB to weaken some of these properties.

This database layer is the first one oriented towards the data modelling of LISP and CLOS. Modelling LISP objects posed no big problem at all, because apart from the objects being dynamically typed most of them could be mapped easily to conventional (or composed) C types. The situation is different for CLOS objects. CLOS is a *procedural reflective* system, describing an object's state and its behavior by the *behavior* of metaobjects associated to the object [AMOP 1991]. The metaobjects are in turn 'normal objects' interpreted as having a descriptive semantic meaning. For relocation issues, PLOB cannot store and call the function bodies of methods by itself in the database layer.[1] Consequently, the data modelling in this layer shows one significant difference compared to true CLOS objects: persistent CLOS objects are defined in terms of their *structure*, and not in terms of the behavior of their associated metaobjects (this could be called *structural reflectiveness*). Since the structure has been chosen to match the assumed structure of CLOS' metaobjects as closely as possible, this mismatch is hardly observable in the top-

[1] In consequence, storing and calling methods here would mean running a fully-fledged LISP and CLOS system in the server, with its state (defined classes, global environment) equalized to that of the client using it. This was beyond the scope of implementing PLOB.

level API layer mapping persistent objects into a transient LISP system. Also, the top-level API layer takes care of hiding this mismatch as far as possible.

4.3 Layers 3 and 4: Client/server interface

The client/server interface uses standard RPCs for establishing a communication between client and server. When transitioning from the first monolithic, non-client/server version to the client/server version, the already existing declarations of the C functions were extended to serve as a base for generating the RPC interface code, too. This resulted in all C declarations wrapped into C preprocessor macros looking like some kind of Interface Definition Language (IDL e.g. of CORBA or COM), from which now both the RPC interface code for the client/server version and the LISP foreign function code for the serverless and client/server versions are generated (figure 3). As stated above, using some communication middleware would have been more appropiate, but at that time no such middleware was available. Figure 4 shows how the generated code integrates into the communication layers of figure 1.

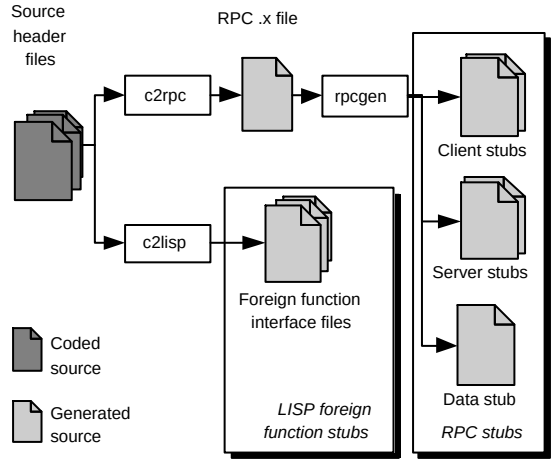

Figure 3: Generated code

Since all RPC calls are client-initiated, there is no way for the server to call the client directly. This made some parts of the code more difficult than others. For example, when the server encounters an error, it cannot call directly a client callback procedure to signal the error to the client's listener. The approach chosen is that the server aborts the failing operation, and an error code and message is transmitted back immediately after the error occurred as the result of the RPC call. The client caching layer checks each RPC call result for an error code and signals an encountered error to the LISP client's listener. Besides error handling, all mapping procedures could also not be implemented by using a direct callback into the client which transfers process control to LISP. Mappings are done instead with iterators [Gamma et al. 1994, p. 257], which maintain the state of a mapping operation explicitly in an object.

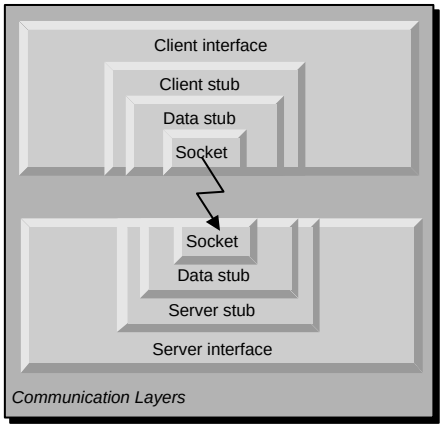

Figure 4: Refinement of communication layers

4.4 Layer 5: Client caching

When extending the very first, non-client/server version into the RPC-based client/server version, it was found that the system became very slow due to the socket-based fragmented data transfer between client and server. The solution was to insert an additional caching layer, with the task of maintaining as much as posssible of a persistent object's state on the client side. Besides that, this layer does a more efficient object allocation by pre-allocating objects in chunks from the server and passing these pre-allocated objects to the higher LISP layer one-by-one for each allocation from the layer above. Since the identity of these new allocated objects is only known to the client which allocated the chunk, locking new allocated objects in the server is not necessary at all; this reduces communication overhead. Also, this layer uses knowledge about a current client's database lock state to speed up opera-

tion. This is done by looking for an exclusive database lock which might have been requested from the client and that has been granted by the server; in this case, the client knows that further lock requests cannot fail, since due to the hierarchical locking all database objects are locked exclusively. So, any subsequent lock requests can be granted immediately on the client's side, without asking the server for locking any more.

Debugging this layer was a pain. The server runs as an extra process, so it is no problem to run it under the control of a debugger. On the client side, this layer is loaded as a library into the LISP process, which made using a debugger impossible for UNIX, since all LISP systems seem to strip all symbolic debug information from libraries loaded at runtime. The only exception was the excellent debugger support by Microsoft's Visual C, which lets you debug a Dynamic Load Library apart from the process using the library.[2]

4.5 Layer 6: LISP/CLOS API

The high level API is the link between the transient LISP objects and the persistent object representation in the lower layers. It takes care of

- opening the database
- converting between persistent and transient representations
- starting and ending transactions
- setting locks on objects being worked upon
- maintaining indexes on objects and
- monitoring the schema evolution on classes.

In transient memory, PLOB uses no special representation for persistent objects; instead, a persistent object exists with two representations, a transient one and a persistent one (figure 5). For CLOS classes declared to be persistent, a transient representation can be chosen on a per-slot basis ranging from a slot being a proxy which allocates no memory in the transient system at all, to a slot representing a shadow-copied version of the state of its persistent counterpart. For CLOS objects which are not declared persistent and instances of LISP built in classes, the object is always represented

[2]Of course, one might ask why some operating systems urgently require a better debugger than others.

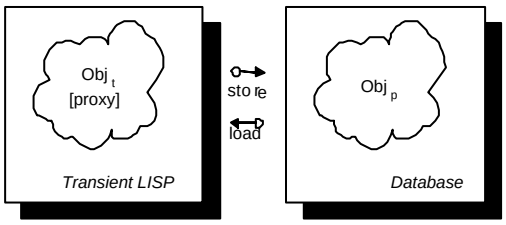

Figure 5: Representation of persistent objects

in the transient system as a shadow copy with the persistent object's state. The reason for this approach is that using knowledge about the internal representation of (transient) objects and modifying these in order to implement persistency is difficult and tends to be very system dependent [Ford et al. 1988].

Because of the different natures of objects found in LISP systems, it was necessary to provide different interfaces for different kinds of objects. For example, it is not possible to monitor the state of a built in object to keep track of changes to the object's state, since this would mean overloading all state-changing built in functions. A simple interface following the store/load paradigm is offered for these objects. For CLOS objects, the Metaobject Protocol [AMOP 1991] offers different degrees of plugging into the access to an object's state, therefore making possible automatic promotion between both representations.

5 Linking LISP and C

For the communication layers between client and server, standard RPCs are used. The reason for this was to make maximal reuse of existing functionality, and to use some mature and generally available tools for communication. Since most of the RPC tools (code generator and runtime library) are based directly on C, it was felt that it would be better not to interface to RPCs directly on the socket level from LISP, but to wrap the RPC calls into an API, shielding the LISP system from doing RPC calls on sockets directly. The first solution would have meant re-coding the RPC data conversion layer again in LISP and using sockets directly (figure 4 on p. 4). The second solution has the advantage that the default data conversion functions generated for RPCs can be used directly and no socket programming is necessary from the LISP level; this way, the data conver-

sion necessary between LISP and C is done by the foreign function interface (FFI) instead.

Another advantage is that for the client/server system no extra LISP license for running PLOB's server is needed, since it is coded in C. Besides that, this decision was also influenced by the missing middleware in LISP.

The foreign function interfaces of both LispWorks and Allegro Common LISP differ considerably in their syntax, resulting in additional coding for portable applications, either subsume the different interfaces under a common interface, or to code the foreign function interface seperately for each LISP system. In PLOB, the first solution was chosen by defining and using an internal 'poor man's foreign function interface, which works on simple foreign function argument types and on statically typed vectors which contain elements of exactly one of these types.

The functionality of both interfaces is quite similar. The LispWorks foreign function interface is more ambitious than that of Allegro Common LISP. For example, it is not possible to directly specify a call-by-reference argument to C in Allegro; instead, a workaround with a one element sized array must be used. The LispWorks Common LISP foreign function interface is type complete as regards the possibility of declaring C types and function arguments; although it seems to be relatively load intensive at runtime.

6 METAOBJECT PROTOCOL

The Metaobject Protocol (MOP) is used to extend CLOS by persistent classes (classes for which instances are persistent). Since classes themselves are also represented by instances, they become persistent. The functionalty of the MOP actually used focuses on extending and monitoring class definitions and on intercepting slot accesses to persistent instances. The basic idea is to specialize a class metaobject class `persistent-metaclass` on `standard-class`; the protocol implemented in CLOS on metaobjects makes it possible to add the desired functionality to CLOS.

The Metaobject Protocol of Common LISP as described in [AMOP 1991] has not been adopted as an official standard, but most CLOS implementations follow the standard described there. Problems showed up where the LISP system used did *not* follow exactly the definition given in [AMOP 1991], or where the definition did not set out clearly an explicit normative reference.

6.1 EXTENDING `DEFCLASS` BY ADDITIONAL CLASS AND SLOT OPTIONS

Class definitions done with `defclass` are extended by taking additional class and slot options to inform PLOB how persistency should be managed for the class' instances. The additional class and slot options implement the following behavior.

Declaring the extent of a slot *(class and slot option)*
The extent of a slot can be declared by a slot option; if passed as a class option, the extent is used for all slots of the class. The extent defines how a slot should be represented, i.e. if it should be represented only in the object's transient representation (transient slot), or only in the persistent representation (proxy slot) or in both of them.

Non-standard object constructors *(class option)*
When creating the transient representation of a persistent object (figure 5 on p. 5), special actions not foreseeable at the time PLOB was coded might be necessary, for example to establish side effects associated with the instance to be created. A class option makes it possible to use a hand-coded constructor for creating the transient representation.

Performance optimizations *(class option)*
Depending on the application using PLOB, it might be possible that objects are 'embedded' into a surrounding object, which means that to reference the surrounding object will also mean to reference the embedded object with a high probability. A class option can be specified to mark a class as having only instances which are each embedded into some surrounding instance. In the client/server mode, this has the effect that an embedded object is always transferred with the surrounding object, thus reducing network traffic. An example would be a `line` class with two `point` instances embedded in it.

Schema evolution *(class option)* The effect of changing a class definition can be specified with a class option. By default, a changed class definition will

be propagated into the database. If the schema evolution option is set, PLOB can be told e.g. to deny any schema evolution on a class, or to allow or deny for structure objects a possible change in their identity which might occur in a schema evolution which enlarges the number of slots.

Declaring an index on a slot values *(slot option)*
On slots of persistent classes, an index can be declared. The index is maintained at slot accesses; some basic queries can be done on the index, retrieving for a single slot value or a range of slot values the associated persistent instances.

Many of the options described are not only available for classes which are declared persistent, but for 'normal' CLOS classes and structure classes, too. For these classes, the options are passed to PLOB in top-level declarations without requiring an actual class definition; the values of these options are then stored in tables besides the class and slot metaobjects.

Slots on persistent objects with an `:allocation` which is `:class` are handled exactly as in CLOS. For implementing proxy slots which are not represented in transient memory at all, it would have been preferable to use the instance allocation protocol as described in [AMOP 1991, p. 99–104]. This protocol allows extra values besides `:instance` and `:class` to be passed to the `:allocation` slot option, with the meaning that a non-standard allocation must be handled by the user. This is in contradiction to the Common LISP standard, which disallows the use of non-standard allocation option values, as described in [CLtLII 1990, p. 824]. To fix this, a workaround has been implemented which patches the internal used structures to allow for user allocated slots (actually, this patch is only implemented for LispWorks Common LISP).

6.2 Monitoring class definitions

Changes done to a class definition are monitored and propagated to the database. Mainly, this implements the schema evolution of classes and their persistent instances. PLOB implements lazy schema evolution, meaning that changes to objects implied by a schema evolution are done at the latest possible point in time. Schema evolution is done destructively: the whole object or a part of it might change due to the schema evolution.

6.3 Slot access

For implementing transparent persistency, it is important to keep track of accesses to an object's state. Slot access monitoring is used to implement the following functionality.

Transaction wrapping Each access to a slot's state is wrapped into an active transaction, to ensure all of the ACID properties (section 4.2 on p. 3) on persistent objects.

Check for schema evolution It is assured that the transient and persistent class definitions are matching: the instance's class is checked if its definition was changed, and, if necessary, the new definition is propagated into the database. The representation of the affected instance is changed, too.

On-demand slot loading The actual slot's state may refer not to the 'real' state, but to a proxy object which references the actual state in the database. In this case, the actual state is loaded from the database when accessing the slot the first time, and the proxy in the slot is replaced by this state. This can reduce memory demands considerably, since otherwise loading an object would mean to load the transitive closure of an object regarding its references, although this transitive closure might never be referenced by the application.

Proxy slot Depending on the class and slot options passed in the class' declaration, the slot might not be represented at all in the transient system. For such a slot, its state is always loaded from or stored in the database.

Write-through slot Being aware of write accesses enables the automatic propagation of state changes done to slots of the transient representation to their persistent counterparts.

Index slot For slots with a declared index, a changed slot state is replicated in the index table.

Slot access can be done in two ways: either by calling an accessor generic function declared on the slot in the class definition (`:reader`, `:writer` or `:accessor` slot option), or by calling `slot-value` or `(setf slot-value)` on an object's slot.

For the first case, [AMOP 1991] defines the generic functions `reader-method-class` and `writer-method-class` [AMOP 1991, p. 224, 242]; these can be used to generate specialized method metaobjects on slots when the class gets finalized.[3] In essence, this means that an application extending CLOS by some functionality gets control over the method code which should be executed for accessing a slot's value when using an accessor generic function. Unfortunately, the algorithm for generating the method code for slot accessors is not stated explicitly in [AMOP 1991], but from other definitions it could be deduced that generic function `make-method-lambda` [AMOP 1991, p. 207] should be called on an instance of a `{reader,writer}-method-class` to generate the accessor's method body. Neither LispWorks Common LISP or Allegro Common LISP implement the generic functions `{reader,writer}-method-class` nor call the generic function `make-method-lambda` on slot accessor methods; instead, implementation specific accessor function bodies are generated without using a standardized protocol. The workaround for both systems is to specialize a method on generic function `finalize-inheritance` called for finalizing a class which replaces the class' accessor methods by those needed for working with PLOB.

The implemented behavior of `slot-value` and `(setf slot-value)` works as expected. The standard defines that calls are delegated to `slot-value-using-class` and `(setf slot-value-using-class)`; by default, for classes with a class metaobject class of `standard-class`, both systems bypass this convention for performance reasons and use some internal functions for fast slot access. As soon as a specialized class metaobject class is used, `slot-value` and `(setf slot-value)` fall back into their standardized behavior, if there are methods specialized on these at all.

[3] Finalizing means establishing a complete (finalized) class' representation at the latest point necessary from its declaration as decided by the runtime system, e.g. before the first instance of a class is created. This is not to be changed with the finalization as defined for Java.

7 ADDRESSING PERSISTENT OBJECTS

After objects have been made persistent, there must be a method to retrieve the stored objects somehow, for example in a later (transient) LISP session. This can be done by two kinds of addressing: using persistent symbols or querying an index.

A persistent symbol works like a transient symbol: it is a named object with persistent properties. A persistent symbol is contained in a persistent package and thus matches the data modelling of LISP as close as possible. The value property of a persistent symbol is also persistent. This means that when the value of a persistent symbol is set, the object representing the value is made persistent.

The second way of addressing a persistent object is by making a query. This will return a list with all persistent objects matching the query's criterion. Queries can only be made for instances of persistent CLOS classes with an index declared on the queried slot.

8 DISTRIBUTION

The system described in this paper is available as free software from `http://plob.sourceforge.net`, available for Linux, Solaris 2, Windows/NT, and IRIX. The LISP systems supported are LispWorks Common LISP up to and including 4.2 and Allegro Common LISP up to and including 6.2. PLOB's home page holds a self-contained archive, containing all binaries and source code (except sources for the persistent heap layer), an installation and a user's guide plus some supplementary documentation. There is also a mailing list with some 80+ subscribers on it.

9 CONCLUSIONS AND FUTURE WORK

This paper desribes the technical design of PLOB, some of the considerations which led to this design and the challenges which had to be faced when implementing the system. The overall experience is that using Common LISP for software development is, to some extent, state of the art; however, the Common LISP standard

needs some modernisation. Compared to other object-oriented languages, Common LISP seems mostly a modern language mostly when looking at the concepts of the core language. Modernisation seems to be more necessary for the language standard outside the core, as can be assumed also from the subjects of the ongoing discussions on the NCITS/J13 (formerly X3J13) mailing list for updating the Common LISP standard. Many of the items which have been discussed have been observed as being problematic when coding PLOB: a standardized MOP, the foreign function binding, containers with weak pointers (arrays and hash tables), and last but not least an API for persistent LISP objects (of course, not necessarily that of PLOB).

There are still many open issues for future work on PLOB. One of the major points is that the server needs a performance improvement for large databases. The schema evolution needs some redesign; it would be preferable to make it more 'dynamic', for example by providing a view onto a persistent object; as objects defining the view, persistent class metaobjects could be used.

REFERENCES

[AMOP 1991] Gregor Kiczales, Jim des Rivières, and Daniel G. Bobrow. *The Art of the Metaobject Protocol*. The MIT Press, Cambridge, Mass., 1991. 4.2, 4.5, 6, 6, 6.1, 6.3, 6.3, 3, 3

[CLtLII 1990] Guy L. Steele Jr. *Common LISP the Language*. Digital Press, Bedford, MA, 2 edition, 1990. 4.2, 6.1

[Cramer 1998] Nichael Cramer. Distributed-OMAR: Reconfiguring a Lisp System as a Hybrid Lisp/(Java) Component. In _, editor, *USLUGM 98*, Berkely, CA, November 1998, pages _–_. _. Remote copy. 3

[Ford et al. 1988] Steve Ford, John Joseph, David E. Langworthy, David F. Lively, Girish Pathak, Edward R. Perez, Robert W. Peterson, Diana M. Sparacin, Satish M. Thatte, David L. Wells, and Sanjive Agarwala. ZEITGEIST: Database Support for ObjectOriented Programming. In K. R. Dittrich, editor, *Advances in ObjectOriented Database Systems*, Berlin, 1988, pages 23–42. Springer-Verlag. 4.5

[Gamma et al. 1994] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, Mass., 1994. 4.3

[Gray & Reuter 1993] Jim Gray and Andreas Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann Publ. Inc., San Francicso, CA, 1993. 4.2, 4.2

[Harlequin 1997] *LispWorks User Guide*, 4.0.1 edition, 1997. Part number LW-4.0.1-UG (HTML files). 2

[Kirschke 1997] Heiko Kirschke. *Persistent LISP Objects User's Guide*, November 1997. Remote copy. 1

[Paepke 1991] Andreas Paepke. PCLOS Reference Manual. Technical report, HP Laboratories, Palo Alto, CA., 1991. 2

Heiko Kirschke is working as a consultant and project manager at the Competence Center e-Business at CSC PLOENZKE AG, Hamburg. He has been working as a software engineer with POET Software GmbH, Hamburg, and as a research assistant for computer science at Hamburg University, where he also earned his MS degree. He has practiced computer science and object-oriented technology for over a decade as a designer, implementer, researcher and consultant. His interests are object-oriented programming languages, object-oriented databases, document processing and software development in general.