

# LINKING AND LOADING IN A PERSISTENT DSM OPERATING SYSTEM

M. SCHOETTNER, O. MARQUARDT, M. WENDE, AND P. SCHULTHESS

[schoettner|marquardt|wende|schulthess]@informatik.uni-ulm.de

Department of Distributed Systems, University of Ulm

89075 Ulm, Germany

## ABSTRACT

Our native Java compiler directly generates runtime structures in a persistent Distributed Shared Memory (DSM). The compiler has been used to build a general purpose PC Operating System (OS) on top of a persistent DSM memory. The persistent DSM operating environment lends itself naturally to an integration of symbol tables, class descriptors and naming during Java program compilation and execution. The linking and loading process is streamlined and the persistent storage property makes a traditional loader redundant.

After a brief overview of the Plurix OS we discuss traditional linking strategies and present extended linking perspectives in the persistent Plurix DSM. This includes a novel immediate linking scheme offering the flexibility of dynamic binding while detecting errors at linking time. The remaining task of class initialization is performed only once by the Plurix Java compiler. To deal with the semantics of distributed Java static class variables we propose extended initialization rules together with a more flexible type checking scheme.

**Keywords:** Linking, Java, Languages, Distributed Shared Memory, Operating Systems.

## 1. INTRODUCTION

Plurix is a lean distributed Operating System (OS) written in Java for the PC platform. We abandon the hardware independence of Java as we do not rely on a Java Virtual Machine (JVM) like JavaOS does [1]. Herewith we gain efficiency and speed. Language based OS development has been successfully demonstrated by native Oberon [2] and others.

The well-known Distributed Shared Memory (DSM) paradigm offers a natural solution for distributing data among several nodes [3]. Applications running on top of a DSM are not aware of data locations. Any reference can either point to a local or a remote memory block. During program execution the OS detects a remote memory access and automatically fetches the desired memory block. Plurix implements a page-based DSM using the built-in memory protection mechanisms of the Intel CPU to detect memory access [7].

The benefits of the persistence property are acknowledged by [4], [19] and others. There are efforts to add persistence to existing languages without disturbing their semantics and implementation [4]. Other researchers are trying to develop OSs with explicit support for persistence [5]. One of our research goals is to provide a persistent DSM enclosing the kernel and compiler. There have been little efforts on adding persistence to a DSM system by now [6].

Our system is developed using our Plurix Java Compiler (PJC) which is an integral part of the OS. Fundamental runtime structures are modeled in Java and, the kernel and the PJC are built on top. The PJC is aware of the persistent DSM and directly creates runtime structures attached to the related symbol information. Persistent symbol information is the base for separate compilation and version management.

Selecting the proper linking time traditionally involves a tradeoff between early error detection and extensibility. The properties of persistence, DSM, and a reference bookkeeping of the memory-management allows Plurix to statically link classes without inheriting the typical drawbacks. Linking in persistent systems has been investigated by others but to the best of our knowledge no similar linking scheme has been proposed [19].

The traditional tasks of a loader are: reading the executable file, adjusting addresses, and performing initialization. The persistent DSM makes a file-system superfluous. As the PJC uses a position independent code generation scheme the only task of a loader is to initialize classes. This is done by the PJC as we were not willing to pay the costs for lazy initialization as postulated by the Java language specification [15].

Classes are shared through the DSM and the problems caused by Java static class variables are similar to those of global variables [7], [20]. The problem has been identified in previous work but no solution was given [17]. We propose user-private classes with extended initialization rules together with a more flexible type checking scheme.

The remainder of this paper is organized as follows. The persistent Plurix DSM environment is briefly reviewed in section two. Section three illustrates traditional linking models together with the new approach of Plurix. The following section discusses extended initialization strategies. Subsequently we compare our contribution with related work.

Finally, we describe the current project status and give an outlook on future work.

## 2. THE PLURIX ENVIRONMENT

### The Operating System

The first Plurix prototype runs within a single LAN segment (10/100 Mbps Ethernet). The central abstraction within our design is a persistent DSM sharing data and code. A node runs several cooperative tasks which are repeatedly called by a central loop much like in the Oberon System [2]. Processing within Plurix is partitioned into short restartable transactions. Together with an optimistic synchronization scheme they realize the main consistency model of Plurix [7], [8]. Memory accesses (read or write) are monitored by the Memory Management Unit (MMU). There may be many read-only replications of a memory page. If a write to a page occurs all read-only replicates are invalidated. Having one global address space for a cluster is no limitation for the emerging 64-Bit address spaces [9], [10]. Memory protection is ensured by the strong typing of the Java language.

Plurix implements *orthogonal persistence* for the Java language meaning any Java object can persist [4]. Memory persistence and backup storage is initially provided by a central disk-server within the Plurix cluster. Solutions for fault-tolerance and recovery will be investigated in future work.

An automatic distributed Garbage Collection (GC) relieves the programmer from the burden of memory management. The GC keeps track of all references to a heap object by using the *backchain concept* [7]. Every heap object includes a backchain pointer locating the first reference pointing to it, see fig.-1. If several references point to an object they are chained together. If the backchain is empty an object is garbage. The compiler supports the backchain by calling a runtime function for every pointer assignment.

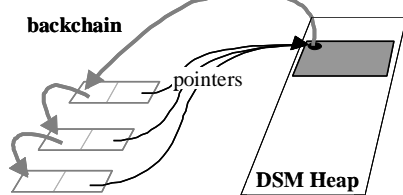


Figure-1, The backchain concept

### The Plurix Java Compiler

Our OS is developed with a proprietary Plurix Java Compiler (PJC) translating source texts into Intel machine instructions. We are reluctant to lose performance at the OS level by interpreting Java byte-code. We also decided against a Just-In-Time (JIT) compiler because the kernel and drivers require minor language extensions [11].

The first version of the PJC does not use an intermediate representation nor state of the art code optimizations. Parser

and code generator are coalesced resulting in a small compact compiler size (~140 kb byte-code). The current code generation is preliminary and uses a single stack allocation strategy.

The PJC is tailored for the persistent DSM and directly creates runtime structures and code segments. Hence, object-, symbol-, library- and exe-files are avoided [11]. If a source text is successfully compiled the symbol class descriptor (SyCD) is automatically registered by the compiler in the name service. The corresponding runtime class descriptor (RtCD) is constructed and attached to the SyCD. The RtCDs are linked via an import table if necessary. Finally the RtCDs are initialized by the compiler and are ready for execution.

Every heap object contains memory management information (e.g. chaining of heap blocks) defined in the class `POBJECT`. Of course, this information is protected against unauthorized access.

The PJC is written in Java and will be the development tool in the Plurix world after being bootstrapped. Currently we are using a cross compilation under MS-Windows.

### A uniform name service

Commercial Operating Systems typically have numerous name services implemented in different components (e.g. file system, address book, ...). The persistent DSM concept facilitates the implementation of a single uniform namespace. An overview of the Plurix namespace is shown in fig.-2. Caching of name service entries is implicitly achieved through the DSM. Access restrictions will be implemented to allow protected private sub-spaces for each user.

Though everything is stored in the DSM only data stored in the “pub” subtree is globally accessible by all nodes. As Java prohibits pointer arithmetic the only way to get a reference to an object is to query the name service.

The subtree “nodes” contains configuration information for individual machines of a cluster. The “users” tree stores the root for each users private data and personal settings. “Grps” defines special sharing views for groups.

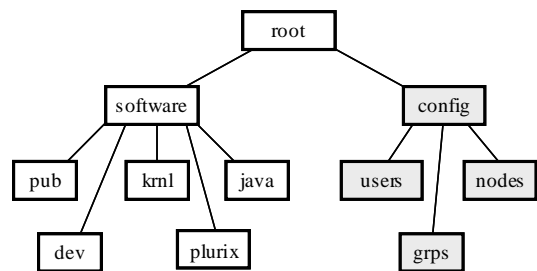


Figure-2, The Plurix namespaces

The “java” subtree contains classes of the java package whereas Plurix specific classes are stored below the “plurix” package. The “knl” package defines the kernel api available only for privileged users. Finally the “dev” space is intended for device driver developers.

The location in the namespace defines the package membership. A directory in the namespace forms a package.

### 3. LINKING

There are two modes for linking: statically and dynamically. Dynamic linking dates back at least to the Multics system (1965). Commercial OSs like MS-Windows (dynamic link libraries) and Unix (shared libraries) support both forms.

#### Packages and Modules

The Java package concept corresponds to the modules of Modula-2. Packages encapsulate a collection of types (classes or interfaces) whereas Modula-2 modules store record types.

In Java terms *exporting* means defining a class or field as `public`. Per default all types are only visible within their own package. *Importing* is the opposite operation and establishes an inter-class relationship. All inter-class relationships must be resolved by a so called *binder* or *linker*. The linking task is of recursive nature because linking a class requires linking of all imported classes before.

Inter-class relationships are not performed by the Java `import` statement which only extends the namespace used for name qualification during compilation (see code sample below). They are implicitly established by the use of class names for static method calls or static variable access. The PJC uses an import table for each runtime class descriptor to administer such inter-class relationships [11].

```
import java.io.*

class imp {
    public static int a = 3;
}

public class tester {
    static void main() {
        imp.a = 5;
    }
}
```

Separate compilation of classes requires that the linker observes the version consistency of all inter-class relationships. The consistency check mechanism should avoid unnecessary trickle down recompilations by invalidating clients by simple changes [12], [13]. Invalidating some class *C* means invalidating all classes importing *C* and so on.

#### Traditional linking strategies

In traditional OSs compiler and linker are separate programs. The compiler translates source texts into object-files containing import and export tables. Inter-module calls are realized via an indirection over an import table. The linker checks consistency and fills in the import table. Hence addresses can be defined at loading time and modules are freely relocatable. Some linkers eliminate this indirection for inter-module calls by modifying the object code and inserting direct calls. This approach requires that the linker can find

each inter-module call. This information consumes considerable memory and time as there is one entry per call but it can be realized by chaining the placeholders within the code itself [14].

*Static linking* implies copying all necessary libraries into the executable file. All target addresses for inter-module references can be determined during linking time and can be directly inserted into the executable file by the linker. Hence procedure calls have maximum efficiency as no indirections are necessary and linking errors can be detected at linking time. On the other side, memory is wasted e.g. linking statically the Microsoft Foundation Classes (MFC) consumes for each application ~1,9MB. Unfortunately, any library modification requires a client recompilation.

*Dynamic linking* is realized by inserting symbolic references into the object files to the library module linked at runtime. Under MS-Windows this works as follows, see fig.-3. The imports are defined in the ".idata" section of the exe file. For every used DLL there is an import descriptor with the symbolic names of the imported functions. The loader overwrites the names with proper addresses. Every call includes one indirection via the import address table.

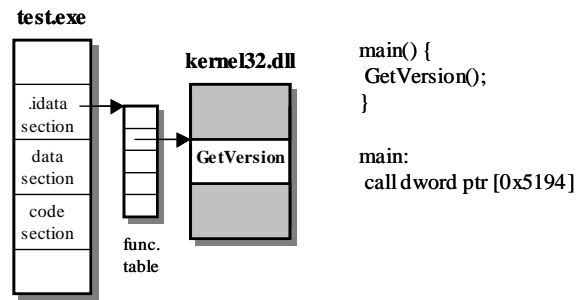


Figure-3, Dynamic Linking in MS-Windows

Dynamic linking eliminates code redundancy which saves memory and therewith loading time. If a shared library is already used by another application it can be directly linked. Furthermore, updated/extended versions (backward compatible) can be used on the fly without recompilation of clients. Java offers extensibility by class inheritance. The main disadvantage is the late error detection. There might be loading failures if the library is removed after linking and before loading. Due to the unknown load address of the library the code needs to be position independent (e.g. all jumps must be PC-relative) or all jumps need to be adjusted by the loader.

*Lazy loading* (or *demand linking*) is a variant where library modules are not loaded until needed by the executing thread. The loading can be done explicitly by the application (e.g. "LoadLibrary" in MS-Windows) or implicitly by a stub code (e.g. JVM). When the stub is invoked it loads the class including all recursively imported types and resolves the symbolic links. The pros and cons are the same as for dynamic linking. Furthermore, lazy linking is also attractive because it minimizes the work of the linker as only those references are resolved which are needed during execution.

On the other side linking errors may even occur during run-time if a library is removed. Unfortunately, each on the fly linking step requires a synchronization of the processors instruction cache which often invalidates a total region of cache entries which is expensive [14].

### Linking in Plurix

Linking in Plurix is performed by the PJC after successful compilation and therefore seems to be static. Though linking is performed during compilation time no memory waste occurs if the library is available and hence shared through the DSM. This way we avoid code redundancy typical for traditional static linking schemes. Furthermore, separate compilation is possible although no separate linker is used. This is possible due to the uniform name service and persistent symbol tables, see fig.4. The symbol class descriptor (SyCD) including method and variable descriptors is registered automatically by the compiler and points to the corresponding runtime descriptor (RtCD). The RtCD contains the method jump table pointing to native code segments and the import table storing references to imported RtCDs [11]. Furthermore, every RtCD is attached to its corresponding SyCD.

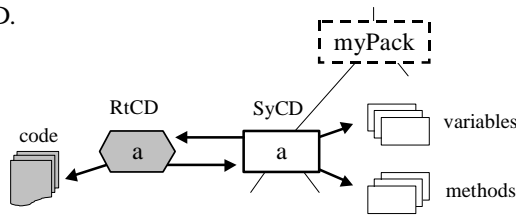


Figure-4, Classes in the name service

A crucial problem in long-living persistent object systems is the handling of type evolution. As symbol information is persistent we have always the full type description and can perform fine granular interface checking [13]. There are two kinds of type evolution: backward compatible or not. If a class *a* is changed in a way it is no longer backward compatible (e.g. deleting a method required by another class), the new resulting type *a\** cannot substitute the older one. In this case the old SyCD-*a* is unregistered from the name service and the new SyCD-*a\** is registered. Note that this does not imply deleting the old descriptors physically, but from now on they are only visible for clients *b* having still a reference to the old type *a*, see fig.-5. All subsequent compilations are automatically using the new type *a\** found via the name service. If all clients of an old invisible type have been deleted the old type *a* is automatically deleted by the garbage collection. Hence we also avoid having dead versions like found in commercial OSs (e.g. unused DLLs in MS-Windows).

If a class is changed but it remains backward compatible the compiler can substitute the old descriptor by using the back-chain. The backchain lets the compiler identify all classes importing the changed class items. Hence a global update can be performed and is propagated through the DSM to all

nodes. Even if a node is concurrently executing code of a type being updated its transaction is aborted and restarted with the new version. Backward compatibility can be tested by the compiler comparing the interfaces of the old and new type (both symbol descriptors are available). The precise definition of *backward compatibility* will be investigated in future work.

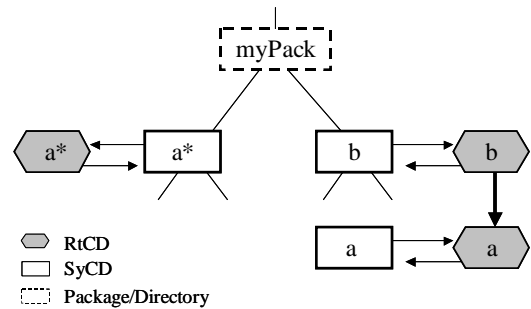


Figure-5, Inter-class relationship and type evolution

We call our binding model *adaptable linking* because of the ability to modify statically linked types during a subsequent compilation without client invalidation and recompilation.

Loading time link errors caused by removed libraries cannot occur in Plurix due to the automatic garbage collection. If a class is linked to another class it cannot be occasionally deleted. An user can only unregister a class from the name service but the garbage collection will detect any references from other importing classes and will not delete it.

Lazy linking was not an option for Plurix though it is implied by the Java language specification [15]. This would require to monitor static class variable access via methods *get* and *put* to detect first time class access. In view of this we were not willing to pay the costs for this indirection and would rather have direct variable access.

The PJC generates position independent code segments (one per method) because all memory objects including code segments may be relocated during runtime. Jumps within a code segment are PC-relative whereas method calls use one indirection via the class descriptor [11]. The cost of this indirection is alleviated by storing the actual class descriptor context in a reserved register [11]. Anyway, hard-coded addresses within code segments are not reasonable in Plurix because these references would violate the memory management constraint to separate references from scalars [11].

In the Plurix context we like to continue the Oberon user interface tradition where text commands simply executed module functions or methods [2]. In Plurix this can easily be realized using persistent symbol tables. If the user clicks on a text Plurix tries to identify class and method names by querying the name service. The persistent symbol information is used to find the proper code segment for execution.

The automation interface of the Microsoft Component Object Model (COM) offers similar functionality. Applications can query method names (including their signatures) of a class and can perform subsequent calls. This technique was intended by Microsoft to support scripting languages like Visual Basic.

Our new proposed *adaptable linking model* thus benefits from the persistent Plurix DSM environment, see table-1. “Module extensibility” means the possibility to bind to new versions of a module without recompiling existing clients.

Table 1., Comparison of linking models

Linking model	Link error detection	Code redundancy	Load time	Module extensibility
Static	Compile time	Redundant	Long	No
Dynamic	Load time	Sharable	Medium	Yes
Lazy	Run-time	Sharable	Short	Yes
Adaptable	Compile time	Sharable	None	Yes

#### 4. LOADING

The task of a loader is to bring an executable file into memory, relocate addresses, and initialize modules. The persistent DSM reduces the tasks of a loader to the initialization. Even this task is performed by the Plurix compiler which creates runtime structures as a compilation output. Hence classes are ready for execution after successful compilation.

The Java initialization rules state: a class or interface is initialized once during the first access [15]. This rule is natural for a Java implementation which is not designed for a persistent DSM. In the Plurix environment it might be confusing for users not seeing the initial values for initialized static class variables because they have been modified by a previous user. This distributed interpretation of the semantics of static class variables might confuse a typical application programmer. Of course, we could prohibit the publishing of such classes but this would be a contradiction to the DSM paradigm. The central question in the persistent Plurix environment is: when or how often is a class initialized. Previous publications have identified the problem but did not propose a solution [16], [17].

We propose an additional class attribute *shared* for clarifying the distributed semantic of static variables. The DSM per default offers only shared classes. Non-shared ones can only be realized by replicating a class descriptor for each user. Replication is automatically performed if a non-shared class from the public namespace is imported. We suggest the following extended class initialization rules:

- shared classes are initialized once during creation,
- non-shared classes having (non-final) initialized static variables are initialized once per user.

Unfortunately, instances of non-shared classes are no longer type equivalent. The Java type test is based on name equivalence which fails for replicated class descriptors.

A Meta Class Descriptor (MCD) automatically generated by PJC offers more type flexibility. The replicated class descriptors are extended by an meta parent pointer referencing the MCD, see fig.-6. The type test is extended to consider possible meta parent pointers and will succeed for instances of non-shared classes.

The problem could also be solved by using a structural comparison for type equivalence checking [18]. However, this would totally change the Java type system and certainly cause a lot of new semantic ambiguities.

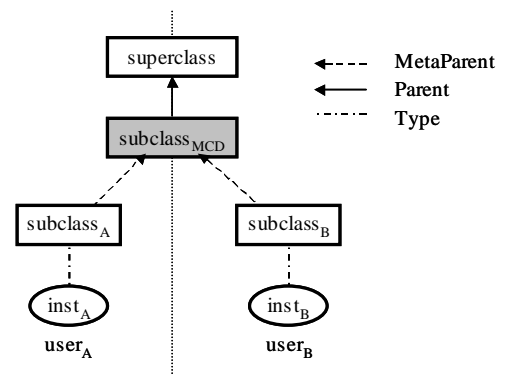


Figure-6, Meta Class Descriptors

#### 5. RELATED WORK

To the best of our knowledge no previous work proposed a similar linking model. Linking in persistent environment was studied by Cutts [19]. He proposed a two phase linking scheme. The initial linking step binds each component to so called locations of other components including identifier resolution and type checking. When a component uses another one during execution a second linking step is performed which retrieves the desired component. This scheme ensures safety by doing type checking and identifier resolution during linking time while offering flexibility required for individual component update. The Plurix approach offers the same advantages while needing only one linking step.

Morrison introduced *hyper programming* where persistent data can be bound to source texts [21]. In this scenario the linking time is even before compilation and offers further perspectives. The embedding of persistent data into texts is natural for a persistent environment and will be implemented in Plurix, too.

Deller and Heiser proposed a dynamic linking scheme for their SASOS system [20]. They were also faced with the problem of not sharing global variables in a shared memory. They use a separate data segment containing private global variables associated with every instantiation of a dynamically linked module.

The data segment is stored in a reserved register. A module descriptor contains pointers to all functions imported from the module and a reference to the data segment of the exporting module. This approach is not applicable for Plurix as migration would be impossible with separate segments. Therefore we decided to solve the problem of non-shared static variables at the language level by replicating class descriptors.

## 6. CONCLUSIONS AND FUTURE WORK

After a brief overview of the persistent Plurix DSM we discussed traditional linking schemes and presented the *adaptable linking model* of Plurix. In Plurix linking is performed at compile time to achieve safety and efficiency.

The DSM property avoids waste of memory as typical for static linking. Persistent symbol tables lay the foundation for separate compilation and allow a safe module update. Together with the reference bookkeeping of the memory management old clients can be relinked to new versions of a class without recompilation. Dynamic linking reduces loading time as only those modules are loaded which are needed during execution. Loading time need not to be considered in a persistent environment. Hence, *adaptable linking* is safe and efficient.

Persistent symbol tables offer further interesting possibilities. There is no more need to distinguish between debug and release versions of programs. An interpretative interface can be designed easily by finding addresses and offsets of classes and methods via the persistent class descriptor. A reflective interface is included at no extra costs.

The task of a loader is reduced to the initialization process which can be performed by the compiler. Therewith a class is ready for execution after successful compilation.

Not sharing static class variables causes problems similar to those of module global variables. They can be solved by a new class attribute *shared* and replication of non-shared class descriptors per user. Such classes are initialized once per user. Extended type compatibility can be reached by the introduction of *meta class descriptors* which make instances of non-shared classes type equivalent.

We have developed a first prototype of the Plurix OS running on three or more Pentium PCs connected via an 100 Mbps Ethernet shown at the CeBIT 2000 fair. We will study further integration aspects between the compiler and the persistent DSM. Elegant version management strategies need to be defined to deal with the problem of type evolution.

## 7. REFERENCES

- [1] "JavaOS", <http://www.sun.com/javaos>, 1997.
- [2] N. Wirth and J. Gutknecht, *Project Oberon – The Design of an Operating System and Compiler*, Addison-Wesley, 1992
- [3] K. Li, "TVY: A Shared Virtual Memory System for Parallel Computing", *Int. Conference on Parallel Processing*, 1988.
- [4] M.P. Atkinson, K.J. Chisholm and R.M. Marshall, "Algorithms for a Persistent Heap", *IEEE Software, Practice and Engineering*, 13(3):259-272, 1983
- [5] A. Lindström, R. di Bona, A. Dearle, S. Norris, J. Rosenberg and F. Vaughan, "Persistence in the Grasshopper Kernel", *Australasian Computer Science Conference, ACSC-18*, pp 329-338, February 1995.
- [6] Morin C. and Puaut I., "A survey of recoverable Distributed Shared Memory Systems", Technical Report Nr 975, IRISA, France, 1995
- [7] S. Traub, "Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen", PhD thesis, University of Ulm, 1996.
- [8] H. T. Kung and J. T. Robinson, "On Optimistic Methods for Concurrency Control", *ACM Transactions on Database Systems*, 1981.
- [9] M. Baker-Harvey, J. Chase, H. Levy, and E. Lazowska, "Opal: A single Address Space System for 64-Bit Architecture", *IEEE Workshop on Workstation Operating Systems*, 1992
- [10] A. Skousen and D. Miller, "Operating System Structure and Processor Architecture for a Large Distributed Single Address Space", *IASTED Parallel and Distributed Computing Conference*, 1998
- [11] M. Schoettner, O. Schirpf, M. Wende and P. Schulthess, "Implementation of the Java language in a persistent DSM Operating System", *Int. Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1999
- [12] R. Hood, K. Kennedy and H. A. Müller, "Efficient Recompilation of Module Interfaces in a Software Development Environment", *ACM SIGPLAN Notices* 22(1), January 1987
- [13] R. B. J. Crelier, "Separate Compilation and Module Extension", PhD Thesis, No 10650, ETH Zurich, 1994
- [14] M. Franz, "Dynamic Linking of Software Components", *Computer*, March 1997
- [15] J. Gosling, B. Joy & G. Steele, "The Java Language Specification", Addison-Wesley, 1996
- [16] A. Malhotra, "Persistent Java Objects: A Proposal", *Int. Workshop on Persistence and Java*, 1996.
- [17] S. Spence, "Distribution Strategies for Persistent Java", *Int. Workshop on Persistence and Java*, 1996.

- [18] R.C.H. Connor, A.B. Brown, Q.L. Cutts, A. Dearle, R. Morrison and J. Rosenberg, "Type Equivalence Checking in Persistent Objects Systems", *Implementing Persistent Object Bases*, A. Dearle, G.M. Shaw and S.B. Zdonik, Morgan Kaufmann, 1990, 151-164
- [19] Q.L. Cutts, "Delivering the Benefits of Persistence to System Construction and Execution", PhD thesis, University of St. Andrews, 1992.
- [20] L. Deller and G. Heiser, "Linking Programs in a Single Address Space", *Usenix Technical Conference*, Monterey, CA, USA, 1999
- [21] Morrison, R., Connor, R.C.H., Cutts, Q.L., Dunstan, V.S. and Kirby, G.N.C., "Exploiting Persistent Linkage in Software Engineering Environments", *Computer Journal*, 38, 1, 1995