

Type Evolution and Version Management in a Persistent Distributed Operating System

M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess

Department of Distributed Systems

University of Ulm, Oberer Eselsberg, 89069 Ulm,

Germany

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

ABSTRACT

Today's commercial Operating Systems (OS) use message passing communication facilities such as Corba, Remote Procedure Calls, and TCP/IP. Distributed Shared Memory (DSM) is an alternative mainly used for scientific computing and specific parallel algorithms. We are currently developing a general purpose PC Operating System using the DSM paradigm as communication media. A new memory consistency model using restartable transactions and optimistic synchronization simplifies the programming of distributed applications. The Plurix system is developed using our Plurix Java Compiler (PJC) translating Java sources directly into Intel machine instructions. The PJC is bootstrapped to the Plurix world where it becomes an integral part of the OS. The benefits of persistence for software development are widely appreciated but appropriate OS and compiler support is necessary to circumvent the inherent problems. Here we investigate the problem of type evolution caused by evolving of persistent types. We briefly review the persistent Plurix environment, the PJC, and our separate compilation scheme exploiting persistent symbol tables. We discuss Java binary compatibility rules for our system and define the stricter term backward compatibility. Furthermore we present mechanisms for smooth type evolution and version management for different type generations.

KEY WORDS: Type Evolution, Persistence, Version Management, Object-Oriented Languages.

1. INTRODUCTION

The Plurix Operating System (OS) uses the well-known Distributed Shared Memory (DSM) paradigm introduced by Keedy [1] and Li [2] and implements a new memory consistency model using restartable transactions in combination with an optimistic synchronization scheme [3]. A primary research goal is to investigate the DSM as a general purpose communication medium e.g. for simplified development of distributed applications.

Orthogonal persistence [4] is another DSM property in the sense that any object reachable from the root of the cluster-wide name service can persist independent of its type. OS, compiler, programs, and data permanently reside in the persistent DSM. The programmer is thus released from the burden of writing the complex serialization and deserialization functions required for file-based systems. Persistence is supported at the OS level much like it is in the Grasshopper project [13]. Questions of fault-tolerance and recovery are dealt with – but not in this paper.

Our compiler is used for OS and application development, directly translating Java source texts into Intel machine instructions [5]. Native code generation is mandatory for OS and driver development in Java. Language based OS development has been successfully demonstrated in systems like Oberon [6]. By using a popular language like Java we hope to increase the acceptance for a new system by the programmer community.

Modification and evolution of existing classes and code are inevitable in any software system. Development tools must provide appropriate support to allow smooth type evolution. These problems have been discussed in file-based environments, considering e.g. adaptation of Java class files during load-time [7]. In a persistent object system the problem is even more fundamental because instances are also affected and may require old class versions to survive. The system must also provide appropriate version management mechanisms. Special persistent languages like Napier-88 have been developed to investigate the problem [14].

The remainder of this paper is organized as follows. In the following second section, we shortly review the properties of the Plurix DSM. Subsequently, we present the compiler architecture tailor-made for the persistent DSM environment. In the fourth section, we define backward compatibility rules for the Plurix environment and show how PJC supports type and instance evolution. Furthermore, we discuss version management strategies. Finally, we compare our system to related work.

2. THE PERSISTENT DSM SYSTEM

The Plurix DSM is organized as a distributed heap storage (DHS) going beyond traditional DSM systems in as much as it shares not only data but also class descriptors and code segments. The current setup runs within a single LAN segment. Each node in a Plurix PC-cluster is controlled by a central loop; threading through a number of cooperative asks [6]. Memory accesses are monitored by the Memory Management Unit. There may be many read-only copies of a memory page until a write to that page occurs. Tasks in the OS are partitioned into restartable transactions. The Plurix memory consistency model features optimistic synchronization of transactions as described in [3].

The DHS is *orthogonally persistent* meaning any Java object is by persistent default [4]. All objects reachable from single cluster-wide name service persist. Memory persistence and backup storage is initially provided by a central disk server, which may be substituted by a distributed solution.

Distributed incremental Garbage Collection (GC) relieves programmers from explicit memory management. GC keeps track of all references to an object by using the *reference backchain* [3]. Every heap object includes a backchain pointer locating the first reference pointing to it. If there are several references to a single object they are chained together. If the backchain of an object is empty no more references to that object exist and it is garbage.

3. THE PLURIX JAVA COMPILER

The Plurix Java Compiler (PJC) transforms Java source texts directly into Intel machine instructions. As PJC is used for OS development we abandoned the hardware independence of Java. PJC itself is written in Java and is bootstrapped into the Plurix OS. We are aware of other native Java compiler projects like Marmot [8] and the GNU Java Compiler [15] but we needed an integrated solution and were reluctant to provide the runtime environment required by those compilers. The first PJC version (< 150 kb byte-code) does not provide an intermediate representation nor code optimizations.

Persistent Symbol Tables

During the semantic analysis the compiler directly registers symbol table entries in the name service. This is achieved by melting compiler scopes with the directory concept. Directories define a scope for their content - classes do the same for their methods and instance variables. Users can browse through the name space and access data and classes. The compiler directly operates on the name service resolving names according to the scoping rules of the Java language.

Even debugging information is readily available without distinction of release and debug versions like in traditional systems. Persistent symbol table entries also lay the foundation for a textual user interface similar to the Oberon tool texts [6]. Any text can contain user commands like *classname.methodname* that are activated by a mouse click and resolved using the name service to execute the desired method.

Runtime Structure Generation

As Plurix implements an orthogonal persistent DSM runtime structures and code segments are created at compile time - bypassing the generation of object-, symbol-, library-, and exe-files. Similarly no separate linker nor a loader is required in our persistent DSM world.

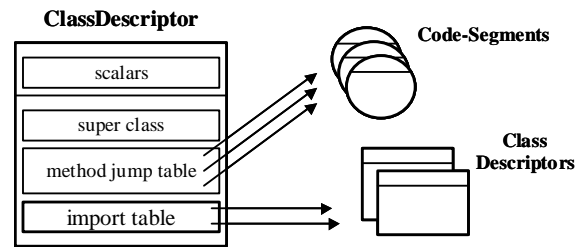


Figure-1, Inter-class relationships & import table

In the course of a successful compilation the runtime structures are bound and initialized by the compiler and are ready for execution. Inter-class relationships are realized by an import table per class descriptor, see figure-1. A detailed description of the runtime structures can be found in [5].

Although binding is performed statically by the compiler we do not sacrifice extensibility. Using the bookkeeping of references realized by the backchain and a cluster-wide name service we are able to alter or upgrade existing types called adaptive linking described previously in [9].

Separate Compilation

By integrating symbol table entries in the cluster-wide name service separate compilation is simplified. Symbol class descriptors (SyCD) are automatically registered by the compiler. A SyCD points to method, variable descriptors, and to the corresponding runtime descriptor, see figure-2. The native code segments are attached to the runtime class descriptors (RtCD).

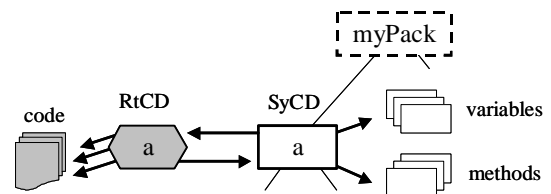


Figure-2, Persistent symbol table entries & runtime structures

The compiler transparently accesses new and already existing classes. The path of base classes like those in “java.lang” is fixed and known by the compiler to prevent overriding and therewith security attacks. All other names are firstly resolved searching the user name space before inspecting the global directory. Therefore, classes can be tested locally before being published for the global cluster. The physical location of classes is transparent and symbol table entries resident on other nodes are automatically fetched by the OS during the first access in a compilation run. If a class of the current compilation package is already stored in the name service then type evolution is about to proceed and a temporary symbol table entry is allocated, see section four and five.

4. TYPE EVOLUTION

Evolution represents the ongoing cycle of activities in development, use, and maintenance of software systems. They evolve over time in response to numerous requirements including bug fixes, extension of functionality, and especially support changes in related software. Because such changes are inevitable, mechanisms must be provided to support smooth type evolution.

We call a class B *client* of a class A if B imports A in some way (e.g. B calls a static method of A). If a client needs to be recompiled due to a modification of an imported class we say the client is *invalidated*. Invalidations are of recursive nature because invalidating a class A also invalidates all the clients of A and so forth. Recompilation of all classes in the system can always restore code consistency and is sufficient for traditional file-based systems. This might be expensive and unnecessary recompilations should be avoided. Furthermore, source code is often unavailable. Of course situations might arise where recompilations require user interaction to remove incompatibilities by editing source code.

Java Binary Compatibility

Due to the wide distribution of Java class files in the Internet it is impossible to invalidate and recompile classes in the case of a modification and source code is often not available. Hence, the Java language specification defines very flexible compatibility rules for type evolution and accepts runtime linking errors which might occur anyway. The Java language specification defines the notion of binary compatible changes: “A change to a type is binary compatible with (equivalently, does not break binary compatibility with) preexisting binaries if preexisting binaries that previously linked without error will continue to link without error” [10]. Binary compatibility should not be mixed with source text compatibility. There are modifications that are binary compatible but the sources cannot be recompiled at all without user editing.

Java permits a lazy binding strategy which binds and initializes at runtime at the time of the first access. Linking errors may occur at runtime but offsets of runtime structures are dynamically determined and based on symbolic entries within the byte-code. Lazy binding and runtime compilation by a Just-In-Time Compiler (JIT) offers flexibility but accepts the risk of runtime errors.

Plurix Backward Compatibility

PJC directly creates runtime structures in the persistent DSM and statically determines their offsets. Source texts are translated directly into machine instructions containing statically determined offsets such as indices into a method jump table. Hence, Java binary compatibility is not sufficient and we need extended rules defining the term *backward compatibility*. We call a modified class A^* backward compatible to its ancestor A if the runtime descriptor of A^* can substitute the one of A without offset invalidations, if no instance variables have been deleted, and if A^* is binary compatible to A. Backward compatibility is stricter than binary compatibility as it requires the runtime structures to be compatible to avoid client invalidations. In this context deleting instance variables is never acceptable because such a modification invalidates all existing instances. Beyond these extended rules binary compatibility is also required to preserve the semantics of the language. These stricter rules offer less flexibility but on the other side provide better reliability by detecting errors at compile- or bind-time.

Type evolution in a persistent object system like Plurix is an important issue because not only code and type descriptors are affected by modifications but also persistent instances which are invalidated when their class descriptor is invalidated. Hence, invalidations ought to be avoided. The development tools should avoid unnecessary invalidations and must also support different type versions (see section five). Evolution occurs on three levels: package, class, and interface. There are four different categories of possible modifications:

1. attribute modifications,
2. adding methods or variables,
3. deleting methods or variables,
4. modifications of type hierarchies.

Changing access attributes may hide names, make them visible or change binding from static to dynamic or vice versa. Clients will be invalidated if a necessary name becomes inaccessible, e.g. a required public method is changed to be private or if the binding is changed. Previously hidden names becoming visible cause no problems to clients but possibly in subclasses.

Extending classes by dynamic methods or instance variables involves an invalidation of all subclasses, because their offsets need to be adjusted due to the replication of dynamic variables and method jump table entries.

Adding new static methods or variables does not cause problems because they are not replicated in subclasses. Adding a method that overloads an existing one may offer a better overloaded alternative for clients (see below). Assume a client calls the method "A.print((short)8);". Due to the implicit conversion rules the method with the Integer-Parameter is selected and "v1" is printed as a result. If the class A is extended by a new method "print(short s)" the old method is still called until the client is recompiled. If the client is recompiled it would bind to the new overloaded method alternative and the output result would be "v2". Nevertheless, the Java language specification tolerates calling the old method and does not require a recompilation [10].

```
public class A {
    static void print(int a) {
        System.out.println("v1");
    }
}

public class A' {
    static void print(int a) {
        System.out.println("v1");
    }
    static void print(short s) {
        System.out.println("v2");
    }
}
```

The removal of static methods or variables always invalidates clients relying on one of those names. The removal of dynamic methods or variables is even worse, as all subclasses including their clients are invalidate because of offset variation.

Modifications related to subtyping are a problem because class descriptor offsets may be invalidated. Inserting or deleting supertypes will invalidate all classes below in the inheritance hierarchy when the moved classes have one or more dynamic methods or instance variables. If existing classes become final all subclasses are disallowed and invalidated.

Interface evolution does not affect classes in SUNs specification but the type system is softened [11]. Normally, if an interface is extended by a new method all classes being a subtype of this interface must implement all the related methods. Subsequently, this would require invalidation of all classes implementing an interface upon an interface extension. Nevertheless, the Java language specification tolerates missing interface methods in the case of interface evolution. In the Plurix context interface modifications may change offsets of subinterfaces that may invalidate classes and require recompilations.

Compiler Support for Type Evolution

Firstly, sufficient type information must be provided to allow fine-grained interface checks and to prevent unnecessary invalidations [12]. Making parts of the symbol table persistent is instrumental for evolution support. We now have old and new symbol information of all types evolving to newer versions and the compiler may arrange the offsets of runtime structures of upgrading candidates in a backward compatible manner. The main goal in this phase is to preserve offsets of all old names in the new class descriptor to enable a subsequent substitution. If for example an old dynamic method is deleted the gap may be filled by a new dynamic method to preserve the offsets for all other old methods. This is reasonable because there might be clients that do not require this deleted method and would be unnecessarily invalidated. We can distinguish two backward compatibility levels: *fully backward compatible* modifications (fbc) and *partially* ones (pbc), see figure-3.

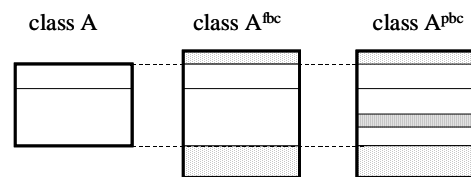


Figure-3, Partial & full backward compatibility

The first kind of evolution is simple and occurs in two cases: unchanged interface or compatible extension. If an upgrading candidate is not extended and not shrunk full backward compatibility is obvious and the new code segments can be attached to the old class descriptor and the classes of the current compilation are also bound to the old version. If a candidate is extended in a fully backward compatible way all clients of the old class descriptor are bound to the new version (using the back-chain) including the clients of the current compilation.

Partial backward compatibility requires fine-grained checks for each imported name of a client. This can be performed by a *tentative recompilation* of all clients of an upgrading candidate or an enhanced binding check. The tentative recompilation works as follows. Assume B is a client of a class A being upgraded to A* and A* is partial backward compatible to A. If the tentative recompilation of class B using A* results no errors A* is backward compatible to B and can be rebound to B. If an error occurred during the compilation B can not be attached to A* and continues to use A. We call this procedure tentative compilation because we hide error messages from the user as we do not want him to edit B and make it compatible to A*. This would possibly invalidate all clients of B and so on and may result in an undesirable recursive invalidation. This approach requires the source code of all clients but the development tools need not to be modified.

An alternative strategy is to provide more detailed inter-class relationship information. Normally import relationships are realized through an import table for each class in a coarse-grained fashion. Finer-grained information is achieved using every entry in the import table as an anchor to a list of all imported methods and variables – every list entry pointing to its corresponding symbol descriptor. Herewith we can identify missing names or violated semantic rules to detect incompatibilities.

Instance Evolution

In a persistent object system it is not sufficient to develop strategies for smooth type evolution. In addition we need to consider instances affected by type modifications. In the case of a backward compatible evolution we have to distinguish two situations either instance variables have been added or have not. If no variables were added the existing instances can be attached to the new type without any changes. If instance variables have been added all existing instances need to be upgraded by instantiating new objects from the new type and copying existing values to the new instance. Furthermore all references to the instance being upgraded need to be adjusted to the new successor using the backchain. The old instances are garbage. If any instance variables have been deleted the old type descriptor must survive and we need to develop a version management for different type generations.

5. VERSION MANAGEMENT

Independent of the techniques applied to the problem of type evolution there will always be situations where modified types are no longer backward or binary compatible, especially in the case of deletions. Recompilations can restore type consistency but not always instance consistency resulting in a coexistence of different class versions. These different versions must be visualized in a way that the user understands the relationships between different generations of instances and classes. Furthermore, unused versions should automatically be deleted when they are no longer needed.

The name service will always show only the newest version of a type that is used automatically for compilations. All previous versions are chained and attached to the newest version and can be viewed by using the Plurix explorer. The name of every old version is automatically extended by a version number that can be used to access older versions for subsequent compilations.

If a client – still attached to an old version of a type – is recompiled after evolution errors may appear. The user might edit the source code and explicitly import the desired old version or make the source compatible to the newest version. If the compiler displays an error due to use of an imported class it checks whether older versions of this class are available.

If yes, it displays the class name of the old version the current runtime structures is bound to. This helps the developer to identify the last used and compatible version among several generations.

During compilation of a class having multiple existing versions backward compatibility of the newest type is always checked against all those versions. Hence numerous versions of a class may collapse after some time and modifications again to a single version.

6. RELATED WORK

The database community is also faced with the problem of type evolution. More recent object-oriented databases (e.g. O₂ [16]) offer support for schema evolution but isolated to individual types only. Typically, data transformation functions must be programmed explicitly. Lerner proposed an approach for compound changes involving multiple types with automatically generated transformation functions [17]. Similar to our strategy she compares sets of types definitions to detect changes. Another approach to schema evolution relies on the simultaneous maintenance of multiple versions of a type. Maintenance complexity grows with the number of type generations. Our system allows automatic collapsing of multiple type versions if a new backward compatible type is available.

Persistent programming languages (e.g. Napier-88, PS-*Algol*, ...) are fundamentally confronted with the problem of type evolution because any type can persist. Napier-88 offers structural type equivalence with implicit subtyping alleviating the problem of type evolution. On the other side implicit subtyping is not very popular - no major object-oriented language offers it - probably because type checking is weakened since it may establish subtype relationships in cases where there is no semantic relationship.

The importance of flexible binding mechanisms to support dynamic modifications in persistent languages has been identified in [18].

Keller investigated component adaptation in a traditional system without requiring source code. A modifier uses the symbolic information stored in the Java class files together with a delta file to adapt the loaded Java class files before the verifier proceeds [19]. He relies on lazy binding and the original Java binary compatibility rules.

7. CONCLUSIONS AND FUTURE WORK

Persistent symbol tables simplify separate compilation and provide the extensive type information necessary for supporting smooth type evolution. Modifications of types and code are inevitable and the development tools must support smooth type upgrades to minimize invalidations of existing code and data.

This is important in the traditional Java applet world as well as for the persistent Plurix DSM. Type evolution in a persistent object system is crucial because existing instances are affected by modifications as well. We learned that the original Java binary compatibilities rules are not sufficient for the Plurix environment where classes are statically translated to machine instructions and bound by the compiler. Hence, we defined the stricter backward compatibility rules extending Java binary compatibility. Subsequently, we compared the effects of different modification categories in our persistent system to the Java specification. We discussed how the Plurix Java Compiler supports smooth type evolution using the cluster-wide name service and the bookkeeping of references called backchain. PJC even supports partial backward compatibility to avoid unnecessary invalidations. Nevertheless, some modifications yield different type versions especially in the case of deleted instance variables where existing instances cannot be attached to a new type version.

The name service will always return the newest version of a class and all compilations will use this version by default. Older generations are attached to the newest version and their names are extended by version numbers. Herewith older classes are still accessible for subsequent compilations although older source codes need to be modified to use the changed class name. If a class is compiled all existing generations are checked for backward compatibility and a list of version may collapse again to a single version. Unused types are automatically freed by the garbage collection on the page server.

8. REFERENCES

- [1] J. L. Keedy and D. A. Abramson, "Implementing a large virtual memory in a Distributed Computing System", *Hawaii International Conference on System Sciences*, Hawaii, USA, 1985.
- [2] K. Li, "IVY: A Shared Virtual Memory System for Parallel Computing", *International Conference on Parallel Processing*, 1988.
- [3] S. Traub, "*Speicherverwaltung und Kollisionsbehandlung in transaktionsbasierten verteilten Betriebssystemen*", PhD Dissertation, University of Ulm, Germany, Distributed Systems Department, 1996.
- [4] M. J. Jordan, T. Prinzezis, M. P. Atkinson, L. Daynes, and S. Spence, "An Orthogonally Persistent Java", *SIGMOD Record*, Volume 25, 1996.
- [5] M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess, "Implementation of the Java language in a persistent DSM Operating System", *International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, USA, 1999.
- [6] N. Wirth and J. Guteknecht, "*Project Oberon – The Design of an Operating System and Compiler*", Addison-Wesley, 1992.
- [7] R. Keller and U. Hoelzle, "Binary Component Adaptation", *European Conference on Object-Oriented Programming*, 1998.
- [8] T.B. Knoblock, R. Fitzgerald, and E. Ruf, "Marmot: An Optimizing Compiler for Java", *Technical Report MSR-TR-99-33*, Microsoft Research, USA, 1999.
- [9] M. Schoettner, O. Marquardt, M. Wende, and P. Schulthess, "Linking and Loading in a persistent DSM Operating System", *International Conference on Parallel and Distributed Computing Systems*, Las Vegas, USA, 2000.
- [10] J. Gosling, B. Joy, & G. Steele, "*The Java Language Specification*", Addison-Wesley, 1996.
- [11] S. Drossopoulou, D. Wragg, S. Eisenbach, "What is Java Binary Compatibility?", *Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 1998.
- [12] R. B. J. Crelier, "*Separate Compilation and Module Extension*", PhD Dissertation, ETH Zurich, Swiss Federal Institute of Technology, 1994.
- [13] R. di Bona, A. Dearle, S. Norris, J. Rosenberg, A. Lindstroem and F. Vaughan, "Persistence in the Grasshopper Kernel", *Australasian Computer Science Conference*, Australia, 1995.
- [14] R. Morrison, R.C.H. Connor, Q.L. Cutts, G.N.C. Kirby, D.S. Munro, and M.P. Atkinson, "The Napier88 Persistent Programming Language and Environment", *Fully Integrated Data Environments*, Springer, 1999.
- [15] Per Bothner, "A Gcc-based Java Implementation", *IEEE Comcon*, 1997.
- [16] F. Ferrandina, G. Ferran, T. Meyer, J. Madec, and R. Zicari, "Schema and database evolution in the o2 object database system", *International Conference on Very Large Databases*, 1995.
- [17] B. Staudt-Lerner, "TESS: Automated Support for the Evolution of Persistent Types", *Automated Software Engineering Conference*, 1997.
- [18] A. Dearle, "Environments: A flexible binding mechanism to support system evolution", *International Conference on System Sciences*, Hawaii, 1989.
- [19] R. Keller und U. Hölzle, "Binary Component Adaptation", *European Conference on Object-Oriented Programming*, 1998.