Lazy Modular Upgrades in Persistent Object Stores

Chandrasekhar Boyapati, Barbara Liskov, Liuba Shrira*, Chuang-Hue Moh, Steven Richman
MIT Computer Science and Artificial Intelligence Laboratory
200 Technology Square, Cambridge, MA 02139
{chandra,liskov,liuba,chmoh,richman}@lcs.mit.edu

Abstract

Persistent object stores require a way to automatically upgrade persistent objects, to change their code and storage representation. Automatic upgrades are a challenge for such systems. Upgrades must be performed in a way that is efficient both in space and time, and that does not stop application access to the store. In addition, however, the approach must be modular: it must allow programmers to reason locally about the correctness of their upgrades similar to the way they would reason about regular code. This paper provides solutions to both problems.

The paper first defines *upgrade modularity conditions* that any upgrade system must satisfy to support local reasoning about upgrades. The paper then describes a new approach for executing upgrades efficiently while satisfying the upgrade modularity conditions. The approach exploits object encapsulation properties in a novel way. The paper also describes a prototype implementation and shows that our upgrade system imposes only a small overhead on application performance.

Categories and Subject Descriptors

D.2.7 [**Software Engineering**]: Maintenance, Enhancement;
D.2.4 [**Software Engineering**]: Program Verification;
D.3.3 [**Programming Languages**]: Language Constructs

General Terms

Design, Verification, Languages

Keywords

Software Upgrades, Presistent Objects, Ownership Types

*Liuba Shrira's address is Department of Computer Science, Brandeis University, Waltham MA 02454.

The research was supported in part by DARPA Contract F30602-98-1-0237, NSF Grant IIS-98-02066, and NTT.

1 Introduction

This paper is concerned with providing efficient automatic upgrades for objects in a persistent object store [4]. Persistent object stores provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. The store acts as an extension of an object-oriented programming language such as Java, allowing programs access to long-lived objects in a manner analogous to how they manipulate ordinary objects whose lifetime is determined by that of the program.

Since persistent objects may live a long time, there may be a need to upgrade them, that is, change their code and storage representation. An upgrade can improve an object's implementation; correct errors; or even change its interface in the face of changing application requirements. Providing a satisfactory way of upgrading objects in a persistent object store has been a long-standing challenge.

A natural way to define upgrades is for programmers to provide a *transform function* [52] for each class whose objects need to be upgraded. A transform function initializes the new form of an object using its current state. The system carries out the upgrade by using the transform functions to transform all objects whose classes are being replaced.

This way of handling upgrades introduces two problems:

1. The system must provide good semantics that let programmers reason about their transform functions locally, thus making it easy to design correct upgrades.
2. The system must run upgrades efficiently, both in space and time.

This paper provides solutions to both problems.

The paper first introduces a set of *upgrade modularity conditions* that constrain the behavior of an upgrade system. Any upgrade system that satisfies the conditions guarantees that when a transform function runs, it only encounters object interfaces and invariants that existed when its upgrade was defined. The conditions thus allow transform functions to be defined *modularly*: a transform function can be considered an extra method of the class being replaced, and can be reasoned about like the rest of the class. This is a natural assumption that programmers would implicitly make in any upgrade system—our conditions provide a grounding for this assumption. This way an upgrade system provides good semantics to programmers who design upgrades.

The paper then describes an efficient way of executing upgrades while satisfying the upgrade modularity conditions. Previous approaches do not provide a satisfactory solution to this problem. An upgrade system could satisfy the conditions by keeping old versions of all objects, since old versions preserve old interfaces and old object states. However versions are expensive, and to be practical, an upgrade system must avoid them most of the time. Some earlier systems [48, 7, 40] avoid versions by severely limiting the expressive power of upgrades (e.g., transform functions are not allowed to make method calls); others [5, 46] limit the number of versions using a stop-the-world approach that shuts down the system for upgrade and discards the versions when the upgrade is complete; yet others [52] do not satisfy the upgrade modularity conditions that enable programmers to reason about their upgrades locally.

Our approach provides an efficient solution to this problem. We perform upgrades *lazily*; we don't prevent application access to persistent objects by stopping the world but instead transform objects just before they are accessed by an application. We do this without requiring the use of versions most of the time. Also, we impose no limitations on the expressive power of transform functions. Yet we provide good semantics: our upgrade system satisfies the upgrade modularity conditions and thus supports local reasoning.

Our approach exploits the fact most transform functions are *well-behaved*: they access only the object being transformed and its encapsulated subobjects. If transform functions are well-behaved, our runtime system provides an efficient way to enforce the upgrade modularity conditions without maintaining versions. If they aren't, we provide an additional mechanism, *triggers*, which can be used to control the order of transform functions to satisfy the conditions. If even triggers are insufficient, we use versions but only in cases where they are needed.

Determining whether a transform function is well-behaved is straightforward, given an understanding of encapsulation. Programmers can use informal reasoning to check this property. Alternatively, the compiler can ensure this property using static checking, provided the programming language is extended to support *ownership types* [11, 12, 14, 15, 24, 25]; we sketch such an extension in the appendix. Ownership types offer a promising approach for making object-oriented programs more reliable, and may become part of future object-oriented languages.

We have implemented a prototype lazy modular upgrade infrastructure in Thor [41, 9], a highly optimized object-oriented database. The paper describes the prototype and discusses the design trade-offs we made to optimize performance in the common case. The paper presents performance results that indicate that the infrastructure has low cost. It has negligible impact on applications that do not use objects that need to be upgraded. We expect this to be the common case because upgrades are likely to be rare (e.g., once a week or once a day). The results also show that when upgrades are needed, the overhead of transforming an object is small.

The paper is organized as follows. Section 2 presents our upgrade modularity conditions. Section 3 describes how our system executes upgrades. Section 4 shows that our system satisfies the upgrade modularity conditions. Section 5 describes our implementation and Section 6 presents performance results. Section 7 discusses related work. Section 8 concludes. The appendix describes ownership types, which can be used to check statically that transform functions are well-behaved.

2 Semantics of Upgrades

This section describes the upgrade model. It also defines the upgrade modularity conditions and explains why they make it easy for programmers to reason about upgrades.

2.1 System Model

We assume a persistent object store (e.g., an object-oriented database) that contains conventional objects similar to what one might find in an object-oriented programming language such as Java. Objects refer to one another and interact by calling one another's methods. The objects belong to classes that define their representation and methods. Each class implements a type. Types are arranged in a hierarchy. A type can be a subtype of one or more types. A class that implements a type implements all supertypes of that type.

We assume that applications access persistent objects within atomic transactions, since this is necessary to ensure consistency for the stored objects; transactions allow for concurrent access and they mask failures. An application transaction consists of calls on methods of persistent objects as well as local computation. A transaction terminates by committing or aborting. If the commit succeeds, changes become persistent. If instead the transaction aborts, none of its changes affect the persistent objects.

Upgrades in a persistent object store can be used to improve an object's implementation, to make it run faster, or to correct an error. They can also be used to extend the object's interface, e.g., by providing it with additional methods, or even to change the object's interface in an *incompatible* way, so that the object no longer behaves as it used to, e.g., by removing one of its methods or redefining what a method does. Incompatible upgrades are probably not common but they can be important in the face of changing application requirements.

2.2 Defining Upgrades

An upgrade is defined by describing what should happen to classes that need to be changed. The information for a class that is changing is captured in a *class-upgrade*. A class-upgrade is a tuple:

$$\langle \text{old-class, new-class, TF} \rangle$$

A class-upgrade indicates that all objects belonging to old-class should be transformed, through use of the *transform function*, TF, into objects of new-class. TF takes an old-class object and a newly allocated new-class object and initializes the new-class object from the old-class object. The upgrade system causes the new-class object to take over the identity of the old-class object, so that all objects that used to refer to the old-class object now refer to the new-class object.

This mechanism preserves object state and identity. The preservation is crucial, because the whole point of a persistent store is to maintain object state. When objects are upgraded, their state must survive, albeit in a modified form as needed in the new class. Furthermore, a great deal of object state is captured in the web of object relationships. This information is expressed by having objects refer to other objects. When an object is upgraded it must retain its identity so that objects that referred to it prior to the upgrade still refer to it.

An upgrade is a set of one or more class-upgrades. When an upgrade changes the interface of a class C *incompatibly*, so that its objects no longer behave as they used to, this may affect other classes, including subclasses of C and classes that use types C no longer implements. All affected classes have to be upgraded as well, so that the new system as a whole remains type correct. A *complete upgrade* contains class-upgrades for all classes that need to change due to some class-upgrade already in the upgrade [5, 28, 30, 52]. Completeness is checked using rules analogous to type checking.

Our system accepts an upgrade only if it is complete. At this point we say the upgrade is *installed*. Once an upgrade has been installed, it is ready to run. An upgrade is executed by running transform functions on all affected objects, i.e., all objects belonging to the old classes.

2.3 Upgrade Modularity Conditions

As we mentioned in the introduction, an upgrade system must guarantee that when a transform function runs, it encounters only interfaces that existed at the time its upgrade was installed and states that satisfy its object's invariants. This guarantee means the transform function writer need not be concerned, when reasoning about correctness of upgrades, with object interfaces and object invariants that existed in the past or will exist in the future. Instead, the transform function can be thought of as an extra method of the old-class: the writer can assume the same invariants and interfaces as are assumed for the other methods.

The job of the upgrade system is to run upgrades in a way that supports this modularity property. There are two different problems that must be solved. First is the question of how to order upgrades relative to application transactions and to other upgrades. Second is the issue of how to order the transform functions belonging to a single upgrade.

2.3.1 Ordering Upgrades

The requirement for ordering of entire upgrades is simple: upgrades are transactions and thus must be serialized relative to application transactions and to one another: a later upgrade must appear to run after an earlier one.

An upgrade transaction transforms all objects of old-classes. We view each transform function as running in its own transaction; each such *transform transaction* is the execution of a transform function on one object. The entire upgrade transaction thus consists of the execution of all the transform transactions for that upgrade.

Now we can state the serializability requirement. A similar condition is given in [52]. We use the notation [$A1$; $A2$] to mean that $A1$ ran before $A2$.

M1. If we have [A; TF(x)], where A is either an application transaction that is serialized after TF's upgrade is installed, or A is a transform function from a later upgrade, this has the same effect as [TF(x); A].

An upgrade system that stops the world to run an upgrade transaction and only allows application transactions to continue after that transaction completes (e.g., [5, 46]) satisfies this condition trivially, since the order [A; TF(x)] won't occur for either later application transactions or later upgrades. An upgrade system that doesn't stop the world will have to ensure that when it runs A before some transform that must be serialized before A, the effect will be the same as if they ran in the opposite order.

2.3.2 Order within Upgrades

Condition M1 says nothing about how the upgrade system chooses the ordering of transforms within an upgrade. The following two conditions constrain this order.

M2. If TF(x) and TF(y) are from the same upgrade and TF(x) (transitively) uses y and we have [TF(y);TF(x)], this has the same effect as [TF(x); TF(y)].

M3. If TF(x) and TF(y) are from the same upgrade and TF(x) does not (transitively) use y and TF(y) does not (transitively) use x, then [TF(y); TF(x)] has the same effect as [TF(x); TF(y)].

Here TF(x) *uses* y if TF(x) reads/writes a field of y or calls a method of y. *Transitively uses* means that this action may occur via uses of intermediate objects, e.g., TF(x) calls a method of z, which calls a method of y.

Condition M2 states that if transform function TF(x) uses object y, the behavior of the system must be the same as if TF(x) ran before TF(y). Condition M3 states that for unrelated objects, the behavior of the system must be independent of the order in which their transforms ran; in this case the upgrade system can choose whatever order it wants for the two transforms.

Some upgrade systems satisfy Condition M2 by using versions (so that when TF(x) runs it sees the old version of y); this is the approach taken in [5, 46]. Others avoid the problem altogether by limiting the expressive power of transforms so that they cannot make method calls, as in [48, 7, 40]. A third possibility (and the direction we follow) is to satisfy M2 by controlling the order of transforms so that when TF(x) uses y, x is transformed before y.

Conditions M1-M3 together ensure upgrade modularity: transform functions encounter the expected interfaces and object invariants because upgrades run in upgrade order, application transactions do not interfere with transform functions, transform functions of unrelated objects do not interfere with each other, and transform functions of related objects appear to run in a pre-determined order (namely an object appears to be transformed before its subobjects). Thus these conditions allow transform functions to be reasoned about locally, as extra methods of old classes. Writers of transform functions can assume the same invariants and interfaces as are assumed for the other methods of old classes.

3 Executing Upgrades

This section describes our lazy upgrade system.

Stopping the world to run an upgrade is undesirable since it can make the system unavailable to applications for a long time. Our system avoids delaying applications by running the upgrade incrementally just in time.

The system runs each transform function as an individual transaction. These transactions are interleaved with application transactions. When an application transaction A is about to use an object that is due to be transformed, the system interrupts A and runs the transform function at that point; this way we ensure that application transactions never observe untransformed objects.

The transform transaction T must be serialized *before* A in the commit order since A uses the transformed object initialized by T. Therefore, if T reads or modifies an object modified by A or if T modifies an object read by A, the system aborts A. A is highly unlikely to abort, however; we discuss how our techniques avoid having to abort A in Sections 4.3 and 4.4.

When T finishes executing, it commits. Then the system continues running A, or if A was aborted, it reruns A.

Our system does not require that an earlier upgrade complete before a later upgrade starts since this might delay the later upgrade for a long time. Instead many upgrades can be in progress at once and several transforms may be pending for an object. The system runs pending transforms for an object in upgrade order. While running transform transaction T, the system might encounter an object that has pending transforms from upgrades earlier than the one that defined T; in this case, the system interrupts T (just as it interrupted A) to run the pending transforms.

4 Enforcing the Modularity Conditions

The lazy approach described in Section 3 ensures that transforms of individual objects run in upgrade order and that applications running after an upgrade never observe objects that need to be transformed. But it does not enforce the upgrade modularity conditions discussed in Section 2.3, and thus it does not provide the desired semantics that allow programmers to reason locally about their transform functions. For example, it's possible that application transaction

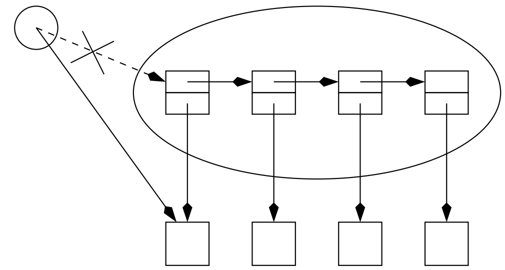

Figure 1: Stack Object With Encapsulated Linked List

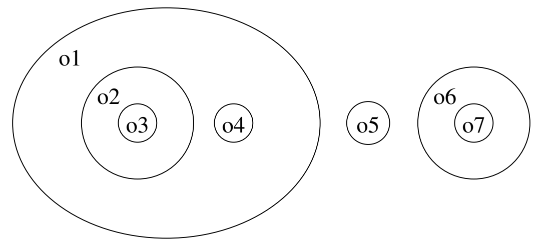

Figure 2: An Object Encapsulation Hierarchy

$A1$ uses object x causing it to be transformed and changing its interface incompatibly; later $A2$ uses y and when TF(y) runs it uses x and encounters the unexpected interface.

For the system to provide good semantics, we must prevent this kind of occurrence. Our approach is based on object encapsulation. Object encapsulation is important in any object-oriented program because it gives programmers the ability to reason locally about program correctness. In our upgrade system, however, object encapsulation provides an additional benefit: it allows our system to support the upgrade modularity conditions without needing versions.

4.1 Object Encapsulation

Reasoning about a class in an object-oriented program involves reasoning about the behavior of objects belonging to the class. Typically objects point to other *subobjects*, which are used to represent the containing object. Local reasoning about class correctness is easy to do if the subobjects are *encapsulated*, that is, if all subobjects are accessible only within the containing object. This condition supports local reasoning because it ensures that outside objects cannot interact with the subobjects without calling methods of the containing object. And therefore the containing object is in control of its subobjects.

However, encapsulation of all subobjects is often more than is needed for local reasoning. Encapsulation is only required for subobjects that the containing object *depends on* [38]:

- An object a *depends on* subobject b if a reads/writes fields of b or calls methods of b and furthermore these reads/writes or calls expose mutable behavior of b in a way that affects the invariants of a.

Thus, a stack implemented using a linked list depends on the list but not on the items contained in the list. If code outside could manipulate the list, it could invalidate the correctness of the stack implementation. But code outside can safely use the items contained in the stack because the stack doesn't call their methods; it only depends on the identities of the items and the identities never change. Similarly, a set does not depend on its elements even if it invokes `a.equals(b)` to ensure that no two elements `a` and `b` in the set are equal, provided the elements are immutable.

In general an object must encapsulate all objects it (directly or transitively) depends on:

- An object x *encapsulates* object y if it maintains an encapsulation boundary such that y and every object it encapsulates is inside the boundary and furthermore if z is outside the boundary, then z cannot access y.

  (An object z *accesses* an object y if z has a pointer to y, or methods of z obtain a pointer to y.)

Figure 1 shows a stack object implemented using a linked list. The nodes in the linked list are encapsulated within the stack object, so that outside objects cannot directly access the list nodes. But the items stored in the stack are not encapsulated in the stack object.

The encapsulation relation forms a hierarchy. Figure 2 shows an example, where objects `o2`, `o3`, and `o4` are encapsulated within `o1`, object `o3` is encapsulated within `o2`, and object `o7` is encapsulated within `o6`.

4.2 Encapsulation and Upgrades

Encapsulation facilitates modular upgrades in our system because it imposes an order on transforms. If y is encapsulated within x, applications must access x before y and therefore x will be transformed before y. This means that when $\mathrm{TF}(x)$ runs it will see the proper interface for y.

However, if $\mathrm{TF}(x)$ accesses some object z referred to by x (directly or indirectly) but not encapsulated within x, it might encounter an unexpected interface or state. Our system provides good semantics without using versions for transforms that don't do this. Such transforms satisfy Condition E:

E. $\mathrm{TF}(x)$ only uses x and objects that x encapsulates.

We say such transforms are *well-behaved.*

Condition E can be checked by informal reasoning, given an understanding of encapsulation boundaries. Understanding these boundaries is a fundamental part of establishing correctness of classes: in general, correctness requires that every object depended on is also encapsulated. Therefore we are not asking more of the programmer than what must already be done. Checking Condition E is also not onerous because the programmer is viewing the transform as an extra method of the old-class. So it is reasonably simple for the programmer to determine whether E holds.

Manual checks can be avoided if the programming language is extended to support *ownership types*; in this case the checking can be done by the compiler. Ownership types [11, 12, 14, 15, 24, 25] are used to declare dependencies: if an object x depends on y, x's class will declare that x owns y. The ownership type system will then guarantee that y cannot be accessed from outside of x, i.e., by objects that x doesn't own directly or transitively. Ownership types combined with effects clauses [44] allow the compiler to track what objects are used by transforms. We have defined an extension to Java [12] that supports ownership types and effects clauses; we include a brief overview of our type system in the appendix. With such a system in place, the compiler can check Condition E at compile time.

4.3 Ensuring Upgrade Modularity

This section shows that our system can ensure Conditions M1-M3, assuming Condition E holds for all TFs.

For any object x affected by an upgrade, our system guarantees that x is accessed before any object encapsulated within x. Thus the system ensures the following conditions:

S1. $\mathrm{TF}(x)$ runs before A uses x or any object encapsulated within x, where A is either an application transaction that ran after TF's upgrade was installed, or A is a transform function from a later upgrade.

S2. If $\mathrm{TF}(x)$ and $\mathrm{TF}(y)$ are in the same upgrade and y is encapsulated within x, then $\mathrm{TF}(x)$ runs before $\mathrm{TF}(y)$.

Now we give informal proofs that when E holds, S1 and S2 ensure that Conditions M1-M3 hold. Our proofs consider only adjacent transactions, but this is sufficient because M1-M3 can be used to reorder sequences containing intervening transactions to achieve adjacency.

M1: If we have $[A;\ \mathrm{TF}(x)]$, where A is either an application transaction that is serialized after TF's upgrade is installed, or A is a transform function from a later upgrade, this has the same effect as $[\mathrm{TF}(x);\ A]$.

Proof: Since A ran before $\mathrm{TF}(x)$, we know from S1 that A does not use x or any object x encapsulates. Furthermore, we know from E that $\mathrm{TF}(x)$ only uses x and objects x encapsulates. Therefore the read/write sets of A and $\mathrm{TF}(x)$ have no object in common and thus the effect is the same as if $\mathrm{TF}(x)$ ran before A. $\square$

M2: If $\mathrm{TF}(x)$ and $\mathrm{TF}(y)$ are from the same upgrade and $\mathrm{TF}(x)$ (transitively) uses y and we have $[\mathrm{TF}(y);\mathrm{TF}(x)]$, this has the same effect as $[\mathrm{TF}(x);\ \mathrm{TF}(y)]$.

Proof: Since $\mathrm{TF}(x)$ (transitively) uses y, we know from E that x encapsulates y. Therefore, we know from S2 that $\mathrm{TF}(x)$ runs before $\mathrm{TF}(y)$. Thus the condition holds trivially because the order $[\mathrm{TF}(y);\ \mathrm{TF}(x)]$ will not occur. $\square$

M3: If $\mathrm{TF}(x)$ and $\mathrm{TF}(y)$ are from the same upgrade and $\mathrm{TF}(x)$ does not (transitively) use y and $\mathrm{TF}(y)$ does not (transitively) use x, then $[\mathrm{TF}(x);\ \mathrm{TF}(y)]$ is equivalent to $[\mathrm{TF}(y);\ \mathrm{TF}(x)]$.

Proof: TF(x) and TF(y) can commute unless there is some object z that is read by one TF and modified by the other. If such an object exists, we know from E that both x and y must encapsulate it. Therefore the existence of z implies that either y is encapsulated within x and z is encapsulated within y, or x is encapsulated within y and z is encapsulated within x. But we know from S2 that an encapsulating object is used before any object it encapsulates. Therefore whichever object encapsulates the other, the TF for that object must use the other before using z, which violates the assumption that neither TF uses the other object. $\square$

When E holds we also get another benefit. Recall from Section 3 that our system will abort an interrupted transaction if it has a read/write conflict with a TF. However, when E holds there will be no conflicts. This is because the interrupted transaction cannot use any object that a pending transform function will use without first causing that pending transform function to run.

4.4 Triggers and Versions

Now we consider what happens when a TF violates Condition E. Condition E states that TF(x) can only use x and objects encapsulated within x. There are two reasons why the condition might not hold.

The first reason is that a TF(x) might use objects that x does not depend on (directly or transitively). For example, the depends-on relation in Section 4.1 is intentionally limited to not include immutable subobjects, since correctness does not require encapsulation of such subobjects. However, if the subobjects are no longer immutable after an upgrade and if a transform function reads such subobjects, Condition E would be violated. But such upgrades are unlikely to happen in practice.

The second reason Condition E may not hold is that an object might not encapsulate subobjects it depends on. This might occur with cyclic objects. It also might occur in the case of iterators [43, 35] and other similar constructs.

Consider, for example, an iterator over a set s. The iterator's job is to return a different element of the set each time its `next` method is called until all elements of the set have been returned. To do this job efficiently, the iterator needs direct access to the objects that represent s, e.g., if s is implemented using a linked list, the iterator must be able to access the nodes in the linked list directly. But the iterator cannot be encapsulated within s because we would like it to be used by objects outside s.

To allow efficient implementation of iterators, a set object does not encapsulate the linked list, even though it depends on it. This is because the iterator is an outside object that can access the list. In fact, what is really happening with iterators is that more than one object depends on some shared subobject. For example, both s and iterators over s depend on the linked list.

Encapsulation violations of this sort do not prevent local reasoning in object-oriented programs, so long as all the code with the shared dependencies is in the same module. If the code is modularized like this, correctness can still be reasoned about locally, by considering the module as a whole. For example, the iterator could be implemented as an inner class of the set class, and modular reasoning would still be possible [12]. However, such encapsulation violations can lead to a violation of E.

4.4.1 Handling Violations of Condition E

When E is violated there are two possible solutions: explicitly order the transform functions so that Conditions M1-M3 are not violated, or use versions. Since the decision about which approach to use requires an understanding of program behavior, the programmer must instruct the system about what to do.

Explicit ordering of transform functions is possible when x and all the unencapsulated objects used by TF(x) are encapsulated within a containing object. For example, suppose the linked list class is being upgraded incompatibly, and as a result a set and all its iterators must also be transformed. If a containing object encapsulates both the set object and its iterator objects, we can force the set and the iterators to be transformed before the linked list by attaching a *trigger* to the class of the containing object.

A trigger is a function that takes an object as an argument and returns a list of objects needing to be upgraded. Triggers are defined as part of an upgrade (in addition to the class-upgrades); such a definition identifies the class being triggered and provides the code for the trigger. The system runs the trigger when an object of the class is first used (after the upgrade is installed); then it processes the list (in list order) and runs any pending transform functions on the objects in the list. The trigger on an object x must be restricted to use only objects x depends on and furthermore to only read those objects. The uses restriction ensures that the trigger itself can be reasoned about modularly; the read-only restriction guarantees that the trigger cannot affect system state. Given these restrictions, triggers provide M1-M3 because they control order: they provide M1 and M2 because [A; TF(x)] and [TF(y); TF(x)] cannot occur.

When there is no containing object, or when there is no way to ensure a correct order for transforms (e.g., because a group of objects with cyclic dependencies is being transformed), we have to fall back on versions. In this case, we keep old versions for any unencapsulated object used by the offending transform function TF(x); for each such object z, we also keep versions for all objects it depends on. Transform functions must be restricted to not modify old versions of objects. Given this restriction on transform functions, versions provide M1-M3, because immutable versions preserve the old interfaces and object states.

Triggers and versions also ensure that a transform does not conflict with interrupted transactions. Furthermore, the system can interact with the user prior to installing an upgrade to help the user include needed triggers and versions. Therefore our system makes it highly unlikely that running a transform will cause interrupted transactions to abort.

4.5 Speeding up Upgrades

Our approach to executing transforms delays their execution until an object is used. We might want to execute them sooner. For example, when a dirty page containing persistent objects is written back to disk, we might want to run pending transforms for objects on that page [5]. Or, we might want a background process to read pages and transform their objects; this might be done in conjunction with garbage collection.

On the other hand, our proofs of Conditions M1-M3 depend on running transforms in the correct order. Specifically we need to be careful about Condition M2: if $\mathrm{TF}(x)$ uses y and y has a pending transform from the same upgrade, we cannot transform y before x. We also cannot transform y before x when the upgrade affecting y is later than the one affecting x; this constraint comes from Condition M1.

Nevertheless we can do some transforms eagerly. This is accomplished by constructing an *upgrade graph*. The graph shows an ordering for old-classes: there is an arrow from old-class C1 to old-class C2 if transforms of C1 objects use C2 objects. Nodes in the graph without in-arrows indicate classes whose objects can be transformed eagerly.

The graph allows opportunistic transforms, e.g., as part of processing an object's page. If the persistent store maintains *extents* (sets that list or contain all objects of a class [20]), this could provide further speed up by allowing the system to find objects eligible for transform. Furthermore, once all objects in a class have been transformed, the node for that class can be removed from the graph, and as a result other classes may become eligible for transform.

5 Implementation

This section describes how we implement upgrades. We describe the general strategy and provide some details of how the implementation works within the Thor object-oriented database. We also sketch an alternative approach that can be used in other persistent object systems.

Thor is a client-server system: Persistent objects reside at servers; application transactions run at client machines on cached copies of persistent objects. Thor uses optimistic concurrency control [1]. Client machines track objects used and modified by a transaction. When a transaction attempts to commit, the client sends a commit request containing information about used objects and states of new and modified objects to one of the servers. The server decides whether the transaction can commit (using two-phase commit if the transaction used objects at more than one server) and informs the client of its decision. More information about Thor can be found in [41, 19, 1, 9].

5.1 Installing Upgrades

Upgrades are installed by interacting with one of the servers. This server checks the upgrade for completeness. It interacts with the user to determine whether Condition E holds; if it doesn't, this may result in a trigger or version being added to the upgrade. If versions are needed, these are also described by class-upgrades, marked as requiring versions. When all needed information has been added to the upgrade, the server notifies clients and other servers about the new upgrade.

Information about transforms, triggers, and versions is attached to class objects of old-classes. E.g., the old-class object points to the transform function.

5.2 Running Upgrades

As mentioned, we interrupt application transactions and transform transactions when we encounter objects that need to be upgraded or have triggers attached to them. This processing involves the following steps:

1. Each time an application transaction, AT, or a transform transaction, TT, uses an object, we check whether that object needs to be transformed or has an attached trigger. If so, we interrupt AT or TT and start a transaction T to run the transform code on that object. This step insures that application code encounters only fully upgraded objects, and pending transforms encounter objects of expected versions.
2. We run transaction T. If T conflicts with an interrupted transaction (reads or modifies a modified object or modifies a read object) we abort all the interrupted transactions including AT.
3. When T completes, we create a version for objects it modifies, if that is indicated.
4. If T has triggered some other transforms we run them provided they are defined by upgrades no later than the upgrade that caused T to run. Note that T is finished executing at this point; we don't interrupt it to run these additional transforms.
5. When there are no triggered transforms left to run, we continue running the interrupted AT or TT, unless these were aborted.
6. When processing is complete (either because the AT is ready to commit, or because the AT was forced to abort), we commit all completed transactions in their completion order. If some transaction's commit fails, we abort those that haven't committed yet and then rerun them in the same order as before. Then we rerun the application transaction if it aborted.

5.3 Implementation in Thor

This section describes an approach that takes advantage of the runtime infrastructure of Thor. We describe an alternate approach in Section 5.4 that does not assume this infrastructure, and thus could be used in any system. More information about the Thor upgrade implementation is contained in [42, 21].

Objects in Thor refer to one another using *orefs* [19]. These are references particular to one of the servers: they identify a page at that server and an object number within that page. Since these references would be expensive to use when running transactions, Thor client machines *swizzle* pointers when they are first used, so that they can be followed efficiently to locate the object being referred to in the client cache. Swizzling is done using an indirection table called the ROT (resident object table). A swizzled pointer points to an entry in the ROT. That entry either points to the object in the client cache, or it is empty.

The most interesting part of the implementation is the technique we use to keep down the cost of Step 1. This step is critical because it requires a test on every method call to determine whether the object whose method is being called is due to be upgraded or has an attached trigger. If the system is to perform well, this test must be inexpensive.

We reduce the cost of this test by maintaining the following invariant:

R1. While an application transaction is running, all non-empty entries in the ROT are for up-to-date objects whose triggers already ran.

This invariant means that while we are running an application transaction, we discover upgrades and triggers when we fill empty ROT entries. ROT entries are filled less often than they are used; therefore we avoid the need to test for upgrades and triggers in the normal case of running method calls on objects that are already in the ROT. As part of filling a ROT entry we look at the object's class object; this is how we discover a pending transform.

The client machine establishes R1 as follows. When it learns of a new upgrade, it clears all ROT entries for objects of old-classes of the upgrade, and it aborts the current transaction if it used objects of these classes. This processing is expensive but the expense is acceptable because upgrades are installed infrequently, e.g., no more than once an hour or once a day.

The invariant is not enough to ensure correct behavior while running transforms, however, because objects in the ROT may be too recent for the transform (i.e., already transformed due to a later upgrade). In this case, if the object is versioned the transform needs to find the appropriate earlier version to use. Therefore, as part of making a method call we test whether we are running an application transaction or a transform transaction. This test is fast: it involves looking at a boolean variable that, because it is used so frequently, ends up in a register or the fastest hardware cache. If the test indicates that we are running a transform transaction, the system does extra processing to find the version if one is needed.

When a transform transaction completes, there might be some old versions in the ROT that were placed there because they were needed to run the transform. These old versions are removed from the ROT at that point.

5.3.1 Versions

When a transform transaction commits, the system determines whether a version is needed. If not, it stores the new object in the same page as the old one if there is room. The new object can actually be bigger than the old one and yet be stored in the same page because orefs are logical, not physical, and because the ROT allows the client machine to move objects around in the cache. If the new version is too big to fit in the page, the original object is changed to a special small surrogate object that points to the new object. If a version is needed, the original object is changed to a surrogate that points to both the old and new versions; the new version is placed in the object's page if possible.

5.3.2 Commits

As part of committing an application transaction, the client sends the servers information about all objects that were transformed during processing of that transaction. For each transformed object it sends the new state, plus the state of the surrogate if one is needed. (Only objects are sent and not their containing pages because Thor uses object-shipping [19].)

5.4 Alternative Approach

The approach described above requires the Thor infrastructure. In particular it relies on the fact that there is an indirection table that can be cleared when the client machine learns of a new upgrade. Many systems will not have a ROT and therefore require a different approach. This section sketches a technique that will work in any environment.

The approach is quite straightforward. Objects typically point to a dispatch vector containing an entry for each of their methods. All objects of the class point to the same dispatch vector. We take advantage of the dispatch vector to handle upgrades.

When a class is due for an upgrade, e.g., it is an old-class, as part of installing the upgrade we modify its dispatch vector to point to special versions of the methods. When such a method runs, we know the object is due to be upgraded (or has an attached trigger that needs to run). The method carries out the check in Step 1: it checks the boolean to determine whether the call is coming from an application transaction or a transform transaction. In either case it proceeds as discussed in Section 5.3.

Additionally, methods of upgraded objects with older versions also do the check, and if they are called by a transform, they carry out the processing to get to the right version.

This approach of using dispatch vectors avoids some version checks done by our implementation: checks are done only in objects due to be upgraded, and in upgraded objects where there are earlier versions.

6 Performance

To evaluate our approach we extended the Thor system to support upgrades and conducted a performance study. The goal of the study was to evaluate the overhead imposed by the upgrade infrastructure on application performance.

Thor is a good basis for this study because it performs well. Earlier studies showed that it delivers comparable performance to a highly-optimized persistent object system implemented in C++ even though the C++ system did not support transactions [41].

We evaluated two kinds of overhead, the *baseline overhead* and the *transform overhead*. The baseline overhead is our main concern. This is the overhead that occurs during normal case execution, when the system does not encounter any objects that need to be upgraded; the overhead is due to checks used to determine whether an upgrade is needed. The transform overhead is the overhead for running transforms. Every upgrade system has such costs; the only concern is whether our transform overhead is reasonable.

We evaluated the overhead by comparing the performance of an application running in two systems, the original Thor system prototype (*ThorBase*), and the prototype that supports upgrades (*ThorUpgrades*). The prototype does not include upgrade installation at the server and subsequent notification to the client. Instead, we configured the client with a sequence of "dormant" upgrades and activated upgrades while running applications.

Our application workloads are based on the single-user OO7 Benchmark [18]; this benchmark is intended to capture the characteristics of various CAD applications, but does not model any specific application. We used OO7 because it is a standard benchmark for measuring object storage system performance. The OO7 database contains a tree of *assembly* objects with leaves pointing to three *composite* parts chosen randomly from among 500 such objects. Each composite part contains a graph of *atomic parts* linked by bidirectional *connection* objects, reachable from a single *root* atomic part; each atomic part has three connections. We used the small OO7 database configuration, where each composite part contains 20 atomic parts. This is a very small database (where a stop-the-world approach would be acceptable) but it is sufficient to allow us to measure the baseline and transform overheads. Our workload choice is conservative for our overhead study because OO7 accesses a large number of small objects and we expect the overhead to be proportional to the number of objects accessed.

We considered both read-only and read-write transaction workloads in our analysis, since upgrades have a different commit cost in workloads with and without modifications. We used the read-only T1 dense traversal, which performs a depth-first traversal of the entire composite part graph (touching every atomic part), and the read-write T2b traversal, which performs a T1 traversal modifying all atomic parts.

Our experiments used a system configuration with a single client and a single server, running on the same machine. The test machine had a 600MHz Intel Pentium III processor and 512MB of memory, and ran Linux 2.4.7.

6.1 Baseline

This section considers the baseline overhead imposed by the upgrade infrastructure when the system does not encounter objects that need to be upgraded. The baseline experiments evaluate two types of application accesses: fast access to an object already installed in the ROT and the slower access to an object that needs to be installed in the ROT. The upgrade infrastructure introduces a small extra cost for an access of an object resident in the ROT, namely a check of the global flag that indicates whether the current transaction is an application or transform transaction. For an access of an object that isn't resident in the ROT, the upgrade code introduces a larger extra cost, which includes the expense of checking if a trigger or transform needs to be run for the object.

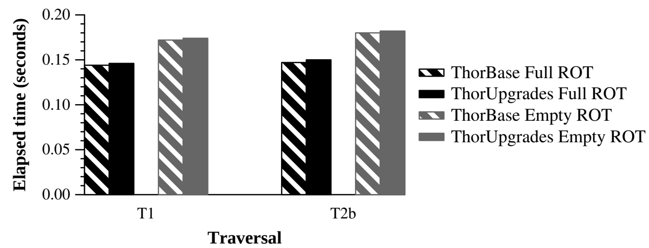

Figure 3: Baseline (no-upgrade) traversal times

To evaluate these costs, we compared application execution times for ThorBase and ThorUpgrades in two environments: with a fully-populated ROT, and with an initially empty ROT. The full ROT comparison is the expected case for most application executions. Conversely, the empty ROT comparison represents a worst case for baseline performance in ThorUpgrades: the maximum amount of work must be performed for each nonresident object access. This is a case that we would not expect to occur normally, as typically only a few empty ROT entrees are encountered when a transaction runs.

Figure 3 shows the execution times for these experiments. The figure shows the costs of running traversals T1 and T2b. We did not include the cost of committing these transactions since we wanted to focus on the extra costs in ThorUpgrades for running the traversals. Also the experiments use a hot cache, since otherwise the cost of fetching objects into the cache would dominate execution time. In either case, the upgrade infrastructure introduces a minimal overhead that is less than 2%.

The overhead would be similar if we use the dispatch vector approach presented in Section 5.4.

6.2 Executing Upgrades

The next experiments measure the transform overhead. We installed an upgrade and ran a database traversal that encounters objects that need to be transformed. The specific upgrade used in these experiments upgrades the atomic part class. Because the goal was to measure the transform overhead, the upgrade uses a trivial transform that minimizes application-specific costs (transforms defined by programmers might do other computations). The new atomic part class has the same methods and fields as the old class, and the transform just copies the fields from the old object to the new. As before, all experiments used a hot cache.

To estimate the transform cost we compared the traversal execution times of ThorUpgrades without upgrades and ThorUpgrades with an atomic part upgrade installed just before the traversal. In these experiments all traversals visit each atomic part multiple times; the version check and transform cost is incurred only on the first visit. By counting the number of objects transformed in traversals T1 (read-only) and T2b (read-write) (9,880 in both cases) we computed the average cost of running a transform in both types of traversals; this is 11.3 μsec for T1 and 11.5 μsec for T2b. This cost includes the processing to start and end a transform as well as the cost to check whether the transform conflicts with the interrupted transaction (Steps 1-3 in Section 5.2). The cost is an over-estimate of the overhead since it includes both the overhead and the running of the transform.

The above calculation does not reflect the entire cost of upgrades because it does not include the cost of committing the transaction that activates the upgrades. There is a commit cost associated with each transformed object. The cost is low if the object was modified by the application transaction (assuming the new object fits in its page and does not require a version), since the application's modifications have to be shipped to the server anyway. Transformed objects not modified by the transaction incur an additional cost since they must be sent to the server in the commit message.

To estimate the average extra commit cost we compared the commit times for T1 and T2b in ThorUpgrades with and without upgrades. Since T1 is read-only, each transformed object incurs the extra commit cost of being sent to the server; the average commit cost per upgraded object is 19.9 μsec. T2b modifies every transformed object and the additional commit cost per upgraded object is 1.0 μsec.

Time per object (μsec)	T1	T2b
Transform	11.3	11.5
Commit	19.9	1.0

Table 1: Extra Upgrade Cost

Table 1 summarizes the extra upgrade costs. The results indicate that our overhead per transform is reasonable. It reflects costs that are incurred by every lazy upgrade system. Each such system must identify objects needing to be transformed, run the transforms, and commit the changes.

7 Related Work

There has been much research on software upgrades and data transformation covering a broad range of research topics. The work on schema or class versioning (e.g., [26, 49, 22]) considers multiple co-existing versions of a schema or class. The work on object instance evolution (e.g., [8, 31]) considers selective transformation of some but not all objects in a class. The work on hot-swapping of modules (e.g., [36, 32, 37]) is concerned with updating a class while there is executing code that is using objects of the class; this work considers issues of type safe access to the same object via multiple potentially incompatible interfaces but does not enforce the upgrade modularity conditions that allow programmers to reason locally about the correctness of their upgrades.

Here we focus on work on schema evolution in persistent object stores (such as object-oriented databases), since this is the work most closely related to our own. In these systems the database has one logical schema to which modifications of class definitions are applied; all object instances are converted (eagerly or lazily, but once and forever) to conform to the latest schema. The schema evolution approach is used in Orion [7], OTGEN [40], O2 [29, 52], GemStone [17, 48], Objectivity/DB [47], Versant [50], and PJama [6, 5] systems, and is the only approach available in commercial RDBMS. An extensive survey of the previous schema evolution systems can be found in [30].

None of the previous schema evolution systems provide a way of executing upgrades efficiently both in space and time, while allowing programmers to reason locally about the correctness of their upgrades. To be practical, systems must avoid keeping old versions of objects most of the times. Some earlier systems [48, 7, 40] avoid versions by severely limiting the expressive power of upgrades (e.g., transform functions are not allowed to make method calls). Others [5, 46] limit the number of versions using a stop-the-world approach that shuts down the system for upgrade and discards the versions when the upgrade is complete

Very few systems support lazy conversion and *complex* (fully expressive) transforms. The work on O2 [29, 52] was the first to identify the problem posed by deferred complex transforms and incompatible upgrades. This work introduced an upgrade modularity condition that is based on the equivalence of lazy and eager conversion. This condition is weaker than our Conditions M1-M3 because it does not consider the interleavings of transforms from the same upgrade.

O2 ensures type safety for deferred complex transforms using a "screening" approach similar to versioning. Unlike our approach, however, analysis in O2 does not take encapsulation into account. When an incompatible upgrade occurs after a complex transform is installed, O2 either activates an eager conversion or avoids transform interference by keeping versions for all objects. This approach is unnecessarily conservative (it switches to eager execution even when E holds). Also, O2 does not solve the problem of applications modifying objects that are then used by transforms from earlier upgrades; this is unsafe because it violates Condition M1.

Implementation details for commercial systems supporting lazy conversion with complex transforms are generally not available. We found limited information for O2, e.g., we found no information about the mechanisms for supporting the atomicity of individual transforms, or about the performance impact of upgrade support on normal case operation. The O2 screening approach co-locates versions of upgraded objects physically near the new version of the object [33]. This requires database reorganization when versions are created. In contrast, our system does not require co-location of object versions; this allows us to preserve clustering of non-upgraded objects without database reorganization and furthermore, we are often able to preserve clustering for upgraded objects as well. Preserving clustering is important for system performance because of its impact on disk access [34].

Some implementation issues caused by complex user-defined transforms arise in eager as well as lazy systems, e.g., either has to support arbitrary order of transforms and access to potentially incompatible transformed objects. The PJama system [5, 30] keeps old and new versions to solve this problem. To provide recoverability and reduce memory demands when converting large datasets, it performs incremental partitioned conversion that creates partitions with old and new versions, and at the end of conversion deletes the old copies by copying the converted partitions. Like our system, PJama uses write-ahead logging to support conversion atomicity and recoverability.

8 Conclusions

Persistent object stores provide a simple yet powerful programming model that allows applications to store objects reliably so that they can be used again later and shared with other applications. Providing a satisfactory way of upgrading objects in a persistent object store has been a long-standing challenge. Upgrades must be performed in a way that is efficient both in space and time, and that does not stop application access to the store. In addition, however, the approach must be modular: it must allow programmers to reason locally about the correctness of their upgrades similar to the way they would reason about regular code. This paper provides solutions to both problems.

This paper defines *upgrade modularity conditions* that any upgrade system must satisfy to support local reasoning about upgrades. These conditions are more general than earlier definitions [52]: they apply to both lazy and stop-the-world upgrade systems; they also apply to both systems that use versions and systems that don't.

The paper also describes a new approach for executing upgrades efficiently while satisfying the upgrade modularity conditions. The approach exploits object encapsulation properties in a novel way. The paper proves that our upgrade system satisfies the upgrade modularity conditions when transforms are well-behaved. We also show that the conditions hold through the use of triggers and versions.

The paper describes a prototype implementation that supports fully expressive, modular, lazy upgrades. The implementation is done in Thor [41, 9]. The paper also describes an alternate implementation approach that can be used in any persistent object system.

The paper presents results of a performance study indicating the infrastructure has low cost. It has negligible impact on applications that do not use objects that need to be upgraded. We expect this to be the common case because upgrades are likely to be rare (e.g., once a week or once a day). The results also show that when upgrades are needed, the overhead of transforming an object is small.

Thus the paper describes a complete solution to the problem of upgrading persistent objects.

Acknowledgments

We are grateful to Dorothy Curtis for helping us with the implementation of the upgrade system. We would also like to thank Daniel Jackson, Viktor Kuncak, Butler Lampson, and the anonymous referees for providing useful comments.

References

[1] A. Adya, R. Gruber, B. Liskov, and U. Maheshwari. Efficient optimistic concurrency control using loosely synchronized clocks. In *ACM SIGMOD International Conference on Management of Data*, May 1995.

[2] O. Agesen, S. N. Freund, and J. C. Mitchell. Adding type parameterization to the Java language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1997.

[3] J. Aldrich, V. Kostadinov, and C. Chambers. Alias annotations for program understanding. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[4] M. P. Atkinson, F. Bancilhon, D. J. DeWitt, K. R. Dittrich, D. Maier, and S. B. Zdonik. The object-oriented database system manifesto. In *SIGMOD Conference*, May 1990.

[5] M. P. Atkinson, M. A. Dmitriev, C. Hamilton, and T. Printezis. Scalable and recoverable implementation of object evolution for the PJama 1 platform. In *Persistent Object Systems (POS)*, September 2000.

[6] M. P. Atkinson, M. J. Jordan, L. Daynes, and S. Spence. Design issues for persistent Java: A type-safe, object-oriented, orthogonally persistent system. In *Persistent Object Systems (POS)*, May 1996.

[7] J. Banerjee, W. Kim, H. Kim, and H. F. Korth. Semantics and implementation of schema evolution in object-oriented databases. In *ACM SIGMOD International Conference on Management of Data*, May 1987.

[8] E. Bertino, G. Guerrini, and L. Rusca. Object evolution in object databases. In *B. Franhoefer and R. Pareschi, editors, Dynamic Worlds, Kluwer Academic Publishers*, 1999.

[9] C. Boyapati. JPS: A distributed persistent Java system. SM thesis, Massachusetts Institute of Technology, September 1998.

[10] C. Boyapati, R. Lee, and M. Rinard. Safe runtime downcasts with ownership types. In *ECOOP International Workshop on Aliasing, Confinement and Ownership in Object-Oriented Programming (IWACO)*, July 2003.

[11] C. Boyapati, R. Lee, and M. Rinard. Ownership types for safe programming: Preventing data races and deadlocks. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[12] C. Boyapati, B. Liskov, and L. Shrira. Ownership types for object encapsulation. In *Principles of Programming Languages (POPL)*, January 2003.

[13] C. Boyapati, B. Liskov, and L. Shrira. Ownership types and safe lazy upgrades in object-oriented databases. Technical Report TR-858, MIT Laboratory for Computer Science, July 2002.

[14] C. Boyapati and M. Rinard. A parameterized type system for race-free Java programs. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2001.

[15] C. Boyapati, A. Salcianu, W. Beebee, Jr., and M. Rinard. Ownership types for safe region-based memory management in Real-Time Java. In *Programming Language Design and Implementation (PLDI)*, June 2003.

[16] G. Bracha, M. Odersky, D. Stoutamire, and P. Wadler. Making the future safe for the past: Adding genericity to the Java programming language. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[17] R. Bretl et al. The GemStone data management system. In W. Kim and F. H. Lochovsky, editors, *Object-Oriented Concepts, Databases, and Applications*. 1989.

[18] M. J. Carey, D. J. Dewitt, C. Kant, and J. F. Naughton. A status report on the OO7 OODBMS benchmarking effort. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1994.

[19] M. Castro, A. Adya, B. Liskov, and A. C. Myers. HAC: Hybrid adaptive caching for distributed storage systems. In *Symposium on Operating System Principles (SOSP)*, October 1997.

[20] R. Catell, editor. *The Object Database Standard: ODMG 3.0.* Morgan Kaufmann, 2000.

[21] Y. Cheung. Lazy schema evolution in object-oriented databases. SM thesis, Massachusetts Institute of Technology, September 2001.

[22] S. M. Clamen. Type evolution and instance adaptation. Technical Report CMU-CS-92-133, Carnegie Mellon University, June 1992.

[23] D. G. Clarke. Object ownership and containment. PhD thesis, University of New South Wales, Australia, July 2001.

[24] D. G. Clarke and S. Drossopoulou. Ownership, encapsulation and disjointness of type and effect. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 2002.

[25] D. G. Clarke, J. M. Potter, and J. Noble. Ownership types for flexible alias protection. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1998.

[26] V. M. Crestana-Jensen, A. J. Lee, and E. A. Rundensteiner. Consistent schema version removal: An optimization technique for object-oriented views. In *IEEE Transactions on Knowledge and Data Engineering (TKDE) 12(2)*, March 2000.

[27] M. Day, R. Gruber, B. Liskov, and A. C. Myers. Subtypes vs. where clauses: Constraining parametric polymorphism. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1995.

[28] C. Delcourt and R. Zicari. The design of an integrity consistency checker (ICC) for an object-oriented database system. In *European Conference for Object-Oriented Programming (ECOOP)*, July 1991.

[29] O. Deux et al. The story of O2. In *IEEE Transactions on Knowledge and Data Engineering (TKDE) 2(1)*, March 1990.

[30] M. A. Dmitriev. Safe class and data evolution in large and long-lived Java applications. Technical Report TR-2001-98, Sun Microsystems, August 2001.

[31] S. Drossopoulou, F. Damiani, M. Dezani-Ciancaglini, and P. Giannini. Fickle : Dynamic object re-classification. In *ECOOP*, 2001.

[32] D. Duggan. Type-based hot swapping of running modules. Technical Report SIT CS 2001-7, Stevens Institute of Technology, Hoboken, NJ 07030, October 2001.

[33] F. Ferrandina and G. Ferran. Schema and database evolution in the O2 Object Database System. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB)*, 1995.

[34] F. Ferrandina, T. Meyer, and R. Zicari. Measuring the Performance of Immediate and Deferred Updates in Object Database Systems. In *Proceedings of the OOPSLA Workshop on Object Database Behavior, Benchmarks and Performance*, 1995.

[35] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[36] M. Hicks, J. Moore, and S. Nettles. Dynamic software updating. In *Programming Language Design and Implementation (PLDI)*, June 2001.

[37] G. Hjalmtysson and R. Gray. Dynamic C++ classes - a lightweight mechanism to update code in running program. In *USENIX Annual Technical Conference*, June 1998.

[38] K. R. M. Leino and G. Nelson. Data abstraction and information hiding. Research Report 160, Compaq Systems Research Center, November 2000.

[39] K. R. M. Leino, A. Poetzsch-Heffter, and Y. Zhou. Using data groups to specify and check side effects. In *Programming Language Design and Implementation (PLDI)*, June 2002.

[40] B. S. Lerner and A. N. Habermann. Beyond schema evolution to database reorganization. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1990.

[41] B. Liskov, M. Castro, L. Shrira, and A. Adya. Providing persistent objects in distributed systems. In *European Conference for Object-Oriented Programming (ECOOP)*, June 1999.

[42] B. Liskov, C. Moh, S. Richman, L. Shrira, Y. Cheung, and C. Boyapati. Safe lazy software upgrades in object-oriented databases. Technical Report TR-851, MIT Laboratory for Computer Science, June 2002.

[43] B. Liskov, A. Snyder, R. R. Atkinson, and C. Schaffert. Abstraction mechanisms in CLU. In *Communications of the ACM (CACM) 20(8)*, August 1977.

[44] J. M. Lucassen and D. K. Gifford. Polymorphic effect systems. In *Principles of Programming Languages (POPL)*, January 1988.

[45] A. C. Myers, J. A. Bank, and B. Liskov. Parameterized types for Java. In *Principles of Programming Languages (POPL)*, January 1997.

[46] Object Design Inc. *ObjectStore Advanced C++ API User Guide Release 5.1*, 1997.

[47] Objectivity Inc. *Objectivity Technical Overview, Version 6.0*, 2001.

[48] D. J. Penney and J. Stein. Class modification in the GemStone object-oriented DBMS. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 1987.

[49] A. H. Skarra and S. B. Zdonik. The management of changing types in an object-oriented database. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, November 1986.

[50] Versant Object Technology. *Versant User Manual*, 1992.

[51] M. Viroli and A. Natali. Parametric polymorphism in Java: An approach to translation based on reflective features. In *Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, October 2000.

[52] R. Zicari. A framework for schema updates in an object-oriented database systems. In *International Conference on Data Engineering (ICDE)*, April 1991.

O1. Every object has an owner.

O2. The owner can either be another object or world.

O3. The owner of an object does not change over time.

O4. The ownership relation forms a tree rooted at world.

Figure 4: Ownership Properties

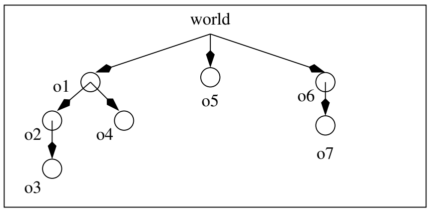

Figure 5: An Ownership Relation

Appendix

A Ownership Types

Ownership types provide a statically enforceable way of specifying object encapsulation. The idea is that an object can *own* subobjects it depends on, thus preventing them from being accessible outside. Ownership types can also be used to statically check Condition E presented in Section 4. This section presents an overview of our type system; more details can be found in [12].

The key to the type system is the concept of object ownership. Every object has an owner. The owner can either be another object or a special owner called world. Our type system statically guarantees the ownership properties shown in Figure 4. Figure 5 presents an example ownership relation. We draw an arrow from x to y if x owns y. In the figure, the special owner world owns objects o1, o5, and o6; o1 owns o2 and o4; o2 owns o3; and o6 owns o7.

Ownership allows a program to statically declare encapsulation boundaries that capture dependencies:

- An object must own all the objects it depends on.

The system then enforces encapsulation: if y is inside the encapsulation boundary of z and x is outside, then x cannot access y. (An object x *accesses* an object y if x has a pointer to y, or methods of x obtain a pointer to y.) In Figure 5, o7 is inside the encapsulation boundary of o6 and o1 is outside, so o1 cannot access o7. An object is only allowed to access: 1) itself and objects it owns, 2) its ancestors in the ownership tree and objects they own, and 3) globally accessible objects, namely objects owned by world.[1] Thus, o1 can access all objects in the figure except for o3 and o7.

[1]Note the analogy with nested procedures: proc P_1 {var x_2; proc P_2 {var x_3; proc P_3 {...}}}. Say x_{n+1} and P_{n+1} are children of P_n. P_n can only access: 1) P_n and its children, 2) the ancestors of P_n and their children, and 3) global variables and procedures.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;

  void push(T<TOwner> value) {
    TNode<this, TOwner> newNode =
      new TNode<this, TOwner>(value, head);
    head = newNode;
  }
  T<TOwner> pop() {
    if (head == null) return null;
    T<TOwner> value = head.value();  head = head.next();
    return value;
  }
}

class TNode<nodeOwner, TOwner> {
  TNode<nodeOwner, TOwner> next;  T<TOwner> value;

  TNode(T<TOwner> v, TNode<nodeOwner, TOwner> n) {
    this.value = v;  this.next  = n;
  }
  T<TOwner> value()               { return value; }
  TNode<nodeOwner, TOwner> next() { return next;  }
}

class T<TOwner> { }

class TStackClient<clientOwner> {
  void test() {
    TStack<this,  this>  s1 = new TStack<this,  this> ();
    TStack<this,  world> s2 = new TStack<this,  world>();
    TStack<world, world> s3 = new TStack<world, world>();
 /* TStack<world, this>  s4 = new TStack<world, this> (); */
}}
```

Figure 6: Stack of T Objects

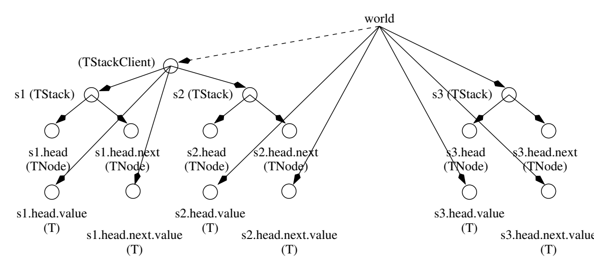

Figure 7: Ownership Relation for TStacks s1, s2, s3

A.1 Owner Polymorphism

We present our type system in the context of a Java-like language augmented with ownership types. Every class definition is parameterized with one or more owners. The first owner parameter is special: it identifies the owner of the corresponding object. The other owner parameters are used to propagate ownership information. Parameterization allows programmers to implement a generic class whose objects have different owners.

An owner can be instantiated with this, with world, or with another owner parameter. Objects owned by this are encapsulated objects that cannot be accessed from outside. Objects owned by world can be accessed from anywhere.

Figure 6 shows an example.[2] A TStack is a stack of T objects. It is implemented using a linked list. The TStack

[2]The example shows type annotations written explicitly. However, many of them can be automatically inferred. See [14, 11] for details.

```
class C<cOwner, sOwner, tOwner> where (sOwner <= tOwner) {
    ...
    TStack<sOwner, tOwner> s;
}
```

Figure 8: Using Where Clauses to Constrain Owners

class is parameterized by stackOwner and TOwner. stackOwner owns the TStack object; TOwner owns the T objects contained in the TStack. The code specifies that the TStack object owns the nodes in the list; therefore the list nodes cannot be accessed from outside the TStack object.

The type of TStack s1 is instantiated using this for both the owner parameters. This means that TStack s1 is owned by the TStackClient object that created it and so are the T objects in s1. TStack s2 is owned by the TStackClient object, but the T objects in s2 are owned by world. TStack s3 is owned by world and so are the T objects in s3. The ownership relation for s1, s2, and s3 is depicted in Figure 7 (assuming the stacks contain two elements each). (The dotted line indicates that every object is transitively owned by world.)

A.2 Constraints on Owners

For every type $T\langle x_1, ..., x_n\rangle$ with multiple owners, our type system statically enforces the constraint that $(x_1 \preceq x_i)$ for all $i \in \{1..n\}$. Recall from Figure 4 that the ownership relation forms a tree rooted at world. The notation $(y \prec z)$ means that y is a descendant of z in the ownership tree. The notation $(y \preceq z)$ means that y is either the same as z, or y is a descendant of z in the ownership tree. Thus, the type of TStack s4 in Figure 6 is illegal because (world $\not\preceq$ this). For a method $m\langle x_{n+1}, ..., x_k\rangle(...)\{...\}$ of an object of type $T\langle x_1, ..., x_n\rangle$, the restriction is that $(x_1 \preceq x_i)$ for all $i \in \{1..k\}$. (These constraints are needed to provide encapsulation in the presence of subtyping. [11] illustrates this point with an example.)

To check ownership constraints modularly, it is sometimes necessary for programmers to specify additional constraints on class and method parameters. For example, in Figure 8, the type of s is legal only if (sOwner $\preceq$ tOwner). We allow programmers to specify such additional constraints using where clauses [27, 45], and our type system enforces the constraints. For example, in Figure 8, class C specifies that (sOwner $\preceq$ tOwner). An instantiation of C that does not satisfy the constraint is illegal.

A.3 Subtyping

The rule for declaring a subtype is that the first owner parameter of the supertype must be the same as that of the subtype; in addition, of course, the supertype must satisfy the constraints on owners. The first owners have to match because they are special, in that they own the corresponding objects. Thus, TStack⟨stackOwner, TOwner⟩ is a subtype of Object⟨stackOwner⟩. But T⟨TOwner⟩ is not a subtype of Object⟨world⟩ because the first owners do not match.

A.4 Inner Classes

Our inner classes are similar to the member inner classes in Java. Inner class definitions are nested inside other classes.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  TStackEnum<enumOwner, TOwner> elements<enumOwner>()
    where (enumOwner <= TOwner) {
    return new TStackEnum<enumOwner, TOwner>();
  }
  class TStackEnum<enumOwner, TOwner>
    implements TEnumeration<enumOwner, TOwner> {

    TNode<TStack.this, TOwner> current;

    TStackEnum() { current = TStack.this.head; }

    T<TOwner> getNext() {
      if (current == null) return null;
      T<TOwner> t = current.value();
      current = current.next();
      return t;
    }
    boolean hasMoreElements() { return (current != null); }
  }
}

class TStackClient<clientOwner> {
  void test() {
    TStack<this, world> s = new TStack<this, world>();
    TEnumeration<this,  world> e1 = s.elements();
    TEnumeration<world, world> e2 = s.elements();
}}

interface TEnumeration<enumOwner, TOwner> {
  T<TOwner> getNext();
  boolean   hasMoreElements();
}
```

Figure 9: TStack With Iterator

Figure 9 shows an example. The inner class TStackEnum implements an iterator for TStack; the elements method of TStack provides a way to create an iterator over the TStack. The TStack code is otherwise similar to that in Figure 6.

Recall from before that an owner can be instantiated with this, with world, or with another owner parameter. Within an inner class, an owner can also be instantiated with C.this, where C is an outer class. This feature allows an inner object to access the objects encapsulated within its outer objects. In Figure 9, the owner of the current field inTStackEnum is instantiated with TStack.this. The current field accesses list nodes encapsulated within its outer TStack object.

An inner class is parameterized with owners just like a regular class. In our system, the outer class parameters are not automatically visible inside an inner class. If an inner class uses an outer class parameter, it must explicitly include the outer class parameter in its declaration. In Figure 9, the TStackEnum declaration includes the owner parameter TOwner from its outer class. TOwner is therefore visible inside TStackEnum. But the TStackEnum declaration does not include stackOwner. Therefore, stackOwner is not visible inside TStackEnum.

Note that in this example, the elements method is parameterized by enumOwner. This allows a program to create different iterators that have different owners. elements returns an iterator of type TStackEnum⟨enumOwner, TOwner⟩. For this type to be legal, it must be the case that (enumOwner $\preceq$ TOwner). This requirement is captured in the where clause.

```
class TStack<stackOwner, TOwner> {
  TNode<this, TOwner> head = null;
  ...
  class TStackEnum<enumOwner, TOwner>
   implements TEnumeration<enumOwner, TOwner> {

    TNode<TStack.this, TOwner> current;
    ...
    T<TOwner> getNext() writes(this) reads(TStack.this){...}
    boolean   hasMoreElements()      reads(this){...}
  }
}

interface TEnumeration<enumOwner, TOwner> {
  T<TOwner> getNext() writes(this) reads(world);
  boolean   hasMoreElements()      reads(this);
}
```

Figure 10: TStack Iterator With Effects

A.5 Encapsulation Theorem

Our system provides the following encapsulation property:

THEOREM 1. *x can access an object owned by o only if:*

1. *$(x \preceq o)$, or*
2. *x is an inner class object of o.*

PROOF. Consider the code: *class* $C\langle f, ...\rangle\{... \; T\langle o, ...\rangle \; y \; ...\}$. Variable y of type $T\langle o, ...\rangle$ is declared within the static scope of class C. Owner o can therefore be either 1) this, or 2) world, or 3) a formal class parameter, or 4) a formal method parameter, or 5) C'.this, where C' is an outer class. We will show that in the first four cases, the constraint (this $\preceq o$) holds. In the first two cases, the constraint holds trivially. In the last two cases, $(f \preceq o)$ and (this $\prec f$), so the constraint holds. In the fifth case, (C'.this $= o$). Therefore an object x of a class C can access an object y owned by o only if either 1) $(x \preceq o)$, as in the first four cases, or 2) x is an inner object of o, as in the fifth case. ☐

A.6 Effects Clauses

Our system also contains effects clauses [44] because they are useful for specifying assumptions that hold at method boundaries and enable modular reasoning and checking of programs. We also use effects with ownership types to check Condition E described in Section 4.

Our system allows programmers to specify *reads* and *writes* clauses. Consider a method that specifies that it writes $(w_1, ..., w_n)$ and reads $(r_1, ..., r_m)$. The method can write an object x (or call methods that write x) only if $(x \preceq w_i)$ for some $i \in \{1..n\}$. The method can read an object y (or call methods that read y) only if $(y \preceq w_i)$ or $(y \preceq r_j)$, for some $i \in \{1..n\}$, $j \in \{1..m\}$. We thus allow a method to both read and write objects named in its writes clause.

Figure 10 shows a TStack iterator that uses effects, but is otherwise similar to the TStack iterator in Figure 9. In the example, the hasMoreElements method reads the this object. The getNext method reads objects owned by TStack.this and writes (and reads) the this object.

When effects clauses are used in conjunction with subtyping, the effects of an overridden method must subsume the effects of the overriding method. This sometimes makes it difficult to specify precisely all the effects of a method. For example, it is difficult to specify precisely all the read effects in the getNext method of the TEnumeration class because TEnumeration is expected to be a supertype of subtypes like TStackEnum and TEnumeration cannot name the specific objects used in the getNext methods of these subtypes. To accommodate such cases, we allow an escape mechanism, where a method can include world in its effects clauses.

```
class IntVector<vOwner> {
  int  elementCount = 0;
  int  size()     reads (this) { return elementCount; }
  void add(int x) writes(this) { elementCount++;  ... }
}
class IntStack<sOwner> {
  IntVector<this> vec = new IntVector<this>();
  void push(int x) writes (this) { vec.add(x); }
}
void m<sO,vO> (IntStack<sO> s, IntVector<vO> v)
  writes (s) reads (v) where !(v <= s) !(s <= v) {
  int n = v.size(); s.push(3); assert(n == v.size());
}
```

Figure 11: Reasoning About Aliasing and Side Effects

Ownership types and effects can be used to locally reason about the side effects of method calls in the presence of subtyping in object-oriented languages. Consider, for example, the code in Figure 11, which shows an IntStack implemented using an IntVector vec. (We adopted this example from [39].) The example has a method m that receives two arguments: an IntStack s and an IntVector v. The condition in the assert statement in m can be true only if v is not aliased to s.vec. In the example, the method m uses a where clause to specify that (v $\not\preceq$ s) and (s $\not\preceq$ v). Since the ownership relation forms a tree (see Figure 4), this constraint implies that v cannot be aliased to s.vec. Furthermore, IntVector.size declares that it only reads objects owned by the IntVector, and IntStack.push declares that it only writes (and reads) objects owned by the IntStack. Therefore, it is possible to reason locally that v.size and s.push cannot interfere, and thus the condition in the assert statement in m must be true.

A.7 Support for Modular Upgrades

Ownership types can be used to statically check a property similar to Condition E in Section 4: they can be used to check whether a transform function uses only owned objects. Also, our system handles inner classes specially to ensure Condition S1 and S2 discussed in Section 4. When a upgrade affects a class, we attach triggers to its inner classes; this is done automatically as part of installing the upgrade. Then when an inner class object is used, the trigger causes the outer object to be transformed.

A.8 Other Applications

Ownership-based type systems have also been used to prevent data races [14] and deadlocks [11] in multithreaded programs, to prevent memory errors [15] in programs that use region-based memory management, and to aid program understanding [3]. Since ownership types require little programming overhead, their type checking is fast and scalable, and they provide several benefits, they offer a promising approach to making object-oriented programs more reliable.