Dynamic Replacement of Active Objects in the Gilgul Programming Language

Pascal Costanza

University of Bonn, Institute of Computer Science III,
Römerstr. 164, D-53117 Bonn, Germany,
costanza@cs.uni-bonn.de, http://www.pascalcostanza.de

Abstract. Gilgul is an extension of the Java programming language that allows for dynamic object replacement without consistency problems. This is possible in a semantically clean way because its model strictly separates the notions of reference and comparison that are usually subsumed in the concept of object identity. This paper sketches problems that occur in attempts at replacements of active objects and presents some solutions, including both variants that preserve consistency and those that trade consistency for timeliness. The latter are enabled by means of the new *recall* construct that even allows for the replacement of objects with non-terminating loops.

1 The TAILOR Project

Unanticipated Software Evolution. Software requirements are in a constant flux. Some changes in requirements can be anticipated by software developers, so that the necessary adaptations can be prepared for, for example by suitable parameterization or by application of appropriate design patterns. However, unanticipated changes of requirements occur repeatedly in practice, and by definition the above suggested techniques cannot tackle them.

To obviate such problems, programming languages and runtime systems should incorporate features that allow for manipulations of program internals without destructively modifying their source code. This leads to an increase in the number of options for unanticipated evolution as well as a decrease in the pressure to prepare for anticipated variability.

In order to provide for this significant simplification of software development, the goal of the TAILOR Project at the Institute of Computer Science III of the University of Bonn [19] is to explore several approaches that enhance programming languages and runtime systems in order to allow for unanticipated software evolution. In doing so, special attention is paid to the following issues.

Component Orientation. The focus on Component-Oriented Software complicates the requirements even further, since components are usually deployed using a compiled format, and their source code is not available for modifications. Even if the source code can be accessed in some cases, destructive modifications are still not feasible, since they would lapse in the presence of new versions of a component.

J. Bishop (Ed.): CD 2002, LNCS 2370, pp. 125–140, 2002.

Reduction of Downtime. Essentially, unanticipated software evolution can take place at two points in time. They can be performed before a program is being linked into its final form, or they can happen during runtime. Changes to software that are carried out before linktime can be made effective only by stopping an old version of a program and starting the new one. This results in downtimes that induce high costs and possibly determine an application's success or failure. Alternatively, runtime systems should be provided with features that allow for subsequent unanticipated evolution of an already active program.

Challenges. Accordingly, the TAILOR Project deals with enhancements of programming languages and runtime systems that allow for unanticipated software evolution on the stringent condition that components are to be included whose source code is not available, and that modifications can still be made on already active programs. This paper presents one of the approaches of the TAILOR Project that rests on the simple idea of dynamic object replacement.

2 Dynamic Object Replacement

In principle, unanticipated evolution can always be dealt with by manual redirection of references. If one knows the reference to an object and wants to add or replace a method or change its class, one can simply assign a new object with the desired properties. The new object can even reuse the old object by some form of delegation [12], so that a recovery of the old state is not needed.

On the conceptual level, however, there are two consistency problems involved in this approach. Firstly, if there is more than one reference to an object, they all must be known to the programmer in order to consistently redirect them. Secondly, even if all references are known, they have to be redirected to the new object one by one. This approach is likely to lead to an inconsistent state of the objects involved if message are sent via these references during the course of the redirections (for example within another thread; see figure 1).

It would be straightforward if we could simply "replace" an object by another one without changing the references involved. Such a replacement would be an atomic operation and hence, would avoid the consistency problems shown above. We have discussed this subject previously on the basis of a specific example in [4].

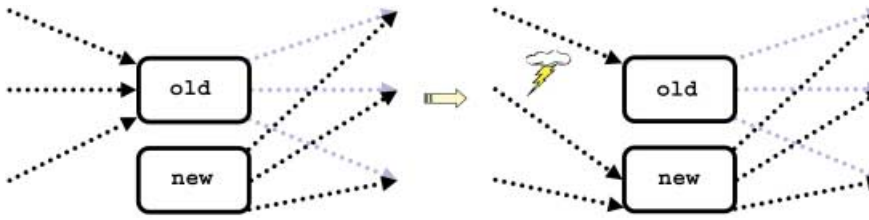

Fig. 1. If an object is replaced by manual redirection of references, messages may be sent to both objects during the replacement, probably leading to an inconsistent state.

In that paper we have also shown that a dissection of the concept of object identity and a strict separation of the included notions of reference and comparison is needed in order to introduce means for dynamic object replacement.

This can be illustrated with an implementation technique called "Identity Through Indirection" in [11]. Here, a reference to an object is realised as an object-oriented pointer (OOP). An OOP points to an entry in an object table which holds the actual memory addresses. Since we do not want to restrict our model to this implementation technique, we abstract from this terminology and say that object references point to entries which hold *referents* that represent the actual objects (see figure 2). Note that an actual implementation of this model does not have to resemble the illustration given here.

In our approach, references are never compared. To be able to compare objects we combine "Identity Through Indirection" with "Identity Through Surrogates" [11]. Each object is supplemented with an attribute that stores a *comparand*. Comparands are system-generated, globally unique values that cannot be manipulated directly by a programmer. The comparison of objects (o1 == o2) then means the comparison of their comparands (o1.comparand == o2.comparand), but they are never used for referencing.[1] Based on this scheme, we outline the programming language Gilgul, a compatible extension of Java, in the following sections. (More details can be found in [2].)

There are four levels that can be manipulated when dealing with variables in Gilgul: the reference and the object level that already exist in Java, and the referent and the comparand level that are new in Gilgul. A class instance

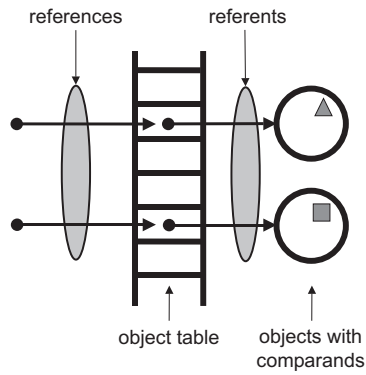

Fig. 2. Gilgul's model: references point to entries in an object table, and each object stores a comparand.

[1] In [11] this attribute is named *surrogate*. Elsewhere, names like "key", "identifier" and "OID" are used for this concept. However, these and other terms found throughout the literature might raise the wrong associations. In our approach, we have coined the artificial word *comparand* to stress that this attribute is a passive entity that is never used for referencing, but strictly within comparison operations only.

creation expression (new MyClass(...)) results not only in the creation of a new object, but also in the creation of a new reference, a new referent and a new comparand. The class instance creation expression returns the reference to the object's referent, which in turn has the comparand among its attributes.

Operations on Referents. In GILGUL, the *referent assignment operator* #= is introduced to enable the proposed replacement of objects.[2] The referent assignment expression o1 #= o2 lets the referent of the variable o1 refer to the object o2 without actually changing any references. Effectively, this means that all other variables which hold the same reference as o1 refer to the object o2, too. This can be realised simply by copying the referent of o2 to the entry of o1 in the object table. Consider the following statement sequence.

```
o1 = new MyClass();
o2 = o1;
o2 #= o3;
```

After execution of the referent assignment, all three variables are guaranteed to refer to the same object o3, since after the second assignment, o1 and o2 hold the same reference (see figure 3).

Figure 3 also illustrates why a strict separation of reference and comparison is needed in order to allow for this kind of manipulations. Assume that you want to compare o2 and o3 after execution of the statement sequence given above, resulting in the scenario on the right-hand side of figure 3. In this situation, comparison of variables without the use of comparands is ambiguous on the conceptual level, since comparison of the references would yield false, whereas comparison of the referents would yield true. The decision for one or the other option would be arbitrary and cannot be justified other than by technical considerations only. Therefore, we opt for comparison of properties stored inside of the objects involved and thus make comparison of variables unambiguous.

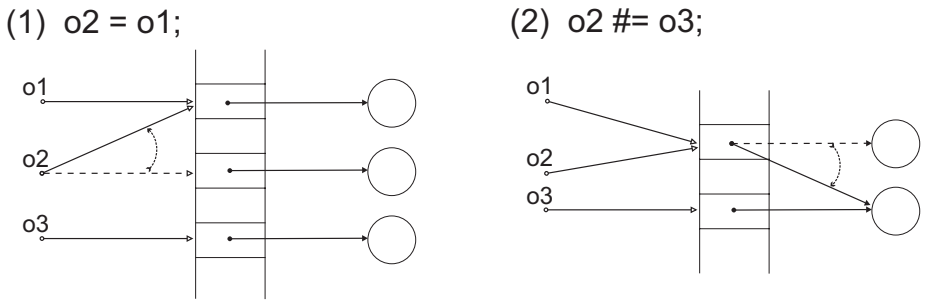

Fig. 3. Referent Assignment: After execution of o2 #= o3, all three variables refer to the same object. Since o1 holds the same reference as o2, it is also affected by this operation.

[2] The hash symbol # is meant to resemble the graphical illustration of an object table.

Note that the referent assignment operator #= is a reasonable language extension due to the fact that the standard assignment operator = copies the reference from the right-hand operand to the left-hand variable, but not the referent stored in the respective entry in the object table.

Since null does not refer to any object, the referent assignment is prevented from being executed on null, by means of both compile-time and runtime checks.

Operations on Comparands. Technically, it is clear that comparands may be copied freely between objects. There are, in fact, good reasons on the conceptual level to allow the copying of comparands. For example, decorator objects usually have to "take over" the comparand of the decorated object so that comparison operations that involve "direct" references to a wrapped object yield the correct result. Other usage scenarios are given in [3].

Comparands are introduced in Gilgul by means of a new basic type comparandtype which can be used to create new comparands via comparand creation expressions (new comparand). By default, the definition of java.lang.Object, and therefore all objects include an instance variable named comparand of this type.

The equality operators == and != that are already defined on references in Java are redefined in Gilgul to operate on comparands, such that o1 == o2 means the same as o1.comparand == o2.comparand, and o1 != o2 means the same as o1.comparand != o2.comparand.

Given these prerequisites, we can let a wrapper "take over" the comparand of a wrapped object in order to make them become equal by simply copying it as follows: wrapper.comparand = wrapped.comparand.

Ensuring the uniqueness of a single object is always possible by assigning a freshly created comparand as follows: o1.comparand = new comparand.

In Java, the equality operator == is only applicable if one operand can be cast to the type of the other, thereby excluding meaningless comparisons of arbitrarily-typed references [8]. Consequently, a comparand assignment of the form expr1.comparand = expr2.comparand is only accepted if expr2 can (possibly) be cast to the type of expr1. This restriction can be lifted by an explicit cast as follows: expr1.comparand = ((Object)expr2).comparand. Other forms of comparand assignment are always accepted.

Since null does not have a comparand it is prevented from being accessed by a combination of both compile-time and runtime checks. This ensures that testing equality against null is guaranteed to be unambiguous.

The actual implementation of comparands is hidden from programmers. In particular, Gilgul prevents comparands from being arbitrarily cast and, for example, does not allow arithmetic operators to be executed on comparands.

Since comparands cannot be manipulated directly, there are no limitations on how they are implemented in a concrete virtual machine. The only requirement they have to fulfil is that if o1.comparand and o2.comparand have been generated by the same (different) class instance or comparand creation expression, then o1.comparand == o2.comparand yields true (false), and o1.comparand != o2.comparand yields false (true). A reasonable and efficient implementation scheme is outlined in [2].

Example of Use. Returning to our given problem, we are now able to apply the new operations to achieve the desired replacement of an object atomically. We can apply oldObject #= newObject to let newObject replace oldObject consistently for all clients that have references to oldObject.

However, one has to be careful when newObject wants to refer to oldObject in order to delegate messages that it cannot handle by itself. In order to avoid unwanted cycles, one must take care to use a fresh temporary variable for forwarding purposes, as follows (see figure 4; refer to [2,4] for more details).

```
// let a new reference refer to oldObject
   tmp #= oldObject;

// use tmp for forwarding
   newObject.orgObject = tmp;

// ensure that equality behaves well
   newObject.comparand = oldObject.comparand;

// tmp, and so newObject.orgObject remain unchanged
   oldObject #= newObject;
```

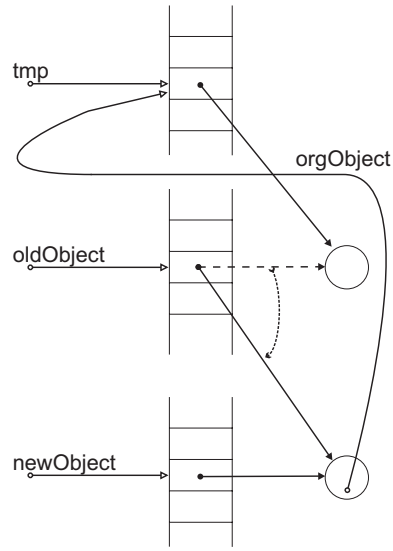

Fig. 4. Application of oldObject #= newObject: When newObject.orgObject holds a different reference to the same object as oldObject beforehand, it will still refer to the former oldObject afterwards, since the temporary reference is not affected. In this way, the state of replaced objects can smoothly be reused.

The actual "replacement" of oldObject is initiated by the last operation, and thus is indeed atomic. Further note that the temporary variable can be used to revert the replacement by application of oldObject #= tmp.

However, this idiom is only needed when newObject needs to reuse oldObject. Otherwise, a "simple" replacement is sufficient. In this case, reversal of a replacement can also be achieved by the use of an additional reference, but it is not needed for forwarding purposes.

As we can see, GILGUL's new operations give the programmer the possibility of "replacing" the former object atomically and thereby relieves him/her from having to deal with any consistency problems that result from many, perhaps unknown, references to the target object. Furthermore, the objects involved need not anticipate such modifications, reducing the complexity of the development of actual components to a great extent.

3 Replacement of Active Objects

Although quite powerful, GILGUL's new operations face a problem when dealing with active objects. Consider figure 5 that depicts an attempt at replacement of object1 that still has a method m() executing on it. After an attempt at replacement of obect1, it is unclear what object the active method m() should continue to execute on afterwards. There are only two options – either it continues to execute on the old object that has been replaced, or it chooses the new object. Both options have serious drawbacks. If m() continues to execute on the old object, it does not reflect the programmer's intention to have the object replaced. If it chooses the new object, this may lead to severe consistency problems, for example, if m() is implemented differently in the class of the new object. This example also illustrates why in the general case, such attempts cannot be detected statically since they might be issued in other objects and especially in totally different source code.

In order to ensure consistency, the default semantics of GILGUL are defined as follows. If the active method m() and the attempt at object replacement are

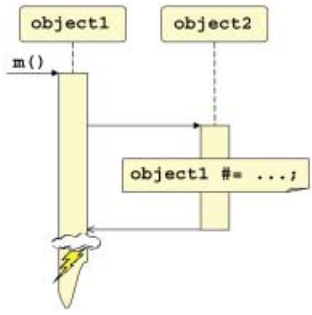

Fig. 5. Replacement of object1 – what object should m() continue to execute on?

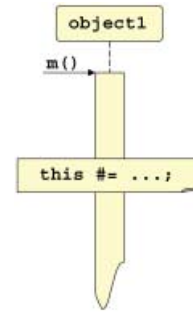

Fig. 6. Replacement of this – method m() is active by definition.

executed by different threads the referent assignment operation blocks until m() (and all other methods that are active on the target object) complete. If they are executed by the same thread (that is, on the same execution stack) the referent assignment operation throws an (unchecked) ReferentAssignmentException.

Replacement of this There are several situations when an object knows best when to be replaced. For example, the State pattern [6] gives the advice "to let the State subclasses themselves specify their successor state and when to make the transition". However, a referent assignment this #= *expression* implies that the method that currently executes on this is active in the same thread by definition (see figure 6).

GILGUL relaxes its restrictions for this case and does not throw a ReferentAssignmentException if it is ensured (by a combination of static and dynamic checks) that after the replacement of this, code will no longer be executed on this. For example, in the case of a void method this can be accomplished by just placing the replacement of this at the end of the method. For non-void methods a new construct is introduced as follows.

```
int m() {
  ...
  return expression with this #= replacement;
}
```

The semantics are as follows. First, the return expression is evaluated. Then, the replacement of this is carried out. Last, the method completes and returns the result of the evaluation in the first step. Together, these steps ensure that the replacement of this is indeed the last statement that gets executed on this.[3]

Another variation of this construct occurs in combination with the throw statement: throw *expression* with this #= *replacement*. The semantics are defined accordingly.[4]

Advanced Requirements. So far, GILGUL does not offer a completely satisfactory solution for replacements of active objects. For the default case, our main goal was to ensure consistency, and consequently, this precludes solutions for the following issues.

- The rule that replacements block until active methods in other threads complete does not by itself handle the situation when the active method consists of a non-terminating loop (for example, in the case of daemon threads).

[3] Possibly, there are still other methods active on the current object. In this case, the rules hold that are given in the previous section.

[4] In previous versions of GILGUL, we have allowed replacements of this to be placed inside of the finally block of a try statement. This would ensure roughly the same semantics (both for return and throw statements). However, the finally block is meant to be used for code that should always execute, even when an exception is thrown. It is more likely that replacements of this should not occur in this case, so we have added the with construct to our design with these slightly different semantics.

– More often than not, administrators might be willing to trade timeliness for (temporary) inconsistency and so want to have an object replaced immediately rather than having to wait for the completion of other methods.

In GILGUL, the concept of *recalls* is introduced to provide support for these cases. In the following sections, we first outline the concept of recalls and afterwards show how they can be combined with referent assignments in order to provide a feasible means to deal with these cases.

Recalls. Exceptions in Java are a means to step out of the standard flow of control, and they can be caught by dedicated exception handlers, possibly at a higher level in the call stack, to revert to a corrected flow. In other words, the current call stack is *unwound* until a matching exception handler is found [8].

A *recall* in GILGUL mimics the behaviour of Java's exceptions, but instead of relying on the definition of dedicated exception handlers, a return to the standard flow takes place as soon as the call stack is clear of a particular receiver during the process of unwinding. For example, assume the throw of a recall on object2 in figure 7. The current call stack is unwound up to the first method call on object2, and then this particular method is called again on object2.

Note that a recall guarantees that immediately before the point of return to the standard flow of control, the current call stack does not contain any method calls to the specified object. So by definition, there is no method active on this object at this point in time (at the marked spot in figure 7).[5]

Just as in the case of exceptions in Java, Recall is a plain class in GILGULand instances thereof can be thrown and caught. It is unchecked, like java.lang.RuntimeException, in order to have it smoothly integrated into existing Java code.

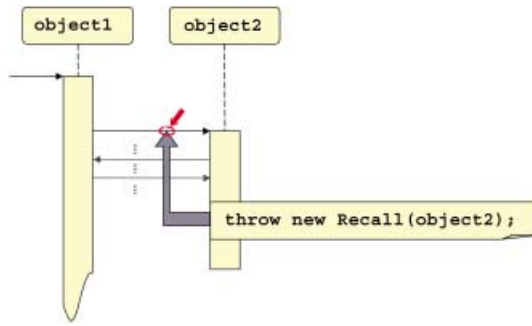

Fig. 7. A *recall* on object2 unwinds the current stack up to the first method call on object2, and then calls this method again. Just before reexecution of this method call (at the marked spot), the call stack is guaranteed to be clear of object2 as a receiver.

[5] If there is no method active on the target object at the moment of the throw of a respective recall, this recall is simply ignored and evaluates to a non-operation.

The catch of a recall can be employed in order to set corresponding objects to a consistent state. However, the respective recall should be rethrown in order to guarantee its effective completion, as follows.

```
try {
  ...                          // do your standard stuff here
} catch (Recall recall) {
  ...                          // reset your object(s)
  throw recall;                // and rethrow the recall
}
```

Since GILGUL's recalls are not exceptions in the strict sense to indicate that something has gone wrong, Recall is not defined as a subclass of java.lang.Exception, but as a subclass of the more general class java.lang.Throwable. This ensures that a general catch of all exceptions does not accidentally catch a recall as well.

Further note that in the presence of parameters to the method call that gets reexecuted by a recall, the (possibly complex) parameter expressions are not reevaluated, but the previously evaluated values are simply reused.[6]

Combination with Referent Assignments. Given these prerequisites, GILGUL's referent assignment can be amended by recalls in order to replace even active objects. This is accomplished by annotating the application of the referent assignment operator accordingly, for example as follows.

```
expression1 #= expression2 with global recall;
```

The options are as follows: a *local recall* throws the recall only for the current thread; a *global recall* throws it for all other threads that have a method executing on the target object, but not the current thread. These options can be combined, as in *expr1* #= *expr2* with local recall, global recall. Then the recall is thrown for both the current thread and other threads. The combination of a local and a global recall can be be abbreviated, as in *expr1* #= *expr2* with total recall.

The actual Recall instance that is thrown in this case takes the left-hand side of the referent assignment expression as a parameter. This means that the respective call stacks are unwound up to the first method call on the object referred to by the left-hand side. The actual replacement of this object is deferred to this point in time, when the call stack is guaranteed to be clear of methods that are active on this object (as a receiver), and just before the reexecution of the respective method call (at the marked spot in figure 7).

As a consequence, by means of the recall construct, the replacement takes place at a point in time when it is safe to carry it out. Afterwards, the standard flow of control is reentered and for example, can return to a thus temporarily terminated loop. Since recalls may be caught during the unwinding of the call

[6] In Java bytecode, the call of a method consists of pushing parameters on the operand stack, and then, as a distinct step, execute an invocation of the respective method. It is only this last step that gets reexecuted by a recall, and the invocation merely reuses the old state of the operand stack.

stack, it is possible to reset the target object (and possibly dependent objects) to a reasonable, consistent state. However, it is not required to provide for such clearance code because recalls are unchecked. Therefore, they can still be thrown even in unanticipated contexts. In the latter case, it is the task of the programmer who wants to replace a specific object to decide if clearance is needed or not and to take the necessary steps.

Relation to Java's Thread Model. There is a close relation of GILGUL's recalls to Java's *interrupts* that are used in order to signal that a specific thread should terminate. In the latter case, no automatic steps are taken by the runtime system to actively stop the thread, but instead all threads are required to regularly check their own interrupt flags and terminate eventually. In order to help programmers to remember to check for interrupts, some standard thread-oriented methods in Java, like wait(...) or Thread.sleep(...), may throw a (checked) InterruptedException when the corresponding flag is set.[7]

Essentially, global recalls in GILGUL mimic this behaviour. Instead of directly throwing recalls in other threads, a global recall merely sets a corresponding flag in each thread. This recall flag is checked in the well-known methods that already check for the interrupt status, namely wait(...), Thread.sleep(...) and so on, and they may throw recalls accordingly. Additionally, the recall flag is checked in the equally well-known Thread.yield(). Since platform independent components are expected to at least occasionally call Thread.yield() for compatibility reasons, and generally make use of the other methods mentioned as well, the recall mechanism will most likely be already applicable in most cases without change of existing components.[8]

A Template for Long-Running Methods. As a preliminary summary, we now give a code template for long-running methods.

```
void longRunningMethod() {
  try {
    for (aVeryLongTime) {
      Thread.yield(); // implicitly check for global recalls
      doYourActualStuff();
    }
  } catch (Recall r) {
    resetToConsistentState();
```

[7] Note that this is the only default means in Java to support the termination of threads. Up to JDK 1.2, the class java.lang.Thread has included methods stop(...) and suspend(...) for explicit termination, but they have since been deprecated for safety reasons [18].

[8] In Java, the method Thread.yield() is used as a hint to the runtime system to indicate where switches between thread contexts may occur, and is essentially a Thread.sleep(0). The occasional call of Thread.yield() is optional in pre-emptive implementations of Java's thread model, but is required in cooperative ones. See [13] for more details on Thread.yield().

```
    throw r; // rethrow the recall
  }
}
```

Please note the following.

- The try-catch block is only required if there is a need for clearance in the presence of recalls. Otherwise, Java's standard exception handling mechanism guarantees the propagation of recalls to higher levels.
- If the longRunningMethod() is the topmost method that has been called for the respective object on the call stack, the throw of a recall is guaranteed to reexecute this method (after object replacement, if thrown in conjunction with a referent assignment). There is no need for the programmer to take explicit steps in order to achieve this behaviour.
- The only requirement that is currently placed on the programmer is to occasionally insert checks for recalls, for example by calling Thread.yield().[9]

State of Implementation. A beta-quality implementation of the complete GILGUL programming language can be freely downloaded as a compiler and runtime system from the GILGUL homepage [7]. The software includes a working implementation of the recall mechanism and its combination with the referent assignment operator, as described in the previous sections.

The GILGUL runtime system is a modification of the Kaffe virtual machine [10], both released under the Gnu Public License. We have put our main focus on providing a stable proof of concept in a manageable amount of time, and not on topnotch industrial-strength performance, given the manpower of just one programmer for the compiler and one for the runtime system. Therefore we have chosen to only modify the pure interpreter part of the Kaffe virtual machine, and have not yet addressed issues of just-in-time or dynamic compilation.

We have also completed a detailed analysis of GILGUL's runtime performance in order to determine the relative speed in comparison to the pure Kaffe virtual machine. We have used standard benchmarks for this purpose, for example VolanoMark [20], and they report an average penalty of less than 5 percent. This is mainly due to some clever optimisations for the general case and, for example, the avoidance of a centralized object table, but instead the use of implementation techniques that make double indirection nearly as fast as direct pointers. Details will be presented in a forthcoming diploma thesis [16].

4 Related Work

Object Identity. GILGUL is the first approach known to the author that strictly and cleanly separates the notions of reference and comparison at the level of a

[9] A programmer may also choose to explicitly check for global recalls by inspection of the respective recall flag. However, we expect programmers to want to do this seldomly, so we have not presented the details here.

programming language. An overview of related work centered around the theme of object identity is given in [4].

Note that this separation of object identity concerns is orthogonal to the usual distinction between reference semantics and value semantics that is found for example in Smalltalk [17], Eiffel [15], and Java [8]. These languages allow programmers to choose from these semantics when comparing variables, but they all still rely on comparison of references in order to establish object identity.[10] For this reason, object identity usually coincides with reference semantics. In contrast, our approach relies on comparands to determine (logical) identity, and this opens up new degrees of flexibility.

Microsoft's component object model COM [1] separates object identity and references to the degree that it generally allows components to return different references for repeated requests of the same interface. However, the special IUnknown reference is required to never change as an exceptional case for comparison purposes. Again, object identity and reference semantics coincide and therefore COM components cannot be dynamically replaced unless prepared for. COM's *monikers* provide an alternative mechanism of object identification, targeted at linking and persistence, that allows the physical implementation of objects to change occasionally. However, it is not used as a general reference mechanism.

Dynamic Object Replacement. The programming language Smalltalk provides an operation become: that enables the programmer to (symmetrically) "swap" two objects without actually changing their references. Early Smalltalk implementations were based on object tables, and so this operation was straightforward to implement. Modern Smalltalk implementations that use direct pointers either take considerable effort to implement become: correctly, or reduce become: to an asymmetrical operation [17]. However, there are still some serious drawbacks. Smalltalk's become: is not type-safe because it does not check for compatible storage layouts of the objects involved. Furthermore, it does not pay any attention to methods currently executing on the objects, but just lets them continue to execute on the "swapped" objects. In contrast, Gilgul's referent assignment respects Java's type system without sacrificing flexibility (as shown in [2]), and introduces means to correctly and explicitly deal with active objects.

Replacement of Active Objects. We are aware of only one approach that allows for dynamic software evolution and tries to deal with non-terminating loops without stopping them, but rather by letting the programmer define states of a loop that allow for quasi-"morphing" to another loop [14].

Another more recent approach for unanticipated software evolution is [5]. That paper describes an extension of the Java Platform Debugging Architecture that is included in version 1.4 of the Java Development Kit. That approach works on a different level of granularity, since it aims at replacements of classes

[10] Whether the actual semantics of comparison is defined in the respective classes or must be determined by choosing from different comparison operators/methods varies from language to language. A thorough examination of these issues is given in [9].

at runtime, not objects as in Gilgul. Furthermore, the issue of replacement in the presence of active objects has not yet been addressed. For the time being, that approach relies on completion of active methods.

5 Conclusions and Future Work

We have designed the programming language Gilgul, a compatible extension to Java. It introduces the new basic type comparandtype and the referent assignment operator #=. It also changes the definition of the existing equality operators == and != according to the Gilgul model. We have shown an example of how to apply Gilgul's new operations for the purpose of dynamic object replacement without the need to deal with consistency problems.

This model is a generalization of what can be expressed in terms of object identity in current object-oriented programming languages. Gilgul offers flexible means for declaring restrictions on which operations are valid on specific referents and comparands, for example in order to prevent the replacement of sensible objects. These restrictions can range from the unrestricted applicability of Gilgul's new operations to the reduction to the traditional stringent restrictions placed on object identity. More details are given in [2].

In this paper, we have also sketched the problems with which dynamic object replacement is faced in the presence of active objects, both in multi-threaded and single-threaded contexts. By default, Gilgul offers means to deal with these situations in a way that preserves consistency. If programmers are willing to trade consistency for timeliness, or even need to break consistency in order to be able to replace objects at all in the case of non-terminating loops, they can take advantage of Gilgul's advanced facilities for these cases.

We have outlined the concept of *recalls* that has been introduced in Gilgul. Like exceptions, recalls unwind the call stack, and can be thrown and caught. Unlike exceptions, the return to the standard flow of control is guaranteed as soon as the call stack is clear of a specific object as a receiver of a method call. At this point in time, the very first call to the specified object is simply reexecuted.

The referent assignment operator can be annotated with several variants of recalls, which means that the actual replacement is deferred until the corresponding call stack is clear of the object to be replaced. Just before reexecution of the first method call to this object, the actual replacement takes place.

Although this combination of the referent assignment operator and a recall might break consistency, target objects are still able to react to the throw of recalls by providing recall handlers for clearance purposes, just like exception handlers in Java. However, even if recall handlers have not been provided, replacements can still be carried out ensuring timeliness, and replaceability in the presence of non-terminating loops. This might sometimes be the last resort before a system shut-down becomes inevitable.

The implementation of the Gilgul compiler and runtime system has just been completed, and we have briefly sketched some of its properties.

Acknowledgements

The author thanks Tom Arbuckle, Michael Austermann, Ferrucio Damiani, Paola Giannini, Peter Grogono, Arno Haase, Günter Kniesel, Thomas Kühne, Sven Müller, James Noble, Markku Sakkinen, Oliver Stiemerling, Clemens Szyperski, Dirk Theisen, Kris De Volder and many anonymous reviewers for their critical comments on earlier drafts and related publications, which led to substantial improvements.

This work is located in the TAILOR Project at the Institute of Computer Science III of the University of Bonn. The TAILOR Project is directed by Armin B. Cremers and supported by Deutsche Forschungsgemeinschaft (DFG) under grant CR 65/13.

References

1. D. Box. *Essential COM*. Addison-Wesley, 1998.
2. P. Costanza *Dynamic Object Replacement and Implementation-Only Classes*. 6th International Workshop on Component-Oriented Programming (WCOP 2001) at ECOOP 2001, Budapest, Hungary.
3. P. Costanza and A. Haase. *The Comparand Pattern*. EuroPLoP 2001, Irsee, Germany.
4. P. Costanza, O. Stiemerling, and A. B. Cremers. *Object Identity and Dynamic Recomposition of Components*. in: *TOOLS Europe 2001*. Proceedings, IEEE Computer Society Press.
5. M. Dmitriev. *Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications*. Workshop on Engineering Complex Object-Oriented Systems for Evolution (ECOOSE) at OOPSLA 2001, Tampa, Florida, USA.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns*. Addison-Wesley, 1995.
7. The Gilgul homepage. http://javalab.cs.uni-bonn.de/research/gilgul/
8. J. Gosling, B. Joy, G. Steele, and G. Bracha. *The Java Language Specification, Second Edition*. Addison-Wesley, 2000.
9. P. Grogono and M. Sakkinen. *Copying and Comparing: Problems and Solutions*. in: *ECOOP 2000*. Proceedings, Springer.
10. The Kaffe homepage. http://www.kaffe.org/
11. S. N. Khoshafian and G. P. Copeland. *Object Identity*. in: *OOPSLA '86*. Proceedings, ACM Press.
12. G. Kniesel. *Type-Safe Delegation for Run-Time Component Adaptation*. in: *ECOOP '99*. Proceedings, Springer.
13. D. Lea. *Concurrent Programming in Java, Second Edition*. Addison-Wesley, 1999.
14. I. Lee. *DYMOS: A Dynamic Modification System*. Dissertation, University of Wisconsin-Madison, USA, 1983.
15. B. Meyer. *Eiffel: The Language*. Prentice Hall, 1992.
16. S. Müller. *Transmigration von Objektidentitäten – Integration der Spracherweiterung Gilgul in eine Java-Laufzeitumgebung* (in German). University of Bonn, Insitute of Computer Science III, diploma thesis, 2002. (in preparation)

17. D. N. Smith. *Smalltalk FAQ*. http://www.dnsmith.com/SmallFAQ/, 1995.
18. Sun Microsystems, Inc. *Java 2 SDK, Standard Edition Documentation, Version 1.3.1*. http://java.sun.com/j2se/1.3/docs/
19. The Tailor Project. http://javalab.cs.uni-bonn.de/research/tailor/
20. VolanoMark Java Benchmarks. http://www.volano.com/benchmarks.html