Programming Distributed Systems with the Delegation-based Object-Oriented Language dSelf

Robert Tolksdorf[1] and Kai Knubben[2]

[1] Technische Universität Berlin, Fachbereich Informatik, FLP/KIT,
Sekr. FR 6–10, Franklinstr. 28/29, D-10587 Berlin, Germany,
mailto:tolk@cs.tu-berlin.de, *http://www.cs.tu-berlin.de/~tolk*
Phone: +49-314-25184, Fax: +49-314-73622
[2] *mailto:cooper@cs.tu-berlin.de*
FULL PAPER

Abstract. Common middleware platforms facilitate a class-based object model. However, this model introduces several dependencies that are disadvantageous for distributed systems. The family of class-less delegation-based languages offers an alternative object model with lesser such dependencies.
dSelf is an extension to the delegation- and prototype-based object-oriented language SELF. It adds distributed objects and transparent remote reference resolution to the languages. As a consequence, dSelf facilitates distributed inheritance and instantiation mechanisms.
We describe the current conception and implementation of dSelf and give examples where the flexibility of dSelf can be used with benefit as a middleware concept for programming distributed systems.

1 Introduction

Concepts of object-orientation are fundamental to a variety of modern middleware platforms such as CORBA or RMI. The separation of interfaces from implementation is a key to the integration of components being heterogenous in design, implementation and underlying technology. Typing and class concepts seem central to the effective construction of distributed systems by reuse.

Looking closer at the kind of object-orientation commonly incorporated by middleware platforms shows that they follow one specific school of thought in object-orientation, that of class-based object-orientation.

Programs in class-based object-oriented languages build on a class-hierarchy rooted usually in some class like *Object* that defines and implements the minimal interface and behavior of instance objects of any subclasses. This hierarchy defines several dependencies. A subclass depends on its superclass by means of an inheritance mechanism. It is commonly implemented by code-sharing for behavior and composition for of an objects state. Any instance object depends on its class as its behavior is defined there. Similar dependencies arise when the notion of type is used: Any assessment of type compatibility between objects and/or messages to them depends on the availability of a type description.

If this model of object-oriented serves as the basis for languages and infrastructures that support distributed objects, these dependencies induce unwanted centralization. Objects are instances of exactly *one* class which is a central object for all its instances then. A subclass has to know about its specific *one* superclasses (or a list of classes in the case of multiple inheritance).

A popular class-based language with built-in support for distributed object systems is Java with its RMI mechanism. Here, the named effects of centralization are somewhat relaxed by the standardized Java class hierarchy. As all `java.*` packages are defined as part of the language, there can be multiple representations of these classes on different machines. With the non-technical means of standardization, this consistent replication of the Java standard libraries is guaranteed.

But still, any application specific class-hierarchy – or the hierarchy "above" the class of a parameter object – has to be available locally. This results in additional communication overhead for passing the class-definitions with an object and additional coordination overhead to deal with versions of the applications classes. There is no way in Java to have the class of an object located on a remote machine, nor to have remote superclasses. The dependencies between instances and their classes and subclasses and their superclasses thus introduces inflexibility in distributing programs.

Using the notion of class is just one school of thought in object-orientation ([Tai99]). The family of *delegation based languages* – often also prototype-based – abandons the concept of classes and is based on objects only. This alternative concept provides object-orientation which is of equal expressive power ([Ste87]) as class-based languages.

Objects are created by *cloning* an existing object instead of instantiating from a class. Inheritance is implemented by *delegating* messages unknown to a an object to a "parent" object. By keeping state separated from behavior, the instance-class relation is implemented. By keeping inherited behavior and state in parent objects, the subclass-superclass relation is implemented. However, there is no notion of class in such languages.

Returning to dependencies named above, delegation based object-orientation replaces them with a single and more suited dependency, that of the parent object. The concept of delegation is inherently communication-oriented, while code-sharing implies a central location. With the work presented in this paper, we explore its usefulness to provide an infrastructure and a language for distributed object systems. We chose the classic delegation-based language SELF ([US87]) for our study and develop a distributed version called *dSelf* ([Knu00,TK01]).

This paper is organized as follows. In the next section, we give a brief overview on the SELF language. Section 3 points to the main contribution of dSelf and the necessary concepts. Then, we describe the extensions to the language for distribution. Section 4 gives some examples for the flexibility gained in dSelf for distributed programming. After that, we present some details on the implementation of the language. We finally give pointers to related work and present conclusions.

2 The Language SELF

SELF [SU95] is a delegation based object-oriented language. Any object consists of a set of named *slots* which each contain objects. As these objects can also be block-objects, the distinction between attributes and methods is neglected in SELF. Any object can be *cloned*, which results in a new object which is an identical copy of the cloned one. A set of initial objects called the "SELF world" is given in a SELF system and accessible by navigating from the known `lobby`-object. The following is an example of a stack object.

```
aStack <- (|
  stack          = array clone.
  top            = 0.
  push: anObject = (top: (top+1). stack at: top Put: anObject).
  pop            = (top: (top-1)).
  getTop         = (^stack at: top)
|)
```

The slot `stack` is an attribute which is initialized with a clone of the universal and empty `array` object provided by the system. Attribute `top` is set to 0 initially. Upon receipt of a message `push`, the counter `top` is incremented and the object passed is stored in the `stack`. Being a clone of the `array` object makes it understand the `at:` message. `pop` and `getTop` remove the topmost element of the stack, resp. return it.

Cloning the object `aStack` would lead to a copy which included any objects that have been put on the stack. In order to implement a class-instance like relation in SELF, one separates behavior from state as follows:

```
stackBehavior <- (|
  push: anObject = (top: top+1. stack at: top Put: anObject).
  pop            = (top: top-1).
  getTop         = (^stack at: top)
|)
stackInstance <- (|
  parent*       <- stackBehavior.
  stack          = array clone.
  top            = 0
|)
```

Here, the object `stackInstance` contains only the state of a stack. If the message `getTop` arrives at the object, it forwards, or *delegates* handling of that message to the object that is referenced in the slot called `parent`. It is marked with a star to select that behavior. The referenced object, `stackBehavior`, contains the methods defining the behavior of the stack.

In order to create a new stack, one clones the `stackInstance`-object which remains in its initial state. Thus, whereas a class describes future objects in class-based languages, any object can serve as an *example* for other objects in a class-less language. `stackBehavior` plus `stackInstance` together define a structure comparable to a class.

Inheritance can also be implemented by delegation. New behavior is added by defining an object that has `stackBehavior` as its parent and adds only new method-slots. Overriding behavior is also possible, as the lookup for matching methods first searches the current object before delegating to the parent. The attributes are inherited by cloning the defining object and perhaps adding new slots. The following is an example for a refinement of the stack into a bounded stack, which limits the number of objects that can be pushed.

```
boundedStackBehavior <- (|
  parent*        <- stackBehavior.
  stackLimit     = 10.
  push: anObject = (top < stackLimit
                     ifTrue:  [parent.push: anObject]
                     ifFalse: ['Stack full' write]
                   )
|)
boundedStackInstance <-
  (stackInstance clone) parent: boundedStackBehavior
```

`stackLimit` plays a role similar to a class variable and thus is not a slot of instances. The `boundedStackInstance`-object is created by cloning `stackInstance` and overwriting the `parent`-slot with a new reference. This mechanism sort of mixes in the attributes defined in `stackInstance` with the changed parent-reference.

While SELF is of equal power in expressing instantiation and inheritance, it provides more flexibility. Examples are:

- Singleton objects are simply defined without the need for any class. If single objects need to be specialized, there is no need to define a subclass.
- The behavior of single objects can be changed during runtime be overwriting a slot.
- The `parent` object itself can be overwritten, thus allowing dynamic classification of objects and inheritance. The drawback is, of course, that there is nearly no way to do type-checking of SELF programs.
- Attributes and methods are unified, thus an attribute can be redefined to be a method in some specialization.

In order to support *distributed* object systems, we take advantage of the high locality of decisions involved and the high degree of encapsulation found. If a message arrives, it is locally determined whether the current object is able to handle it. There is no need to access a class-object, as in Smalltalk. If the lookup fails, then the delegation principle emphasizes the classic *communication oriented* style of interaction in object-oriented systems: A message is sent to the parent-object. Exactly this is the way of interaction amongst remote machines.

Also, delegation implies ease of change in the schema of objects. As the behavior of objects is not distributed by copying code, there is no need to propagate changes to this code to the distributed objects.

We extend SELF with concepts to support a remote delegation schema. In the following section, we outline these.

3 dSelf Concepts

dSelf extends SELF with distribution of objects. The dSelf system comprises a runtime system – the *dSelf Virtual Machine* – and a dSelf compiler, which both are written in Java.

By being an extension of SELF, dSelf also is capable of *dynamic inheritance* or *dynamic classification* of objects. As the parent-slots are usual data slots, they can be changed at runtime by assignment. The above example object `boundedStackInstance` demonstrates this: An instance of `stackInstance` is created by cloning which "inherits" its methods from `stackBehavior`. Immediately after creation, it is reclassified to inherit behavior from `boundedStackBehavior` by an assignment message to the `parent`-slot.

Adding distribution to SELF leads to two characteristics that are the main contributions of dSelf:

1. *Distributed instantiation* means that objects contain slots of data or behavior common to other objects – the "instances" – do not have to be co-located with these. Any clone of `stackInstance` can be anywhere in the world at places different from the `stackInstance`-object and the `stackBehavior`.
2. *Distributed inheritance* means that objects delegation messages to other objects – their "superclasses" – do not have to be co-located with these either. `boundedStackBehavior` and `stackBehavior` are independent wrt. their location.

Both characteristics lead to more flexibility. First, there is no need for co-location which usually leads to increased communication overhead – eg. to deliver application specific classes with object as in Java-RMI – and increased coordination effort to keep distributed copies consistent.

Second, the ability to keep just one copy of the definition of behavior of objects, makes changes much more easy. It suffices to change one object only to affect a whole distributed system. Also, there is no coordination overhead in synchronizing such a change.

3.1 Distributed dSelf Objects

Distribution in dSelf means that objects referenced in slots can be located on remote dSelf Virtual Machines. Thus, the system introduces navigable remote references to objects located on another dSelf Virtual Machine.

In order to access a remote object, a connection to its remote Virtual Machine and its lobby object must be established. Two primitive messages are understood by all objects for this purpose:

1. `_AddSlot:` ⟨*slotname*⟩ `ConnectedTo:` ⟨*URL*⟩
2. `_AddSlotIfAbsent:` ⟨*slotname*⟩ `ConnectedTo:` ⟨*URL*⟩

Argument ⟨*slotname*⟩ is a string, that specifies a lexically correct data slot and ⟨*URL*⟩ defines the location of the remote virtual machine (e.g. '//127.0.0.1:1099/myVM').

The behavior of these primitives is similar to the built-in messages _AddSlot: and _AddSlotIfAbsent: of SELF, except that the slots will contain a reference to a remote lobby. By this slot one can access and modify each remote object reachable from this lobby.

Once a connection to a remote lobby has been established, all its slots and its contents are transparently accessible and modifiable like local ones. An example from the dSelf console illustrates this:

```
lobby _AddSlot: 'remoteVM' ConnectedTo: '//127.0.0.2/remoteVM'
lobby remoteVM _AddSlots: (| x = 42 |)
-> lobby(r)
lobby remoteVM _Print
lobby(r): (|stackBehavior = stackBehavior(r). x = 42 |)
-> nil
```

Here, a connection to a remote lobby is stored in a local slot called `remoteVM` and a new slot `x` with content 42 is added to this remote lobby. After that, the output of the primitive `_Print` shows that a slot `x` has been added on this other machine. The trailing `(r)` after `lobby` or `stackBehavior` marks the reference as remote. The lookup algorithm implemented in dSelf is the same as specified in [ABC+95], except that remote references are followed across machine-boundaries.

As SELF, dSelf distinguished between ordinary and primitive objects, however, possible performance gains are higher in the distributed case. Remote references are created if and only if their target is an ordinary object. Primitive objects – these are built-in types like integers and strings – are treated differently, because their values are always copied during an access by a remote object.

The introduction of primitive objects is justified by efficiency considerations. It is more efficient to access a local copy of an primitive object than operating via a remote reference across the borders of a Virtual Machine. In contrast to SELF, dSelf considers the booleans `true` and `false` and the initial value `nil` also primitive objects.

Obviously, the disadvantage of introducing primitive objects in the distributed case is a loss of flexibility, as the possibility of inconsistency of the parent objects primitive objects has to be dealt with. All primitive objects of one kind (e.g. integers) have only one slot named `parent` that refers to one ordinary object. This object, in turn, consists of slots that are shared by all instances of the primitive objects (e.g. + or - for integers). But these slots can differ from one Virtual Machine to another one.

To ensure consistency, the user either has to leave these objects untouched or make its copies consistent by some extra code like this:

```
0 parent _AddSlots: remoteVM parents smallInt
```

In this example the slots of the parent object (assumed to be accessible in `parents`) of integers that are located on a remote Virtual Machine are copied to the local integers by using the primitive message `_AddSlots:` that adds all slots of the argument to its receiver. All slots whose names match with the added slots are replaced by the new ones.

Figure 1 illustrates the state of two dSelf Virtual Machines. The first one consists of a lobby with three slots `top`, `remoteStackInstance` and `remoteVM` with its initial content `nil`. The second VM consists of a slot a slot `stackInstance` that refers to the previously introduced stack object, which has a data slot `top` containing the number 42. We can now create a link from `local VM` to the lobby of `remote`

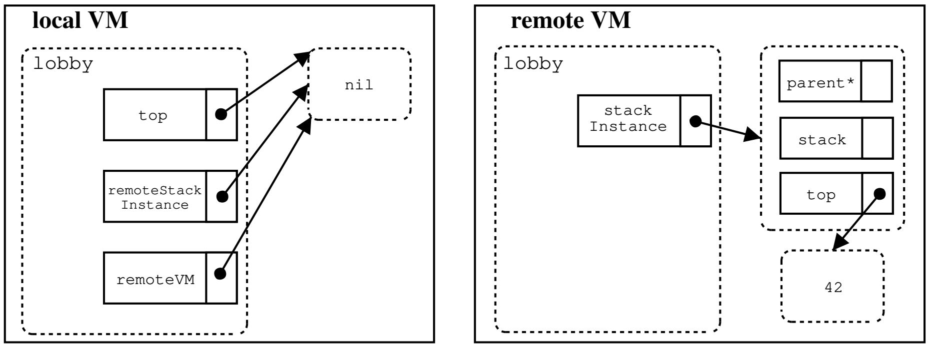

Fig. 1. Local references within VMs

`VM` in the slot `remoteVM` by the following step:

```
lobby _AddSlot: 'remoteVM' ConnectedTo: '//127.0.0.2/remoteVM'
```

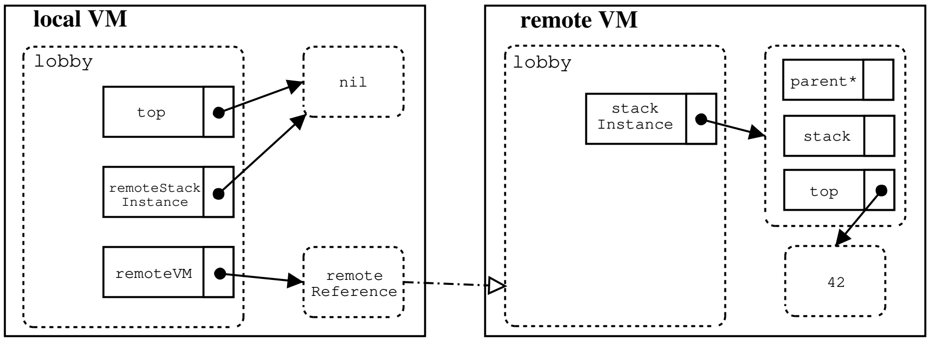

Fig. 2. A reference to a remote VM

As in figure 2, there now exists a local reference (displayed as a solid arrow) to a remote reference object that realizes the link (displayed as a dashed arrow) to the lobby at `local VM`. Now we can access and modify the remote dSelf Virtual Machine like the local one by using this link. For example this code:

```
lobby top: lobby remoteVM stackInstance top.
lobby remoteStackInstance: lobby remoteVM stackInstance.
```

results in the state shown in figure 3.

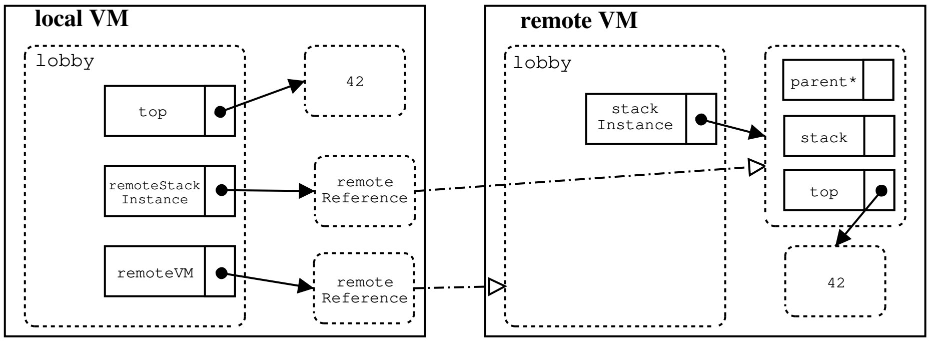

Fig. 3. A reference to an object in a remote VM

The slot `top` now contains a local copy of the primitive integer object `42` and `remoteStackInstance` refers by using a remote reference to the ordinary data object that is located at `remote VM`.

With remote references, we can demonstrate distributed instantiation with the following example.

```
localStackInstance <- (|
  parent*        = remoteVM stackBehavior.
  stack          = array clone.
  top            = 0
|)
```

In this example a slot `localStackInstance` is created at the local Virtual Machine. It consists of the data slots `stack`, `top` and the parent slot `parent` that refers to `stackBehavior` object that is located at the remote Virtual Machine. Thus, the "class" can be remote from the "instances" as all messages whose selector do not match with one name of the local slots are delegated across the network.

3.2 Evaluation of Slots and Methods

Methods always consist of two parts. The first optional part consists of the local slot declarations and the second one of its code:

(| ⟨*local slots*⟩ | ⟨*code*⟩)

The slots declared in a method are always evaluated in the context of the lobby at declaration time and the code will be evaluated in the context of the calling object when it is invoked.

In a distributed setting this means that ordinary objects created with the slot declarations are always located at the dSelf Virtual Machine, where the surrounding method was created. Instead, ordinary objects created during the execution of code are always located at the dSelf Virtual Machine, where the method was invoked. For example:

```
m = (| local1 = (). local2| local2: () )
```

In this example two local slots are declared. Both contain an empty ordinary data object, but their locations differ. `local1` is located on the dSelf-VM, where `m` was declared and `local2` is located on the dSelf-VM, where the object invoking `m` is located. The practical effect is that each access to `local1` causes communication across the network, while each access to `local2` is local.

Different from the original SELF version 4, dSelf supports local methods which were first introduced in SELF version 3 but dropped from the language later. Local Methods are methods whose declarations are nested in another method.[1] For example:

```
dupStack <- (|
  parent* = stackBehavior.
  dup = (
    |push: anObject = (top: top+1. stack at: top Put: anObject)|
    push: top
  )
|)
```

In this example an object `dupStack` inheriting from `stackBehavior` is declared. A new method slot `dup` is added for duplicating the top element of the stack. `dup` has a local method slot `push:` that is identical to the method with the same name at `stackBehavior`.

Local methods behave like ordinary methods, but are only accessible within the context of the code of their surrounding method.

Local methods minimize network access in a distributed context. As parent objects can be remote, calling methods "inherited" from them, causes network traffic. Local methods can be used to transport and co-locate these methods with an object, thus decreasing communication across the network.

3.3 Concurrent objects and synchronization

The original SELF definition seems to support processes and semaphores. However, the description in [ABC+95] is vague about the respective modules and functionality. The SELF 4.0 implementations for Solaris and MacOS seem only to pass through the operating system support for synchronization.

dSelf introduces concurrency implicitly with the mere existence of multiple virtual machines that each start one thread of control. Thus, even without the ability to start a thread within the language, dSelf needs a synchronization facility.

In the current version, dSelf uses a very primitive concept to support synchronization. Every non-primitive object understands the primitive messages `_Lock` and

[1] Not to be confused with inner methods, those are nested in expressions.

_Unlock which each take as argument a reference to an object that wants to obtain the lock. The common scheme to obtain a lock on some object $\langle o \rangle$ thus is $\langle o \rangle$ _Lock: self.

The additional argument seems superfluous at first. However, the dynamic inheritance in dSelf makes it necessary. Assume that objects a and b both contain a slot data. A third object c now refers to a as its parent. In some method of c a lock to the "inherited" data is obtained. If the method now changes the parent slot to point to b, then a subsequent unlock-message still reaches a slot data. This one now belongs to b and is not necessarily locked. data of a will never receive an unlock-message.

While dynamic inheritance is supported in dSelf, and the inconsistent situation can still be programmed, but the additional parameter of _Lock and _Unlock allow it to check whether the unlocking object really obtained a lock before. If not, a runtime error is raised.

The current support for synchronization in dSelf is rudimentary and not complete. We plan to introduce a complete concurrency model in a future version.

4 Usage Examples

This section shows in selected examples where dSelf provides interesting flexibility serving as an object-oriented middleware. While several common services of middleware are currently not present in dSelf, we focus on the consequences of applying a quite different object model at the baseline of middleware for distributed programming.

4.1 Reclassification

Objects in large distributed systems are subject to changes which can be induced by technical changes and/or changes in the business requirements.

In class-based objects models, the instance-of relation is usually considered being static, that is, an object cannot change its class during runtime. In delegation-based system, support for reclassification is more common. In SELF – and therefore also in dSelf – the parent slot of objects is mutable. Changing the contained reference changes the way that messages not understood are delegated. Thus, the object is reclassified.

For our simple stack example, we can consider the changing requirement that objects on the stack have to be stored securely. For brevity, we omit all aspect of how that security is establish.

An object secureStackBehavior contains the code to store and retrieve objects securely:

```
secureStackBehavior <- (|
  parent*          <- stackBehavior.
  push: anObject  = (⟨store object securely⟩).
  getTop          = (⟨retrieve object securely⟩).
|)
```

Now, if `myStack` is a stack object whose parent currently points to `stack-Behavior` it can easily be reclassified to be a secure stack by `myStack parent: secureStackBehavior`. In reality, some code to secure any already existing objects on the stack will have to be added.

`myStack` now has been reclassified to be an instance of a secure stack. It can well be moved up the class hierarchy by changing the `parent` slot again to refer to the normal stack implementation. Note that neither SELF nor dSelf contain any means to prevent changes to `parent` slots that lead to circular and wrong chains of references. In dSelf, circularities are detected during lookup.

4.2 Implementation Changes and their Propagation

Instead of reclassification of instances, one might also need to change the behavior for all objects that share it. In class-based languages, most often, this is not possible, eg. in Java where classes are compiled. In the case of distributed systems, the behavior specification, ie. code, might have been copied to several places, as is the case with Java RMI. So even if classes can be changed, there exists the problem of how to propagate these changes. Even more, the propagation should be consistent – in RMI, versioning had to be introduced for this purpose. The propagation issue is not only relevant for the class-object relation, but also for any behavior shared amongst super- and subclasses.

In delegation-based systems, only the `parent` object has to be updated and any changes are activated automatically with the next message lookup.

For the simple stack example, we want to change the behavior so that `push` logs all objects stored in a systemwide central log-object.

```
stackBehavior <- (|
  push: anObject = (top: top+1.
                    stack at: top Put: anObject.
                    centralLog log: anObject)
|)
```

The code changes the contents of the `push` slot whose value is an executable block of code. For any future `push` messages, the new behavior is activated to both the instance objects, as well as for the subclass objects systemwide.

A real application of this easy propagation of changes is found in Web-based distributed workflow systems that support adaption of the workflow definition. Here, a workflow schema could be represented in a parent object in some method and the instances of the workflow are objects that delegate the lookup of the next work item to their parent. Changes to the workflow schema then can be done at the parent object without any need to propagate it to the instances. Also, the parent could well be exchanged, for example when the flow of work has to change due to exception. We believe, that it is worthwhile to understand this as a dynamic reclassification of workflow instances and think that a distributed Self provides an attractive platform to implement such a mechanism.

4.3 Policies for Data Slots

Class-based languages distinguish two kinds of field in objects: methods and attributes. As in SELF, slots in dSelf contain values that are either blocks of code or data. As any slot can contain either of them, methods and attributed are unified. A data slot can be turned into a method slot an vice versa, while keeping the interface of the object unchanged.

For the bounded stack example above, on can overwrite the `stackLimit` field with a method that manages the stack based on some policy. For example, the stack limit might be based on a upper bound of available memory which we assume to be stored in a central statistics object:

```
boundedStackBehavior <- (|
  stackLimit     = ((centralStatistics memAvailable) / 10000).
|)
```

This policy is introduced at runtime and applies to all instance objects immediately. Now, any access to the `stackLimit` slot leads to a recalculation of its value.

4.4 Object specialization

In class-based languages, all objects are equal in their structure and behavior. In order to introduce special objects that differ in just one aspect, one has to construct a subclass and instantiate them. In SELF and dSelf, one can specialize objects simply by changing their slots and leaving `parent` unchanged.

While the above managed bounded stack introduces a general rule for the size of all stack objects, one can easily exempt an object from the rule by overwriting that slot again.

This can be done at creation time with:

```
priviledgedBoundedStackInstance <-
  ((stackInstance clone) parent: boundedStackBehavior) stackLimit: 9999
```

Here, a newly created clone of a normal stack is immediately specialized. This cal also be done later at runtime with

```
priviledgedBoundedStackInstance stackLimit: 5000
```

Note that the `priviledgedBoundedStackInstance` itself can be cloned for new objects, thus the specialized object in turn can serve as a class like entity.

5 dSelf Implementation

The dSelf-System consists of a dSelf Virtual Machine and a compiler. The compiler translates dSelf code to executable Java objects. The dSelf Virtual Machine receives input from the user and returns the output of the executed code. dSelf was implemented in Java by using RMI for the communication between objects, that are located in different dSelf Virtual Machines.

5.1 The Compiler

The compiler of dSelf was written by using the scanner-generator JFlex [Kle00] and the parser-generator CUP [Hud99]. It is based on the grammar as described in [ABC$^+$95] with some minor extensions and corrections.

The compiler provides two interfaces, one for the dSelf Virtual Machine and one as a stand-alone program for the user. The interface for the dSelf Virtual Machine makes it possible to compile an expression "on-the-fly". This generates a Java object which can be executed directly by invoking a method without a compile-link-cycle.

The interface for the user makes the compiler accessible as a separate Java program. A dSelf program is compiled to a Java class that can be loaded dynamically to a running dSelf system. The manual usage of the compiler is normally not necessary. When dSelf code it to be read, the loader first looks, if there is a Java class file with the same name and newer age. If there is one, then this precompiled version is linked to the system, if not then the script will be loaded and compiled. The precompiled version loads much faster, because there is no need to scan and parse its content by the dSelf compiler again.

5.2 The dSelf Virtual Machine

The dSelf Virtual Machine receives the input of the user and returns the output of the program. Currently it appears in two possible views, that can be chosen by the user. The first takes textual input from a terminal. The second view is a GUI as shown in figure 4, that was realized with the Swing libraries of Java. The GUI is more comfortable than the terminal view, because it supports more features like a slot browser and a history function, that repeats previous executed inputs.

As the dSelf virtual machine runs on any JVM, it can take advantage of utilizing any JVM based support to integrate other object models, such as the Java CORBA interface and others. The development of an interface from dSelf to Java objects is currently underway.

6 Related Work

There are several implementations of SELF for various platforms available. However, none of these adds distribution to the language, so we claim that dSelf is the first extension of SELF to support distribution. To our knowledge, there is no reference to distribution concepts in the original SELF literature. Also, there is a body of work on distributed OOP languages ([BGL98]), but still, none of these takes advantage of delegation and classlessness as dSelf does. Also, we know of no middleware system that facilitates a class-less object model.

The family of Actor-languages ([AH87]), however, has aspects similar to a distributed Self. Both as classless approaches to OO, both use delegation and employ a cloning mechanism. However, in Actors, the default mode of interaction by message passing is asynchronous and the buffer for incoming messages is explicitly represented with the so-called mailbox.

Obliq ([Car95]) is a recently proposed language for wide-area programming. In contrast to SELF, and therefore dSelf, Obliq is not delegation based. Messages have

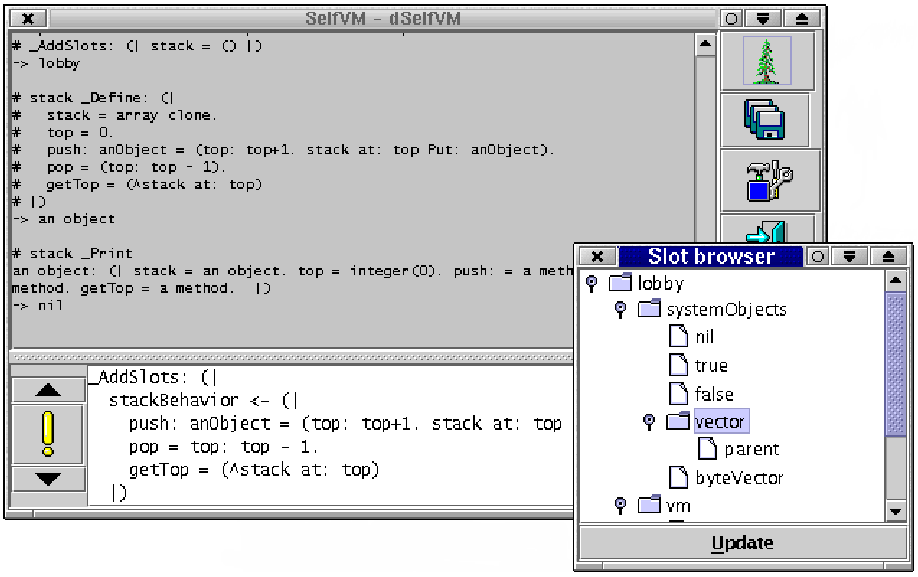

Fig. 4. The user interface of dSelf

to be understood by one of the methods an object contains. Instead of linking objects by a parent, Obliq copies methods during cloning. While this still allows to express inheritance-relations, it emphasizes the autonomy of object in a wide-area setting. In dSelf, this can be expressed with the help of local methods.

dSelf follows a slightly different goal. While delegation also enhances flexibility wrt. distribution, the slightly lower autonomy of objects raises *flexibility of change*. It is possible to change the behavior of delegating objects *themself* by changing the single, possibly remote, parent object. In Obliq, one can only change the behavior of a remote member of an object.

Repo ([Mac99]) is an extension of Obliq which introduces replication of objects. In addition to the Obliq notion of remote objects, Repo introduces replicated objects that are kept consistent and simple objects that are replicated but can become inconsistent. dSelf contains no notion of replicated objects but this might be introduced by extending the assignment mechanism to slots.

7 Conclusion and Outlook

Common middleware platforms build on a class-based object model. We have experimented with using a delegation-based, class-less object model to program distributed systems.

The language dSelf extends SELF with distributed objects and transparent resolution of remote references. This adds distributed inheritance and distributed instantiation to the language. dSelf thus adds flexibility in programming and executing distributed object-oriented programs.

Using delegation lessens dependencies in a distributed system. We have shown several examples where the concept can be applies with interesting advantages.

The current implementation of dSelf does not contain some mechanisms proposed for SELF. Amongst these are prioritized multiple inheritance, visibility modifiers for slots for encapsulation, mirror objects for reflection and a more comfortable initial dSelf world including support for graphical objects. These issues are subject for a reworked implementation.

Conceptually, we plan to study the extension of dSelf with the notion of processes and their coordination. Further flexibility of dSelf program might be gained from introducing mobile dSelf objects.

More information on dSelf and a downloadable version of the system can be found at http://www.cs.tu-berlin.de/˜tolk/dself.

References

[ABC+95] Ole Agesen, Lars Bak, Craig Chambers, Bay-Wei Chang, Urs Hölzle, John Maloney, Randall B. Smith, David Ungar, and Mario Wolczko. *The SELF 4.0 Programmer's Reference Manual*. Sun Microsystems, Inc. and Stanford Unversity, 1995.

[AH87] Gul Agha and Carl Hewitt. Concurrent programming using actors. In A. Yonozawa and M. Tokoro, editors, *Object-Oriented Concurrent Programming*, Computer Systems Series, pages 37–53. MIT Press, Cambridge, MA, 1987.

[BGL98] Jean-Pierre Briot, Rachid Guerraoui, and Klaus-Peter Löhr. Concurrency and distribution in object-oriented programming. *ACM Computing Surveys*, 30(3):291–329, September 1998.

[Car95] L. Cardelli. A Language with Distributed Scope. *Computing Systems*, 8(1):27–59, January 1995.

[Hud99] Scott E. Hudson. *CUP (v0.10k) LALR Parser Generator for Java*. Georgia Institute of Technology, Visualization and Usability Center, July 1999. http://www.princeton.edu/˜appel/modern/java/CUP/.

[Kle00] Gerwin Klein. *JFlex Version 1.3, The Fast Lexical Analyzer Generator for Java*, Oktober 2000. http://www.jflex.de.

[Knu00] Kai Knubben. Verteilte Implementierung der objektorientierten Programmiersprache SELF. Master's thesis, Technische Universität Berlin, 2000. In German.

[Mac99] B. MacIntyre. Repo: An Interpreted Language for Exploratory Programming of Highly Interactive, Distributed Applications. Technical Report Technical Report 99-34, Graphics, Visualization & Usability (GVU) Center at Georgia Tech, 1999.

[Ste87] Lynn Andrea Stein. Delegation is inheritance. *ACM SIGPLAN Notices*, 22(12):138–146, December 1987.

[SU95] Randall B. Smith and David Ungar. Programming as an Experience: The Inspiration for Self. In Walter G. Olthoff, editor, *ECOOP'95—Object-Oriented Programming, 9th European Conference*, volume 952 of *Lecture Notes in Computer Science*, pages 303–330, Åarhus, Denmark, 7–11 August 1995. Springer.

[Tai99] Antero Taivalsaari. Classes vs. Prototypes: Some Philosophical and Historical Observations. In James Noble, Antero Taivalsaari, and Ivan Moore, editors, *Prototype-Based Programming. Concepts, Languages and Applications*, chapter 1, pages 3–16. Springer Verlag, 1999.

[TK01] Robert Tolksdorf and Kai Knubben. dSelf – A Distributed SELF. Technical Report KIT-Report 144, Technische Universität Berlin, 2001.

[US87] David Ungar and Randall B. Smith. Self: The power of simplicity. *ACM SIGPLAN Notices*, 22(12):227–242, December 1987.