A Tutorial on dSelf

Kai Knubben

27th September 2001

Contents

1 Installation **5**
- 1.1 Requirements . . . 5
- 1.2 Starting dSelf . . . 6
  - 1.2.1 Using the jar-file for execution . . . 6
  - 1.2.2 Using the class-file for execution . . . 6
- 1.3 The contents of the jar-file . . . 6

2 The dSelf Virtual Machine and Compiler **9**
- 2.1 The dSelf Virtual Machine . . . 9
  - 2.1.1 Starting the dSelf Virtual Machine . . . 9
  - 2.1.2 Using the dSelf Virtual machine . . . 12
    - 2.1.2.1 The Terminal . . . 12
    - 2.1.2.2 The GUI . . . 13
- 2.2 The Compiler . . . 16
  - 2.2.1 How to start the compiler . . . 16

3 The dSelf Language **19**
- 3.1 Distributed dSelf Objects . . . 19
- 3.2 Prioritized multiple inheritance . . . 20
- 3.3 Method-holder-based privacy semantics . . . 21
- 3.4 Binary Methods . . . 22
- 3.5 Primitive Objects . . . 22
- 3.6 Mirrorobjects . . . 23
- 3.7 Local Data Slots . . . 23
- 3.8 Local Methods . . . 24
- 3.9 The primitive Messages . . . 25
  - 3.9.1 Primitive Messages for Objects of Type Integer . . . 26
  - 3.9.2 Primitive Messages for Objects of Type Short . . . 28
  - 3.9.3 Primitive Messages for Objects of Type Long . . . 30
  - 3.9.4 Primitive Messages for Objects of Type Float . . . 32
  - 3.9.5 Primitive Messages for Objects of Type Double . . . 35
  - 3.9.6 Primitive Messages for Objects of Type String . . . 37
  - 3.9.7 Primitive Messages for Objects of Type ByteVector . . . 38
  - 3.9.8 Primitive Messages for Objects of Type ObjectVector . . . 39

3.9.9 Primitive Messages for Ordinary Objects 40
3.9.10 Primitive Messages for the Debugger 42
3.9.11 Other primitive Messages 43

4 The Grammar 45

5 Further Information 47

Chapter 1

Installation

This small tutorial describes step-by-step how to install and use dSelf. For informations about the distribution rough draft please read [1]. As dSelf is a derivate of SELF you will find an overview about the language in [2] and informations about its basic design philosophy in [4]. The design and implementation of dSelf is described in more detail in [9].

1.1 Requirements

At first you need a file called *dSelf_v1_0_2.jar* (the version numbers may change in future releases), which is available at `http://www.cs.tu-berlin.de/~tolk/dself/`.

If you just want to use the precompiled version of dSelf as it is delivered in the jar-file, you only need the *Java Runtime Environment* [11]. The necessary version depends on your usage. If you want to start the *dSelf Virtual Machine* with a GUI, you will need the version 1.2 of the Java Runtime Environment (or update an old version for Swing-support), but when you only need a terminal for working, an older version may be sufficient. Anyway, to have the latest release of Java certainly will be the best choice.

If you want to compile dSelf by yourself, you will need some additional tools:

JDK	The Java Development Kit, Standard Edition, v 1.3 [12]
JFlex	A tool for generating the scanner of dSelf[7]
CUP	A tool for generating the parser of dSelf[6]
JavaDeps	A tool for automatic dependency tracking[8]
gmake	The GNU Make utility [13]

1.2 Starting dSelf

There are two ways for using dSelf. You can either start the *dSelf Virtual Machine* directly without extracting the jar-file, or by calling its class-file after extracting it. In the latter case you also have access to the compiler of dSelf and so it is recommend to extract it. Anyway, both ways will be described in the next two sections. For both approaches please make sure that the class-path variable of Java is set correctly to the directory that contains dSelf.

1.2.1 Using the jar-file for execution

If you execute the jar-file directly, the *dSelf Virtual Machine* will be started. You have to start the *Java Virtual Machine* with the option **-jar**, just like in the following example:

```
java -jar -Djava.security.policy=dSelf.policy dSelf_v1_0_2.jar -t -cr
```

As you can see, there are some additional informations about the security policies necessary, given by the option **-Djava.security.policy=** . Its argument must be a *Java Security Policy File* that can be created by using *policytool* which is part of the Java distribution. Alternatively you can extract one from the distributed dSelf file, by typing :

```
jar xf dSelf_v1_0_2.jar dSelf.policy
```

But be aware that this policy permits everything !

The options **-t** and **-cr** are processed by the *dSelf Virtual Machine* and will be described in a later section.

1.2.2 Using the class-file for execution

If you want to start the class file of the *dSelf Virtual Machine* directly, you first need to extract the jar-file with tool *jar* that is delivered with the Java environment. To do this, type the following line:

```
jar xf dSelf_v1_0_2.jar
```

Now you will find a file called **dSelfVM.class** in the current directory, that can be executed by calling the *Java Virtual Machine* like this:

```
java -Djava.security.policy=dSelf.policy dSelfVM -cr
```

1.3 The contents of the jar-file

When you have extracted the jar-file, you will find several files and directories. The most interesting contents are:

dSelfVM.class The *dSelf Virtual Machine*

dSelfComp.class The *dSelf Compiler*

gpl-license The license of the GPL

dSelf/ A directory with the implementation files of dSelf

doc/ A directory with the descriptions of the implementation, generated by *javadoc*

Chapter 2

The dSelf Virtual Machine and Compiler

This section describes the usage of the dSelf Virtual Machine and the dSelf Compiler. All start-parameters are explained and the basic elements of the dSelf Virtual Machine-GUI are explained in detail.

2.1 The dSelf Virtual Machine

The dSelf Virtual Machine is main component of dSelf. It evaluates the scripts and makes it possible to change the state of the system interactively by providing a shell. The shell can be started either in terminal or GUI mode.

2.1.1 Starting the dSelf Virtual Machine

The previous sections already described how to start the *dSelf Virtual Machine*, so here only the options will be enumerated. All possible options are:

-h gives a little help about the usage

-v verbose, give informations when starting

-t start with terminal front-end

-cr create registry when not found

-f <name> load script when starting

-b <name> bind the VM to the given name (default: dSelfVM)

-r <name> re-bind the VM to the given name

-ds activate debugger of scanner at start

-dc activate parser-debugger(CUP) at start

-dp activate parse-tree debugger (with parenthesis)

-di activate parse-tree debugger (with indention)

-dl activate search path debugger (lookup algorithm)

Printing the Usage

The option **-h** prints a little list with all possible options that can be used. It looks similar to the list above.

Verbose mode

With **-v** you can start in *verbose mode* to obtain some additional information about the state of the *dSelf Virtual Machine*. The output that will be generated looks like this:

```
Name of VM...........................“vm” (bound)
Chosen front-end.....................GUI
Load file............................"smallWorld.dSelf"
debugger of scanner..................disabled
debugger of parser(CUP)..............disabled
Generate flat parse-trees............enabled
Generate indented parse-trees........disabled
Show search path of lookup algorithm..disabled
Try to bind the name of the VM to the RMI-Registry.....OK
Initialize the dSelf-world............................OK
Generate the GUI......................................OK
```

This information can be very useful if something went wrong at start time.

Choosing a view

The default view of the *dSelf Virtual Machine* is a GUI that was created with Swing. If you want for some reason (e.g. you haven't installed Swing) to start with a terminal instead, you can do this with the option **-t**.

Creating a RMI-Registry

As dSelf is a distributed language, there must be a way to connect to another *dSelf Virtual Machine* by using some kind of middleware. The middleware used for dSelf is Java-RMI. In order to locate remote machines, dSelf must be connected to a registry. You can do this by starting a new registry manual by starting the program *rmiregistry* that is provided by Java environment. Alternatively you can advise the *dSelf Virtual Machine* to create a new one itself by using the option **-cr** if there is no registry already running. This can be useful when you start some instances of dSelf, each with an own registry, which are all located on different hosts and therefore using separate address spaces.

Note that the life time of an automatically created registry is bound to the life time of the *dSelf Virtual Machine* that created it ! So if you start two instances of dSelf on one machine by using the option **-cr**, the first one will create a new registry and the second one will use it, too. If the first one terminates, the second one won't find the registry any more, because it terminated with the first *dSelf Virtual Machine* ! As you can expect this will lead to undesirable situations. In case of some instances that are located on the same machine, starting the registry manual is the best choice.

Binding to the Registry

In the previous section one saw how to create a RMI-Registry. Now we want to use it with the options **-b <name>** or **-r <name>**. When several *dSelf Virtual Machines* were started, one would like to connect them. In order to connect them, one first needs a name to identify it. The IP-address of its location provides not enough information, because there could be more instances of dSelf on the same computer. With option **-b <name>** one can bind *dSelf Virtual Machine* the with a given name to the registry. It might happen that for some reason the preferred name is already bound to the registry, while no other program is using it. E.g., when one kills the dSelf process, its virtual machine has no chance to unbind its name at the registry. In case of such a situation, one can re-bind the name to the registry by using the option **-r <name>**, while the old binding is deleted.

If both options are omitted, the default name "dSelfVM" is used, but then one should be sure that only one *dSelf Virtual Machine* is running on the same computer.

Starting a script

After the *dSelf Virtual Machine* has booted, one can advice it to start a script with the given name immediately with the option **-f <name>**. This has the same effect as typing this input directly after booting:

```
'<name>' _RunScript
```

The debuggers

The *dSelf Virtual Machine* offers several debuggers to the user. They provide information of the current state of the scanner, parser or *lookup algorithm* of dSelf.

The debugger of the scanner

With option **-ds**, the debugger of the scanner will be enabled. It will print all scanned tokens on the standard output stream.

The debugger of the parser

With option **-dc**, the debugger of the parser (CUP) will be enabled. It will print all actions (shift/reduce) that were done by the parser. For further information about it refer to the documentation of [6]. This option is very useful in combination with option **-ds**.

Printing the parse tree

Do you have problems by analyzing the grammar of dSelf ? With option **-dp** the parser will generate a parse tree for each parsed input. The structure of this tree is represented by using parenthesis. The grammar can be found in [9].

Printing the parse tree with indention

The previously mentioned option printed the parse tree in a compact but not very readable way. Option -di prints the same tree more attractive by using indention for displaying its structure.

Printing the search paths

dSelf uses - like SELF - an *lookup algorithm* for resolving the inheritance dependencies of objects. By using option **-dl** one can advice the *dSelf Virtual Machine* to print the search path, when an object is delegating a message to another object. This might be useful for debugging and learning purposes. As the *lookup algorithm* of dSelf is equal to its counterpart in SELF, please refer for a description to [2].

2.1.2 Using the dSelf Virtual machine

There are two possibilities for working with the dSelf Virtual machine. The first one is a terminal that needs only few resources and the second one is a more comfortable GUI. The latter one needs Swing and could be a bit slow on older computers.

2.1.2.1 The Terminal

When the *dSelf Virtual machine* is started in terminal mode, one will get a shell like this:

```
Welcome to dSelf !
#dSelf[0]> _Print
lobby: (| systemObjects* = systemObjects. vm* = vm. |)
-> nil
#dSelf[1]> _AddSlots: (|
x = (| a. b. c |)
|)
-> lobby
```

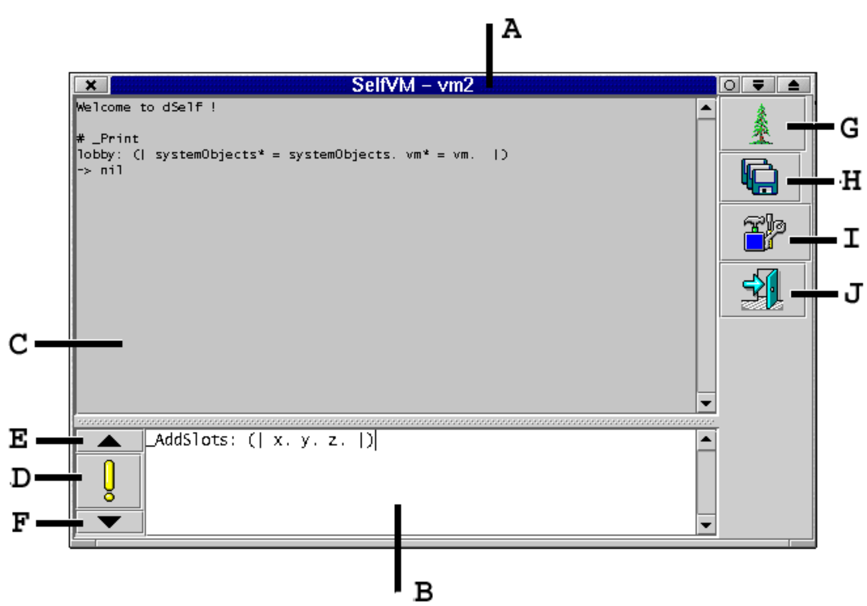
```
#dSelf[2]> x _Print
an object: (| a <- nil. b <- nil. c <- nil. |)
-> nil
#dSelf[3]>
```

After the *dSelf Virtual machine* has been started, one will find a prompt that is waiting for some input. Now one can enter any (syntacticly correct) command. After pressing the *enter* key, the input will be evaluated and executed directly.

An input can be split over more than one line. The terminal uses internally a parenthesis counter, that recognizes the end of an input when the number of the open parentheses matches with its closed counterparts.

2.1.2.2 The GUI

When one starts the *dSelf Virtual machine* without the option **-t**, a window will appear:



The window consists of several elements:

[A] The title bar contains an text of the form "dSelfVM <name>", where <name> is the name that was specified by the options **-b <name>** or **-r <name>**.

[B] This text field accepts the input of the user. In contrast to the terminal, it treats the *enter key* not as an "end of input" signal and there is no parenthesis counter. The input is evaluated by clicking on the input button (see [D]) .

[C] This text field displays the output of the *dSelf Virtual Machine*. For ease of reading, each user-input is declared with a leading hash-mark sign ("#"). Previously generated outputs can be reached by using the scrollbar.

[D] By clicking on the input button one advises the *dSelf Virtual Machine* to evaluate the content of the input field. Alternatively one can press the *enter key* in combination with the *ALT key* for accelerated input.

[E] By clicking on the up-array button one can respell the previously entered inputs. The GUI has an history function that remembers previous inputs.

[F] The down-arrow button offers the opposite effect of the up-arrow button.

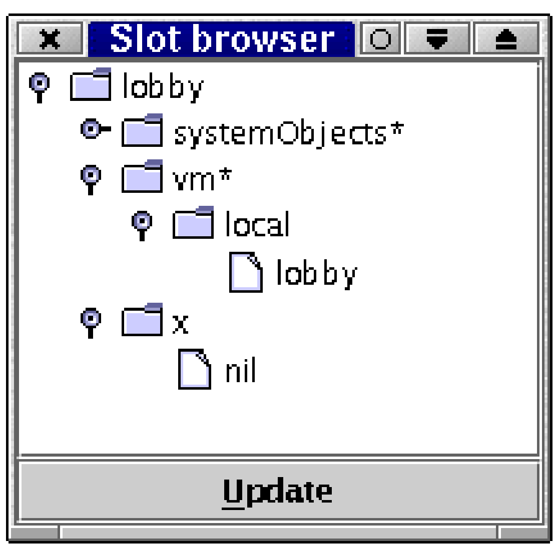
[G] This button opens the slot browser of the *dSelf Virtual Machine* that is described in a later section.

[H] Opens a dialog for choosing a script that will be loaded and evaluated. Has the same effect as '<script name>' _RunScript on the input text field.

[I] Opens a dialog for changing the preferences like color and fonts.

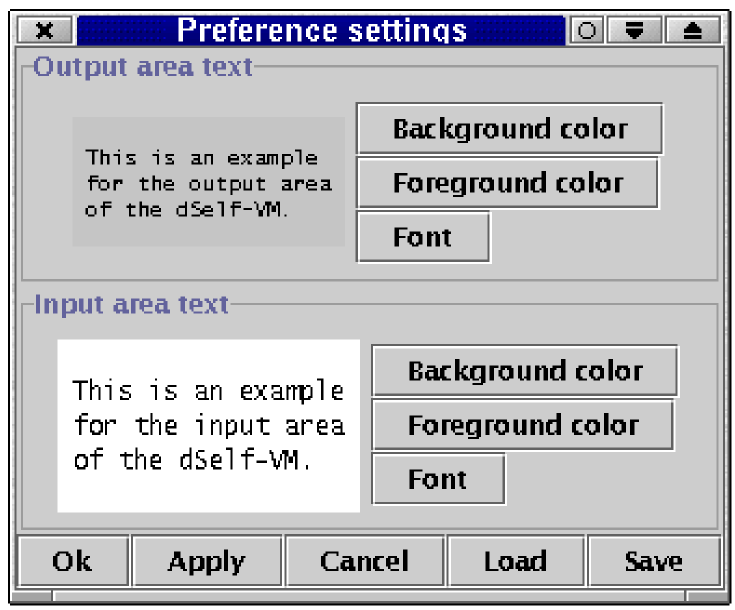
[J] Exit the *dSelf Virtual Machine.*

The Slot Browser



By using the slot browser one can explore the slot contents of the objects of dSelf. The tree can be expanded by clicking on the nodes, where the rood node represents the *lobby*. The slot browser don't updates its contents automatically, so one can force it to update its nodes by clicking on the button that is located at the bottom.

The Preferences Window



The window for the preference settings offers the user the possibility to customize the appearance of the *dSelf Virtual Machine*. The colors of the text fields and the chosen fonts can be selected. The text fields give an impression about the choices made by the user without changing the current settings of the main window.

2.2 The Compiler

The compiler of dSelf offers two interfaces. One for the user as a separate program and one for the *dSelf Virtual Machine* as an class that evaluates inputs "on-the-fly". As the purpose of this tutorial is not to introduce into compiler techniques, the latter is not explained here. For further information on this topic, the reader is referred to [9].

2.2.1 How to start the compiler

The compiler can be started by typing this the command line of a shell:

```
java dSelfComp [options] <filename>
```

The compiler of dSelf can be started with several options:

-h prints the usage

-r recognizer (don't generate code)

-j invoke java-compiler and generate class file

-jc <name> the name of the java-compiler (default: javac)

-jm <size> set preferred size for methods(default: 10000 Bytes)

-c <path> set the CLASSPATH for java compiler (default: ".")

-ds print debug-information of scanner

-dc print debug-information of parser (CUP)

-dp print the parse-tree (with parenthesis)

-di print the parse-tree (with indentation)

Printing the Usage

The option **-h** prints a little list with all possible options that can be used. It looks similar to the list above.

Recognizer

When the compiler starts with option **-r** it will generate no code. This is useful, if the user just wants to check the syntax of a script.

Invoke the Java Compiler

As a default setting, the compiler of dSelf only generates code for a new Java class in a Java file ("filename.java") and the compiler of Java must be called by the user in order to get a class file ("filename.class") . With option **-j**, this work is done by the compiler is of dSelf.

Preferred Java Compiler

With option **-jc <name>** the user can specify the name of the preferred Java compiler. As default setting "javac" is called.

Class path for Java Compiler

With option **-c <path>** the user can specify the class path, that is needed by the Java compiler.

Method length

The dSelf compiler generates as its result the source code for a new Java class that represents the translated dSelf program. By creating methods for this class it might happen that the generated code becomes huge. Due to limitations of some *Java Virtual Machines*, it is not possible to create methods, whose size is larger than 64 kBytes. To handle this problem, the user can specify with the option **-m <size>** the preferred size of the generated methods. Without this option, the size of the generated methods is about 10 kBytes and shouldn't be a problem for a *Java Virtual Machine.*

The debuggers

The *dSelf Virtual Machine* offers several debuggers to the user. They provide information of the current state of the scanner and the parser of dSelf.

The debugger of the scanner

With option **-ds**, the debugger of the scanner will be enabled. It will print all scanned tokens on the standard output stream.

The debugger of the parser

With option **-dc**, the debugger of the parser (CUP) will be enabled. It will print all actions (shift/reduce) that were done by the parser. For further information about it refer to the documentation of [6]. This option is very useful in combination with option **-ds**.

Printing the parse tree

Do you have problems by analyzing the grammar of dSelf ? With option **-dp** the parser will generate a parse tree for each parsed input. The structure of this tree is represented by using parenthesis.

Printing the parse tree with indention

The previously mentioned option printed the parse tree in a compact but not very readable way. Option -di prints the same tree more attractive by using indention for displaying its structure.

Chapter 3

The dSelf Language

In this chapter the language dSelf itself will be described. As dSelf is a derivate of SELF, only the differences to SELF are explained. So the reader is expected to know SELF and its basic concepts which are described in [2, 3, 4] [1].

dSelf was developed with the claim to be as close to SELF as possible. Unfortunately this was not possible for each detail because of the distribution of dSelf, so we were forced to implement some features in a different manner. Hence, dSelf is a derivate of SELF and not a 100% compatible clone, because the semantics of some language constructs differ in a distributed context(e.g., boolean objects). Furthermore, some features of SELF weren't implemented in dSelf because they didn't prove them self in practice like *prioritized multiple inheritance* or the *method-holder-based privacy semantics*[2]. The differences to SELF are explained in the next sections.

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 provides 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 (e.g. '//127.0.0.1:1099/myVM').

[1] It is planned to extend this tutorial with an introduction section to dSelf in a future release.

[2] This features were implemented in SELF in some earlier versions. In later releases (e.g., SELF 4.0) they were removed because of the bad experiences the authors made with it.

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 [2], except that remote references are followed across machine-boundaries.

3.2 Prioritized multiple inheritance

In [10] the authors of SELF introduced an extension for their language called *prioritized multiple inheritance*. With this new concept, the user is able to order the parent slots (*ordered inheritance*) of an object by declaring its name with additional asterisks. When the *lookup algorithm* (an algorithm that realizes the inheritance mechanism, see [2] for details) reaches an object with more than one parent slot, it first visits the object contained in the parent slot with the least trailing asterisks. If that search fails, it goes on with the second one and so on until a matching slot was found. When two or more slots with the same priority are found, each of them will be examined in an arbitrary order (*unordered inheritance*).

```
object = (|
  parent1* = (...).
  parent2** = (...).
  parent3*** = (...).
  ...
|)
```

In the example above, three parent slots with different priorities were declared. If **object** would receive a message which one's selector wouldn't match with the name of one of its slots, it has to delegate this message to its parent objects which are available via its parent slots. When no priority would be given, then it would be send to all parents and ambiguous situations could occur when more than one matching slot would be found. To prevent such a situation, one can prioritize the parent slots in order to the influence the inheritance semantics, so in the example only the object in slot **parent1** would receive the message. If this message was understood its corresponding slot would be evaluated and all slots with similar names in **parent2** and **parent3** ignored.

This extention sounds simple, but in [5] Randall B. Smith and David Ungar wrote:

> Early in the evolution of Self we made three mistakes: prioritized multiple inheritance, the sender-path tie-breaker rule, and method-holder-based privacy semantics. [...] But each feature also caused us no end of confusion. The prioritization of multiple parents implied that Self's "resend" (call-next-method) lookup had to be prepared to backup down parent links in order follow lower-priority paths. The resultant semantics took five pages to write down, but we preserved. After a year's experience with the features, we found that each of the members of the Self group had wasted no small amount of time chasing "compiler bugs" that were merely unforseen consequences of these features. It became clear that the language had strayed from its original path.

Because of the authors experiences, we decided not to support prioritized multiple inheritance in dSelf and other solutions are currently investigated.

3.3 Method-holder-based privacy semantics

Also in [10] the authors of SELF introduced an extension for their language called *method-holder-based privacy semantics*. The language was extended with a circumflex and an underscore symbol which describe the visibility of slots. The circumflex declares a slot as public and the underscore a private slot that is only accessible for the object that contains it. For assignment slot declarations are combinations of this symbols possible where the one symbol declares the accessibility of the assignment slot itself and the other one the accessibility of its corresponding data slot.

For example:

```
object = (|
    ^ theAnswer = (^ 42).
    _ someText  = 'I'm private'.
  ^ _ aNumber   <- 1.
  _ ^ someSlot  <- 'foo'
|)
```

The method slot **theAnswer** represents a public slot that is accessible to all other objects, while **someText** is a private data slot which is only visible to **object**. **aNumber** declares a public data slot whoes corresponding assignment slot **aNumber:** is a private member of this object. **someSlot** is assignable for all objects but its content can only be read by **object** because it is private.

The reason, why we did not take on this concept the same as described in the previous section.

3.4 Binary Methods

In SELF it is impossible to combine different operators in one expression without parenthesizes. An expression like

```
x + y + z
```

is allowed, while

```
x + y - z
```

is not. If one wants to write such a formula, one has to embed **x + y** or **y - z** into parenthesis because there is no build-in mechanism for dissolving the priority of operators like in C or Java. In dSelf it is allowed to write such an expression without parenthesizes and the associativity in left to right, i.e., **x + y** in the example is evaluated first.

The reason for this behavior is of technical nature[3] and is described in [9] in more detail.

3.5 Primitive Objects

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.

Type	Size	Declaration	min. Value	max. Value
Short	16 Bit	0_s or 0_S	-32768	32767
Integer	32 Bit	0	-2147483648	2147483647
Long	64 Bit	0_l or 0_L	-9223372036854775808	9223372036854775807
Float	32 Bit	0.0	±3.40282347E+38	±1.40239846E-45
Double	64 Bit	0.0_d or 0.0_D	±1.79769313486231570E+308	±4.94065645841246544E-324

[3]dSelf uses an LALR(1) parser generator for syntax analysis which is a context free grammar while the parser of SELF is “handmade” without this limitations. For detecting such dependencies one would need the power of Chomsky type 1 grammars (context sensitivity).

The types listed in the above table are the different kinds of numbers implemented in dSelf[4]. The underscore sign is necessary because of the radix notations. As in SELF in dSelf one can specify a value by using bases from 2 to 36 followed by 'r' or 'R'. For bases greater than 10 the characters 'a' through 'z' represent digit values 10 through 35. Some possible values are for example:

```
2r1001, 16rff12, 36r1zl, 36r1zl_l
```

Without an underscore it wouldn't be possible to determine, if **36r1zl** represents the integer value 2577 or the long value 71.

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.

3.6 Mirrorobjects

In dSelf are no *mirror objects* implementet till now. Such objects are used in SELF for examing and manipulating objects in a comfortable way. When mirror objects will be also available in dSelf, one will have the capability to manipulte local and remote objects.

3.7 Local Data Slots

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

[4] dSelf was implemented in Java, so the types correspond not only by chance.

```
(| <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.

3.8 Local Methods

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.[5]

For example:

```
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
|).
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

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

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.

3.9 The primitive Messages

The primitive messages described in this section are all available in dSelf. They are declared with a leading underscore sign that signifies that they are build-in messages and cannot be modified by the user. Some of these messages can fail resulting in a runtime-error. Such errors can be caught by an additional **IfFail:** keyword with a following block object that comprises some code that shall be executed in case of such situation. This block can have two arguments with information about the kind of error and the name of the failing message. For example:

```
x _At: 4 IfFail: [
  | :error. :name|
  (name, 'failed with:', error) print.
]
_At: failed with: badIndexError
```

Here, the primitive message **_At:** failed with a **badIndexError**, because 4 was out of vector bounds. Because of this, two strings with the necessary information were generated and sent to the trailing **IfFail:** block for (simple) exception handling. A **IfFail:** block can have one of three forms:

unary messages:

<object> <messagename>IfFail: <block>
(e.g. 12 _IntAsFloatIfFail: [...])

binary messages:

<object> <operator> <object> IfFail: <block>
(e.g. 12 + 'a' IfFail: [...])

keyword messages:

<object> <keyword> <object> {<keyword> <object>}
(e.g. 12 _IntAdd: 'a' IfFail: [...])

Note that there is no separator between an unary message and the trailing "IfFail:"-keyword !

The following errors can occur:

error name	kind of error
badTypeError	The receiver of this message or one of its arguments has a wrong type.
badIndexError	The specified position in a vector was out of bounds.
divisionByZeroError	Occures when attempted to divided a number by zero.
ioError	An error occured when accessing a file.
primitiveFailedError	A primitive message failed, e.g. when a wrong argument-value was given.
securityError	Occures when the current primitive has no permission to acces a demanded resource.

3.9.1 Primitive Messages for Objects of Type Integer

_IntAdd:

Adds the argument object of type *integer* to the receiver. Possible over- and underflows are not checked.

_IntAnd:

The receiver and argument objects (both of type *integer*) are combined by using the the logical AND-operation.

_IntArithmeticShiftRight:

The resulting object corresponds to the receiver, right-shifted by the number of bits specified by the argument (an *integer*) while the sign bit is preserved.

_IntAsDouble

Returns the receiver as an object of type *double*.

_IntAsFloat

Returns the receiver as an object of type *float*.

_IntAsLong

Returns the receiver as an object of type *long*.

_IntAsShort

Returns the receiver as an object of type *short*. Possible over- and underflows are not checked.

_IntDiv:

Divides the receiver by the argument object, also of type *integer*. May fail with *divisionByZeroError*.

_IntEQ:

Checks the receiver and argument object for equivalence. When two objects are *_IntEQ:* they are also *_Eq:*.

_IntGE:

Returns *true*, if the receiver is greater or equal to the argument object of type *integer*. Otherwise *false* is returned.

_IntGT:

Returns *true*, if the receiver is greater than the argument object of type *integer*. Otherwise *false* is returned.

_IntLE:

Returns *true*, if the receiver is less or equal to the argument object of type *integer*. Otherwise *false* is returned.

_IntLT:

Returns *true*, if the receiver is less than the argument object of type *integer*. Otherwise *false* is returned.

_IntLogicalShiftLeft:

The resulting object corresponds to the receiver, left-shifted by the number of bits specified by the argument (an *integer*). There is no checking for overflow.

_IntLogicalShiftRight:

The resulting object corresponds to the receiver, right-shifted by the number of bits specified by the argument (an *integer*) while the sign bit is perserved.

_IntMod:

Calculates the receiver modulo the value of the argument object of type *integer*. May fail with *divisionByZeroError*.

_IntMul:

Multiplies the receiver with the argument object of type *integer*. Possible over- and underflows are not checked.

_IntNE:

Returns true, if the receiver is unequal to the argument object of type *integer*. Otherwise *false* is returned.

_IntOr:

The receiver and argument objects (both of type *integer*) are combined by using the the logical OR-operation.

_IntSub:

Subtracts the argument object of type integer from the receiver. Possible over- and underflows are not checked.

_IntXor:

The receiver and argument objects (both of type *integer*) are combined by using the the logical XOR-operation.

3.9.2 Primitive Messages for Objects of Type Short

_ShortAdd:

Adds the argument object of type *short* to the receiver. Possible over- and underflows are not checked.

_ShortAnd:

The receiver and argument objects (both *short*) are combined by using the the logical AND-operation.

_ShortArithmeticShiftRight:

The resulting object corresponds to the receiver, right-shifted by the number of bits specified by the argument (a *short*) while the sign bit is preserved.

_ShortAsDouble

Returns the receiver as an object of type *double*.

_ShortAsFloat

Returns the receiver as an object of type *float*.

_ShortAsInt

Returns the receiver as an object of type *integer*.

_ShortAsLong

Returns the receiver as an object of type *long*.

_ShortDiv:

Divides the receiver by the argument object, also of type *short*. May fail with *divisionByZeroError*.

_ShortEQ:

Checks the receiver and argument object of type *short* for equivalence. When two objects are *_IntEQ:* they are also *_Eq:*.

_ShortGE:

Returns *true*, if the receiver is greater or equal to the argument object of type *short*. Otherwise *false* is returned.

_ShortGT:

Returns *true*, if the receiver is greater than the argument object of type *short*. Otherwise *false* is returned.

_ShortLE:

Returns **true**, if the receiver is less or equal to the argument object of type short. Otherwise *false* is returned.

_ShortLT:

Returns *true*, if the receiver is less than the argument object of type *short*. Otherwise *false* is returned.

_ShortLogicalShiftLeft:

The resulting object corresponds to the receiver, left-shifted by the number of bits specified by the argument (a *short*). There is no checking for overflow.

_ShortLogicalShiftRight:

The resulting object corresponds to the receiver, right-shifted by the number of bits specified by the argument (a *short*) while the sign bit is preserved.

_ShortMod:

Calculates the receiver modulo the value of the argument object of type *short*. May fail with *divisionByZeroError*.

_ShortMul:

Multiplies the receiver with the argument object of type short. Possible over- and underflows are not checked.

_ShortNE:

Returns *true*, if the receiver is unequal to the argument object of type *short*. Otherwise *false* is returned.

_ShortOr:

The receiver and argument objects (both *short*) are combined by using the the logical OR-operation.

_ShortSub:

Subtracts the argument object of type *short* from the receiver. Possible over- and underflows are not checked.

_ShortXor:

The receiver and argument objects (both of type *short*) are combined by using the the logical XOR-operation.

3.9.3 Primitive Messages for Objects of Type Long

_LongAdd:

Adds the argument object of type *long* to the receiver. Possible over- and underflows are not checked.

_LongAnd:

The receiver and argument objects (both *long*) are combined by using the the logical AND-operation.

_LongArithmeticShiftRight:

The resulting object corresponds to the receiver, right-shifted by the number of bits specified by the argument (a *long*) while the sign bit is preserved.

_LongAsDouble

Returns the receiver as an object of type *double*.

_LongAsFloat

Returns the receiver as an object of type *float*.

_LongAsInt

Returns the receiver as an object of type *integer*. Possible over- and underflows are not checked.

_LongAsShort

Returns the receiver as an object of type *short*. Possible over- and underflows are not checked.

_LongDiv:

Divides the receiver by the argument object, also of type *long*. May fail with *divisionByZeroError*.

_LongEQ:

Checks the receiver and argument object of type *long* for equivalence. When two objects are *_IntEQ:* they are also *_Eq:*.

_LongGE:

Returns *true*, if the receiver is greater or equal to the argument object of type *long*. Otherwise *false* is returned.

_LongGT:

Returns *true*, if the receiver is greater than the argument object of type long. Otherwise *false* is returned.

_LongLE:

Returns *true*, if the receiver is less or equal to the argument object of type *long*. Otherwise *false* is returned.

_LongLT:

Returns *true*, if the receiver is less than the argument object of type *long*. Otherwise *false* is returned.

_LongLogicalShiftLeft:

The resulting object corresponds to the receiver, left-shifted by the number of bits specified by the argument (a *long*). There is no checking for overflow.

_LongLogicalShiftRight:

The resulting object corresponds to the receiver, right-shifted by the number of bits specified by the argument (a *long*) while the sign bit is preserved.

_LongMod:

Calculates the receiver modulo the value of the argument object of type *long*. May fail with *divisionByZeroError*.

_LongMul:

Multiplies the receiver with the argument object of type *long*. Possible over- and underflows are not checked.

_LongNE:

Returns *true*, if the receiver is unequal to the argument object of type *long*. Otherwise *false* is returned.

_LongOr:

The receiver and argument objects (both *long*) are combined by using the the logical OR-operation.

_LongSub:

Subtracts the argument object of type *long* from the receiver. Possible over- and underflows are not checked.

_LongXor:

The receiver and argument objects (both longs) are combined by using the the logical XOR-operation.

3.9.4 Primitive Messages for Objects of Type Float

_FloatAdd:

Adds the argument object of type *float* to the receiver. Possible over- and underflows are not checked. Possible over- and underflows are not checked. Maximum values are positive or negative infinity.

_FloatAsDouble

Returns the receiver as an object of type *double*.

_FloatAsInt

Returns the receiver as an object of type *integer*.

_FloatAsLong

Returns the receiver as an object of type *long*.

_FloatAsShort

Returns the receiver as an object of type *short*. Possible over- and underflows are not checked.

_FloatCeil

Returns the greatest integral value greater or equal to the receiver object by rounding towards positive infinity. The resulting object is of type *float*.

_FloatDiv:

Divides the receiver by the argument object, which is also of type *float*. May fail with *divisionByZeroError*.

_FloatEQ:

Checks the receiver and argument object of type *float* for equivalence. Different to *_Eq:*, *_FloatEQ:* distinct between 0.0 and -0.0 i.e., 0.0 *_FloatEq:* -0.0 returns *true* and 0.0 *_Eq:* -0.0 returns *false*.

_FloatFloor

Returns the greatest integral value less or equal to the receiver object by rounding towards negative infinity. The resulting object is of type *float*.

_FloatGE:

Returns *true*, if the receiver is greater or equal to the argument object of type float. Otherwise *false* is returned.

_FloatGT:

Returns *true*, if the receiver is greater than the argument object of type *float*. Otherwise *false* is returned.

_FloatLE:

Returns *true*, if the receiver is less or equal to the argument object of type float. Otherwise *false* is returned.

_LongLT:

Returns *true*, if the receiver is less than the argument object of type *float*. Otherwise *false* is returned.

_FloatMod:

Calculates the receiver modulo the value of the argument object of type *float*. May fail with *divisionByZeroError*.

_FloatMul:

Multiplies the receiver with the argument object of type *float*. Possible over- and underflows are not checked. Maximum values are positive or negative infinity.

_FloatNE:

Returns *true*, if the receiver is unequal to the argument object of type *float*. Otherwise *false* is returned.

_FloatPrintString

Returns the receiver as a *string* object.

_FloatPrintStringPrecision:

Returns the receiver as a *string* object. The argument object of type *integer* specifies the number of digits after the decimal point.

_FloatRound

Rounds the receiver towards the nearest integral value. 0.5 is rounded towards the next even number, e.g., 1.5 rounds to 2 and 2.5 rounds to 2. The resulting object is also of type *float*.

_FloatSub:

Subtracts the argument object of type *float* from the receiver. Possible over- and underflows are not checked. Maximum values are positive or negative infinity.

_FloatTruncate

Rounds the receiver towards the nearest integral value. Numbers are rounded towards zero, e.g., 1.x is rounded to 1.0 and -1.x rounds to -1. The resulting object is also of type float.

3.9.5 Primitive Messages for Objects of Type Double

_DoubleAdd:

Adds the argument object of type *double* to the receiver. Possible over- and underflows are not checked. Possible over- and underflows are not checked. Maximum values are positive or negative infinity.

_DoubleAsFloat

Returns the receiver as an object of type *float*.

_DoubleAsInt

Returns the receiver as an object of type *integer*.

_DoubleAsLong

Returns the receiver as an object of type *long*.

_DoubleAsShort

Returns the receiver as an object of type *short*.

_DoubleCeil

Returns the greatest integral value greater or equal to the receiver object by rounding towards positive infinity. The resulting object is of type *double*.

_DoubleDiv:

Divides the receiver by the argument object, which is also of type *double*. May fail with *divisionByZeroError*.

_DoubleEQ:

Checks the receiver and argument object of type *double* for equivalence. Different to *_Eq:*, *_FloatEQ:* distinct between 0.0 and -0.0 i.e., 0.0 *_FloatEq:* -0.0 returns *true* and 0.0 *_Eq:* -0.0 returns *false*.

_DoubleFloor

Returns the greatest integral value less or equal to the receiver object by rounding towards negative infinity. The resulting object is of type *double*.

_DoubleGE:

Returns *true*, if the receiver is greater or equal to the argument object of type *double*. Otherwise *false* is returned.

_DoubleGT:

Returns *true*, if the receiver is greater than the argument object of type *double*. Otherwise *false* is returned.

_DoubleLE:

Returns *true*, if the receiver is less or equal to the argument object of type *double*. Otherwise *false* is returned.

_DoubleLT:

Returns *true*, if the receiver is less than the argument object of type *double*. Otherwise *false* is returned.

_DoubleMod:

Calculates the receiver modulo the value of the argument object of type *double*. May fail with *divisionByZeroError*.

_DoubleMul:

Multiplies the receiver with the argument object of type *double*. Possible over- and underflows are not checked. Maximum values are positive or negative infinity.

_DoubleNE:

Returns *true*, if the receiver is unequal to the argument object of type *double*. Otherwise *false* is returned.

_DoublePrintString

Returns the receiver as a *string* object.

_DoublePrintStringPrecision:

Returns the receiver as a *string* object. The argument object of type *integer* specifies the number of digits after the decimal point.

_DoubleRound

Rounds the receiver towards the nearest integral value. 0.5 is rounded towards the next even number, e.g., 1.5 rounds to 2 and 2.5 rounds to 2. The resulting object is also of type *double*.

_DoubleSub:

Subtracts the argument object of type *double* from the receiver. Possible over- and underflows are not checked. Maximum values are positive or negative infinity.

_DoubleTruncate

Rounds the receiver towards the nearest integral value. Numbers are rounded towards zero, e.g., 1.x is rounded to 1.0 and -1.x rounds to -1. The resulting object is also of type *double*.

3.9.6 Primitive Messages for Objects of Type String

_BitSize

The receiver of this message is a *string* that contains the words 'short', 'integer', 'float', 'long' or 'double'. The result is an *integer* object that specifies the number of bits for the corresponding data type.

_RunScript

This primitive message causes the system to load and execute a script whose name is specified by the receiver.

_StringAsByteVector

Converts to receiver to an object of type *bytevector*. Note that the receiver is an unicode *string* with 2 Bytes for each character, so the kind of conversion towards a byte representation depends on the under laying platform !

_StringAt:

Returns an object of type *integer* between 0 and 65535 that represents the unicode character at the specified position (an *integer*).

_StringConcatenate:

Returns the concatenation (a *string*) of the receiver of this message with its argument object of type *string*.

_StringGetSubFrom:

Returns the suffix (also a *string*) of the receiver starting at the specified position of type *integer*.

_StringGetSubFrom:To:

Returns the substring of the receiver that is located between the specified positions (both of type *integer*).

_StringPrint

Prints the content of the receiver on the standard output device.

_StringSize

Returns the size of the receiver as an object of type *integer*.

3.9.7 Primitive Messages for Objects of Type ByteVector

_ByteAt:

Returns an integer between 0 and 255 that represents the byte at the specified position (an *integer*) of this *bytevector*. May fail with *badIndexError*.

_ByteAt:Put:

Puts an integer at the specified position (an *integer*) of this bytevector. The value must be a number between 0 and 255, otherwise an error will occur. May fail with *badIndexError*.

_ByteSize

Returns the size of the receiver as an object of type *integer*.

_ByteVectorAsString

Converts this *bytevector* to a *string* object. Note that this conversion depends on the underlying platform (the operating system).

_ByteVectorCompare:

Compares two objects of type *bytevector* and returns an object of type integer that represents the result of the comparison. The values can be -1 (less than), 0 (equal to) or 1 (greater than). When both *bytevectors* are identical 0 ibeginnings returned, otherwise all bytes at the same index are compared, beginning from the lowest index. When two bytes at the same index differ, then 1 is returned, if the byte of the first *bytevector* is greater than the corresponding byte of the latter. Otherwise -1 is returned. When the first bytevector is a prefix of the argument bytevector the value1 is returned, in opposite case -1.

_ByteVectorConcatenate:

Returns the concatenation (a *bytevector*) of the receiver of this message with its argument object of type *bytevector*.

_CloneBytes:Filler:

Clones an prefix of the receiver up to the position specified by *_CloneBytes:* (an *integer*). When the specified position is greater than the size of the receiver, the rest of the new *bytevector* is filled with the byte (an *integer* between 0 and 255) specified by *Filler:*.

_CopyByteRangeDstPos:Src:SrcPos:Length:

Copies a number of bytes (specified by *Length:*, an *integer*) into the receiver *bytevector* at the position specified by *DstPos:* (an *integer*) from the bytevector at the position specified by *SrcPos:* (an *integer*). May fail with *badIndexError*.

_StringPrint

Prints the content of the receiver as a *string* on the standard output device. Note that this conversion depends on the underlying platform (the operating system).

3.9.8 Primitive Messages for Objects of Type ObjectVector

_At:

Returns the object at the specified position (an *integer*) of this *vector*. May fail with a *badIndexError*.

_At:Put:

Puts an object at the specified position (an *integer*) of this *vector*. May fail with *badIndexError*.

_Clone:Filler:

Clones a prefix of the receiver up to the position specified by *_Clone:* (an *integer*). When the specified position is greater than the size of the receiver, the rest of the new *vector* is filled with the object specified by *Filler:*.

_CopyRangeDstPos:Src:SrcPos:Length:

Copies a number of objects (specified by *Length:*, an *integer*) into the receiver *vector* at the position specified by *DstPos:* (an *integer*) from the *vector* at the position specified by *SrcPos:* (an *integer*). May fail with *badIndexError*.

_Size

Returns the size of the receiver as an object of type *integer*.

3.9.9 Primitive Messages for Ordinary Objects

_AddSlot:ConnectedTo:

Adds a new slot to the receiver that refers to the lobby of a remote *dSelf Virtual Machine*. The name of this slot is specified by the first argument, which is a *string* that represents a lexically correct dataslot name. The second argument is also of type *string* and specifies the location of the remote *dSelf Virtual Machine*. This URL has the following form: `//hostname:port/vm_name`, e.g. `//127.0.0.1:1099/myVM`. The values for the hostname and port are optional and set by default to the local host and port number 1099 (the default port of the RMI-Registry).

_AddSlotIfAbsent:ConnectedTo:

Has the same effect as *_AddSlot:ConnectedTo:*, except that this action is omitted, if a slot with the same name already exists.

_AddSlots:

Adds the slots of the argument object to the receiver. New slots with the same names replace old slots.

_AddSlotsIfAbsent:

Adds the slots of the argument object to the receiver. New slots with the same names don't replace old slots.

_Clone

Returns a clone of the receiver by copying it with shallow-copy semantics.

_Define:

Redefines the receiver of this message. All old slots are deleted an the slots of the argument object are added.

_Describe

Returns a description of the receiver by printing all annotations of its slots.

_Eq:

Returns *true* if both objects are identical, otherwise *false* is returned.

_GetSlotNames

Returns an *objectvector* with the receivers slot names as *string* objects.

_ObjectID

Returns the ID (an *integer*) of the receiver that is unique **within this** *dSelf Virtual Machine*. Note that two objects on different machines can have the same ID by chance !

_Perform:

Sends an unary message whose name is specified by the argument (a *string*) to the receiver. E.g., *x _Perform: 'foo'* means the same like *x foo*. Primitive messages (messages starting with an underscore) can't be send in this way.

_Perform:With:{With:}

Sends an keyword message whose name is specified by the arguments (*strings*) to the receiver. Each argument is separated by up to 20 *With:* keywords. E.g., *x _Perform: 'foo:Bar:' With: 1 With: nil* is identical to *x foo: 1 Bar: nil*. Primitive messages (messages starting with an underscore) can't be send in this way.

_PerformResend:

Resends (indirected resend) an unary message whose name is specified by the argument (a *string*) to the parents of the receiver. E.g., *x _PerformResend: 'foo'* is equivalent to *resend.foo* inside object *x*. Primitive messages (messages starting with an underscore) can't be resend in this way.

_PerformResend:With:{With:}

Resends (indirected resend) a keyword message whose name is specified by the arguments (*strings*) to the parents of the receiver. Each argument is separated by up to 20 *With:* keywords.

E.g., *x _PerformResend: 'foo:Bar:' With: 1 With: nil* has the same semantics as *resend.foo: 1 Bar: nil* inside of object *x*. Primitive messages (messages starting with an underscore) can't be resend in this way.

_Perform:DelegatingTo:

Resends (directed resend) an unary message whose name is specified by the argument (a *string*) to the parent of the receiver specified by the latter argument (a *string*). E.g., *x _Perform: 'foo' DelegatingTo: 'p'* has the same semantics as *p.foo* inside of object *x*. Primitive messages (messages starting with an underscore) can't be resend in this way.

_Perform:DelegatingTo:With:{With:}

Resends (directed resend) a keyword message whose name is specified by the arguments (*strings*) to the parent of the receiver specified by the second argument (a *string*). Each argument is separated by up to 20 *With:* keywords. E.g., *x _Perform: 'foo:Bar:' DelegatingTo: 'p' With: 1 With: nil* has the same semantics as *p.foo: 1 Bar: nil* inside of object *x*. Primitive messages (messages starting with an underscore) can't be resend in this way.

_Print

Prints a description for the receiver that consists of a list of all slot names.

_RemoveAllSlots

Removes all slots of the receiver.

_RemoveSlot:

Removes the slot that is specified by the argument (a *string*) of this message.

3.9.10 Primitive Messages for the Debugger

_DebugCUPOn

With this message, the debugger of the parser (CUP) will be enabled. It will print all actions (shift/reduce) that were done by the parser. For further information about it refer to the documentation of [6]. This option is very useful in combination with *_DebugScannerOn*.

_DebugCUPOff

Disables the debugger of the parser.

_DebugScannerOn

With this message, the debugger of the scanner will be enabled. It will print all scanned tokens on the standard output stream.

_DebugScannerOff

Disables the debugger of the scanner.

_DebugSearchPathOn

Enables the lookup debugger of the *lookup algorithm* and shows the search path for messages. For details about the inheritance mechanism in SELF/dSelf see [2] or [9].

_DebugSearchPathOff

Disables the debugger of the *lookup algorithm*.

_DebugFlatParseTreeOn

With this message the parser will generate a parse tree for each parsed input. The structure of this tree is represented by using parenthesis.

_DebugFlatParseTreeOff

Disables the parse tree debugger.

_DebugIndentedParseTreeOn

The previously mentioned message *_DebugFlatParseTreeOn* printed the parse tree in a compact but not very readable way. This message prints the same tree more attractive by using indention for displaying its structure.

_DebugIndentedParseTreeOff

Disables the parse tree debugger.

3.9.11 Other primitive Messages

_Credits

Returns a message about the copyright of dSelf.

_CurrentTimeString

Returns a *string* with informations about the current time and date.

_DirPath

Returns a *string* that contains the current directory path. The directory path represents the root directory of all scripts that are executed by calling *_RunScript*.

_DirPath:

Replaces the old directory path by the argument of type *string*.

_GarbageCollect

Advises the Virtual Machine to start the garbage collector now.

_Memory

Returns the size of the currently available amount of memory. The result object is of type *long*.

_OperatingSystem

Returns a *string* that describes the underlying operating system and processor type.

_Quit

Exits dSelf.

_TimeReal

Returns the number of milliseconds since 1. January 1970 GMT. The result object is of type *long*.

Chapter 4

The Grammar

expression -> constant | unary-message | binary-message | keyword-message

subexpression -> '(' ['|' '|' ['\^{ }']] expression ')' | expression

constant -> **self** | number | string | object | **nil** | **true** | **false**

unary-message -> receiver unary-send | indirected-unary-send | directed-unary-send

unary-send -> identifier

binary-message -> receiver binary-send | indirected-binary-send expression | directed-binary-send expression

binary-send -> operator expression

keyword-message -> receiver keyword-send | indirected-keyword-send expression { cap-keyword expression } | directed-keyword-send expression { cap-keyword expression }

keyword-send -> small-keyword expression { cap-keyword expression }

receiver -> [subexpression]

object -> data-object | block

data-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] ')'

method-object -> '(' ['|' ['{' '}' '=' string] slot-list '|'] code ')'

block -> '[' ['|' slot-list '|'] [code] ']'

slot-list -> [unannotated-slot-list] {annotated-slot-list [unannotated-slot-list]}

annotated-slot-list -> '{' string slot-list '}'

unannotated-slot-list -> { slot '.'} slot ['.']

code ->{ expression ‘.’} [‘^’] expression [‘.’]

slot -> arg-slot | data-slot | binary-slot | keyword-slot | unary-slot

arg-slot -> argument-name

data-slot -> slot-name | slot-name ‘<-’ expression | slot-name ‘=’ expression

unary-slot -> identifier‘=’ method-object

binary-slot -> operator ‘=’ method-object | operator [identifier] ‘=’ method-object

keyword-slot -> small-keyword {cap-keyword} ‘=’ method-object | small-keyword identifier {cap-keyword identifier} ‘=’ method-object

slot-name -> identifier | parent-name

parent-name -> identifier ‘*’

Chapter 5

Further Information

If you want more information about dSelf, you should visit the homepage of dSelf at: `http://www.cs.tu-berlin.de/~tolk/dself/`. There are some documents located about the basic principles of dSelf and my diploma thesis [9] that describes dSelf in more detail, concerning its design and implementation. I'm sorry, but it's available only in German. But with this tutorial the most important topics of the thesis are available in English now. The thesis might still be useful for non German readers, who are looking for information about the implementation of dSelf, as it contains many class-diagrams.

There are some diploma-theses that will extend dSelf concerning topics like concurrency and progressive multiple-inheritance techniques, so visiting the dSelf homepage occasionally will be worth it.

Bibliography

[1] dSelf - A Distributed SELF, Robert Tolksdorf and Kai Knubben, KIT - REPORT 144, ISSN 0931-0436, Fakultät IV, Projektgruppe KIT, Technische Universität Berlin, April 2001. Available at http://www.cs.tu-berlin.de/~tolk/dself/

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

[3] Prototyped-Based Application Construction Using Self 4.0, Mario Wolczko and Randall B. Smith. Published by Sun Microsystems, Inc. and Stanford University, 1995

[4] SELF: The Power of Simplicity, David Ungar and Randall B. Smith. Lisp and Symbolic Computation: An International Journal, Kluwer Academic Publishers, 4. 3. 1991.

[5] Programming as an Experience: The Inspiration for Self, Randall B. Smith and David Ungar, Sun Microsystems Laboratories

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

[7] JFlex Version 1.3, The Fast Lexical Analyzer Generator for Java by Gerwin Klein, October 2000. http://www.jflex.de

[8] JavaDeps version 1.0.4. Automatic Dependency Tracking for Java by Steven Robbins, 1998. http://www.cs.mcgill.ca/~stever/software/JavaDeps/

[9] Verteilte Implementierung der objektorientierten Programmiersprache SELF (in German), Kai Knubben, Technische Universität Berlin, 2000. Available at http://www.cs.tu-berlin.de/~tolk/dself/

[10] Parents are Shared Parts of Objects: Inheritance and Encapsulation in SELF, Craig Chambers and David Ungar and Bay-Wei Chang and Urs Hölzle, Lisp and Symbolic Computation volume 4, number 3, pages 207–222, 1991

[11] Java Runtime Environment v 1.2, Sun Microsystems. Available at http://java.sun.com/products/jdk/1.2/jre/index.html

[12] Java 2 SDK, Standard Edition, v 1.3, Sun Microsystems. Available at http://java.sun.com/j2se/index.html

[13] GNU Make, Copyright (C) 1997, 1998, 1999, 2000 Free Software Foundation, Inc., 59 Temple Place - Suite 330, Boston, MA 02111, USA. Available at http://www.gnu.org/software/make/make.html