SCOLA - A Users Guide

Andreas Plüschke

November 26, 2002

1 Introduction

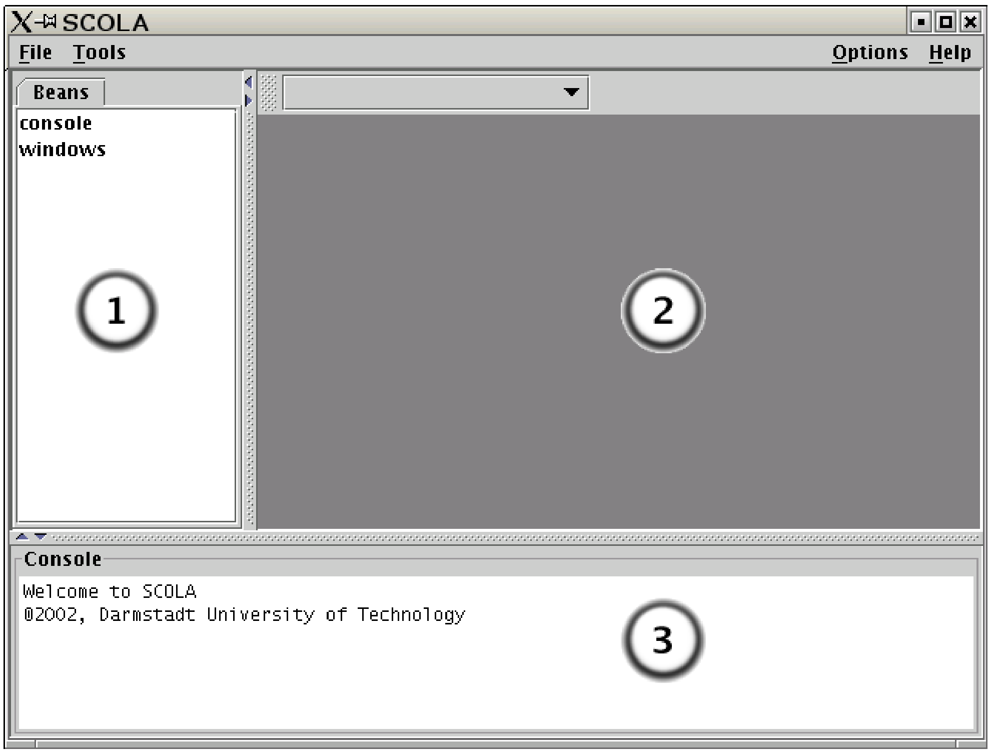
As part of the Thesis [Plü02] not only the framework for the *Data Type Components* was done, but additionally an interactive tool has been implemented using this framework for the core data types: SCOLA – Shell for Contextual Logic Applications. The idea behind this tool is to provide a platform for implementing and testing new algorithms and enhancements for the core *Data Type Components*, which can be used by mathematicians and other scientists. The core features of the tool are:

- Interactive development and research due to the use of scripting languages
- Support for several scripting languages (e. g. BeanShell, JPython, and NetRexx)
- Concurrent use of the scripting languages
- Object exchange between scripts of one language and between scripts written in different scripting languages
- Default implementations of the data types ordered set, lattice, and Formal Context as *Data Type Components*

These features are possible by using IBM's Bean Scripting Framework [IBM] that enables Java applications to embed scripting engines. By implementing adapter classes this framework allows to integrate further scripting languages. So it is possible to extend our program by new languages without the need to change any line of code of the existing classes.

Combining the Bean Scripting Framework[1] with the default implementation of the Data Type – Processors pattern enables the interactive shell to become a powerful tool. Within this chapter we will introduce the application and explain the main steps using it.

[1]Although the name of the framework underlines the use of Beans it also works with any Java object.



```
Welcome to SCOLA
©2002, Darmstadt University of Technology
```

Figure 1: The SCOLA main window

2 Tutorial

Within the next subsections we will go through the most common used options while working with the tool. Since one goal implementing this tool is to provide an application which can be extended, the information given here is related to the version in October 2002. After finishing the diploma thesis this software project will be made available as a new open-source project hosted on SourceForge [SCO].

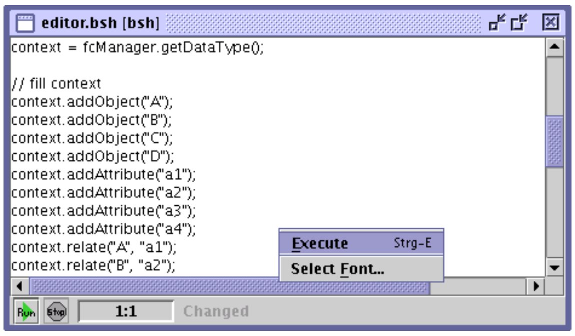
2.1 The User Interface

After starting SCOLA the main graphical user interface is shown centred on the screen. Figure 1 shows a screen-shot. The main window is divided into three main parts. The tabs region is marked with circle no. 1, the desktop pane by no. 2, and the console sub-window labelled by circle no. 3.

The tabs region is for extension purposes. At the moment it shows a single tab pane, which lists registered Beans. Registered Beans are means provided by the Bean Scripting Framework to share Java objects between several scripting engines supported by the framework[2]. A Bean registered to the BSF manager can be accessed from any scripting language supported by the framework using Strings as keys[3]. However, in future this region shall be used to embed further information

[2]Further details can be found in [BSF99]

[3]We will give some more details on this mechanism later.



```
context = fcManager.getDataType();

// fill context
context.addObject("A");
context.addObject("B");
context.addObject("C");
context.addObject("D");
context.addAttribute("a1");
context.addAttribute("a2");
context.addAttribute("a3");
context.addAttribute("a4");
context.relate("A", "a1");
context.relate("B", "a2");
```

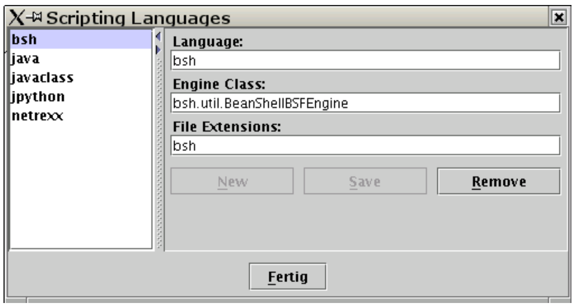
Figure 2: An internal frame for editing a script

panes.

The desktop pane provides a multiple-document interface. It is capable to handle numerous text windows used to edit the scripts executed within this environment. Such an internal frame is shown in Figure 2. Besides the purpose as editor the scripts are also executed from within these windows either using the shortcut `CTRL+E`, the pop-up menu – as shown in the figure – or the *Run* button, which is part of the info line at the bottom of the window. The execution can be halted using the *Stop* button to the right of the *Run* button.

The third region is the console pane at the bottom of the main window. The output of every script executed is redirected to this console. Additionally if a script expects input from `stdin`[4], this data can be entered here. Both the tabs sub-window and the console pane can be un-docked from the main window and reside in their own window. To activate this invoke the context specific pop-up menu pressing the right mouse button either on the console pane or the head-line of the tabs region. Using the pop-up menu of the console pane the user can additionally set the font used to print the output; likewise the pop-up menu of the internal frames provides an item to show a fonts dialog.

2.1.1 Adding further Scripting Engines



bsh
java
javaclass
jpython
netrexx

Language: bsh
Engine Class: bsh.util.BeanShellBSFEngine
File Extensions: bsh

Figure 3: The dialog to edit the registered scripting engines

Another customising dialog can be invoked using the Options menu. The dialog

[4]`stdin` = standard in; normally this is the user input on a command line

Scripting Languages is used to edit the registered scripting engines. Figure 3 shows a screen-shot of the dialog box. Users can use this dialog to add further scripting engines by specifying the language name, the class name of the Java engine class, and the used file name extensions for this scripting language. Thus SCOLA can be adopted easily to support further scripting languages.

2.2 Developing Scripts

The main purpose of SCOLA is – on the one hand – to provide a test platform for processor classes and completely new *Data Type Components* and on the other hand to be used as an experimentation platform for new algorithms and results of research in the domain of *Formal Concept Analysis* and *Conceptual Knowledge Processing*. Therefore SCOLA enables the user to write and execute scripts.

2.2.1 A First Example

For a first example we will use the BeanShell scripting language, which in fact is the Java programming language extended by common scripting language conventions and syntax. Most noticeably the BeanShell interpreter allows dynamic typing of variables, but it can execute standard Java code, too. Therefore code written for the BeanShell interpreter can be adopted easily to build a *real* Java application later.

```
print("Hello, world!");
```

Listing .1: Hello, world! written in the BeanShell Java dialect

Listing .1 shows – in good tradition – a program printing Hello, world! onto the console. To do this start SCOLA and select the menu item New in the File menu. Selecting this item will bring up a dialog used to select one of the registered scripting engines. We select bsh for BeanShell; then a new internal frame like the one shown in Figure 2 will be added to the desktop pane[5]. Using the new window we can enter the code and immediately start the program using the *Run* button. The output should occur in the console pane.

2.2.2 Using Data Type Components

How can one now use a *Data Type Component* like the default implementations for a Formal Context or an Ordered Set? As in plain Java the developer must import the belonging package. Now by invoking the constructor of a manager class one gets the reference of a new manager instance, but implicitly the user gets a reference on a new instance of the belonging data type class as well as references on processor objects, one for each applicable processor class. Using the manager's reference the

[5]Naturally it will contain no lines of code

```
01 : import scola.math.impl.sets.Memory.*;
02 :
03 : \\  create new changeable Ordered Set component
04 : setManager = new MemoryOrderedSetManager(true);
05 :
06 : \\  retrieve set reference
07 : set = setManager.getDataType();
08 :
09 : \\  modify the set
10 : set.link("Assembler", "C");
11 : set.link("Assembler", "Cobol");
12 : set.link("Assembler","Algol 60");
13 : set.link("Lambda calculus", "Algol 60");
14 : set.link("Assembler", "Fortran");
15 : set.link("Fortran", "Fortran 77");
16 : set.link("Logic calculus", "Prolog");
17 :
18 : \\  show line diagram of ordered set
19 : frame(setManager.getProcessor("Visualisation").createCanvas());
```

Listing .2: Generates an ordered set of some programming languages and shows a line diagram of it

user can retrieve the data type instance and the several processors. This enables the users to fulfil their tasks.

Listing .2 shows an example. After importing the package containing the manager class the script creates a new component instance in line 4. The next statement retrieves the data type reference, which is used to manipulate the ordered set in the lines 10 until 16. Finally a frame[6] is created containing a canvas which shows the ordered set as a line diagram[7] like Figure 4.

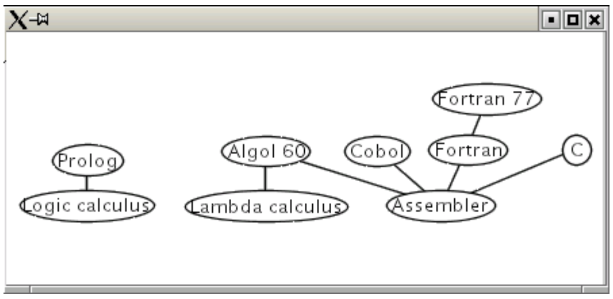
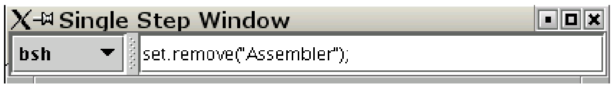


Figure 4: The dialog to edit the registered scripting engines

[6]frame is a special BeanShell command

[7]Mostly the nodes of the diagram are not placed very well by the automatism. This is because at the moment only a poor algorithm for node placing is implemented



```
set.remove("Assembler");
```

Figure 5: The *Single Step Window*

2.2.3 Getting more interactively

We can move the nodes of the line diagram using the mouse. However we cannot add or remove nodes or links. One can open a new BeanShell scripting window using the NEW menu item and writing a line like set.link("C", "C++");. But, after executing this code, the line diagram will not update. However, when we select line 19 (cf. Listing .2) in the first scripting window and execute it[8] via the *Run* button another frame will be opened showing the changed ordered set. So we see that the content of the data type has changed, but the diagram view hasn't. To fix this we now replace line 19 of the script shown in Listing .2 with the following statement:

```
frame(setManager.getProcessor("Visualisation").createListeningCanvas());
```

Executing this statement will bring up a frame showing the line diagram like before, but now the canvas is linked with the data type and receives events if the content of the data type changes. Thus the canvas can be updated automatically. To test this we now use the so-called *Single Step Window* (Figure 5 shows a screenshot of it), which can be opened using the TOOLS menu. This window allows to enter single lines of code, which will be executed immediately after the user presses the `Enter` key. Additionally the user can select the used scripting language by a combo box.

2.2.4 Note

Using the *Single Step Window* and accessing variables in second runs (either from the same window or from another window using the same scripting language) is not possible for all scripting languages supported by the Bean Scripting Framework. Rather it depends on the scripting engines which delegate the execution. For example BeanShell uses an interpreter, which preserves the state between a sequence of invocations, whereas other engines – like the NetRexx engine – compile the script and execute it afterwards. Thus the state cannot be preserved. For further information users should read the documentation coming with the different engines.

2.2.5 Declaring and Registering Beans

IBM's Bean Scripting Framework provides a mechanism which enables scripts to exchange objects between several scripting engines and executions of scripts. This mechanism allows to avoid the problems mentioned above and can be used for any Java object.

[8]By selecting text we can execute only parts of a script.

Therefore the framework uses an object registry, which maps names (represented by Strings) to objects. Additionally the framework distinguishes between two kinds of announcing a BeanAs already mentioned, any Java object can be used: *declaring* and *registering*. Thereby an object, which has been declared, will automatically be registered, but not the other way round. This affects scripts. Declared objects can be accessed directly by its defined name in the registry, whereas registered objects must be retrieved first by using a language dependant mechanism.

What are the reasons for a second mechanism besides *declaring* objects? The answer is due to technical reasons. The authors of the framework allow scripting engines not to support the declaring mechanism. For declaring the documentation says: engines are expected to make declared beans "pre-available" in the scripts as far as possible. However, it is not a must for engines to implement this feature. Thus registering an object provides more flexibility for authors of scripting engines because the exact mechanism, how a registered object can be retrieved, is not specified.

A feature of SCOLA is the *Beans* tab as part of the tabs region. It provides a list of all declared and registered objects. To make the difference visible registered objects are shown with additional braces on the list. To test the exchange of objects we can do the following steps after we have executed Script .2: first we register the ordered set instance to the Bean Scripting Framework. For this purpose we enter `bsf.registerBean("set", set)` in the *Single Step Window* and execute this as BeanShell script[9]. In the Beans tab a new entry (set) should occur. Now we change the scripting language to jpython and enter `print bsf.lookupBean("set")`. Executing this code fragment results in the following output on the console pane:

```
scola.math.impl.sets.Memory.MemoryRWOrderedSet@21e5f0 :
C: [(Assembler), ()]
Assembler: [(), (C, Cobol, Algol 60, Fortran)]
Cobol: [(Assembler), ()]
Algol 60: [(Assembler, Lambda calculus), ()]
Lambda calculus: [(), (Algol 60)]
Fortran: [(Assembler), (Fortran 77)]
Fortran 77: [(Fortran), ()]
Prolog: [(Logic calculus), ()]
Logic calculus: [(), (Prolog)]
```

Table 1: Textual representation of an OrderedSet instance

This is a textual representation of the ordered set we created using the Listing .2. However, executing `print set` in the *Single Step Window* as JPython code will result in an exception. To change this we run

```
bsf.declareBean("set", set, scola.math.sets.RWOrderedSet.class)
```

as BeanShell script and now in the Beans tab the entry (set) will be replaced by

[9]Select bsh in the combo box

set[10]. Within any scripting language supporting the declaring mechanism it should now be possible to access the ordered set instance using the variable name set.

2.2.6 Predefined Beans and Helper Classes

After starting SCOLA the Beans tab shows already two entries, console and windows. Both provide methods to change the user interface programmatically. The variable windows allows to access all opened internal frames, whereas console enables the user to access the console pane (e. g. to clear it).

Additionally a helper class is made available by the package scola.tools.shell.helper. The class Window allows to create JFrame and JDialog instances more flexible than the BeanShell command frame we used in Listing .2. A further feature provided by this class is a small browser, which can be used to show JavaDoc HTML pages. Additional information for the program come with some example scripts demonstrating several features of SCOLA and how to use the program.

References

[BSF99] IBM TJ Watson Research Center, Hawthorne, NY 10532. *Bean Scripting Framework User's Guide*, December 1999. Part of the documentation of [IBM].

[IBM] Bean Scripting Framework.
http://oss.software.ibm.com/developerworks/projects/bsf.

[Plü02] Andreas Plüschke. Design of a component based framework for conceptual knowledge processing. Diploma thesis, Darmstadt University of Technology, 2002.

[SCO] SCOLA – Shell for Contextual Logic Applications.
http://sourceforge.net/projects/scola.

[10]Now without braces to signalise that this name is used for a declared object