JOHANNES KEPLER
UNIVERSITÄT LINZ
Netzwerk für Forschung, Lehre und Praxis

Augmenting GCSpy with Scripting Functionality

Bakkalaureatsarbeit
(Projektpraktikum)

zur Erlangung des akademischen Grades

Bakkalaureus der Technischen Wissenschaften

in der Studienrichtung

Informatik

Angefertigt am *Institut für Systemsoftware*

Betreuung:

o.Univ.-Prof. Dr. Dr. h.c. Hanspeter Mössenböck
Dipl.-Ing. Thomas Schatzl

Eingereicht von:

Benjamin Dallinger

Linz, 25. 01. 2011

Abstract

GCSpy is a powerful heap visualization tool which alleviates the observation and analysis of memory managed systems during the development process. Because of its flexible and loosely coupled architecture the application can be easily integrated into various environments without much effort. Furthermore the implementation of the visualizer in Java ensures high portability and robustness. For this reason GCSpy is a useful instrument especially for developers working on efficient garbage collection algorithms in virtual machines.

Goal of this project is the integration of a scripting engine and the development of an efficient interface that allows script-based access to all important functionality in GCSpy. This enables developers to adapt the displayed information to their individual preferences by loading and running scripts in the visualizer. In addition, this enhancement provides script-based control of the visualization process which allows users to utilize GCSpy like a classical debugger. For example, this allows the use of breakpoints to e.g. stop the visualization when specific conditions are met.

Kurzfassung

GCSpy ist ein mächtiges Werkzeug zur Heap-Visualisierung, welches die Beobachtung und Analyse von speicherverwalteten Systemen während des Entwicklungsprozesses erleichtert. Dank seiner flexiblen und lose gekoppelten Architektur kann die Applikation ohne großen Aufwand in unterschiedliche Umgebungen integriert werden. Die Implementierung der Visualisierung in Java garantiert außerdem hohe Portabilität und Robustheit. Aus diesen Gründen ist GCSpy ein nützliches Instrument speziell für Entwickler, welche an effizienten Garbage Collection Algorithmen für virtuelle Maschinen arbeiten.

Ziel dieses Projekts ist die Integration einer Scripting Engine und die Entwicklung einer effizienten Schnittstelle, welche einen skriptbasierten Zugriff auf sämtliche wichtigen Funktionen in GCSpy gestattet. Dies ermöglicht Entwicklern die Anpassung der angezeigten Informationen nach ihren individuellen Bedürfnissen, indem Skripte in der Visualisierung geladen und ausgeführt werden. Außerdem erlaubt diese Erweiterung die skriptgesteuerte Kontrolle des Visualisierungsprozesses, was Benutzern erlaubt, GCSpy wie einen klassischen Debugger zu verwenden und z.B. die Visualisierung unter Verwendung von Haltepunkten zu szoppen, sobald bestimmte Bedingungen eintreten.

Acknowledgment

At this point I want to offer my regards and blessings to all of those who supported me in any respect during the completion of this bachelor project. Especially I am thankful to my supervisor, DI Thomas Schatzl, whose encouragement, guidance and support from the initial to the final level enabled me to develop an understanding of the subject.

Lastly, I want to dedicate this work to my family who supports me as good as possible and always conveys great interest in my work. Without their patience, understanding, and most of all love, the completion of this work would not have been possible.

Danksagung

An dieser Stelle möchte ich mich bei all jenen bedanken, die mir während der Anfertigung dieser Bakkalaureatsarbeit zur Seite gestanden haben. Ein besonders Dankeschön geht an meinen Betreuer DI Thomas Schatzl, der mir durch seine Anregungen half, ein Verständnis für die Problemstellung zu entwickeln und mich während des gesamten Projektverlaufes tatkräftig unterstützte.

Zuletzt möchte ich dieses Arbeit meiner Familie widmen, welche mich nicht nur so gut es geht unterstützt, sondern auch ständig großes Interesse an meiner Arbeit zeigt. Ohne ihre Geduld, Verständnis und vor allem Liebe wäre die Fertigstellung dieses Projektes nicht möglich gewesen.

Contents

Chapter 1

Approach and motivation

1.1 What is GCSpy?

GCSpy is a powerful heap visualization tool which alleviates the observation and analysis of memory managed systems during the development process. It provides functionality for collection, transmission, storage and replay of memory management information.

Because of its flexible and loosely coupled architecture the application can be easily integrated into various environments without much effort. Furthermore the implementation of the visualizer in Java ensures high portability and robustness. For this reason GCSpy is an important instrument especially for developers working on garbage collection algorithms in virtual machines.

1.2 Idea and task description

During development of memory managed systems and especially garbage collection algorithms it is a big advantage to have a graphical representation of the heap state instead of pure numerical data. Due the facts that every developer has individual preferences and each development task has distinct requirements, it is important that a visualization tool like GCSpy offers large range of flexibility.

The main task of this thesis is to improve GCSpy's capabilities by making the current data and views available to small scripts that augment and control the visualization during tracing. Interesting new example applications may provide new views derived from existing data (e.g. provide summaries etc), enable the user to quickly find interesting data (e.g. breakpoints conditional on particular values, or value highlighting in the views based on thresholds) and other interesting ways of helping to understand (i.e. debug) the traced data.

So the goal is the integration of a scripting engine into the visualizer and the development of an efficient interface that allows script-based access to all important functionality in GCSpy.

1.3 Further structure of this thesis

The next chapter takes a closer look on GCspy with a focus on the visualizer, summarizes the previous work on the application and compares it with some related tools. Chapter 3 defines the requirements for the implementation and documents some design decisions. Moreover it contains some example scripts for common use-cases. Chapter 4 analyzes the visualization process and describes the implementation details. The final chapter 5 contains a reflection about the result of this project and some ideas for future work on GCSpy with a focus on scripting tasks. The appendix specifies the fundamental use-cases for script administration and utilization, as well as the detailed scripting interface.

Chapter 2

Introduction to GCSpy

This chapter describes GCspy in more detail and takes a closer look at the current implementation of the visualizer and its actual functionality, followed by a short overview of the previous work on the application and a comparison with some related tools.

2.1 Overview

The framework gathers the heap and memory management information from the observed system and transforms it into a visual representation for better perception and understanding. The developer is able to analyze dynamic memory behavior and profile system performance based on graphical information instead of numerical data. This can be done efficiently under realistic conditions when running real programs resulting in large heaps, because the cost of storage, transmission and visualization is independent of the heap size. [Printezis02]

In order to maximize flexibility and keep the influence on the observed system as small as possible, the application is divided into two parts and implemented as a server-client architecture communicating over TCP/IP as shown in figure 2.1. This technique allows a minimal impact remote observation and also eases the adaption for other environments because changes only have to be applied to the server module. Furthermore this method enables the visualizer to connect to and disconnect from a running system at any time. [Printezis02]

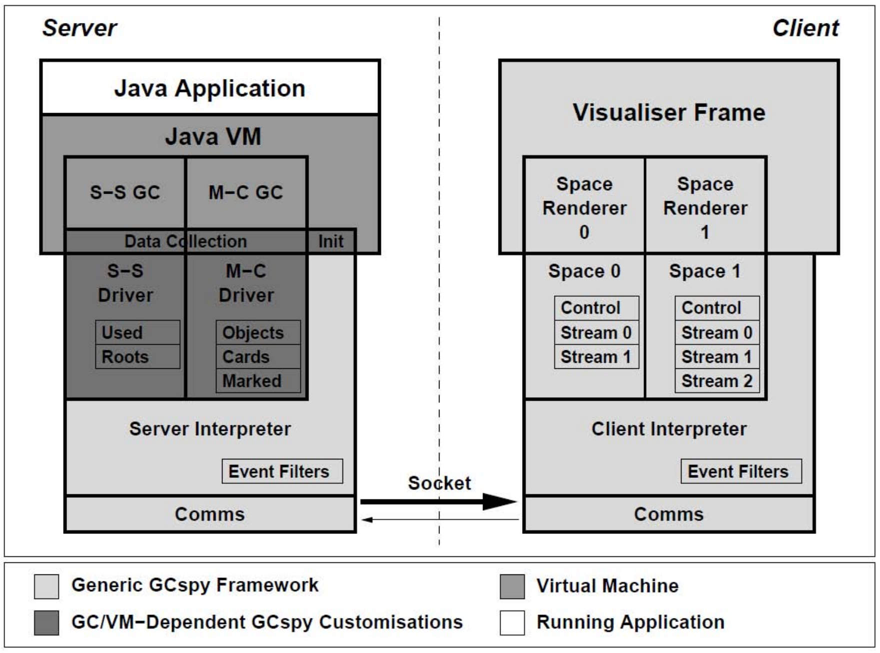
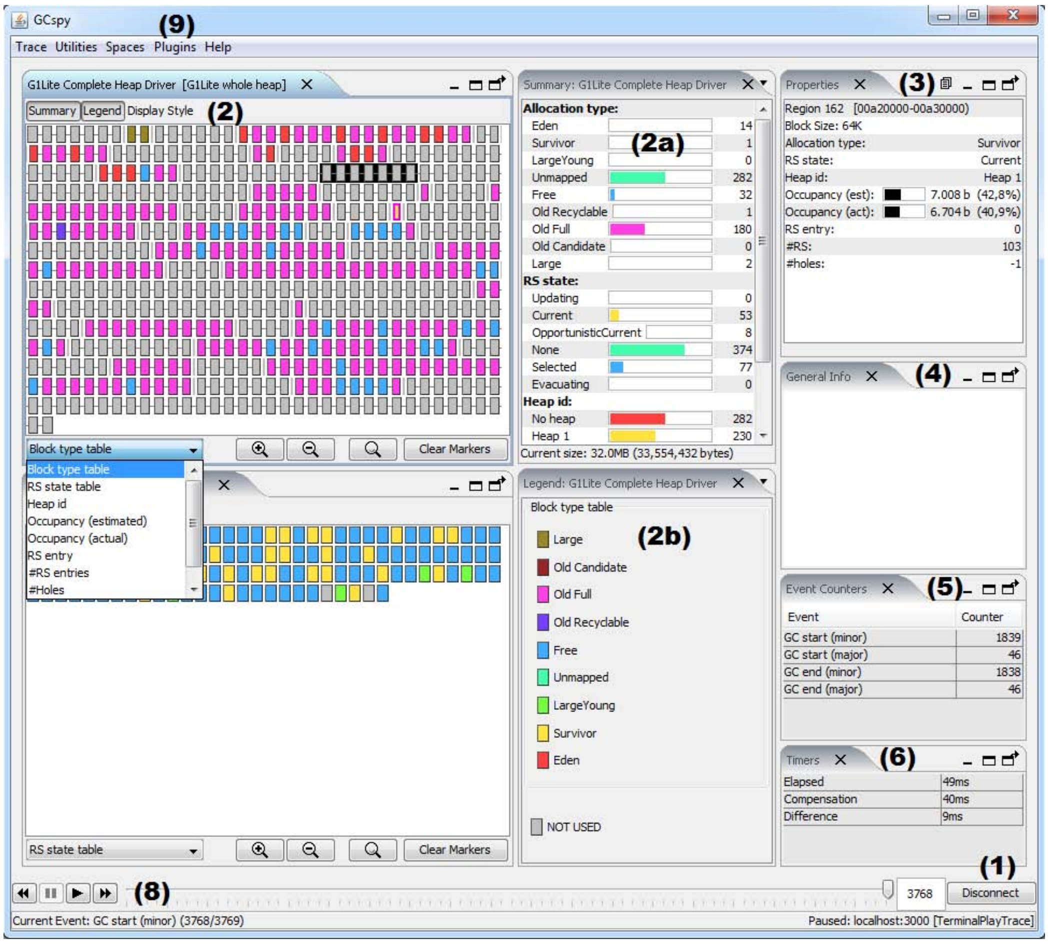


Figure 2.1: GCSpy's architecture [Printezis02]

The memory management information on the host runtime system is gathered by individual drivers for every single aspect that should be considered in the observation. Outside the gathering period the runtime costs are negligible. The server collects the heap information from the drivers and serializes it for transmission over TCP/IP. Currently the server module is available in Java, C and C++. On the client side, which is exclusively responsible for the data visualization, the received data is de-serialized and transformed into in a object oriented data structure which gets processed by the visualizer. The client application is completely implemented in *Java* using *Swing* for the GUI, which ensures high portability, reusability and robustness. In addition to the stream data GCSpy supports the transmission of special commands which allow the control of the visualization process, i.e. stop and resume the playback. [Printezis02]

2.2 GUI and visualization

This section provides an overview of GCSpy's GUI and the visualization part of the application. It also gives a short description of the basic components and their functionality to get a fundamental knowledge for further analysis and discussions. Figure 2.2 below shows GCSpy running with a loaded trace file and all relevant dockable windows open. The connection dialog, the display style dialog and the event filter settings are implemented as independent frames as shown in figures 2.3, 2.4 and 2.5.



Allocation type:	
Eden	14
Survivor	1
LargeYoung	0
Unmapped	282
Free	32
Old Recyclable	1
Old Full	180
Old Candidate	0
Large	2
RS state:	
Updating	0
Current	53
OpportunisticCurrent	8
None	374
Selected	77
Evacuating	0
Heap id:	
No heap	282
Heap 1	230

Current size: 32.0MB (33,554,432 bytes)

Block type table
- 🟫 Large
- 🟥 Old Candidate
- 🟪 Old Full
- 🟪 Old Recyclable
- 🟦 Free
- 🟩 Unmapped
- 🟩 LargeYoung
- 🟨 Survivor
- 🟥 Eden
- ⬜ NOT USED

Region 162 [00a20000-00a30000)
Block Size: 64K

Allocation type:	Survivor
RS state:	Current
Heap id:	Heap 1
Occupancy (est):	7.008 b (42,8%)
Occupancy (act):	6.704 b (40,9%)
RS entry:	0
#RS:	103
#holes:	-1

Event	Counter
GC start (minor)	1839
GC start (major)	46
GC end (minor)	1838
GC end (major)	46

Elapsed	49ms
Compensation	40ms
Difference	9ms

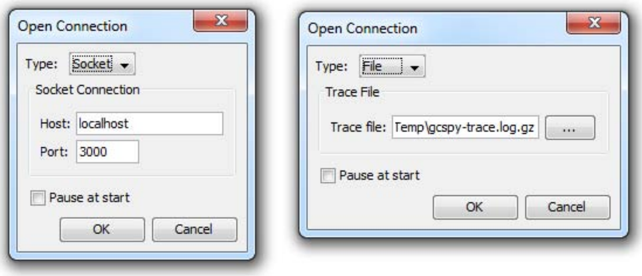
Current Event: GC start (minor) (3768/3769)

Paused: localhost:3000 [TerminalPlayTrace]

Figure 2.2: GCSpy's visualizer

1. **Connection dialog**

   At application start or after pressing the *Connect* button the user is requested to choose either a server socket for connection or a stored trace file as shown in figure 2.3. If the user opens a trace file, the application invokes a pseudo server that processes the stored trace data from the file and provides it to the visualizer. In addition the user can decide if the visualization should start immediately after connection or if the process should pause until pressing the *Play* button.



Type: Socket
Socket Connection
Host: localhost
Port: 3000
☐ Pause at start

Type: File
Trace File
Trace file: Temp\gcspy-trace.log.gz
☐ Pause at start

Figure 2.3: Connection dialog

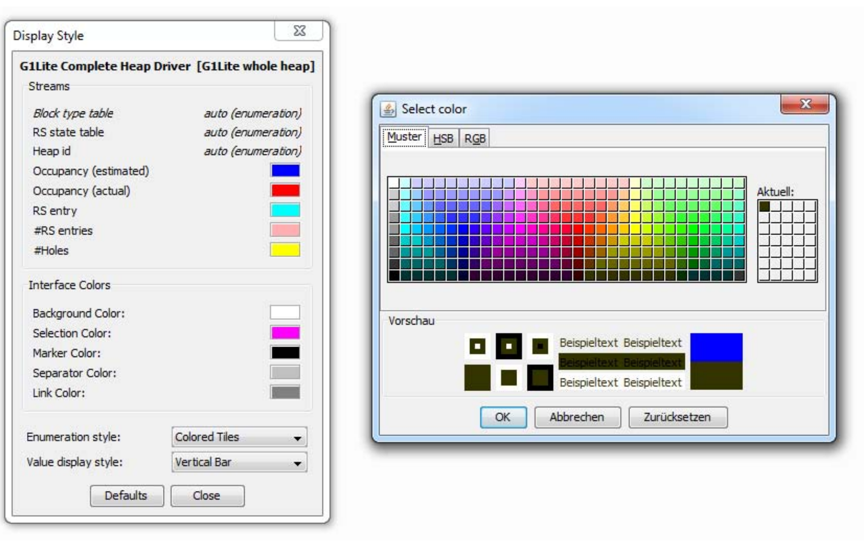
2. **Space Manager**

   A *Space Manager* is a graphical representation of a space in GCSpy's user interface which contains a constant number of tiles showing the data values for the currently selected stream. The number of spaces, streams and tiles is given from the actual trace data. Depending on the type of the selected stream the tiles appear as colored rectangle for enumerations or vertical chart bars for value-based streams. The actual stream can be selected from pop-up list in the lower left corner. It is also possible to display multiple streams at the same time by selecting the option <*Show multiple streams*> from the stream menu.

   In order to gather information about the meaning of particular tile colors, every *Space Manager* provides a *Legend* button, which opens a dockable window that shows the tile colors and the corresponding enumeration name (2b). Further every *Space Manager* offers a *Summary* window (2a). If the stream currently selected in the *Space Manager* is a enumeration the summary shows the sum of the particular tile types. The chart bars are named and have the same color as the corresponding enumeration type which spares a look on the legend.

   Every *Space Manager* enables the user to individually set markers on single or multiple tiles by right-clicking them. The marked tiles are highlighted with a border whose color can be set in the *Display Style* dialog of the particular *Space Manager*. A right click on a marked tile removes the marker. The *Clear Markers* button removes all markers in a *Space Manager*.

   Also every *Space Managers* provides the possibility to zoom in and out or enlarge a small section of the tile space. The *Display Style* dialog provides several settings tho customize the appearance and the colors.


Display Style

G1Lite Complete Heap Driver [G1Lite whole heap]

Streams

Block type table	*auto (enumeration)*
RS state table	*auto (enumeration)*
Heap id	*auto (enumeration)*
Occupancy (estimated)	🟦
Occupancy (actual)	🟥
RS entry	🟦
#RS entries	🟥
#Holes	🟨

Interface Colors

Background Color:	⬜
Selection Color:	🟪
Marker Color:	⬛
Separator Color:	⬜
Link Color:	⬛

Enumeration style: Colored Tiles

Value display style: Vertical Bar

Figure 2.4: Display style dialog

3. **Properties**

   If a tile is selected in the space manager by left click, the *Properties* window shows some information like name, index and block size, as well as the actual values for each stream in the space. Values in enumeration type streams are displayed with their name while percentage type streams show the numerical value in combination with a horizontal chart bar.

4. **General Info**

   This window contains a simple text area that shows some general information of the given trace data noted by the implementer. In this example this data field is empty.

5. **Event Counters**

   The *Event Counters* window shows the actual number of the particular event types. In this example there are two types of GC activities called major and minor events. Further the application differentiates between the start and the end of every GC activity. This results in the four event types GC start (minor), GC start (major), GC end (minor) and GC end (major) which may differ depending on the actual use-case.

6. **Timers**

   The *Timers* utility displays the elapsed time between the events and the amount of time used for compensation. This two values are provided from the *Client Interpreter* at every event, the difference between them is calculated at display time.

7. **Event Filters**

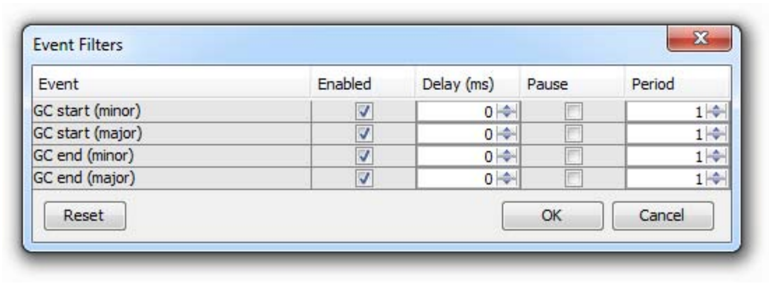
   In order to reject event information which is uninteresting for the user and control the visualization process based on event properties, GCSpy provides an event filter mechanism implemented in the server module. All filtered information does not get collected and transmitted by the server, so that the filters achieve a smaller impact on the performance of the target system. The *Event Filters* dialog shown in figure 2.5 allows the manipulation of the filter settings.

   - Enabled: Only enabled event types are collected and transmitted by the server.

   - Delay: Server delays for a given period after transmissions of specific event types, allowing the user to slow down the visualization process.

   - Pause: Server pauses after transmission of specific types of events, which results in a primitive breakpoint functionality.

- Period: Transmission of specific event types happens only every n times to reduce the transmission rate.



Event	Enabled	Delay (ms)	Pause	Period
GC start (minor)	✓	0	☐	1
GC start (major)	✓	0	☐	1
GC end (minor)	✓	0	☐	1
GC end (major)	✓	0	☐	1

Figure 2.5: Event filters dialog

8. **Navigation**

   The visualization process can be controlled in the bottom navigation area. The panel provides functions to pause and restart the visualization, walk through the event history in single steps or navigate to specific event using the event slider. Furthermore the panel displays information about the current connection state, the actual trace data and some event informations.

9. **Plug-ins**

   The actual implementation of GCSpy offers a plug-in management framework in order to include further functionalities. Unfortunately this plug-ins are hard coded within the application code which causes a lack of flexibility and maintainability. During the build process every plug-in gets compiled which results in one Java JAR file for every plug-in. The plug-in manager searches a defined directory for JAR files and loads every found plug-in at application start. Currently GCSpy includes plug-ins for histogram visualization and textual representation.

2.3 Previous work

Initially GCSpy was incorporated into Sun's *RJVM*, Sun's *HotSpot* and IBM's *JikesRVM* as described in [Printezis02]. In [Printezis01] Printezis and Jones adapted the framework for observation of the *Train Garbage Collector*. In [Singh07] Singh and Ranu created a server implementation for the *JikesRVM*.

A previous bachelor project focused on bringing the code and the visualization up to current standards, visualization and replay of the available data work very well now. For example, similar to a debugger, the user can interactively step back and forth between heap states, can set breakpoints in a limited way, can watch heap information and more. [Hofer10]

The latest enhancements on the framework primarily covered actions on cleaning up the code, make it more efficient and add functionality. The coupling between the server side data collection and the client visualization application was loosened by establishing an indexing component, storing the incoming trace data into a file so that it is possible to pause the visualization process and explore the event history without influencing the data transmission between server and client. In addition the illustration of tiles showing percental values was improved by changing the visualization method from variable color intensity to vertical bars with variable filling degree. [Hofer10]

2.4 Comparison with related applications

Main goal of the project is to get more comfortable debugging functionality into GCSpy. In order to get information about the abilities of actual debuggers and tracing tool we will compare GCSpy with some related applications and summarize the functionality which is interesting for GCSpy. Furthermore we think about if and how particular functions can be applied to a tracing tool like GCSpy using the possibilities of this project.

- Eclipse Debugger

  The *Eclipse* IDE includes a debugging function which allows the user to set an arbitrary number of breakpoints inside the code. *Eclipse* stops the program execution at this positions and allows a stepwise continuation. The actual code line is highlighted in the editor before execution. It is possible to step over methods and functions or resume the execution until the next breakpoint is reached. At the same time the debugger grants value access for all currently active variables. [Eclipse]

  In the figurative sense this breakpoint functionality can be adopted into GCSpy by giving playback control access to the scripting engine. If a certain condition is reached in the trace data the visualization stops and the user is enabled to walk forward or backward event by event. The corresponding tile gets highlighted using GCSpy's marker function which is already implemented. Value information can be obtained from the Properties window which can be automatically actualized providing script-based access to the cursor.

- HPROF Heap / CPU Profiling Tool

  HPROF is a simple command line profiling tool which comes with the *J2SE* platform. It is a JVM native agent library which is dynamically loaded through a command line option at startup and becomes part of the JVM process. Users can request various types of profiling features like CPU usage, heap allocation statistics, complete heap dumps, as well as states of all monitors and threads to track down and isolate performance problems involving memory usage. The tool generates textual or binary data which can be used with tools like *HAT (Heap Analysis Tool)* to browse the allocated objects by running queries against a heap dump. [HPROF]

Currently GCSpy is a pure heap visualization tool which eases the inspection of the heap structure, but not able to gather information about allocated objects. Such functionality might be also interesting for GCSpy but requires changes also on the server side of the application. Because this thesis only considers a scripting extension for the visualizer this would be a topic for another project.

- Heapviz

  Heapviz is a tool for observation of runtime behavior of complex systems which involves algorithms for aggregating and abstracting individual objects to create a more succinct summary of the heap using single representative elements for large containers and provide interactive query and navigation methods to expand or collapse regions of the heap, inspect individual objects and field values, search for objects or classes based on type and explore the connectivity of the object graph. [Aftandilia10]

  Such techniques to explore the object structure of the heap would also be an interesting functions for GCSpy, especially when debugging complex applications. As already mentioned in the previous bullet such actions would exceed the possibilities of this project but are considerable for further improvements on the application.

Chapter 3

Design

Now that we have seen some related applications and got an idea about the approach of the thesis, we define the detailed requirements, as well as some design decisions for the implementation. Therefore we must find a capable and highly expressive scripting language and an appropriate interpreter implementation that can be utilized in GCSpy's visualizer. Then the layout and the appearance of the new GUI components for script management and debugging have to be defined. Finally we determine some design directives for the software architecture and the scripting interface.

3.1 Requirements

This section defines the requirements for the implemetation and discusses the new functionality that GCSpy should obtain by the integration of a scripting engine. The first part lists the common functionality for comfortable script management. The following subsections describe the functionality that should be provided through the scripting interface. For the list of corresponding use-cases see appendix A.

3.1.1 Script management

In order to make script administration tasks as easy as possible, it is important to provide a practical and comfortable script management interface. This requires a clearly arranged layout and a manageable number of control elements. To achieve uniformity the interface should be implemented as a GUI element that matches the design of existing utility components in GCSpy. It has to offer fundamental functionality that is necessary for efficient script handling:

- Provide an overview of all registered scripts
- Create new scripts
- Load scripts from the file system

- Remove scripts from the visualizer
- Edit scripts using the systems default editor if supported, or allow the definition of a alternative command line
- Change script's execution state (enable/disable)
- Give information if scripts have been modified
- Allow the user to comment scripts
- Allow direct interaction between user and scripting engine (debugging console)
- Provide efficient error handling (automatically disable and mark faulty scripts and show error information)

3.1.2 Navigation

In some situations it might be useful to have the ability to control the visualization process via script command. Especially a breakpoint function which stops the visualization if a specific condition is fulfilled, i.e. a specific data field reaches a threshold, is one of the most requested features. For the sake of completeness the API should provide access to all relevant control functions available in the existing user interface.

- Pause the visualization
- Resume the visualization
- Navigate one single event forward or backward
- Set the current event

3.1.3 Spaces and streams

The current implementation of GCSpy is only able to handle a fixed number of spaces defined in the server-side data collection part of the framework. For a more flexible work flow GCSpy would benefit of the ability to create new virtual spaces out of existing trace data without changing a single line of code in the server implementation. This requires a number of fundamental functionality:

- Create new space data objects by script
- Create new stream data objects by script
- Deploy new GUI components at runtime
- Make the trace data accessible to scripts

3.1.4 Markers and cursor

In practice it is often useful for developers to quickly identify tiles of interest. For that reason a function would be profitable, that automatically highlights important tiles in a space when specific conditions are fulfilled. The scripting interface has to provide functionality to access respectively modify the marker state of single tiles in a space.

- Make maker information accessible to scripts
- Allow scripts to set markers
- Support individual maker colors for every tile

3.1.5 Control information

In addition to the stream values GCSpy offers control information for every tile in a space which results in links and separators between them. To use this function in virtual spaces the scripting API has to provide access to this controls.

- Let scripts read the control information of native spaces
- Allow scripts to set the control information of virtual spaces

3.1.6 Event and timing information

GCSpy currently offers access to event and timer information as described in chapter 2. In order to use this data for debugging issues, it should also be available in the scripting engine. Further the event filters function should be considered in the script API.

- Make event information accessible to scripts
- Make timer information accessible to scripts
- Allow scripts to modify the event filters settings

3.1.7 Logging

To be able to write arbitrary memory management or script state information into external files, the scripting interface should provide basic logging functionality.

3.1.8 External libraries

The python engine should be capable to use external python libraries for realization of further extensions. The python standard libraries should be available native in GCSpy without installation of further software modules.

3.2 Scripting language

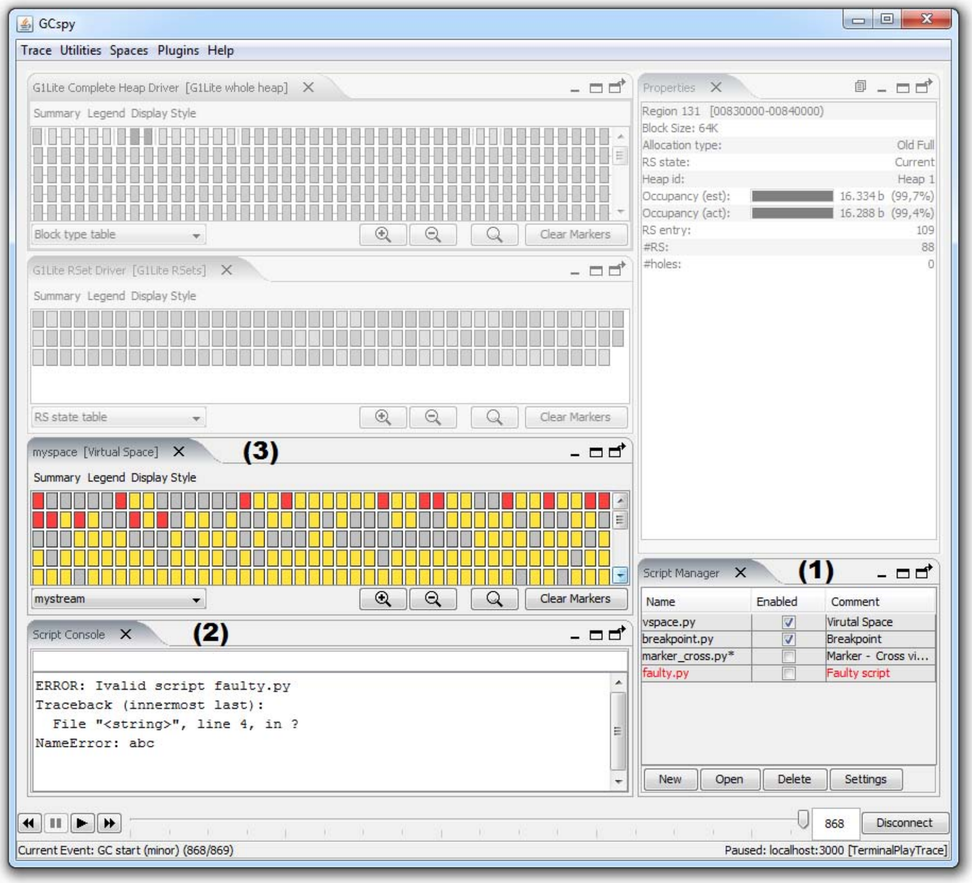
An appropriate scripting language needs among others high expressiveness, as well as a simple and easy to learn syntax. *Python* is such a powerful programming language and satisfies both requirements. Compared to other high level languages it provides a good amount of functionality and its syntax is relatively tense, retaining the same expressiveness. Another important argument is the huge number of available libraries that can be utilized to extend GCSpy's abilities.

As the visualizer of GCSpy is written in *Java*, an appropriate *Python* interpreter for this project should be available as a *Java* library. The open source project *Jython* meets all requirements and allows an easy integration of *Python* functionality into existing *Java* applications. The interpreter allows the execution of single commands and whole scripts, as well as exchange and access of data object in both directions. [Jython]

We use the standalone version of Jython because it already contains the python standard libraries, so they can be accessed without installing further modules on the system. Part of this library is the logging module which can be easily utilized to satisfy the requirement for an adequate logging facility in a flexible way. See the logging documentation for more details. [Logging]

3.3 GUI

To grant an efficient script handling, the GUI has to be augmented with two new elements. The first one is a *Script Manager* (1) that gives an overview of the registered scripts and their current state, as well as the ability to manipulate and administrate them. The second one is a *Script Console* (2) which shows script output or error information from the script interpreter and allows the developer to enter single commands for debugging issues. Figure 3.1 shows the integration of this new GUI elements in the visualizer, as well as an exemplary *Virtual Space* (3) dynamically generated and incorporated by a script.


GCspy
Trace Utilities Spaces Plugins Help
G1Lite Complete Heap Driver [G1Lite whole heap]
Summary Legend Display Style
Block type table
Clear Markers
G1Lite RSet Driver [G1Lite RSets]
Summary Legend Display Style
RS state table
Clear Markers
myspace [Virtual Space]
(3)
Summary Legend Display Style
mystream
Clear Markers
Script Console
(2)
Properties
Script Manager
(1)
New
Open
Delete
Settings
868
Disconnect

Region 131 [00830000-00840000)
Block Size: 64K

Allocation type:	Old Full
RS state:	Current
Heap id:	Heap 1
Occupancy (est):	16.334 b (99,7%)
Occupancy (act):	16.288 b (99,4%)
RS entry:	109
#RS:	88
#holes:	0

Name	Enabled	Comment
vspace.py	✓	Virutal Space
breakpoint.py	✓	Breakpoint
marker_cross.py*	☐	Marker - Cross vi...
faulty.py	☐	Faulty script

```
ERROR: Ivalid script faulty.py
Traceback (innermost last):
  File "<string>", line 4, in ?
NameError: abc
```

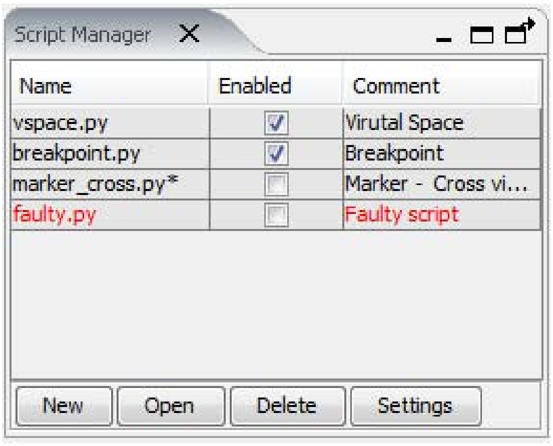
Current Event: GC start (minor) (868/869)
Paused: localhost:3000 [TerminalPlayTrace]

Figure 3.1: New GUI elements: Script Manager and Script Console

The *Script Manager* shows all registered scripts in a list. The first column holds the name of the script, which is equivalent to the corresponding file name. If a script has been modified, it gets labeled with a bullet on the end of the name. The second column holds a check box that indicates the current execution state of the script. It also allows the user to enable or disable a script by left click on the check box. The third column contains a comment, which is parsed from the first appearing comment inside the script content. If a script causes an error during execution it gets disabled automatically and marked as faulty using red font color in the script list.

The control area contains four buttons as shown in figure 3.2:

- New: Creates a new script using a stored template (see UC1.1)
- Open: Opens a script from a file (see UC1.2)
- Delete: Removes a script form the *Script Manager* (see UC1.4)
- Settings: Opens the *Script Settings* dialog.

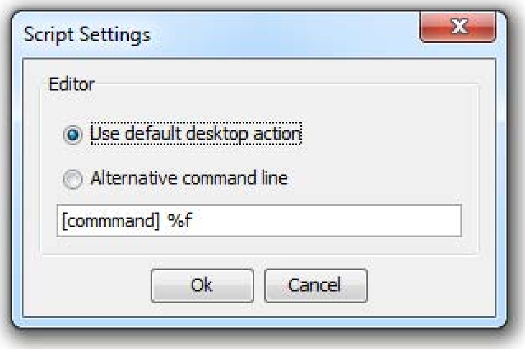


Script Manager

Name	Enabled	Comment
vspace.py	✓	Virutal Space
breakpoint.py	✓	Breakpoint
marker_cross.py*	☐	Marker - Cross vi...
faulty.py	☐	Faulty script

Figure 3.2: Script Manager

It is also possible to open scripts in an external editor, which is either the default editor defined in the operating system or the application defined it the alternative command string option inside the *Script Settings* dialog (see figure 3.3). The second method is recommended if the OS does not provide a default desktop operation.



Script Settings

Editor

[✓ Use default desktop action]

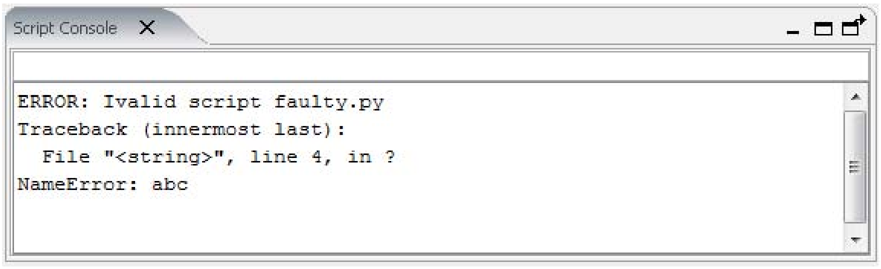
Alternative command line

[commmand] %f

Figure 3.3: Script Settings

To edit a script the user double clicks the desired line in the script list. While the script is opened in the particular editor, it gets disabled in the *Script Manager* and marked as modified. When the user has finished his changes, he safes the updated content to the file and closes the editor. After that he reactivates the script in the *Script Manager* which causes the application to reload the script's content from the file.

The second GUI component needed for scripting is the *Script Console* shown in figure 3.4. In the top area it contains a command line which enables the developer to directly execute single commands and functions. The console runs in the same context as the scripts registered in the *Script Manger* which alleviates script debugging tasks. It also provides a input history function for convenient recall of previously entered commands using the up- and down arrow keys.


Script Console

```
ERROR: Ivalid script faulty.py
Traceback (innermost last):
  File "<string>", line 4, in ?
NameError: abc
```

Figure 3.4: Script Console

The text area below shows the output of the scripting engine and accordant error information in case of a fault. It is also possible to print individual content using the accordant print function in the scripting interface.

3.4 Scripting interface

Another important design task is the scripting interface itself. For some use-cases it is necessary to distinguish between commands which are executed only once at load time and commands which are executed at every event. During creation of a virtual space, for example space and streams first must be initialized and deployed to the main frame once when script is loaded. Additionally the space data has to be updated at every event. For this reason scripts in GCSpy have to provide two basic functions for initialization and event handling as shown in the following code fragment.

```python
"""First comment gets parsed and displayed in the Script Manager"""

class NewScript:

  def __init__(self):

    """Initialization section gets executed only once at load time"""

  def event(self, curEvent, eventID, elapsedTime, compensationTime):

    """Event section gets executed at every event"""
```

Listing 3.1: Basic script structure

When the script interpreter loads a script, it instances the first class found inside the script and automatically calls its *init* method once, provided that such a method exists. If the main class defines an *event* method, it gets executed at every event. Additional utility classes and methods are not considered directly, but are available in the script interpreter's context and can be invoked from initialization or event method.

The first comment within the script gets parsed at load time and displayed in the *Script Manager*. This enables the user to place short notes to gain a better overview on loaded scripts. In the initialization section definition of variables must be done in the scripts context (self) to avoid name interferences with oder running scripts. The event section provides some important event and timing parameters which can be used to flexibly filter the execution of the scripts event function using simple conditions.

- curEvent: The serial number of the current event.
- eventID: The type of the current event.
- elapsedTime: Time passed since the last event.
- compensationTime: Amount of time used for compensation.

The syntax for the scripting API should be equivalent to the *Java* code used in the object structure of the visualizers implementation. This brings benefits when directly dealing with native data objects like *Stream*, *Space* or *Event* because the user does not have to distinguish between API and object syntax.

3.5 Example Scripts

This section shows some example scripts for the most representative use-cases described in appendix A. For detailed information about the used commands see the API documentation in appendix B.

All scripts are described and explained via inline comments. The used trace data contains two spaces. Space 0 is the main space and contains the whole heap information, while space 1 holds information about the result sets.

3.5.1 Breakpoint

The script observes the main space and stops the visualization when any tile in stream 3 (occupancy estimated) reaches a threshold of 10000. It remembers breakpoints already visited and ignores them in the further process. Tiles which triggered a visualization stop get marked and the latest one gets selected with the cursor so that the properties window displays the tile information.

```
class BreakPoint:

  def __init__(self):
    """Get number of tiles in space 0 and define threshold of 10000"""
    self.tilenum = gcs.getSpace(0).getTileNum()
    self.threshold = 10000;
    self.memory = []
  def event(self, curEvent, eventID, elapsedTime, compensationTime):
    """Iterate over all tiles in space 0"""
    for i in range(0, self.tilenum):
      """Define breakpoint condition (stream value reached threshold)"""
      if gcs.getVal(0, 3, i) > self.threshold and not
      (i in  self.memory):
        """Let cursor point on the tile and highlight it with a marker"""
        gcs.setCursor(0, i)
        gcs.setMarked(0, i)
        """Flag the tile as visited"""
        self.memory.append(i)
        """Stop the visualization and exit iteration"""
        gcs.pause()
        break
```

Listing 3.2: Breakpoint

3.5.2 Virtual Spaces

This example script creates a new virtual space containing one stream which displays a summery of old and young generation tiles. It gathers the information from stream 0 (block type table) inside the main frame.

```
from java import awt

class VirtualSpace:

  def __init__(self):
    """Define references on space 0 (main space) and """
    """stream 0 (block type table) and get number of tiles"""
    self.complete = gcs.getSpace(0)
    self.blocktype = self.complete.getStream(0)
    self.tilenum = self.complete.getTileNum()
    """Create a new space with one stream"""
    self.myspace = gcs.createSpace("myspace", self.tilenum,
    1, "Region ", "Block Size: 64k")
    self.enumnames = "Young", "Old", "Unused"
    self.mystream = gcs.createStream(self.myspace,
    "mystream", 0, 2, 0, 0, "", "", gcs.PRES_ENUM, 0,
    awt.Color.red, self.enumnames)
    gcs.addSpace(self.myspace)
    self.myspaceID = self.myspace.getID()
    """Get indices of young generation"""
    self.eden = gcs.nameToIndex(self.blocktype, "Eden")
    self.survivor = gcs.nameToIndex(self.blocktype, "Survivor")
    self.largeyoung = gcs.nameToIndex(self.blocktype, "LargeYoung")
    """Get indices of old generation"""
```

```
        self.oldfull = gcs.nameToIndex(self.blocktype, "Old Full")
        self.oldcandidate = gcs.nameToIndex(self.blocktype, "Old Candidate")
        self.oldrecyclable = gcs.nameToIndex(self.blocktype, "Old Recyclable")
        self.large = gcs.nameToIndex(self.blocktype, "Large")

    def event(self, curEvent, eventID, elapsedTime, compensationTime):
        """Filter unwanted events"""
        if eventID == 0 or eventID == 1:
            data = []
            young_sum = 0
            old_sum = 0
            """Iterate over all tiles in source space 0"""
            for i in range(0, self.tilenum):
                """Get tile value for stream 0 (block type table)"""
                val = gcs.getVal(0, 0, i)
                if val == self.eden or val == self.survivor or
                val == self.largeyoung:
                    """Summarize young generation"""
                    young_sum+=1
                    data.append(0)
                    gcs.setControl(self.myspaceID, i, gcs.CTRL_USED)
                elif val == self.oldfull or val == self.oldcandidate or
                val == self.oldrecyclable or val == self.large:
                    """Summarize old generation"""
                    old_sum+=1
                    data.append(1)
                    gcs.setControl(self.myspaceID, i, gcs.CTRL_USED)
                else:
                    """Unused tiles"""
                    gcs.setControl(self.myspaceID, i, gcs.CTRL_UNUSED)
                    data.append(2)
            """Create summary for the virtual stream"""
            summary = young_sum, old_sum, self.tilenum-young_sum-old_sum
            self.mystream.setSummary(summary)
            """Set data of the virtual stream"""
            gcs.setStreamData(self.mystream, data)
```

Listing 3.3: Virtual Space

3.5.3 Markers

The script marks every tile inside the main space where the value of stream 3 (occupancy estimated) reaches a threshold of 16000 and also the corresponding result set in space 1.

```
class Marker:
  def __init__(self):
    """Get number of tiles in space 0 and define threshold of 16000"""
    self.tilenum = gcs.getSpace(0).getTileNum()
    self.threshold = 16000;

  def event(self, curEvent, eventID, elapsedTime, compensationTime):
    """Clear all markers in both spaces"""
    gcs.clearMarkers(0)
    gcs.clearMarkers(1)
    """Iterate over all tiles in space 0"""
    for i in range(0, self.tilenum):
      """Define marking condition (stream value reached threshold)"""
      if gcs.getVal(0,3,i) > self.threshold:
        """Set a marker on the tile and mark corresponding result set"""
        gcs.setMarked(0,i)
        gcs.setMarked(1,gcs.getVal(0, 5, i))
```

Listing 3.4: Marker

3.5.4 Event Filters

This example script slows down the visualization process from event index 200 to 300 using the Event Filter functionality. It reduces the event types delivered from the server to GC start events (minor and major), GC end events are ignored. Furthermore it uses the delay functionality to slow down the visualization.

```
class Filter:

  def event(self, curEvent, eventID, elapsedTime, compensationTime):
    """Define start condition (event index = 200)"""
    if curEvent == 200:
      """Configure which types of events are enabled"""
      enabled = 1, 1, 0, 0
      gcs.efSetEnabled(enabled)
      """Set delay for enabled event types to 200ms"""
      delays = 200, 200, 0, 0
      gcs.efSetDelays(delays)
    """Define end condition (event index = 300)"""
    elif curEvent == 300:
      """Restore default settings for event filters"""
      gcs.efEnableAll()
      gcs.efClearDelays()
```

Listing 3.5: Event Filters

3.5.5 Logging

This example script demonstrates the logging functionality using the python standard library. It observes the main space and emits a log entry when any tile in stream 3 (occupancy estimated) reaches a threshold of 16000. Tiles already visited are remembered and ignored in the further process.

```python
import logging

class Logging:

  def __init__(self):
    """Setup logger"""
    logging.basicConfig(filename='example.log',
      level=logging.DEBUG, filemode='w')
    self.logger = logging.getLogger('Logging example script')
    """Get number of tiles in space 0 and define threshold of 16000"""
    self.tilenum = gcs.getSpace(0).getTileNum()
    self.threshold = 16000;
    self.logger.info('Threshold = ' + str(self.threshold))
    self.memory = []

  def event(self, curEvent, eventID, elapsedTime, compensationTime):
    """Iterate over all tiles in the main space"""
    for i in range(0, self.tilenum):
      """Get value of stream 3 (occupancy estimated)"""
      val = gcs.getVal(0,3,i)
      """Define logging condition (tile value reached threshold)"""
      if val > self.threshold and not (i in  self.memory):
        """Emit log information and mark tile as visited"""
        self.logger.info('Event ' + str(curEvent) +  ' - Tile '
          + str(i) + ' exceeded threshold: ' + str(val))
        self.memory.append(i)
```

Listing 3.6: Logging

Chapter 4

Implementation

This chapter describes implementation details and required modifications on the visualizer for the integration of the scripting engine. In order to find the right strategy, the first step implies a detailed analysis of the visualization process. Next part will cover integration of the script interpreter and components for script management. Then functionality defined in the requirements must be made accessible to the scripting engine.

4.1 Analysis of the visualization process

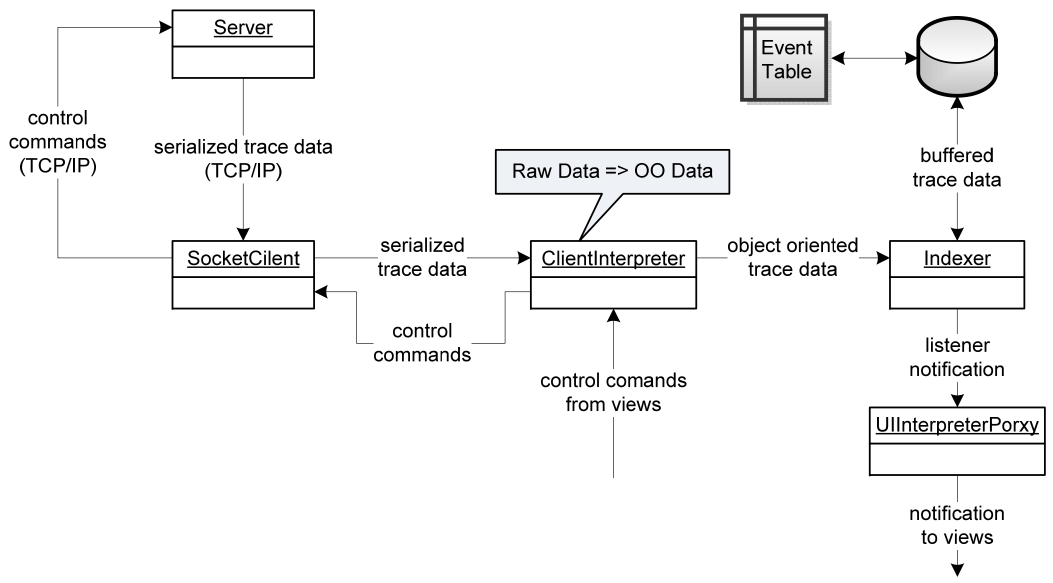


Figure 4.1: Data flow

Figure 4.1 shows data and control flow between server and client. As already mentioned in the introduction GCSpy is implemented as a server-client architecture. The server serializes the gathered trace data and transfers it to a client interpreter over TCP/IP. This interpreter converts raw data into an object oriented data structure and passes it to the visualizer. View actualization is carried out through a listener architecture using a proxy which delegates notifications to all registered views. This architecture grants better separation between data model and GUI components.

Between client and visualizer there also exists an indexer which buffers incoming data into an index file. This procedure allows local event history access inside the visualizer when data transmission is paused. Inside the user interface this is realized through the navigation panel containing control buttons and event slider. The history stored in the index file provides all events from connection time to the latest event received from the server. In the opposite direction the client interpreter is able to send control commands to the server to steer the data transmission, i.e. stop and resume the data transfer and send updates for the event filter settings.

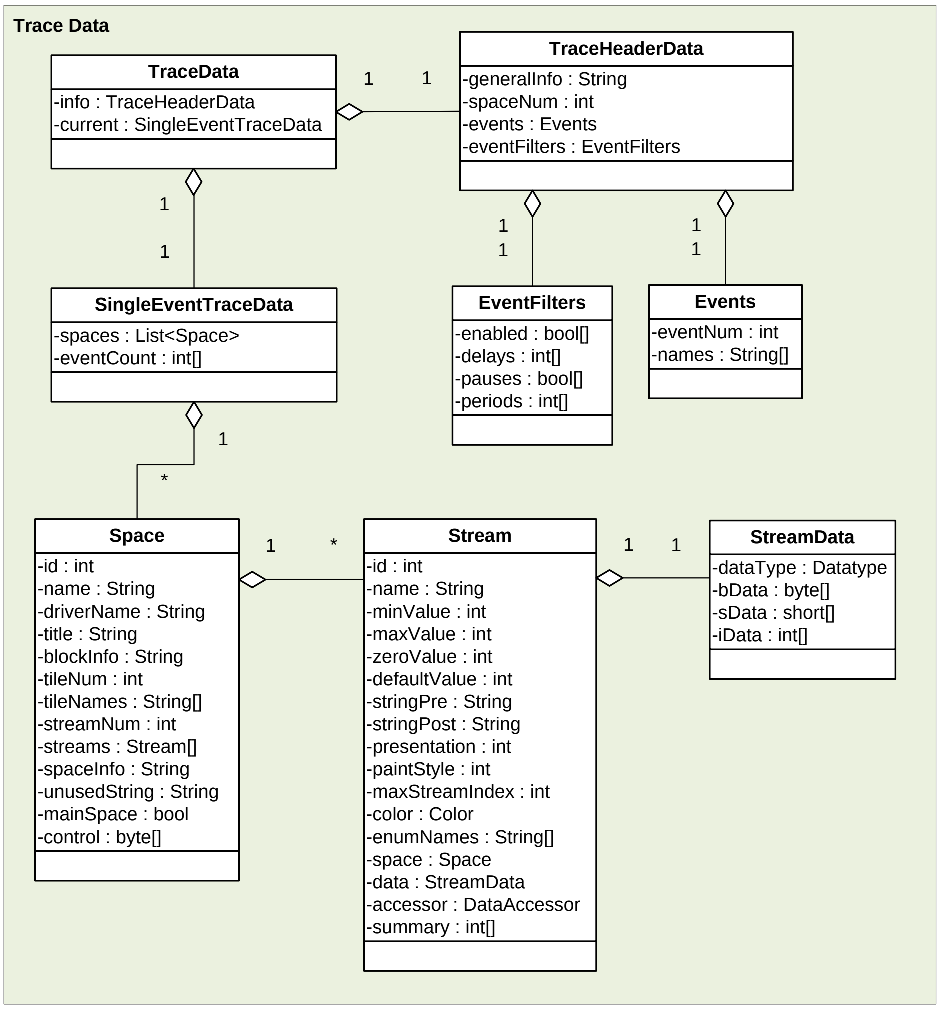


Figure 4.2: Trace data structure

The detailed class structure of the trace data is shown in figure 4.2. Main component of the diagram is the *TraceData* object which contains the snapshot of the memory structure for a single event, represented by an object named *SingleEventTraceData*. It mainly consists of a number of *Spaces* and *Streams* holding heap information. Furthermore *SingleEventTraceData* provides an integer array with counters for every singe type of event.

In addition to the single event data *TraceData* contains a header holding some general information like the total number of spaces, details about the different types of events, as well as the *EventFilters* responsible for data transmission control between server and client.

SingleEventTraceData contains at least one *Space* which again consists of one or multiple *Streams*. All of them provide a unique integer identifier, as well as several fields holding data like name, driver, title, block and space information. Also every tile has its own name which is represented by an string array. Furthermore the control information is stored for every space as a byte array. See [Hofer10] for further information about the control functionality.

GCSpy supports different types of streams. Tiles in enumeration streams display different memory states using distinct colors. Tiles in percentage streams show arbitrary data values in form of a vertical chart bar. Parameters used in the particular stream depend on its type, for example the enumeration names are only used in enumeration streams. All streams comprise ether trace data of type byte, short or integer and provide access to them by accordant accessors.

```java
private void playOne(int nextEvent) {
  ...
  FileInputStream f = new FileInputStream(indexFile);
  long framePos = indexer.getFramePosAt(nextEvent);
  FileUtils.seek(f, framePos);

  int size = (int) indexer.getFrameSizeAt(nextEvent);
  byte[] data = new byte[size];
  FileUtils.read(f, data, 0, data.length);

  ByteArrayInputStream in = new ByteArrayInputStream(data);

  SingleEventTraceData trace = new SingleEventTraceData(
    interpreterData.getSpaceNum());
  for (int i = 0; i < indexer.getNumSpaces(); i++) {
    trace.setSpace(i, indexer.getSpace(i));
  }
19      EventInterpreter interpreter = new EventInterpreter(in, trace);
        interpreter.execute();

22      for (int i = 0; i < indexer.getNumSpaces(); i++) {
          proxy.fireSpaceListeners(indexer.getSpace(i));
        }
25
        proxy.fireEventListeners(interpreter.getLastEventId(),
          interpreter.getLastElapsedTime(),
28        interpreter.getLastCompensationTime());

        curEvent = nextEvent;
31      ...
      }
```

Listing 4.1: Function playOne in MainFrame

An important task is to find an adequate position to apply event-based script execution. Code fragment 4.1 shows the original *playOne* function inside the *MainFrame* class of the visualizer which gets executed every time the position of the event slider changes and also when a new event arrives.

Parameter *nextEvent* represents the index of the event to show next inside the visualizer. Between line 3 and 11 system reads the index file containing the complete event history including the latest event received from the server and locates the correct frame position according to the given event index to be displayed. Then it constructs a *SingleEventTraceData* object for the event from the indexer information from lines 13 to 17. After that system starts an *EventInterpreter* processing the information (line 19, 20) and fires all space and event listeners using the *UIInterpreterProxy* from line 22 to 28.

After the *EventInterpreter* has handled the data, there is a point where scripting can be applied without much effort. The scripting engine additionally processes the information before the listeners are notified. The next section describes the required changes on this method in detail.

4.2 Required changes on the system architecture

This section describes the required modifications on the visualizer. Figure 4.3 shows an abstract overview on the static class structure including components responsible for script interaction.

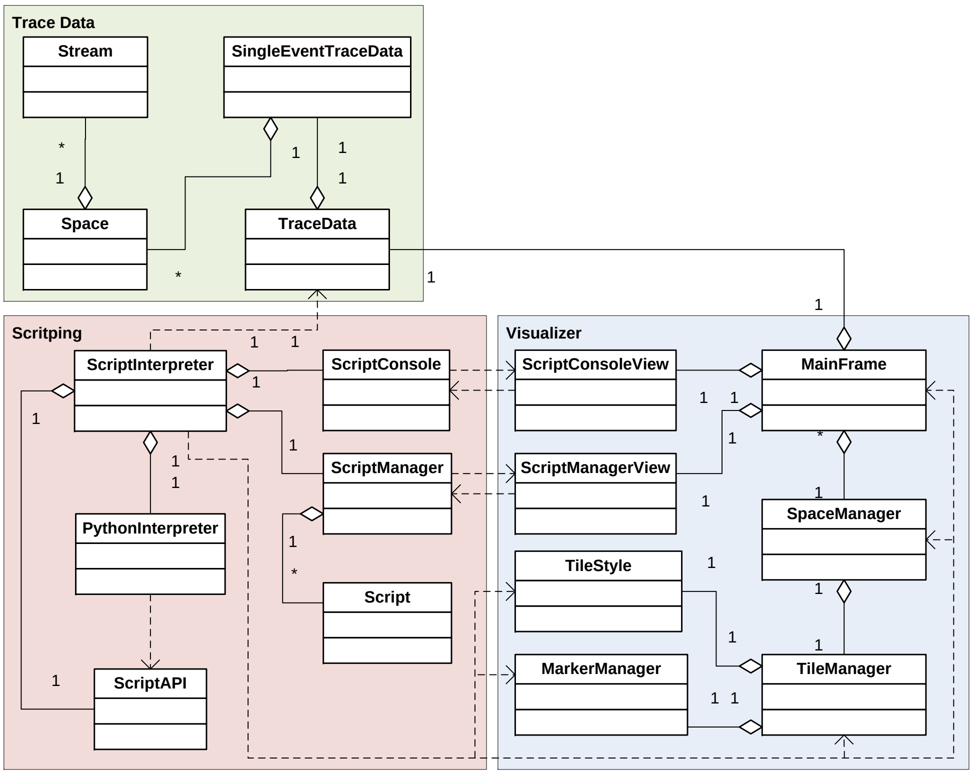


Figure 4.3: Overall class structure

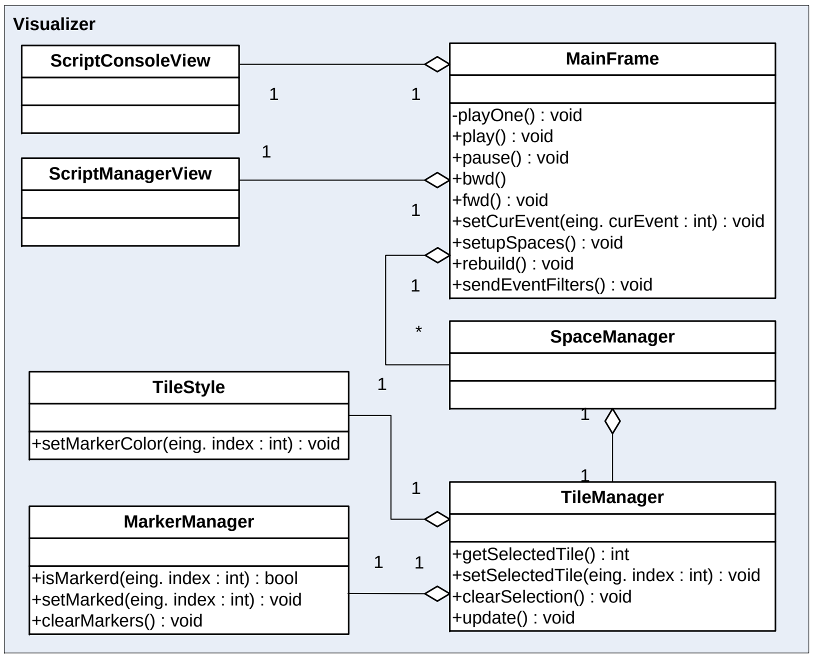


Figure 4.4: Visualizer class structure

Figure 4.4 shows a class diagram for the visualizer. Its central component is the *MainFrame* which contains the primary GUI components. Most of the interactive control functionality is directly reachable through this class. [Hofer10]

In order to gain access to the visualizers functionality the script interpreter has to interact with some of its components. For navigation and control of the visualization process the interpreter has to call accordant methods inside the *MainFrame* which result in the same actions like the navigation buttons and the event slider. Furthermore the *MainFrame* has to consider virtual spaces during setup of the *SpaceManagers* accessing the *ScriptInterpreter* and also provide a function to rebuild the GUI if the number of spaces changes at runtime. Also the method for sending the event filter settings to the server must be accessible to the script interpreter.

For the marker and cursor functionality the script interpreter has to make use of the *TileManager* which is part of the *SpaceManager*. The *TileManager* itself allows request and manipulation of the current cursor position and also holds a *MarkerManager* which is provides equivalent functionality for the markers. Every *TileManager* has a *TileStyle* object which has been modified to allow the use of individual marker colors instead of one common color per space.

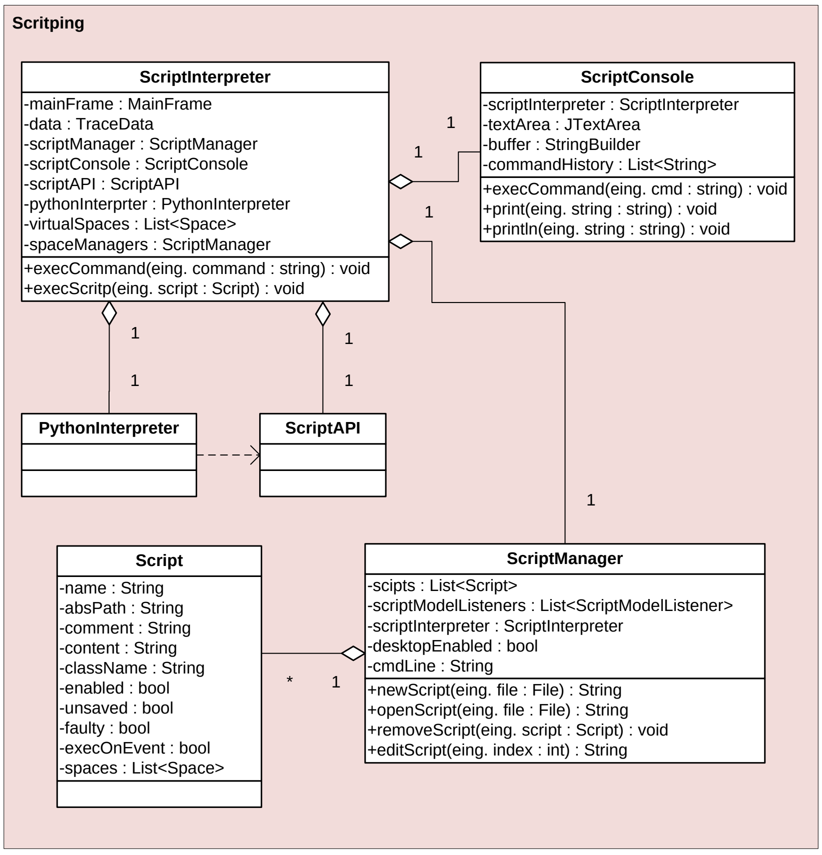

Scritping
ScriptInterpreter
-mainFrame : MainFrame
-data : TraceData
-scriptManager : ScriptManager
-scriptConsole : ScriptConsole
-scriptAPI : ScriptAPI
-pythonInterprter : PythonInterpreter
-virtualSpaces : List<Space>
-spaceManagers : ScriptManager
+execCommand(eing. command : string) : void
+execScritp(eing. script : Script) : void
ScriptConsole
-scriptInterpreter : ScriptInterpreter
-textArea : JTextArea
-buffer : StringBuilder
-commandHistory : List<String>
+execCommand(eing. cmd : string) : void
+print(eing. string : string) : void
+println(eing. string : string) : void
PythonInterpreter
ScriptAPI
Script
-name : String
-absPath : String
-comment : String
-content : String
-className : String
-enabled : bool
-unsaved : bool
-faulty : bool
-execOnEvent : bool
-spaces : List<Space>
ScriptManager
-scipts : List<Script>
-scriptModelListeners : List<ScriptModelListener>
-scriptInterpreter : ScriptInterpreter
-desktopEnabled : bool
-cmdLine : String
+newScript(eing. file : File) : String
+openScript(eing. file : File) : String
+removeScript(eing. script : Script) : void
+editScript(eing. index : int) : String
1
1
1
1
1
1
1
1
*
1

Figure 4.5: Scripting class structure

Figure 4.5 shows a more detailed view on the scripting components including all data fields, functions and dependencies. For the integration of the *Python* interpreter we introduce a new component called *ScriptInterpreter* holding an instance of the *Jython* interpreter itself and responsible for all kinds of script execution. Furthermore this class is the central node for all interaction between scripts and the remaining system.

For this issue the interpreter maps the *ScriptAPI* into the python interpreter under the variable name *gcs*. The object contains all necessary methods and functions to allow script-based interactions with the visualizer. The particular commands are delegated to the *ScriptInterpreter* which provides the requested data and calls the accordant methods in the visualizer. Furthermore the *ScriptInterpreter* holds a list of all *Space-Managers* currently registered in the *MainFrame*, as well as a list of all virtual spaces created by scripts.

The *ScriptInterpreter* provides different methods for script execution. The first one runs single commands in the python interpreter which is necessary for the debugging console. The second one executes a whole script which is needed once at script initialization time to define variables, create virtual spaces and streams and register the scripts methods. The last one invokes the event method of all registered scripts to set actions at event time.

In order to keep a clear separation between data model and interpreter we introduce a *ScriptManager* holding a list of all registered scripts and responsible for all script administration issues. The *ScriptManagerView* represents the accordant interface inside the GUI. The same strategy was used for the *ScriptConsole* which is also separated into view and data model. See chapter 3 for further information about these two components.

```
private void playOne(int nextEvent) {
  ...
  EventInterpreter interpreter = new EventInterpreter(in, trace);
  interpreter.execute();

  // Get event and timing information from EventInterpreter and
  // pass it to the ScriptInterpreter, execute all registered scripts
  int eventID = interpreter.getLastEventId();
  int elapsedTime = interpreter.getLastElapsedTime();
  int compensationTime = interpreter.getLastCompensationTime();
  scriptInterpreter.execOnEvent(curEvent, eventID,
    elapsedTime, compensationTime);

  // Fire listeners of native spaces
  for (int i = 0; i < indexer.getNumSpaces(); i++) {
    proxy.fireSpaceListeners(indexer.getSpace(i));
  }

  // Fire listeners of virtual spaces
  int spacenum = indexer.getNumSpaces();
  for (int i = spacenum; i < spacenum +
    scriptInterpreter.getSpaceNum(); i++) {
    proxy.fireSpaceListeners(scriptInterpreter.getSpace(i));
  }

  proxy.fireEventListeners(eventID, elapsedTime, compensationTime);
  ...
}
```

Listing 4.2: Modified playOne function

For event-based script execution the *playOne* introduced in the previous section has to be modified so that it invokes the scripts event function and considers the virtual spaces during notification of the views. Code fragment 4.2 above shows how it works in detail. After the *EventInterpreter* has processed the trace data system calls a method inside the *ScriptInterpreter* which executes the event method of all registered scripts with the event and timing parameters gathered form the *EventInterpreter*. Then in addition to the notification of the views for the native spaces system has to fire space listeners for the virtual ones. Their number and the virtual spaces itself are requested from the *ScriptInterpreter*.

Chapter 5

Conclusion and future work

GCSpy is now featured with an integrated scripting engine which allows use of python language inside the visualizer. Now developers are free to create new individual spaces and streams showing information calculated by python scripts. Furthermore they may now use common breakpoint functionality to control the visualization process.

Also the script-based marker access with individual color support eases debugging by dynamically highlighting important tiles inside spaces. The API grants access to all trace data, event and timing information, as well as the event filter functions to reject unwanted information or slow down the visualization.

The GUI provides a script management utility, as well as a debugging console which allows direct script interaction. Another accomplishment of this project is the fact that now there are no barriers for future enhancements using the huge variety of available python libraries.

In order to achieve more convenience it should be possible to response spaces and streams by their index or short name. The current data structure considers only a field for the full name which is not very efficient because of its length. This extension requires changes also on the server module of GCSpy.

At the moment, GCSpy offers a plug-in interface including several plug-ins for histogram visualization and recording as described in chapter 2. Unfortunately, they are not very interesting for the developer because they do not give new information.

During the latest improvements the visualizer gained an indexing mechanism which made the history examination more comfortable and reduced the importance of this histogram plug-ins. Furthermore, the entire plug-in framework is difficult to maintain or modify because it is hard coded in Java.

The new scripting functionality offers the opportunity to re-implement the whole plug-in framework in a more reasonable and flexible way utilizing some external graphic libraries like *matplotlib*.

Appendix A

Use-cases

This appendix lists the fundamental user interactions for script management inside the GUI and some common use-cases for script utilization, i.e. the basic usage of the scripting interface. Scripting offers a wide application spectrum, especially when using external libraries to extend functionality. For this reason not all possible use-cases can be considered.

All of the following textual use-cases include only the two actors *User* and *System*. One general precondition is a running GCSpy application which is connected to a server or running with a trace file. Also the utilities necessary for script administration (*Script Manager*, *Script Console*) have to be open.

A.1 Script management

UC1.1	**Create new script**
Description	Create a new script file and deploy it to the visualizer.
Preconditions	-
Postconditions	File system contains a new script file in the chosen location. Script is registered in *Script Manager* and enabled by default.
Normal Course	1. User clicks the *New* button in the *Script Manager*. 2. User defines name and location for the script file using a standard file chooser dialog (default name is *new*). 3. System adds an empty script object to *Script Manager* and creates the corresponding script file.
Failure	a. File with desired name already exists in chosen location. User must confirm to overwrite existing file. b. Script with desired name already registered in *Script Manager*. Show accordant error Message and abort action.

Table A.1: UC1.1 - Create new script

UC1.2	**Load script from file**
Description	Load a script from an existing file and deploy it to the visualizer.
Preconditions	At least one script file is available in the file system.
Postconditions	Script is registered in *Script Manager* and enabled by default. View contains entry showing scripts filename, comment and state. GUI elements created by the script get integrated into the main frame.
Normal Course	1. User clicks the *Open* button in the *Script Manager*. 2. User browses a file chooser dialog for the desired script file and clicks the *Open* button. 3. System creates a script object form the file content and deploys it to the *Script Manager*. 4. System changes scripts execution state to enabled. GUI elements created in the scripts initialization function get integrated into the main frame. If a event function was defined it will be executed every time an event occurs.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.2: UC1.2 - Load script from file

UC1.3	**Edit script**
Description	Change a scripts content using an external editor.
Preconditions	At least one script is registered in the *Script Manager*.
Postconditions	Script is up to date, all changes have taken effect in the visualizer.
Normal Course	1. User selects desired script form the *Script Manager* by double click. 2. System opens script using the default editor configured in the OS. 3. System deactivates script and marks it as edited in the *Script Manager* which appears as a star at the end of the filename. 4. User edits script data and safes changes to the file using the external editor. 5. User reactivates the edited script in the *Script Manager*. 6. System updates script data from file and enables it for execution.
Failure	a. Default desktop action not supported by the OS: System shows accordant error message. User is enabled to start an arbitrary external editor using the alternative command in the *Script Settings*. b. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.3: UC1.3 - Edit script

UC1.4	**Delete script**
Description	Remove a script from the visualizer.
Preconditions	At least one script is registered in the *Script Manager*.
Postconditions	Deleted script is not longer registered in the *Script Manager* and does not get executed anymore. All GUI elements created by the script are removed from the main frame. The associated script file stays untouched.
Normal Course	1. User selects one or multiple scripts in the *Script Manager* and clicks the *Delete* button. 2. System disables all selected scripts and removes all virtual spaces created by them from the main frame. 3. System removes all selected scripts from the *Script Manager*.
Failure	-

Table A.4: UC1.4 - Delete script

UC1.5	**Change scripts execution state**
Description	Enable respectively disable a script for execution.
Preconditions	At least one script is registered in the *Script Manager*.
Postconditions	• Enabled: GUI elements created in scripts initialization section get integrated into the main frame. If an event function is defined it will be executed each time an event occurs. • Disabled: GUI elements created by the script are removed from the main frame. The scripts event function does not get executed anymore until it gets enabled again but script is still registered in the *Space Manager*.
Normal Course	1. User clicks the *Enabled* check box of the desired script in the *Script Manager*. 2. If the script was enabled before, system sets its state to disabled. It is still registered in the *Script Manager*, but does not get executed any more. All GUI elements created by the script are cleared from the main frame. 3. If the script was disabled before, system sets its state to enabled. All instructions declared in the initialization section of the script get executed. System integrates all GUI elements created by the script into the main frame.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.5: UC1.5 - Change execution state

A.2 Script utilization

UC2.1	**Breakpoints**
Description	Stop visualization process when specific condition is fulfilled.
Preconditions	The visualization process is running.
Postconditions	Visualization is paused and the user is able to walk through the trace data event by event using the GUI's navigation controls or by command line in the *Script Console*. All stream's current data values can be obtained from the *Properties* window or using the output section of the *Script Console*.
Normal Course	1. User creates a new script as described in UC1.1. 2. User defines a condition within the scripts event section accessing the native trace data at which the visualization should stop. 3. User applies the *pause* command to stop the visualization. 4. System executes the script at every event. If the defined condition is fulfilled the visualization stops.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.6: UC2.1 - Breakpoints

UC2.2	**Virtual spaces and streams**
Description	Create a new virtual space including several virtual streams by processing data from native ones.
Preconditions	-
Postconditions	Main frame contains the new virtual space.
Normal Course	1. User creates a new script as described in UC1.1. 2. In the script's initialization section user defines a new virtual space and at least one virtual stream. 3. User applies the *addSpace* command to deploy the new space to the main frame. 4. In the scripts's event section user accesses the native trace data and processes it for the virtual streams. 5. User updates virtual streams data and the corresponding summary.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.7: UC2.2 - Virtual spaces and streams

UC2.3	**Cursor**
Description	Let the cursor dynamically point on tiles of interest.
Preconditions	The visualization process is running.
Postconditions	Visualization is stooped and cursor points on desired tile.
Normal Course	1. User creates a new breakpoint script as described in UC2.1. 2. At every breakpoint the user additionally sets the cursor position to the tile that caused the visualization stop. 3. System executes the script at every event and stops the visualization when a breakpoint is reached. 4. System sets the cursor position to the actual breakpoint tile. The *Properties* window displays detail information for the selected tile.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.8: UC2.3 - Cursor

UC2.4	**Markers**
Description	Use the marker functionality to dynamically highlight tiles of interest.
Preconditions	-
Postconditions	Desired tiles get marked dynamically accordant to the defined conditions.
Normal Course	1. User creates a new script as described in UC1.1. 2. User defines one or multiple conditions at which specific tiles should get marked. 3. User applies accordant API commands to set and remove markers in arbitrary spaces and apply individual marker colors. 4. System executes script at every event and dynamically sets respectively removes markers if defined conditions are fulfilled.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.9: UC2.4 - Markers

UC2.5	**Control information**
Description	Set the control information for tiles in virtual streams and use GC-Spy's ability to display memory structure.
Preconditions	-
Postconditions	In addition to stream data the virtual space is provided with control information.
Normal Course	1. User creates a new virtual space script as described in UC2.2. 2. Within the scripts event section user defines one or multiple conditions at which control information for specific tiles should change. 3. User applies accordant control commands to set tiles to either used or unused or set links and separators between them. 4. System executes the script at every event and dynamically sets control information for all tiles in the virtual stream.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.10: UC2.5 - Control information

UC2.6	**Event filters**
Description	Use GCSpy's event filter functionality to slow down the visualization at critical sections or to dynamically limit observation detail during visualization.
Preconditions	The visualization process is running.
Postconditions	Event filter settings change dynamically according to the definitions in the script and influence the data transmission from the server.
Normal Course	1. User creates a new script as described in UC1.1. 2. Within the scripts event section user defines one or multiple conditions at which the event filter settings should change. 3. User applies desired event filter commands for changing state, delay and period for specific event types. 4. System executes the script at every event and adjusts the server-side event filters according to the definitions.
Failure	a. Script is faulty: Script automatically gets disabled and marked as faulty in the *Script Manager*. The output area of the *Scripting Console* shows an accordant error message.

Table A.11: UC2.6 - Event filters

Appendix B

API

The development of an efficient and easy to use scripting API is one of the most important requirements of this project. This appendix gives a detailed description to every single function. For a better understanding, chapter 4 contains a couple of example scripts for some common use-cases. Generally the command syntax is **gcs.<function>**

B.1 Navigation

The following commands allow script-based control of the visualization process. They are modeled according to the GUI's navigation panel and provide the same functionality. Developers are enabled to use this methods for debugging issues like setting breakpoints.

Type	**Function**	**Description**
void	pause()	Stops the visualization process equivalent to the *Pause* button. If the visualization is already stopped, the action has no effects.
void	play()	Starts respectively resumes the visualization process equivalent to the *Play* button. If the visualization is already running, the action has no effects.
void	fwd()	Navigate one single event forward equivalent to the *Forward* button. Only possible if visualization is paused.
void	bwd()	Navigate one single event backward equivalent to the *Backward* button. Only possible if visualization is paused.
void	setCurEvent(int eventIndex)	Set the current event to a specific index equivalent to the *Event Slider*. Only possible if visualization is paused and *eventIndex* lies within valid range.

Table B.1: API - Navigation

B.2 Spaces and streams

This part of the API covers space and stream handling tasks. The functions allow access to the stream data, creation and manipulation of virtual spaces and streams, as well as their integration into the main frame.

Type	**Function**	**Description**
const int	PRES_PLAIN	Present a tile as it is.
const int	PRES_PLUS	Present a tile as maximum if its value exceeds the maximum for the stream.
const int	PRES_MAX_VAR	Max. value from iteration over stream.
const int	PRES_PERCENT	Present as percentage of a fixed maximum.
const int	PRES_PERCENT_VAR	Present as percentage but use the value of corresponding tile in stream *maxStreamIndex* for its maximum.
const int	PRES_ENUM	Select presentation from *enumnames*.
TraceData	getData()	Returns the current trace data.
int	getVal(int spaceIndex, int streamIndex, int tileIndex)	Returns the value of a specified tile in a stream or -1 if not found.
Space	getSpace(int spaceIndex)	Gets space with given index or *null* if not found.
Space	getSpace(String spaceName)	Returns the space with given name.
Stream	getStream(int spaceIndex, int streamIndex)	Returns the stream with given index in a space or *null* if not found.
Stream	getStream(int spaceIndex, String streamName)	Returns the stream with given name in a space or *null* if not found.
Space	createSpace(String name, int tileNum, int streamNum, String title, String blockInfo)	Returns a new virtual space object with provided properties. User has to define a name for the space, the number of tiles, the number of streams, a title which is used as prefix for the tile index (e.g. Region 122) and an optional block information (e.g. Block Size: 64k).
Stream	createStream(Space space, String name, int minVal, int maxVal, int zeroVal, int defaultVal, String pre, String post, int presentation, int maxStreamIndex, Color color, String[] enumNames)	Returns a new stream object with provided properties. User has to define the target space, a name for the stream, value range and default value, optional pre- and postfix for the streams string representation, the presentation style (PRES_*), the, maxStreamIndex for PRES_PERCENT_VAR, the tile color and a list of names for PRES_ENUM.
void	setStreamData(Stream stream, int[] data)	Updates the stream with the provided data.
void	addSpace(Space space)	Adds the given space to the main frame.
void	removeSpace(Space space)	Removes given space form the main frame.

Table B.2: API - Spaces and streams

B.3 Markers and cursor

This commands provide access to the cursor and also the markers for every space. The cursor can used to point on the actual tile of interest during execution of breakpoint scripts so that the *Properties* window always displays accordant information automatically. The markers can be utilized to highlight arbitrary tiles in every native and virtual space and also support individual marker colors.

Type	**Function**	**Description**
boolean	isMarked(int spaceIndex, int tileIndex)	Returns if a specific tile in a space is marked.
void	setMarked(int spaceIndex, int tileIndex)	Marks one specific tile inside a space given its index.
void	setMarked(int spaceIndex, int tileIndexStart, int tileIndexEnd)	Marks multiple tiles inside a space given start and end index.
void	clearMarkers(int spaceIndex)	Clears all markers inside a space equivalent to the *Clear Markers* button.
void	setMarkerColor(Color color, int spaceIndex, int tileIndex)	Sets the marker color for a specific tile.
void	resetMarkerColors(int spaceIndex)	Resets all markers in a space to default color.
int	getCursor(int spaceIndex)	Returns the cursors current position or -1 if cursor is not set in the given space.
void	setCursor(int spaceIndex, int tileIndex)	Moves the cursor to a specific tile in a space.
void	clearCursor(int spaceIndex)	Removes the cursor form a space.

Table B.3: API - Markers and cursor

B.4 Control information

The following functions cover the utilization of control information in native and virtual spaces. The interface provides access to control information for single tiles or entire spaces. The different control types are encoded in a control byte represented by five constants defined in the API.

Type	Function / Constant	Description
const byte	CTRL_USED	Used tiles represent busy memory areas and display accordant detail information.
const byte	CTRL_BACKGROUND	Background tiles appear as gray boxes and does not show any information.
const byte	CTRL_UNUSED	Unused tiles appear as gray boxes and does not show any information.
const byte	CTRL_SEPARATOR	Draws a separator between a tile and its left neighbor.
const byte	CTRL_LINK	Draws a link between a tile and its right neighbor.
byte[]	getControl(int spaceIndex)	Returns all control information of a space as a byte array.
byte	getControl(int spaceIndex, int tileIndex)	Returns the control information for a specific tile in a space.
void	setControl(int spaceIndex, byte[] control)	Sets the control information for all tiles in a space.
void	setControl(int spaceIndex, int tileIndex, byte control)	Sets the control information for a specific tile in a space.

Table B.4: API - Control information

B.5 Event information

This commands provide access to the event information for the actual space, as well as the *Event Counters* accordant to the accordant utility in the GUI.

Type	Function	Description
String[]	getEventNames()	Returns the event names for the actual trace data as a string array.
String	getEventName(int eventIndex)	Returns the name of a specific event type.
int	getEventNum()	Returns the number of different event types for the actual trace data.
int[]	getEventCounts()	Returns the number of events for every type received since connection as an integer array.

Table B.5: API - Event information

B.6 Event filters

This are the commands for event filter access and manipulation according to the *Event Filters* dialog in the GUI. Developers are enabled to control the data transmission behavior of the server module.

Type	**Function**	**Description**
boolean	efGetEnabled(int eventIndex)	Returns if a specific event type is enabled in the event filters. Disabled events are ignored by the server.
void	efSetEnabled(int eventIndex, boolean enabled)	Enables or disables a specific type of event.
boolean[]	efGetEnabled()	Returns enabled settings for all event types.
void	efSetEnabled(boolean[] enabled)	Sets enabled settings for all event types.
void	efEnableAll()	Enables all event types for transmission.
void	efDisableAll()	Disables all event types for transmission.
int	efGetDelay(int eventIndex)	Returns delay settings for a specific event type. Server delays for given time in ms after transmissions of this events.
void	efSetDelay(int eventIndex, int delay)	Sets the delay for a specific type of event.
int[]	efGetDelays()	Returns the delay settings for all different event types.
void	efSetDelays(int[] delays)	Sets the delay settings for all different event types.
void	efClearDelays()	Sets the delay for all event types to zero.
boolean	efGetPause(int eventIndex)	Returns the pause option for a specific type of event in the event filters. Server pauses after transmission of this events.
void	efSetPause(int eventIndex, boolean pause)	Sets the pause option for a specific event.
boolean[]	efGetPauses()	Returns pause settings for all event types.
void	efSetPauses(boolean[] pauses)	Sets the pause settings for all event types.
void	efClearPauses()	Deletes pause settings for all event types
int	efGetPeriod(int eventIndex)	Returns the actual period settings for a specific type of event in the event filters. Transmission of this event types happens only every n times.
void	efSetPeriod(int eventIndex, int period)	Sets the period for a specific type of event.
int[]	efGetPeriods()	Returns period settings for all event types.
void	efSetPeriods(int[] periods)	Sets the period settings for all event types.
void	efResetPeriods()	Resets period for all event types to 1.

Table B.6: API - Event filters

B.7 Utilities

In addition to the main functionality the API provides some commands which allow access to the *Script Console* and assist during script implementation.

Type	**Function**	**Description**
void	print(int val)	Print an integer value on the *Script Console*.
void	print(string s)	Print a string value on the *Script Console*.
int	nameToIndex(Stream stream, String enumName)	Returns the index for a specific enumeration name in a given stream or -1 if not found.

Table B.7: API - Utilities

List of Figures

List of Tables

Bibliography

[Aftandilia10] E. Aftandilian, S. Kelley, C. Gramazio, N. Ricci, S. Su, S. Guyer, *Heapviz: Interactive Heap Visualization for Program Understanding and Debugging*, ACM Symposium on Software Visualization 2010 (SOFTVIS10), Salt Lake City, USA.

[Eclipse] IBM Developer Works, *Debugging with the Eclipse Platform*, `http://www.ibm.com/developerworks/library/os-ecbug/`, Retrieved on December 17, 2010.

[Hofer10] P. Hofer, *Visualization of Heaps with GCSpy*, Bachelor thesis, JKU Linz, Austria, to be published.

[HPROF] Sun Developer Network, *HPROF: A Heap/CPU Profiling Tool in J2SE 5.0*, `http://java.sun.com/developer/technicalArticles/Programming/HPROF.html`, Retrieved on December 17, 2010.

[Jython] The Jython Project, *Jython: Python for the Java Platform*, `http://jython.org`, Retrieved on December 17, 2010.

[Logging] Python Documentation, *logging - Logging facility for Python*, `http://docs.python.org/py3k/library/logging.html`, Retrieved on December 17, 2010.

[Printezis01] T. Printezis, R. Jones, *Visualizing The Train Garbage Collector*, Proceedings of the 2002 International Symposium on Memory Management (ISMM02), Berlin, Germany.

[Printezis02] T. Printezis, R. Jones, *GCSpy: An Adaptable Heap Visualization Framework*, Proceedings of the 17th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications 2002 (OOPSLA02), Seattle, USA.

[Singh07] V. Singh, S. Ranu, *Extending GCSpy for JikesRVM*, Technical Report CMPCS 263, Department of Computer Science, University of California, Santa Barbara, USA.

List of abbreviations

API	Application Programming Interface
CPU	Central Processing Unit
GC	Garbage Collection
GUI	Graphical User Interface
IDE	Integrated Development Environment
IP	Internet Protocol
I/O	Input/Output
JVM	Java Virtual Machine
J2SE	Java 2 Standard Edition
OS	Operating System
TCP	Transport Control Protocol
UI	User Interface
UML	Unified Modeling Language
VM	Virtual Machine

Curriculum vitae

Personal data

Name	Benjamin Dallinger
Date of birth	March 18, 1983
Family status	Single
Citizenship	Austria
Parents	Evelyn Dallinger Norbert Wüstner
Siblings	None

Education

1989-1993	Primary school, Nußdorf am Attersee
1993-1997	Secondary school, BRG Vöcklabruck
1997-2002	Technical school for mechanical engineering, HTL Vöcklabuck
June 2002	Final examination passed with distinction
2002-2003	Compulsory military service
2003-2007	Employee in mechanical engineering department
March 2007	Higher education entrance qualification
Since 2007	Study of computer science, JKU Linz

Eidesstattliche Erklärung

Ich erkläre hiermit an Eides statt, dass ich die vorliegende Arbeit selbstständig und ohne fremde Hilfe verfasst, andere als die angegebenen Quellen nicht benützt und die den benutzten Quellen wörtlich oder inhaltlich entnommenen Stellen als solche kenntlich gemacht habe.

Linz, am 25.01.2011 Benjamin Dallinger