GCSpy Heap visualization with scripting

GCSpy [1] is a generic heap visualization framework for collection and replay of memory management behavior. With only a few changes to the system in which it is incorporated it can visualize various information in a number of ways similar to an off-line debugger.

GCSpy has already been successfully incorporated into many existing environments (e.g. Hotspot JVM [2], Jikes RVM [3] and others) showing its adequateness for a wide range of applications.

A previous bachelor project focused on bringing the code and the visualization up to current standards. Standard 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. The image below shows a typical situation during inspecting a trace:

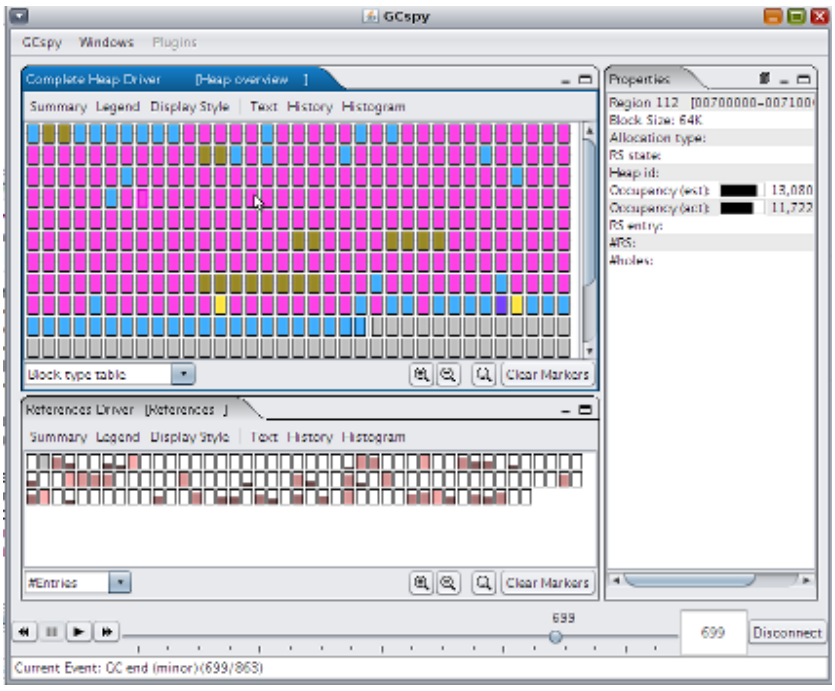

Figure 1: GCSpy visualizer

The main task of this bachelor's 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 (ie. “debug”) the traced data.

The task is not to implement (i.e. hardcode) these features in the existing Java code base, but provide access to the existing trace data and GCSpy functionality to some existing scripting engine, and allow execution of these small scripts at specific events for flexibility. These scripts should be hosted in a scripting engine like Jython [4], JRuby [5], or similar.

The project consists of the following subtasks:

- familiarize yourselves with the current GCSpy code: read existing documentation, build it, run sample traces and maybe implement minor improvements. Also determine the types of data currently available to the visualizer and investigate the visualization process.
- compare with existing tools (e.g. debugger or better some tracing tools)
  - summarize their functionality which is interesting for a tracing tool
  - think about if and how particular functionality may be applied to an off-line tracing tool like the GCSpy visualizer using the possibilites of this project.
- determine affected components and changes
  - determine where in the existing (cyclic) visualization process scripts may be applied to what effect, and what data needs to be exposed to them to achieve the desired functionality
  - also define the user use-cases on a high-level view, i.e. describe how the user will be able to add scripts (where?) to achieve what functionality and determine the required changes
- prototyping/implementing changes in the GCSpy visualizer
  - implement necessary UI related changes
  - host the scripting engine of choice in the visualizer.
  - prototype various types of scripting APIs which will be used by the user and select one. Try to make the API as easy to use as possible.
  - if not already present, define interfaces between Java code and the scripting engine for the data and functionality that needs to be exposed
  - implement any required changes

The expected goals for this thesis are that there are example scripts that

- can control the tracing itself, for example automatically pause the tracing if certain conditions in the data are met (e.g. “pause if the total memory consumption is > 80%”)
- augment existing views with new (calculated) data values (e.g. in a generational garbage collector setting, display eden and survivor areas as belonging to a single young generation)
- create custom views of the traced data. E.g. create a new view where each block represents an aggregate of either eden, survivor or old generation memory areas with some specific data values, which is kept current every time new data arrives.
- […] insert your favourite features showing off your code here ...]

Note that all the functionality described above is already available, the only effort required is to be able to glue them together using scripts easily and execute them using a scripting engine.

Basic experience in working with source control systems is useful. In particular, Mercurial[6] will be used for versioning. Basic knowledge and interest in memory management is recommended.

Programming language: Java and one of Jython, JRuby or even a different one in agreement with the advisor

Procedure, deliverables:

- detail description of the GCSpy visualization process from raw data to data display. Summary of available data within the GCSpy framework, outline of enhancements made possible by adding a scripting engine (2-4 pages)
- implementation
- after a more intensive beginning phase, short weekly (possibly later bi-weekly) meetings about recent issues, the current progress, and planning for the next time are expected to be held.
- at the end of the bachelor's project a written report of at least 30-40 pages has to be handed in. It should at least contain the following contents: problem description, analysis of the visualization process, comparison with features of other program analysis tools (e.g. debugger, tracing) and how they can be applied, user documentation also including a small introduction in scripting on the basis of a sample script, description of the end-user scripting API, changes to the visualizer, and summary containing ideas for possible further work.
- since the efforts are part of an open source project, the written documentation should be in English.

Advisor: Dipl.-Ing. Thomas Schatzl (thomas.schatzl@jku.at, room HF305 at the SSW institute)

References:

[1] GCSpy homepage: http://www.cs.kent.ac.uk/projects/gc/gcspy/
[2] Hotspot/OpenJDK homepage: http://www.openjdk.org/
[3] Jikes RVM: http://jikesrvm.org/
[4] Jython: http://jython.org
[5] JRuby: http://jruby.org
[6] Mercurial: http://mercurial.selenic.com/wiki/