JavaScript Debugger Using Data Structure Visualization
JavaScriptのデータ構造可視化を用いたデバッガ

東京工業大学大学院
情報理工学研究科数理・計算科学
学籍番号：12M54060
徐駿剣

2013年度修士論文
指導教員 脇田 建 准教授

June 10, 2014

Abstract

Debugger is used to test and debug target programs by stepping through the program and examining the current program state by evaluating the values of expressions or checking the stack trace. However, the character-based expressions evaluation provides limited insight into the current program state. This paper examines how a debugger can provide a higher-level, more informative visualization based on data structure. Except static view, the debugger also allows generating animation while target object is modified. However, high-level visualization typically relies on user augmented source code. This paper enables the debugger to understand the data structure of target object by externally supplied semantic information which is written in a declarative language.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation

Program visualization systems translate program into visual shapes. It is often used in algorithm animation systems where algorithm behavior is visualized by producing the abstraction of the data and the operations of the algorithm. Such visualization also can be used in a debugger, as it allows for better understanding on the behavior of the program and help perceive which parts of the program does not function correctly.

A debugger is used to test and debug target programs. It offers many sophisticated functions such as running a program step by step, pausing the program at some event or specified instruction by means of a breakpoint, and examining the current state by evaluating the values of expressions or checking the stack trace. However, the character-based expressions evaluation provides limited insight into the current program state. Modern IDEs intend to solve this problem by offering a dedicated view to watch expressions. As a typical example of object-oriented programming language, when the variable being watched is of reference type, its fields is listed in a tree-view table. The root of this list represents the value itself. If a field is also of reference type, it can be further expanded in the same manner. Fields of primitive type can not be further expanded.

However, such visualization still has deficiencies. Firstly, all of the objects are visualized in a tree-view table, and hence lack of important semantic information. Secondly, The whole view will be refreshed while stepping through the program. It is nearly impossible to check the modification parts and find the relevance between two succeeding states.

This paper examines how a debugger can visualize data structure and generate smooth animation automatically while according object is modified via externally supplied semantic information. Instead of displaying object in a general way, we wish to produce more helpful and informative visualization.

1.2 Main Contribution

1.3 Outline of the Thesis

Chapter 2

Related Research

2.1 Debugger

2.2 Visual Debugger

Chapter 3

System Proposal

3.1 Declarative language

3.2 Animation Mechanism

To generate valid data structure animation, we must prove the consistency between the visualization and program. A program is composed of a series of instructions which are executed orderly. Under the control of debugger, program is being paused until step requests come. Every time the program pauses, it represents a new program state. What we need to prove is that every time the new program state has been generated, the animation will respond to it correctly.

Object graph details the relationships between objects. We can get one object graph $G = (V, E)$ for a given object at certain program state. The vertex set V_p is defined by objects which refer to that given object directly or indirectly, and the edge set E is defined by reference relationships among those objects.

By iterating the object graph using next actions, we will get a subgraph of it, $G_p = (V_p, E_p)$. Visual nodes are defined based on this graph.

Visual nodes including two types, node and edge, are created by create actions. They are also a graph, $G_v = (V_v, E_v)$. The vertex set is defined by visual nodes of node type. They are related to V_p. The relationships can be defined as a non-injective and surjective function $f : V_p \to V_v$ (***TODO***: illustration). The edge set is defined by visual nodes of edge type.

3.3 Data Structure Visualization and Animation

Chapter 4

System Implementation

4.1 V8 Debugger Protocol

V8 is able to debug the JavaScript code running in it. The debugger related API can be used in two ways, a function based API using JavaScript objects and a message based API using a JSON base protocol. The function based API is for in-process usage, while the message based API is for out-process usage. This system is implemented with message based API. Each protocol packet is defined in JSON format and can to be converted to string.

4.1.1 Protocol Packet Format

All packets have two basic elements called seq and type. The seq field holds the consecutive allocated sequence number of the packet. And type field is a string value representing the packet is request, response or event.

```
{
  "seq" : <number>,
  "type": <type>,
  ...
}
```

A request packet has the following structure.

```
{
  "seq"      : <number>,
  "type"     : "request",
  "command"  : <command>,
  "arguments": ...
}
```

A response packet has the following structure. If command failed, the success field will be set as false and message field will contain an error message.

```
{
  "seq"         : <number>,
  "type"        : "response",
  "request_seq": <number>,
  "command"     : <command>,
  "body"        : ...,
  "running"     : <is the VM running after sending the message>,
  "success"     : <boolean indicating success>,
  "message"     : <error message>
}
```

An event packet has the following structure.

```
{
  "seq"  : <number>,
  "type" : "event",
  "event": <event name>,
  "body" : ...
}
```

4.1.2 V8 Debugger Protocol Features

V8 debugger provides various commands and events for us to better understand runtime situation. However, this research focus on data structure visualization. Only following features are used in order to implement basic debugger features.

- **Request** continue
- **Request** evaluate
- **Request** lookup
- **Request** source
- **Request** setbreakpoint
- **Request** clearbreakpoint
- **Event** break
- **Event** exception

The request continue makes V8 start running. It also can be used to make V8 step forward, including step in, step over, and step out. The request evaluate is used to evaluate a expression with depth one. If the result is object type and still contains object type field, the object handle will be returned. Here request lookup can be used to lookup objects based on their handle. By recursively using lookup, we can get the deep copy of the target object. Request source is used to retrieve source code for a frame by indicating the frame and range. Each script running on Node.js will be wrapped within a wrapper function. We have to remove the header and tail of Node.js wrapper manually. Request setbreakpoint is used to add breakpoint. Target script and line number have to be indicated in the request. And clearbreakpoint is used to remove breakpoint set by request setbreakpoint. Breakpoint number required from request setbreakpoint have to be indicated in the request. There also exists other kinds of requests like backtrace to require stacktrace from the current execution state, and frame, scope to require related information.

4.1.3 Use Cases of V8 Debugger Protocol

4.2 System Architecture

4.3 Asynchronized Communication Model

4.4 Declarative Language Parser

4.5 Visualization Generation and Updating

Chapter 5

Examples

5.1 Quick Sort

5.2 Math Expression Tree

5.3 AVL Tree

Chapter 6

Summary

Bibliography