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 23, 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

To solve the problems mentioned in last chapter, our system used traversal-based method proposed by [5] to traverse object to design a new declarative language oriented for JavaScript. On top of this, we proposed a mapping mechanism to generate the mapping relationship from object graph to visual nodes. Making use of the mapping relationship, the system is able to generate visualization and animate it automatically while stepping through program. Instead of providing rich kinds of data structure visualization, our research focuses on how users can display and animate the objects they are interested in handily and how the system works in the background.

3.1 From Object to Visual Shapes

To generate valid data structure visualization and animation, we must at first prove the consistency between the program and visualization. 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 need to be proved is that the initial program state is being correctly visualized and every time the new program state is generated, the animation will respond to it correctly. However, program state contains too much information. A computer program stores data in variables, which represent storage locations in the computer's memory. Program state contains all contents of these memory locations. In contrast, users always have limited interest and perception at a time. We intend to help users understand the program from any angel he is interested in. Instead of visualizing the whole program state like Heapviz [1], this system always focuses on visualizing one object but allows switching targets at any time.

3.1.1 Variable Name And Object

In JavaScript, a variable is a storage location and an associated symbolic name which contains a value. JavaScript: The Definitive Guide [3] introduces about data types in JavaScript. JavaScript allows you to work with three primitive data types: numbers, strings of text, and boolean truth values. JavaScript also defines two trivial data types, null and undefined, each of which defines only a single value.

In addition to these primitive types, JavaScript also supports a composite data type known as object. Just like other object-oriented programming languages, an object in JavaScript is composed of a collection of values with either primitive values or objects. An object can represent an unordered collection of named values or an ordered collection of numbered values. The latter case is called an array. Although array is also an object, it behaves quite differently and have to be considered specially throughout the paper.

JavaScript also defines a few other specialized kinds of objects. **Function** is a subprogram that can be called externally or internally in case of recursion. **Date** creates a object that represents a single moment in time. **RegExp** creates a object that represents a regular expression for matching text with a pattern. **Error** creates a object that represent syntax or runtime errors that can occur in a JavaScript program. Because this research focuses on user-customized data structures, these four types of objects have their own specific data structures hence will not be considered any specially.

We selected object as the target to be visualized. This seems trivially different from previous research like prestigious data structure visualization system, jGRASP [2] that always uses variable name as the target of visualization. Although there is no problem using variable name previously because they are all about static visualization. Our research introduces animation hence the problem has tremendously changed. The fundamental difference is whether there exists substantive relationship between two consecutive states. Animation is visualizing the changes between two consecutive states, so it have to proceed on the former state. That is why variable name can not be used here as the target of visualization in that even the variables with the same name may refer to different objects or go out of scope after stepping through the program. Both situations may lead to meaningless animation because original mapping relationship can not be adapted for new object with different structure. However, variable name still need to be used as an entry to start watching certain object in that users' interest focus on text code rather than memory content.

3.1.2 Mapping Mechanism

Object graph details the relationships of 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 $f : V_p \to V_v$. The edge set is defined by visual nodes of edge type.

3.1.3 Animation Semantics

3.2 Declarative language

The declarative language is used to generate visual

The principles that were considered when we designed this system are as follows:

1. **Expression Ability**
2. **Flexibility**

3.2.1 Data Model

Although this system is constructed based on JavaScript, it theoretically suits all object-oriented programming languages like C++ and Java and any other programming language in which an object is constructed in a recursive way, which means an object is composed of other primitive type values or objects. Hence the object can be traversed and matching actions can be executed to generate the topology of visual nodes.

3.2.2 Patterns

A pattern aligns a series of predicates to be matched when object comes. The first action whose predicate is matched will be executed.

3.2.3 Actions

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 JSON based 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. The 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 assigned sequence number of the packet. And type field is a string value representing the packet is request, response or event. Each request will receive a response with the same request seq number as long as the connection still works. And additional events will be generated on account of particular requests or system errors. Each packet has the following structure.

```
{
  "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 fails, 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 has various commands and events providing detailed runtime information. However, this research focus on the 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

Request "continue" makes V8 start running or stepping forward, including stepping in, stepping over, and stepping out. Although step count can be indicated in the arguments, we always set it as 1.

Request "evaluate" is used to evaluate a expression. However, if the result is object type that contains other fields, all fields will be represented as their object handle. Hence we have to use request "lookup" to lookup objects based on their handle. As a result, we can get the deep copy of any object by recursively using request "lookup".

Request "source" is used to retrieve source code for a frame. Frame and code range have to be indicated in the arguments. Note here that each script file running on node.js is wrapped within a wrapper function. Hence we have to remove the header and tail before showing it to users.

Request "setbreakpoint" is used to add breakpoint. Target file/function and line number are essential here. Request "clearbreakpoint" is used to remove breakpoint set by request "setbreakpoint". Breakpoint number which can be received from request "setbreakpoint" has to be indicated in the arguments. There also exists other kinds of requests like request "backtrace" which is used to require stacktrace information, request "frame" which is used to require frame information and so on.

4.1.3 Response Object Serialization

As discussed in 4.1.2, request "evaluate" and "lookup" may contain objects as part of the body. All objects are assigned with an ID called handle. Object identity[4] is that property of an object which distinguishes each object from all others. Although the handle can be used to identify objects here, it has a certain lifetime after which it will no longer refer to the same object. The lifetime of handles are recycled for each debug event.

For objects serialized they all contains two basic elements, handle and type. Each object has following the structure.

```
{
  "handle": <number>,
  "type"  : <"undefined", "null", "boolean", "number",
             "string", "object", "function">,
  ...
}
```

For primitive JavaScript types, the value is part of the result.

- 0 →

  ```
  {
    "handle": <number>,
    "type"  : "number",
    "value" : 0
  }
  ```

- "hello" →

```
{
  "handle": <number>,
  "type"  : "string",
  "value" : "hello"
}
```

- true →

```
{
  "handle": <number>,
  "type"  : "boolean",
  "value" : true
}
```

- null →

```
{
  "handle": <number>,
  "type"  : "null",
}
```

- undefined →

```
{
  "handle": <number>,
  "type"  : "undefined",
}
```

An object is encoded with additional information.

{a:1,b:2} →

```
{
  "className"           : "Object"
  "constructorFunction": { "ref": <number> },
  "handle"              : <number>,
  "properties"          : [{ "ref": <number> }, ...],
  "protoObject"         : { "ref": <number> },
  "prototypeObject"     : { "ref": <number> },
  "text"                : "#<Object>",
  "type"                : "object"
}
```

An function is encoded as an object with additional information in the properties name, inferredName, source and script.

function(){} →

```
{
  "handle"              : <number>,
  "type"                : "function",
  "className"           : "Function",
  "constructorFunction": { "ref": <number> },
```

```
  "protoObject"         : { "ref": <number> },
  "prototypeObject"     : { "ref": <number> },
  "name"                : "",
  "inferredName"        : "",
  "source"              : "function(){}",
  "script"              : { "ref": <number> },
  "scriptId"            : <number>,
  "position"            : <number>,
  "line"                : <number>,
  "column"              : <number>,
  "properties"          : [{
                              "name": <string>,
                              "ref" : <number>
                           }, ...]
}
```

4.2 System Architecture

This is a full-stack JavaScript system which consists of three components.

1. The debuggee node.js program is running on V8.
2. Server is also running on node.js.
3. Client is running on browser.

Server side is responsible for starting running debuggee program along with V8 debugger and communicate with it using V8 debugger protocol. It responds to the client and require according information from V8 debugger. When response arrives, this component is also responsible for informing client to update along with required information. Because the communication with V8 debugger is an asynchronized process whereas the system logic is basically synchronized, this component must be able to handle it carefully. Client side provides IDE-like debugging experience with an embedded editor to show the source code of the debuggee program, breakpoint management and series of stepping buttons. It also be responsible for visualization work. Figure4.1 shows the system architecture.

4.2.1 Implementation Technique

Protocol.js ensures data integrity in that some responses may be separated into several chunks. Client.js is provides basic features by encapsulating the communication with V8 debugger. Animator module generates and updates graph for target object with the given script. It is feasible to build different interfaces, like command interface or GUI interface.

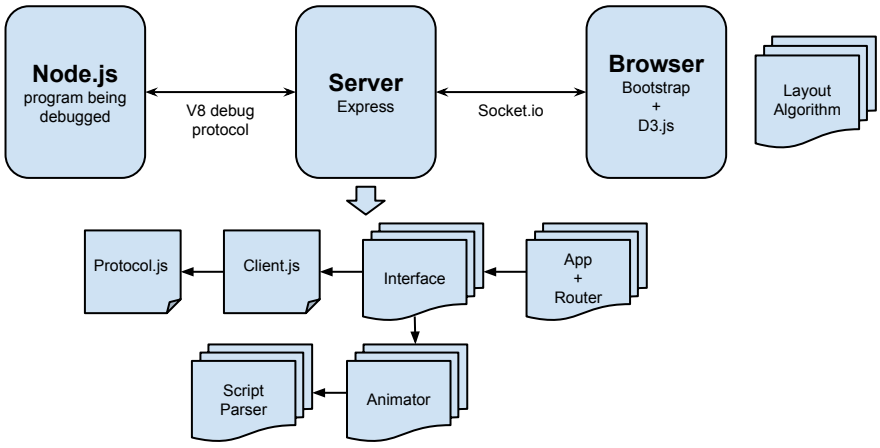

Figure 4.1: System Architecture

On the client side, we used bootstrap for faster and easier web GUI development. Visualization related work is finished by D3.js which is an open-source project about visualizing. D3.js is a JavaScript library for manipulating documents based on data. D3 helps you bring data to life using HTML, SVG and CSS. D3 's emphasizes on combining powerful visualization components with a data-driven approach to DOM manipulation.

We also used other open-source libraries and tools like underscore, async.js, jquery, require.js, buckets, grunt, jasmine to help the development work on both sides.

4.2.2 Asynchronized Communication Mechanism

Another mentionable implementation technique used here is the asynchronized communication mechanism. Most I/O-related API containing TCP/IP communication provided by node.js is in asynchronized way. On the other hand, all requests sent from client via websocket is also in asynchronized way. As a result, it is very complicated to keep program running correctly because the overall control flow across three components is based on synchronization logic. Firstly, blocking message queue is used to handle the requests

from client. All requests are stored in the queue and will be handled one by one. On the server side, Async.js is used to help manage asynchronized code. Although node.js is famous for its speed and single-thread model, its coding style is difficult to maintain on account of endless nested callback. Async.js is a tool to help alleviate this problem.

4.3 Declarative Language Parser

4.4 Visualization Generation and Updating

Chapter 5

Examples

5.1 Quick Sort

5.2 Math Expression Tree

5.3 AVL Tree

Chapter 6

Summary

Bibliography

[1] Edward E. Aftandilian, Sean Kelley, Connor Gramazio, Nathan Ricci, Sara L. Su, and Samuel Z. Guyer. Heapviz: Interactive heap visualization for program understanding and debugging. In *Proceedings of the 5th International Symposium on Software Visualization*, SOFTVIS '10, pages 53–62, New York, NY, USA, 2010. ACM.

[2] James H. Cross, II, T. Dean Hendrix, Jhilmil Jain, and Larry A. Barowski. Dynamic object viewers for data structures. In *Proceedings of the 38th SIGCSE Technical Symposium on Computer Science Education*, SIGCSE '07, pages 4–8, New York, NY, USA, 2007. ACM.

[3] David Flanagan. *JavaScript: The Definitive Guide*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 3rd edition, 1998.

[4] Setrag N. Khoshafian and George P. Copeland. Object identity. *SIGPLAN Not.*, 21(11):406–416, June 1986.

[5] J.L. Korn and A.W. Appel. Traversal-based visualization of data structures. In *Information Visualization, 1998. Proceedings. IEEE Symposium on*, pages 11–18, Oct 1998.