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

June 9, 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 Introduction

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 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.

1.2.1 Visual Nodes

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.

Bibliography