Diplomarbeit

Automatisches Bestimmen fehlerverursachender Programmzustände in Eclipse

12. Februar 2006

Karsten Lehmann
lehmann@st.cs.uni-sb.de

Betreuer: Prof. Dr. Andreas Zeller

Gutachter: Prof. Dr. Andreas Zeller
Prof. Dr. Reinhard Wilhelm

Lehrstuhl für Softwaretechnik
Fachbereich Informatik
Universität des Saarlandes

Zusammenfassung

Betrachtet man einen fehlerhaften Programmlauf als Abfolge mehrerer Programmzustände: Jeder Zustand induziert einen Folgezustand bis zu dem Auftreten des Fehlers. Welche Variablen und Werte von Variablen innerhalb eines Programmzustandes sind für den Fehler relevant? Unter Zuhilfenahme eines funktionierenden Programmlaufes können diese Variablen automatisch ermittelt werden und damit dem Entwickler wertvolle Hinweise auf die Fehlerursache geben.

Dieses Verfahren konnte bisher mit Erfolg auf Programmzustände von C-Programmen angewendet werden. In dieser Arbeit zeigen wir, wie die bestehenden Algorithmen für Java Programme adaptiert wurden. Das Ergebnis ist DDSTATE, ein *Delta Debugger*, voll integriert in die Java Entwicklungsumgebung Eclipse.

Danksagung

Endlich ist es geschafft! Die Wochen und Monate der Anstrengungen eine erfolgreiche Diplomarbeit zu erarbeiten sind vorbei. Jetzt ist es an der Zeit, den aktuellen Stand der Ergebnisse darzulegen. Zu dem Gelingen der Arbeit haben viele Menschen beigetragen, denen ich sehr zum Dank verpflichtet bin. Durch sie ist es erst möglich gewesen, die Thematik der Arbeit in die Realität umzusetzen.

Allen voran möchte ich meinem Betreuer Prof. Dr. Andreas Zeller für das interessante Thema danken und sein Vertrauen in mich, die Aufgabenstellung dieser Arbeit umzusetzen. Seine Anregungen halfen mir stets technische Probleme zu überwinden. Ebenso einen recht herzlichen Dank an den Zweitgutachter Prof. Dr. Reinhard Wilhelm.

Ein Dankeschön geht an alle Mitarbeiter des Lehrstuhls für Softwaretechnik der Universität des Saarlandes. Thomas Zimmermann war immerzu hilfsbereit, wenn ich Probleme hatte. Die Diskussionen mit ihm unterstützten mich einen Weg der Problemlösung zu finden. Vielen Dank auch an Holger Cleve, der durch sein umfassendes Wissen im Bereich Delta Debugging mir immer wieder bei konzeptionellen Schwierigkeiten zur Seite stand. Durch die Unterstützung im Bereich Byte Code Engeneering zur Modifikation von Java Bytecode half mir Valentin Dallmeier einige grundlegende Probleme dieser Arbeit zu lösen. Nicht zu vergessen Stephan Neuhaus und Christian Lindig, die immer ein offenes Ohr für mich und guten Rat auf meine Fragen hatten.

Während des Studiums lernt man viele Menschen kennen. Zusammen mit meinem Kommilitone und Freund Martin Burger habe ich viele Partys gefeiert, aber sicherlich auch genauso hart gearbeitet. Vielen Dank für die Freundschaft und die schöne Zeit miteinander. Für die wertvollen Hinweise und Verbesserungsvorschläge zur schriftlichen Ausarbeitung bedanke ich mich bei Jörg Brück, Holger Cleve, Valentin Dallmeier, Roland Fischer, Martin Mehlmann, David Schuler und Thomas Zimmermann. Nicht zuletzt möchte ich mich bei all meinen Freunden bedanken, die während meines Studiums mir immer zur Seite standen.

Für die Unterstützung und Hilfe während meines schulischen Werdegangs sowie des Studiums, besonders innerhalb der letzten Monate, möchte ich ein ganz großes Dankeschön an meine Familie aussprechen. Meine Eltern gaben mir von Anfang an, die Freiheit und Möglichkeiten meine persönlichen Ziele und Träume zu verfolgen. Meine Geschwister Elke und Ursula unterstützten mich in allen schulischen Entscheidungen und waren stets für mich da, wenn ich einmal verzweifelte oder nicht mehr weiter wusste. Sie gaben mir Halt und Kraft in schwierigen Zeiten, und immer wieder den Mut meine Ziele weiter vor Augen zu behalten. Voller Stolz schaue ich heute auf die Ergebnisse dieser Arbeit und kann nur sagen – ich bin froh, dass es euch gibt.

Inhaltsverzeichnis

1 Einleitung

Softwarefehler, auch Bugs genannt, kosten nach einer Studie des Commerce's National Institute of Standards and Technology (NIST) die US-Amerikanische Wirtschaft jährlich 59,5 Milliarden US-Dollar. Diese Kosten teilen sich größtenteils in Kosten, die durch etwaige Betriebsausfälle entstehen und Kosten, die zur Fehlerbehebung notwendig sind.

Die Softwaretechnik klassifiziert Fehler in Programmen nach folgenden Kategorien (Wikipedia, 2005):

Syntaxfehler sind Verstöße gegen die grammatischen Regeln der benutzten Programmiersprache. Einen Syntaxfehler verhindert bei einigen Programmiersprachen bereits die Kompilierung des fehlerhaften Programms.

Laufzeitfehler (Bugs) sind alle Arten von Fehlern, die auftreten während das Programm abgearbeitet wird. Da diese Fehler die Programmlogik und damit die Bedeutung des Programmcodes betreffen, spricht man hier auch von *semantischen Fehlern*.

Designfehler sind Fehler im Grundkonzept, entweder bei der Definition der Anforderungen an die Software, oder bei der Entwicklung des Softwaredesigns, auf dessen Grundlage das Programm entwickelt wird.

Regressionsfehler sind Fehler, die in einer früheren Programmversion bereits behoben wurden, aber in einer späteren Programmversion wieder auftauchen.

In der folgenden Ausarbeitung beschäftigen wir uns speziell mit Laufzeitfehlern. Böhm hat in seinen Studien festgestellt, dass, je früher man in dem Entwicklungs- und Lebenszyklus einer Software einen Fehler entdeckt, umso geringer sind die entstehenden Kosten für diesen Fehler (Boehm, 1981). Damit wir semantische Fehler in der Programmausführung erkennen, müssen wir das Programm testen.

> Testing is the process of executing a program with the intent of producing some problem (Zeller, 2005, Kapitel 3).

Durch frühzeitiges Testen sind wir in der Lage Fehler schon während der Implementierungsphase aufzudecken und somit Kosten zu sparen. Jeder Fehler, der erst bei Produktivität der Software festgestellt wird, ist sehr kostspielig. Eventuell muss die Software erneut validiert, Akzeptanztests durchlaufen und ein Patch erstellt werden. Dieser Zyklus kann sehr zeitaufwendig werden. Die Folge sind unkalkulierbare Kosten und Unzufriedenheit bei Hersteller und Kunden.

Umso wichtiger ist es, die verschiedenen Komponenten der Software ausgiebig zu testen. Doch was geschieht, wenn wir durch einen Test einen Fehler gefunden haben? Die Programmierer müssen sich auf die Suche nach der *Ursache* des Fehlers begeben (engl. Debugging). Dies bedeutet, wir suchen die Codestelle(n), die für den Fehler verantwortlich sind. Diese Stellen im Quellcode werden *Defekte* genannt. Wird der Defekt in einem Programmlauf ausgeführt, kann ein ungewünschter Programmzustand (*Infektion*) entstehen. Dieser infizierte Zustand propagiert sich weiter, bis es zu einem für den Anwender sichtbaren *Fehler* kommt.

Wenn wir zwei Testläufe haben, einen fehlerhaften und einen funktionierenden, können wir den Programmierer bei seiner Fehlersuche unterstützen. Unter Anwendung eines automatischen Verfahrens zur Fehlersuche, namens Delta Debugging, sind wir in der Lage die fehlerrelevanten Ursachen des fehlerhaften Programmlaufs automatisch zu bestimmen. Dies geschieht praktisch per Knopfdruck und kann uns Zeit und die damit verbundenen Kosten ersparen. Ziel dieser Arbeit ist die Entwicklung von DDSTATE – ein Werkzeug, integriert in die Entwicklungsumgebung Eclipse, zum automatischen Bestimmen fehlerverursachender Programmzustände von Java-Programmen.

Zur Überprüfung der Korrektheit eines Programmlaufs verwenden wir *Komponententests*.

1.1 Komponententests

Jedes nichttriviale Programm ist in eine Zahl von unabhängigen *Komponenten* (engl. Units) aufgeteilt (Zeller, 2005, Kapitel 3). Dies sind z.B. Unterprogramme, Funktionen, Libraries, Module, etc. Obwohl Komponenten eines der ältesten Programmierkonzepte darstellen, wurde für viele das automatisierte Testen auf Komponentenebene erst in den letzten Jahren interessant. Vor allem bei dem *Extreme Programming* (XP), der bekannteste Prozess aus der Klasse der agilen Prozesse der Softwareentwicklung, werden *Komponententests* (engl. *Unit-Tests*) eingesetzt.

Mit Hilfe von Komponententests sind wir in der Lage kleine Programmbereiche auf deren semantische Richtigkeit zu prüfen. Komponententests werden in der Implementierungsphase erstellt und können auch schon vor der zur testenden Komponente bestehen – dies ist eines der Prinzipien von Extreme Programming. Man spricht dann von testgetriebener Software-Entwicklung (engl. Test Driven Software Development).

Heutzutage gibt es für die meisten Programmiersprachen Testframeworks, um Komponententests zu schreiben. *JUnit* ist ein Komponententest-Framework, welches es uns ermöglicht, Unit-Tests für Java zu schreiben (JUn, 2005). Basierend auf JUnit gibt es viele verschiedene Erweiterungen, um z.B. Datenbankoperationen, die Korrektheit von XML-Dateien, etc. zu überprüfen.

Für die Anwendung von DDSTATE werden zwei JUnit-Testfälle benötigt – einen der fehlschlägt ($r\text{✘}$) und einen der funktioniert ($r\text{✔}$). Unterschiede im Programmlauf beider Testläufe sind dafür verantwortlich, dass die Tests zu einem unterschiedlichen Ergebnis führen. An vorher definierten Stellen innerhalb der Programmläufe bestimmen wir die Unterschiede der beiden Läufe. Wir

```
1  public class CarConfiguratorTest extends junit.framework.TestCase {
2
3    /** Create an empty car and add all desired accessories */
4    public void testCreateCarBottomUp() {
5      Car myCar = new Car();
6      myCar.setEngine(new Engine(100, false));
7
8      myCar.setNumberOfDoors((short) 4);
9      myCar.addAccesory(AccessoryPool.COLD_WEATHER_PACKAGE);
10     myCar.setExteriorColor(ColorConstants.WHITE);
11     myCar.setInteriorColor(ColorConstants.BLACK);
12
13     junit.framework.Assert.assertTrue(myCar.isConfigurationValid());
14   }
15
16   /** Create a sports car with the provided factory method */
17   public void testCreateSportsCar() {
18     Car myCar = Car.createSportsCar();
19
20     myCar.setNumberOfDoors((short) 2);
21     myCar.setExteriorColor(ColorConstants.RED);
22     myCar.addAccesory(AccessoryPool.ESP);
23     myCar.setInteriorColor(ColorConstants.BLACK);
24     myCar.addAccesory(AccessoryPool.AUTOMATIC_TRANSMISSION);
25
26     junit.framework.Assert.assertTrue(myCar.isConfigurationValid());
27   }
28 }
```

Abbildung 1.1: JUnit-Testfall: Validierung des »Car-Configurator «

werden die Unterschiede untersuchen und automatisch die Teilmenge der Unterschiede berechnen, die für den Fehler in $r_{\times}$ verantwortlich ist.

Beispiel: Car-Configurator

Ein Automobilhersteller stellt seine Software »Car-Configurator« seinen Vertriebspartnern zur Verfügung. Mit Hilfe des Programms sind die Unternehmen in der Lage die Konfiguration eines Autos zu bestimmen und zu validieren. Der Quellcode des Beispiels befindet sich in Anhang A.3. Auch wir möchten die Software nutzen und entwerfen zunächst zwei Komponententests, um die Korrektheit der Funktionalität zu überprüfen (siehe Abbildung 1.1).

Die Testfunktion `testCreateCarBottomUp` erzeugt, unter Verwendung des Standardkonstruk-

tors, ein Instanz vom Typ `Car`. Diesem Objekt fügen wir einen Motor mit einer Leistung von 100 PS hinzu. Danach simulieren wir die Wünsche eines Kunden. Zum Beispiel wählen wir für das Interieur die Farbe Schwarz und fügen als Sonderzubehör das Winterpaket hinzu. Eine Auswahl verschiedener Zubehörpakete findet sich in der Klasse `AccessoryPool` wieder.

Die zweite Testfunktion `testCreateSportsCar` nutzt eine Fabrikmethode um eine Instanz von `Car` zu erzeugen. Wir wollen eine sportliche Variante eines Fahrzeugs zusammenstellen. Deshalb entscheiden wir uns für einen 2-Türer mit roter Außenfarbe. Zusätzlich fügen wir dem Auto als Sonderzubehör ein elektronisches Stabilitätssystem (ESP) und Automatikgangschaltung hinzu.

In beiden Testmethoden überprüfen wir die Korrektheit der Autokonfiguration unter Verwendung der JUnit-Klasse `Assert`. Diese Klasse bietet verschiedene Möglichkeiten Werte und Referenzen auf Korrektheit bzw. Gleichheit zu prüfen. Wenn die Überprüfung fehlschlägt, wird ein `AssertionFailedError` geworfen – der Komponententest ist dann ebenfalls fehlgeschlagen. Um zu überprüfen, ob die Konfiguration an den `Car`-Objekten entsprechend den Anforderungen der Software vorgenommen wurde, rufen wir auf dem jeweiligen `Car`-Objekt die Funktion `isConfigurationValid` auf. Liefert diese `true`, beendet sich der Test ohne Fehler (✔). Wird `false` zurückgegeben schlägt die Testausführung fehl (✘).

Wenn wir beide Tests ausführen, läuft die erste Funktion ohne Fehler. Die Konfiguration des Fahrzeugs wurde somit entsprechend der Anforderung der Software gewählt. Diesen erfolgreichen Testlauf nennen wir $r_{✔}$. Die Ausführung des zweiten Tests schlägt fehl ($r_{✘}$). Die `Assert`-Klasse des JUnit-Frameworks wirft eine `AssertionFailedError`, da die Konfiguration des Fahrzeugs nicht gültig ist.

Es stellt sich die Frage, welche Unterschiede (*Deltas*) in den Konfigurationen der beiden `Car`-Objekte für das Fehlschlagen von $r_{✘}$ verantwortlich sind. Genauer gesagt, interessieren wir uns für die Unterschiede in den *Programmzuständen* beider Testläufe.

1.2 Programmzustände und Speichergraphen

Unter einem Programmzustand versteht man die Menge aller Variablen und die Werte dieser Variablen, die zu einem definierten Zeitpunkt innerhalb der Programmausführung *aktiv sind*. Die Ausführung eines Programms kann als Abfolge von mehreren Zuständen angesehen werden. Dabei induziert ein bestimmter Zustand immer den gleichen Folgezustand. Betrachten wir jeweils einen Zustand aus $r_{✔}$ und $r_{✘}$, sind die Unterschiede zwischen den Zuständen die *Ursache* für den Fehler.

Um die Unterschiede zwischen zwei Programmzuständen zu berechnen, werden wir den Programmzustand als gerichteten Graphen extrahieren. Wie in Zimmermann u. Zeller (2002) beschrieben ist der *Speichergraph* eine geeignete Repräsentation für Programmzustände. Dabei werden die Werte als Knoten, die Variablen als Kanten zwischen den Knoten dargestellt. Der

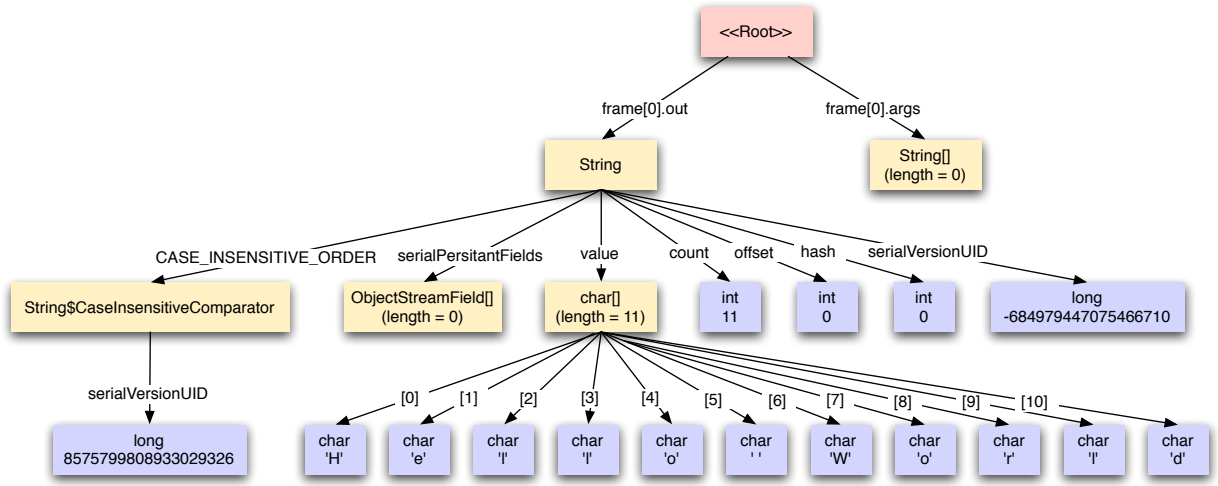


Abbildung 1.2: Speichergraph des Programmzustandes von HelloWorld.java, extrahiert beim Erreichen der Zeile 5 während der Programmausführung.

Speichergraph bietet eine von Speicheradressen abstrahierte Sicht auf die komplette Datenstruktur eines Programms.

Betrachten wir uns den Speichergraphen an einem einfachen Beispiel. Das nachfolgende Java-Programm gibt auf der Konsole `"Hello World"` aus. Den resultierenden Speichergraphen, der den Programmzustand repräsentiert, wenn wir die Programmausführung in der Zeile 5 anhalten, ist in Abbildung 1.2 dargestellt.

```
1 public class HelloWorld {
2
3   public static void main(String[] args) {
4     String out = "Hello World";
5     System.out.println(out);
6   }
7
8 }
```

Beispiel: Car-Configurator (2)

Die Konfiguration des Objekts der Variable `myCar` finden wir, nach dem Setzen der einzelnen Fahrzeugeigenschaften und Zubehörpakete, komplett im Programmspeicher wieder. Durch den Vergleich des Programmzustands $P_{\checkmark}$ (in Zeile 13 der Methode `testCreateCarBottomUp`) mit

dem Programmzustand $P_{\times}$ (in Zeile 26 der Methode `testCreateSportsCar`) können wir alle *Zustandsunterschiede* und somit alle Unterschiede in der Konfiguration der beiden Fahrzeuge identifizieren.

Die Unterschiede stellen die Ursache für den Fehler dar. Jedoch sind nicht alle Unterschiede in der Konfiguration für den Fehler – die ungültige Konfiguration – relevant. Wir gehen davon aus, dass nur eine kleine Teilmenge aller Unterschiede in den Konfigurationen der Fahrzeuge für den Fehler verantwortlich ist. Ziel ist es diese Teilmenge – die *fehlerrelevanten Unterschiede* – zu bestimmen. Dazu verwenden wir *Delta Debugging*.

1.3 Delta Debugging

Delta Debugging automatisiert die wissenschaftliche Methode der Fehlersuche (Zeller, 2005, Kapitel 6). Der Algorithmus isoliert Fehlerursachen, indem er die Menge der fehlerverursachenden Umstände minimiert. Dies geschieht voll automatisch. Wir benötigen lediglich einen automatisierten Test, welcher entscheidet, ob ein Fehlverhalten der ausführenden Anwendung vorliegt.

Es gibt verschiedene Bereiche, in denen die Anwendung von Delta Debugging erfolgreich eingesetzt wird.

- Minimierung fehlerverursachender Eingaben

  Eine der ersten erfolgreichen Anwendungen von Delta Debugging beruht auf der Minimierung fehlerverursachender Eingaben. Denkbare Eingaben sind beispielsweise Konfigurationsdateien von Programmen oder auch HTML-Seiten von einem Web-Browser. Zeller u. Hildebrandt (2002) zeigten, dass die Durchführung von nur 57 Tests genügte, um eine HTML-Seite aus 896 Zeilen auf die Zeile zu reduzieren, die den Mozilla Web-Browser zum Absturz brachte.

- Lokalisierung fehlerverursachender Änderungen bei dem Auftreten von Regressionsfehlern

  Jeder Programmierer kennt sicherlich folgende Situation: »Gestern lief mein Programm, heute nicht mehr. Warum?«. Durch die Anwendung von Delta Debugging können wir die fehlerrelevanten Programmänderungen von Regressionsfehlern bestimmen. In Zeller (1999) werden die Algorithmen beschrieben, um von einer Menge fehlerverursachender Codeänderungen die fehlerrelevanten zu berechnen.

- Fehlerverursachende Änderungen in Programmzuständen von C++-Programmen

  Oftmals genügt es nicht die Eingaben eines Programmlaufs zu untersuchen, um den Grund der fehlerhaften Ausführung zu finden. Der eigentliche Fehler, beruhend auf einem Defekt im Quellcode, manifestiert sich erst während der Programmausführung. Dies geschieht zum Beispiel, indem eine fehlerhafte Berechnung oder ein nicht gewünschter Kontrollfluss abgearbeitet wird. Eine Möglichkeit die Fehlerursache zu lokalisieren besteht, indem

wir Programmzustände innerhalb des fehlerhaften Laufs näher untersuchen. Gegenüber einem funktionierenden Programmlauf muss der fehlerhafte Lauf Unterschiede im Programmzustand aufweisen – sie führen ja schließlich zu einem unterschiedlichen Ergebnis. Diese Unterschiede können wertvolle Hinweise auf die Ursache des Fehlers geben.

Mit ASKIGOR wurde ein öffentlicher Debugging-Server entwickelt, mit dem sich fehlerrelevante Zustandsunterschiede isolieren lassen (Sof, 2005a). Der Server wendet das Verfahren zur Bestimmung der fehlerverursachenden Variablen an mehreren Stellen innerhalb der Programmausführung an. Das Ergebnis ist eine Ursache-Wirkungskette in der Form:

- Zunächst hatte `argc` den Wert `3`,
- danach wurde `a[2] = 0`
- und deshalb enthielt die Ausgabe `"0"` – dies wiederum ist der Grund, warum das Programm fehlschlug.

Beispiel: Car-Configurator (3)

Durch den Delta Debugging-Algorithmus sind wir in der Lage die fehlerverursachenden Unterschiede auf die fehlerrelevanten zu minimieren. DDSTATE implementiert die bisher vorgestellten Techniken und findet so für das Car-Configurator-Beispiel insgesamt drei fehlerrelevante Zustandsunterschiede (siehe Abbildung 1.3):

1. In $r_{\text{✘}}$ wurde gegenüber $r_{\text{✔}}$ der Wert der Variable `elementCount` von 1 auf 2 erhöht.
2. In dem fehlerhaften Lauf zeigt die Objektreferenz des ersten Eintrags des Objektarrays `elementData` auf das Objekt der Referenz `AccesoryPool.ESP`.
3. In dem $r_{\text{✘}}$ zeigt die Objektreferenz des zweiten Eintrags des Arrays `elementData` auf das Objekt der Referenz `AccesoryPool.AUTOMATIC_TRANSMISSION`.

Wie sind diese Informationen zu interpretieren und wo werden die angegebenen Variablen deklariert? Wenn wir uns in der Tabelle den Tooltiptext zu dem dritten Eintrag anschauen, sehen wir, von welchem Objekt die Variable `elementData` referenziert wird: Die Variable `myCar`, welche eine Instanz des Objekts `Car` referenziert, deklariert eine Variable `accessories` und diese wiederum deklariert die Variable `elementData`. Durch die Namensgebung können wir erahnen, dass `accessories` wohl ein Container innerhalb des `Car`-Objekts ist, indem die Zubehörpakete verwaltet werden. Dies erklärt auch den Unterschied der Variable `elementCount`, da in $r_{\text{✔}}$ nur ein während in $r_{\text{✘}}$ zwei Zubehörpakete dem Fahrzeug hinzugefügt wird.

DDSTATE findet *1-minimale* fehlerrelevante Zustandsunterschiede. Dies bedeutet, wenn wir nur eine Änderung, welche DDSTATE als fehlerrelevant berechnet hat, nicht anwenden, tritt der Fehler nicht mehr auf. Auf das Beispiel bezogen heißt das, wenn die Variable `elementCount` nicht

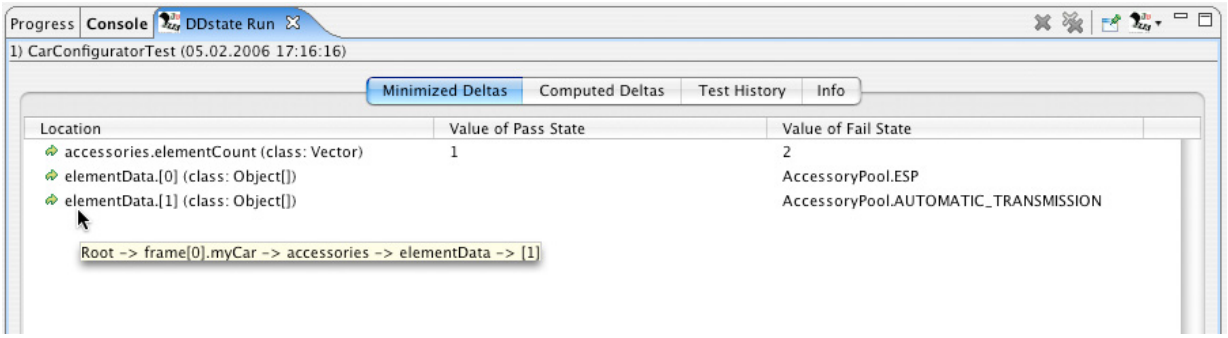


Abbildung 1.3: Minimale fehlerrelevante Menge von Zustandsunterschieden

auf 2 gesetzt oder eines der Zubehörpakete hinzugefügt wird, wäre die Konfiguration nicht ungültig. Eine direkte Folgerung ist, dass die Verwendung der Pakete ESP und automatische Gangschaltung in Kombination nicht erlaubt ist. Um dies zu überprüfen entfernen wir ein beliebiges Zubehör und führen den Test wieder aus. Das Ergebnis ist, dass der Testfall ohne Fehler durchläuft.

DDSTATE ist ein Debugging-Werkzeug, welches als Erweiterung (engl. Plugin) für die Java-Entwicklungsumgebung *Eclipse* verfügbar ist.

1.4 Eclipse

Auf der Internetseite des Open-Source-Projektes Eclipse (IBM, 2005) wird dieses wie folgt beschrieben:

> Eclipse is an open source community whose projects are focused on providing an extensible development platform and application frameworks for building software. Eclipse provides extensible tools and frameworks that span the software development lifecycle, including support for modeling, language development environments for Java, C/C++ and others, testing and performance, business intelligence, rich client applications and embedded development. A large, vibrant ecosystem of major technology vendors, innovative start-ups, universities and research institutions and individuals extend, complement and support the Eclipse Platform.

Eclipse ist ein Plattform für »alles Mögliche und nichts im Besonderen«. Die Java-Entwicklungsumgebung (engl. Integrated Development Environment; kurz: IDE) ist nur ein spezielles Werkzeug, welches auf der Plattform aufsetzt. Diese integriert neben einem Debugger auch das Komponententestframework JUnit.

Eclipse bietet durch seinen integrierten Plugin-Mechanismus eine besonders einfache Art die bereits in Eclipse vorhandene Funktionalität zu erweitern. Diese Funktionalität kann über so

genannte Extensionpoints – dieses sind wohl definierte Schnittstellen der jeweiligen Plugins – auch von eigenen Plugins genutzt und erweitert werden.

Wir verwenden Eclipse, da es JUnit, und somit ein Testframework zur Erstellung automatisierter Tests, unterstützt. Des Weiteren stellt die implementierte Debugger-Architektur eine komfortable Lösung dar, um auf den Zustand eines Programms zuzugreifen und Modifikationen an diesem vorzunehmen. Unter Verwendung der Plugin-Architektur können wir DDSTATE als Erweiterung zu Eclipse anbieten, welches sich über die bestehenden Installations- und Updatemechanismen von Eclipse installieren lässt (siehe Anhang A.1).

1.5 Ziel dieser Arbeit

Die Aufgabenstellung dieser Diplomarbeit war die Entwicklung von DDSTATE – einem Eclipse-Plugin zur Unterstützung der automatischen Fehlersuche auf Programmzuständen von Java-Programmen. Wie wir an dem »Car-Configurator«-Beispiel gesehen haben, können wir durch Identifizierung der Zustandsunterschiede einen Hinweis auf das Fehlschlagen eines Testlaufs bekommen. Ein Unterschied in der Eingabe – in dem Beispiel die Konfiguration eines Fahrzeugs – kann sich durch Berechnungen über mehrere Programmzustände propagieren, bis schließlich, ein für den Anwender sichtbarer Fehler entsteht.

In dem Beispiel dient die Konfiguration eines Fahrzeugs als Eingabe zur Berechnung der Korrektheit der Konfiguration. Die Unterschiede zwischen den Konfigurationen der Fahrzeuge aus beiden Testläufen sind Fehlerursachen. Somit sind auch Unterschiede zwischen den Programmzuständen Fehlerursachen. Jeder Zustandsunterschied wiederum ist eine Auswirkung von früheren Unterschieden. Die Kette aller Unterschiede bildet daher eine Ursache-Wirkungskette über den kompletten Programmlauf. Wenn wir diese Ursache-Wirkungskette berechnen können, haben wir eine solide Basis, die Entstehung des Fehlers nachzuvollziehen (Zeller, 2005, Kapitel 14).

Um die fehlerrelevanten Unterschiede zweier Programmzustände zu bestimmen, müssen wir folgende Probleme lösen:

1. Zugriff auf den Speicher und Abbildung des aktuellen Programmzustands in einem Speichergraphen.

2. Vergleich zweier Speichergraphen, um die Menge aller Graph- bzw. Zustandsunterschiede zu bestimmen.

3. Manipulation eines Programmzustands.

4. Minimierung der berechneten Unterschiede auf die fehlerrelevanten Unterschiede unter Anwendung von Delta Debugging.

Wenn wir die fehlerrelevanten Unterschiede an einer bestimmten Stelle innerhalb der beiden Testläufe berechnet haben sind wir in der Lage eine Ursache-Wirkungskette zu erstellen, indem wir an verschiedenen Stellen innerhalb der Programmausführung die Fehlerursachen berechnen. Da bei der Ausführung eines Testlaufs eventuell tausende oder auch Millionen von Zuständen durchlaufen werden, müssen wir die Stellen finden, an denen wir einen Hinweis auf den Defekt bekommen? Durch die Anwendung des von Cleve u. Zeller (2005) beschriebenen Verfahrens zur Suche nach *Ursache-Übergängen*, werden wir eine Ursache-Wirkungskette erstellen, die uns möglichst genau den Defekt im Quellcode lokalisiert.

1.6 Bisherige Arbeiten

Bei der Vorbereitung und Implementierung von DDSTATE waren einige Vorarbeiten von großer Hilfe. Zu erwähnen ist die Diplomarbeit von Philipp Bouillon (Bouillon, 2004) in der er den Delta Debugging-Algorithmus als Eclipse-Plugin implementiert hat.

Christoph Lauer beschreibt in seiner Diplomarbeit »Speichergraphen für Java Programme« (Lauer, 2004), wie Speichergraphen von Programmzuständen extrahiert und verglichen werden. Des Weiteren untersuchte er die technischen Möglichkeiten bestehende Objekte im Speicher zu manipulieren und neue zu erzeugen.

Cleve u. Zeller (2005) beschreiben in ihrer Arbeit ein Verfahren, um den Defekt im Quellcode mit Hilfe von fehlerverursachenden Zustandsunterschieden zu lokalisieren. Dabei suchen sie nach Ursache-Übergängen (engl. Cause Transitions; kurz: CTS). Dies sind Stellen innerhalb der Programmausführung, an denen einige Variablen aufhören für den Fehler relevant zu sein, während andere Variablen beginnen. Einer Cause Transition liegt immer eine Berechnung zu Grunde. Diese Berechnungen erweisen sich als besonders gute Stellen, um eine Fehlerbehebung (engl. Bugfix) vorzunehmen. Die Auswertung des Verfahrens an der Siemens Test Suite ergab, dass die Cause Transitions einen fehlerverursachenden Defekt zweimal so gut lokalisieren können, als die (zum Zeitpunkt der Auswertung bekannten) besten Verfahren.

1.7 Aufbau dieser Arbeit

Diese Studie liefert keine Einführung in die Verfahren und Algorithmen von Delta Debugging und die erforderlichen Prozesse, um Delta Debugging zur Berechnung von fehlerrelevanten Zustandsunterschieden zu nutzen. Eine ausführliche Behandlung der Themen findet sich in Zeller (2005). Die Ausarbeitung konzentriert sich auf die Umsetzung der verschiedenen Verfahren und Techniken in Java und für Java-Programme.

Der Inhalt der Arbeit gliedert sich nach der Reihenfolge der Verfahren, wie wir sie bei der Berechnung, Übertragung und Minimierung der Zustandsunterschiede anwenden. In Kapitel 2 betrachten wir den Zustand eines Java-Programms. Dabei wird ein Objektmodell erstellt, durch das

der Zustand in Form eines Speichergraphen abgebildet werden kann. Durch den Vergleich zweier Speichergraphen bestimmen wir alle Unterschiede der jeweiligen Programmzustände und somit die Ursachen für den fehlerhaften Testlauf (Kapitel 3). Um die fehlerrelevanten Unterschiede zu bestimmen, werden wir den funktionierenden Programmzustand verändern und überprüfen, wie diese Änderungen sich auf das Testergebnis auswirken. Wie die berechneten Deltas *angewendet* werden, um den Programmzustand zu manipulieren wird in Kapitel 4 beschrieben.

In Kapitel 5 betrachten wir an einem Beispiel, wie wir DDSTATE verwenden, um die fehlerrelevanten Unterschiede zwischen zwei Testläufen zu extrahieren. Ebenfalls exemplarisch zeigen wir, wie DDSTATE durch Anwendung des Cause Transition Algorithmus, die Suche nach dem Defekt im Quellcode unterstützt. In Kapitel 6 werden die Grenzen der Anwendung und bekannte Probleme kurz erläutert. Abschließend kombiniert Kapitel 7 die Ergebnisse der vorherigen Kapitel. In Anhang A.1 befindet sich eine Anleitung zur Installation von DDSTATE mit Hilfe von Eclipse. Anhang A.2 enthält Abbildungen der Speichergraphen zu dem Car-Configurator-Beispiel, dessen Quellcode sich in Anhang A.3 befindet.

2 Extrahieren von Programmzuständen

Der Programmzustand bildet eine Momentaufnahme eines Programmlaufs zu einem bestimmten Zeitpunkt während der Programmausführung. Wir werden später sehen, wie wir den Zustand eines Programms nutzen, um Hinweise auf den Defekt im Programmcode zu bekommen. Doch zuerst müssen wir den Zustand erfassen. In diesem Kapitel untersuchen wir, was ein Programmzustand umfasst. Wir entwickeln ein Objektmodell, um den Zustand als gerichteten Graphen – einen so genannten *Speichergraphen* – abbilden zu können. Wir werden erklären, wie wir den Graphen erfassen, d.h. aus dem Programmzustand *extrahieren*. Ziel ist es, die Struktur von Variablen und Werten innerhalb des Zustands abstrahiert von dem Speicher zu erfassen. Durch diese Abstraktion wird erst ein Vergleich von Programmzuständen aus unterschiedlichen Programmläufen ermöglicht.

2.1 Der Zustand eines Programms

Was ist der Zustand eines Java-Programms? Analysieren wir zunächst die Struktur des Quellcodes, so gliedert sich dieser in verschiedene Pakete. Pakete wiederum enthalten Klassen, welche Attribute, Konstruktoren, Methoden, Konstruktor-Blöcke, statischen Blöcke oder auch innere Klassen definieren. Wenn wir ein Programm ausführen, so werden Klassen durch die Java Virtual Machine (kurz JVM) geladen, Objekte erzeugt und Methoden ausgeführt. Methoden können Parameter und lokale Variablen definieren. Ein Programmzustand umfasst alle Variablen, die während der Programmausführung *aktiv* sind, also deren referenzierte Werte und Objekte sich im Programmspeicher befinden. Dazu gehören alle erzeugten und referenzierten Objekte und deren Attribute. Zusätzlich alle Werte und Objekte lokaler Variablen und Parameter einer Methode, sowie alle Werte und Objekte statisch deklarierter Klassenvariablen.

Ein Programmzustand ist eine Momentaufnahme zu einem gewissen Zeitpunkt der Programmausführung. Da sich der Zustand während der Programmausführung durch Berechnungen fortlaufend ändert, müssen wir den Zeitpunkt, an dem wir den Programmzustand betrachten, definieren. Doch wie können wir dies tun? Betrachten wir die Programmausführung als eine Folge von Maschinencodeinstruktionen so ist der Befehlszähler ein geeignetes Maß. Dieser ist jedoch sehr nah an der Maschinenebene und bei der Ausführung von Java Programmen für den Programmierer oder Anwender nicht ersichtlich.

```
1  public class CarConfiguratorTest extends junit.framework.TestCase {
2
3    /** Create an empty car and add all desired accessories */
4    public void testCreateCarBottomUp() {
5      Car myCar = new Car();
6      myCar.setEngine(new Engine(100, false));
7
8      myCar.setNumberOfDoors((short) 4);
9      myCar.addAccesory(AccessoryPool.COLD_WEATHER_PACKAGE);
10     myCar.setExteriorColor(ColorConstants.WHITE);
11     myCar.setInteriorColor(ColorConstants.BLACK);
12
13     junit.framework.Assert.assertTrue(myCar.isConfigurationValid());
14   }
```

Abbildung 2.1: »Car Configurator«-Beispiel: Erfolgreiche JUnit-Testmethode ($r_{\checkmark}$)

Deshalb suchen wir eine benutzerfreundlichere Möglichkeit einen Programmzeitpunkt zu definieren. Diese sollte näher an dem programmierten Quelltext liegen. Eine Stelle innerhalb des Quelltextes können wird durch Angabe der Klasse und der Zeilennummer innerhalb der Klasse eindeutig definieren. Untersuchen wir den Programmlauf so kann diese Zeile nie, einmal oder auch mehrmals ausgeführt werden. Der Zeitpunkt t, welcher eine Stelle innerhalb der Programmausführung eindeutig bestimmt, definiert sich daher wie folgt:

Definition 1 (Programmzeitpunkt)**.** Eine Stelle bzw. Zeitpunkt $t = (d, z, n)$ innerhalb eines Programmlaufs wird eindeutig bestimmt durch die Angabe von:

- d – Dem voll qualifizierten Namen einer Klasse. Dieser setzt sich aus dem Paket, indem die Klasse definiert wurde, und dem Klassennamen zusammen.
- z – Die Zeilennummer innerhalb der Klasse.
- n – Die Anzahl, wie oft die Zeile z in der Klasse d in einem Programmlauf bis zum Erreichen des Programmzeitpunkts ausgeführt wird.

Wenn wir einen Programmzustand betrachten, so ist dies vereinfacht gesagt der Zustand des Programms, wie man ihn im Speicher des Computers vorfindet. Um allerdings später Unterschiede zwischen den Programmzuständen herauszufinden, wollen wir eine Abstraktionsebene, welche sich auf Variablen und Werte bezieht.

Definition 2 (Programmzustand)**.** Ein Programmzustand P_t beinhaltet alle, zu dem Zeitpunkt t der Programmausführung durch das Programm aktiven Variablen und deren dazugehörigen Werte.

Betrachten wir die Funktion `testCreateCarBottomUp` von dem »Car Configurator«-Beispiel. Angenommen wir wollen den Programmzustand P_t mit $t = (CarConfiguratorTest, 13, 1)$ untersuchen (siehe Abbildung 2.1), d.h. den Zustand, wenn wir die Programmausführung in der Klasse `CarConfiguratorTest` direkt vor dem ersten Ausführen der Zeile 13 anhalten. Dazu müssen wir alle, zu dem Zeitpunkt durch das Programm aktiven Variablen und deren Werte und Objekte erfassen. Dies ist beispielsweise die lokale Variable `myCar` der Funktion `testCreateCarBottomUp`, sowie alle Variablen die von `myCar` referenziert werden. Eine vollständige tabellarische Auflistung, der im Speicher existenten und referenzierten Variablen, finden wir in Abbildung 2.2.

Wir können feststellen, dass für ein relativ kleines Beispiel sehr viele Variablen im Speicher angelegt werden. Dabei wurde in diesem Beispiel darauf verzichtet die Stringobjekte in ihre Attribute und referenzierten Objekte aufzusplitten (siehe Abschnitt 2.7).

Ein Nachteil der tabellarischen Auflistung ist, dass wir nur schwer erkennen können, ob Variablen auf das gleiche Objekt im Speicher zeigen. So verweist z.B. die Wagenfarbe des Autos `exteriorColor` auf die, in `ColorConstants` definierte Konstante `WHITE`. Außerdem ist es sehr schwer in einer Tabelle Strukturen zu erkennen. Darum verwenden wir *Speichergraphen*. Die Notation G_t beschreibt den Speichergraphen zu dem Programmzustand P_t. Den resultierenden Speichergraphen zu dem »Car-Configurator«-Beispiel sehen wir in Abbildung 2.3. Wie wir sehen werden die primitiven Werte bzw. die Objekte als Knoten, die Variablen welche diese Werte benennen als Kanten dargestellt.

Wir sehen, dass ein Speichergraph ein komplexes Gebilde aller während der Programmausführung aktiven Variablen und deren referenzierten Werte und Objekte ist. In dem folgenden Abschnitt zeigen wir, wie man einen Speichergraphen an einer bestimmten Stelle innerhalb des Programmlaufs extrahiert.

2.2 Extrahieren eines Speichergraphen

Einen Programmzustand haben wir definiert als die Menge der aktiven Variablen und der von diesen Variablen referenzierten Werte und Objekte. Doch wie bekommen wir Zugriff auf die Variablen? Dazu verwenden wir den in Eclipse eingebauten Debugger.

Wenn wir ein Java-Programm starten, so wird, beginnend mit der `main` Funktion, jede aufgerufene Funktion auf einem Aufrufstapel (engl. Call Stack) abgelegt. Wenn eine Funktion beendet wird – entweder durch eine `return`-Anweisung oder eine Ausnahme (engl. Exception) – wird diese Funktion wieder vom Aufrufstapel entfernt. Jede Funktion die noch nicht beendet wurde, findet sich somit in dem Aufrufstapel wieder. Die Reihenfolge innerhalb des Stapels richtet sich nach der Reihenfolge der Funktionsaufrufe während der Programmausführung. Dabei ist die `main` Funktion immer die erste und unterste Funktion auf dem Stapel.

Variable	Type	Wert
frame[0].this	CarConfiguratorTest	64dc11
frame[0].myCar	Car	1ac1fe4
fName	String	5a23bc
doors	short	4
price	long	8150
interiorColor	String	5978fff
exteriorColor	String	6bdcc29
engine	Engine	161d36b
power	int	100
diesel	boolean	false
accessories	Vector	ecd7e
elementCount	int	1
serialVersionUID	long	-276760561404898439
modCount	int	1
capacityIncrement	int	0
elementData	Object[]	1b16e52
elementData[0]	Object	64ade8
elementData[1]	Object	null
WHITE	String	6bdcc29
BLACK	String	5978fff
RED	String	1b891
COLD_WEATHER_PACKAGE	AccessoryPool $COLD_WEATHER_PACKAGE	64ade8
ESP	AccessoryPool$ESP	f5a486b
AUTOMATIC_TRANSMISSION	AccessoryPool $AUTOMATIC_TRANSMISSION	165af

Abbildung 2.2: Variablen und Werte von P_t mit $t = (CarConfiguratorTest, 13, 1)$

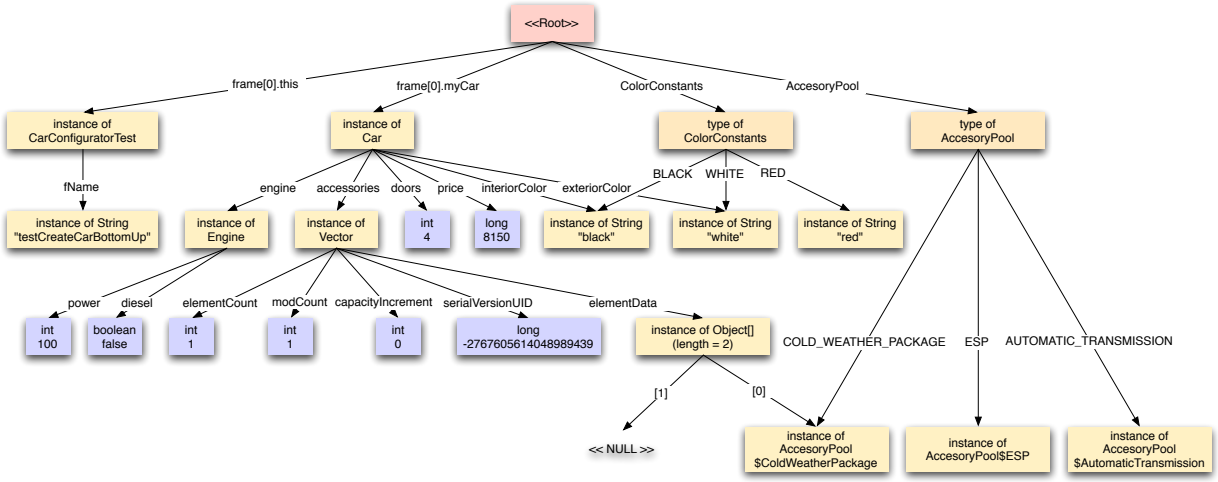


Abbildung 2.3: Speichergraph von P_t mit $t = (CarConfiguratorTest, 13, 1)$

Die verschiedenen Funktionen innerhalb eines Aufrufstapels werden als Stack Frames (kurz: Frame) bezeichnet und enthalten Informationen über die Parameter dieser Funktion und alle in der Funktion verwendeten Variablen und Werte. Wir bezeichnen diese Variablen als *lokale Variablen*.

Definition 3 (Aufrufstapel)**.** Wir definieren A_t als den Aufrufstapel zur Zeit $t = (d, z, n)$ während der Programmausführung. Wenn auf dem Aufrufstapel m Frames liegen bezeichnet $A\,[x]_t$ den x-ten Frame innerhalb des Aufrufstapels ($0 \leq x \leq m - 1$). Dabei gilt, dass der oberste Frame, und somit der Frame, an dem das Programm angehalten wurde, die Nummer 0 bekommt.

Neben den lokalen Variablen interessieren uns auch alle *globalen Variablen* des Programms, d.h. alle Variablen in allen während der Programmausführung geladenen Klassen oder Interfaces, welche als `static` deklariert wurden. Daher untersuchen wir die von der JVM geladenen Klassen bzw. Interfaces, da diese die globalen Variablen direkt referenzieren.

Die Menge der lokalen und globalen Variablen deklarieren wir als *Basisvariablen*, die konkreten Werte bzw. Instanzen der Variablen als *Basisobjekte*. Allgemein betrachtet wird der Zustand eines Objektes durch die Werte seiner Attribute (Objektattribute) bestimmt. Diese Attribute werden in den zugehörigen Objektklassen definiert. Eine primäre Einteilung der Attribute erfolgt in *primitive Attribute* oder *Objektattribute*, welche Referenzen auf weitere Objekte sind.

Von den Basisvariablen ausgehend können wir alle Variablen und Werte des Programmzustands erreichen. Um einen kompletten Zustand zu erhalten, extrahieren wir, zusätzlich zu den Basisobjekten, alle im Speicher befindlichen Objekte und Werte. Dies tun wir, indem wir die nicht primitiven Basisobjekte *entfalten*, d.h. wir fügen die referenzierten Objekte dieser Attribute dem Speichergraphen hinzu. Dieses Verfahren wenden wir rekursiv auf alle noch nicht entfaltenden

```
1  public IRootNode createMemoryGraph(IJavaThread thread) {
2
3    IRootNode resultGraph = new RootNode(passingRun, ddRunID);
4
5    // Step 1: Add all local variables to the graph
6    this.addLocalVariables(thread, resultGraph);
7
8    // Step 2: Add all global variables to the graph
9    this.addGlobalVariables(thread, resultGraph);
10
11   // Step 3: Unfold the graph
12   resultGraph.unfoldSubgraph(monitor);
13
14   return resultGraph;
15 }
```

Abbildung 2.4: Algorithmus zum Extrahieren eines Speichergraphen

Objekte des Graphen an und extrahieren somit den Zustand aller im Speicher befindlichen, referenzierten Objekte und Werte.

Um einen Speichergraphen zu erstellen, müssen wir zunächst die Programmausführung starten und an dem festgelegten Programmzeitpunkt $t = (d, z, n)$ innerhalb des Programmlaufs anhalten (siehe Abschnitt 2.4). Wenn wir die Programmausführung (mit Hilfe des Debuggers) gestoppt haben, extrahieren wir den Speichergraphen mit Hilfe der in DDSTATE implementierten Klasse `GraphExtractor` (siehe Abbildung 2.4). Dieser führt folgende Schritte aus:

1. Erstellen eines Startknotens (*Rootknoten*), an dem wir die Basisvariablen verankern werden. Der Rootknoten dient nur als Verankerungsknoten für die Basisvariablen und entspricht keinem Wert oder Objekt im Programmspeicher.

2. Extrahieren der Basisvariablen mit Hilfe des Aufrufstapels (siehe Abschnitt 2.5).

   a) Extrahieren der lokalen Basisvariablen.

   b) Extrahieren der globalen Basisvariablen.

3. Entfalten der Basisobjekte (siehe Abschnitt 2.6).

Bevor wir die Extraktion des Speichergraphen beschreiben, definieren wir zunächst die Struktur eines Speichergraphen (Objektmodell).

2.3 Das Objektmodell eines Speichergraphen

Ein Speichergraph besteht aus Knoten und gerichteten Kanten. Dabei repräsentieren die Knoten Werte und Objekte innerhalb des Speichers und die Kanten Variablen, die diese Werte referenzieren.

Um die Werte und Variablen innerhalb des Programmspeichers eines zu untersuchenden Programmes (engl. Debuggee) zu erfragen oder auch zu manipulieren, bietet Java eine Reihe von Schnittstellen (engl. Interfaces). Durch diese Schnittstellen können wir mit der virtuellen Maschine kommunizieren und somit Informationen über den Programmspeicher erhalten oder auch Manipulationen an Variablen und Werten vornehmen. Eine Übersicht, über die von Java bereitgestellten Interfaces, befindet sich in der *JPDA – Java Platform Debugger Architecture* (Sun, 2005b).

Eclipse bietet ebenfalls Möglichkeiten auf den Programmspeicher einer laufenden Anwendung zuzugreifen und nutzt dabei das von der JPDA etablierte *Java Debugging Interface* (kurz: JDI). Teile der Funktionalität in Eclipse sind gekapselt in dem Plugin `JDI Debug Model`, welches wiederum ein Bestandteil des Java Development Tools Feature[1] (kurz JDT Feature) ist. Dieses Plugin stellt drei interessante Interfaces zur Verfügung, die eine Abstraktion der Werte und Variablen innerhalb des Programmspeichers beschreiben:

`IJavaVariable` – Dieses Interface repräsentiert eine aktive Variable innerhalb des Debuggees. Sie bietet Funktionen um den Namen, deklarierten Typ und Wert der Variable zu erfragen.

`IJavaValue` – Dieses Interface repräsentiert einen Wert innerhalb des Programmspeichers. Dieser kann einerseits ein primitiver Wert, ein Objekt bzw. als Spezialisierung eines Objekts auch ein Array darstellen. Gemäß dieser Einteilung wurden in Eclipse folgende Subschnittstellen definiert: `IJavaPrimtiveValue`, `IJavaObject` und `IJavaArray`.

`IJavaType` – Dieses Interface repräsentiert den Typ einer Variable bzw. des Werts der Variable. Dabei unterscheiden wir zwischen dem *deklarierten Typ*, d.h. dem Typ, welcher der Variable im Quellcode zugewiesen wurde und dem *konkreten Typ*. Dies ist der Typ, den der Wert der Variable zur Programmausführung besitzt.

Über das Interface können wir den Namen des Typs erfragen. Handelt es sich bei dem Typ um ein *Referenztyp*, d.h. ein Typ der ein weiteres Objekt referenziert, können wir zusätzlich auf die global (als `static`) definierten Variablen und deren Werte zugreifen. Referenztypen werden weiter unterteilt in den Typ eines Arrays (`IJavaArrayType`), einer Klasse (`IJavaClassType`) bzw. einer Schnittstelle (`IJavaInterfaceType`). Neben den Referenztypen gibt es noch die *primitiven Typen*. In Java unterscheiden wir acht verschiedene primitive Typen: `short`, `int`, `long`, `float`, `double`, `byte`, `char` und `boolean`.

[1] Features sind in Eclipse eine Gruppierung von inhaltlich zusammengehörender Plugins

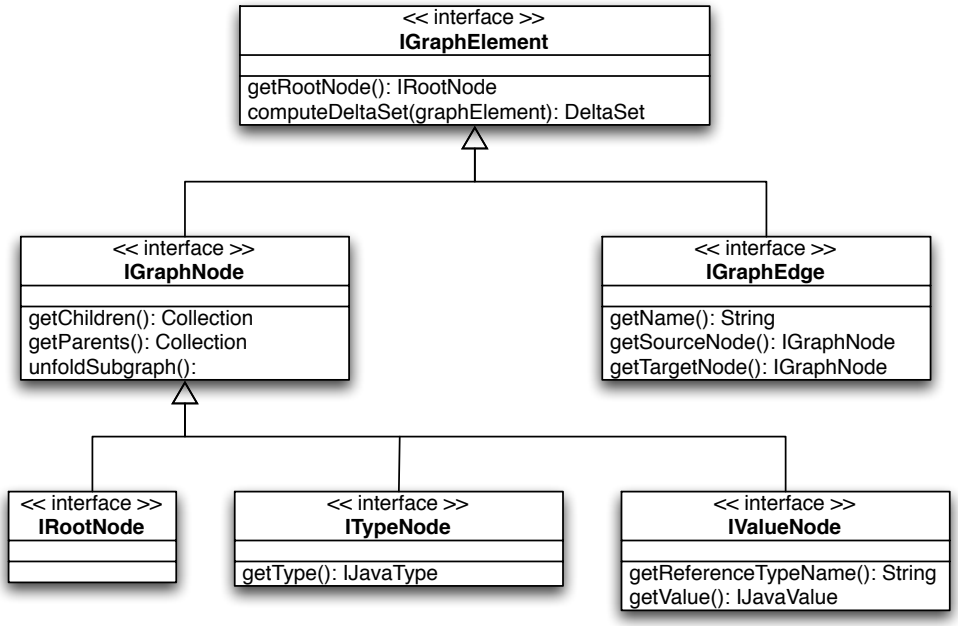


Abbildung 2.5: UML Klassendiagramm: `IGraphElement`

Diese drei Interfaces bieten uns eine high-level objektorientierte Sicht auf den Speicherinhalt des zu untersuchenden Programms. Um den kompletten Programmzustand zu erfassen, müssen wir sowohl die Variablen, deren Werte und alle Referenztypen (alle geladenen Klassen und Interfaces) in dem Speichergraphen abbilden. Dabei werden wir die Variablen als Kanten, die Werte einer Variablen und die Typen als Knoten beschreiben. Als Einstiegspunkt fügen wir dem Graph einen speziellen Knoten, den *Rootknoten* hinzu. An diesem werden wir später die Basisvariablen als Kindknoten verankern. Der Rootknoten dient rein als Verankerungspunkt und hat kein Pendant im Speicher.

Im Folgenden definieren wir die Interfaces für das Objektmodell des Speichergraphen (siehe Abbildung 2.5). Dieses hat den Vorteil, dass wir unabhängig von einer Implementierung bleiben und diese auch bei Bedarf austauschen können. Da wir die Struktur des Graphen objektorientiert aufbauen, definieren wir zuerst eine Schnittstelle namens `IGraphElement`. Diese beschreibt allgemein ein Element im Speichergraph. `IGraphElement` wird durch ein Interface für Kanten (`IGraphEdge`) und Knoten (`IGraphNode`) erweitert. Unter dem Knoten definieren wir ein Interface für den Rootknoten (`IRootNode`). Zusätzlich jeweils ein Interface, welches die Werte innerhalb des Programmspeichers repräsentiert (`IValueNode`) und die Typen dieser Werte (`ITypeNode`).

Untersuchen wir die einzelnen Graphelemente genauer:

`IGraphElement` – Dieses Interface wird von jedem Element in dem Speichergraphen implementiert. Die hier definierten Methoden gelten somit für Knoten, als auch für Kanten in

dem Speichergraphen.

- `getRootNode`: Jedes Graphelement gehört zu einem bestimmten Graphen. Jeder Graph hat genau einen Rootknoten. Diese Methode gibt den Rootknoten (Graphen) zurück, zu dem dieses Graphelement gehört.
- `computeDeltaSet`: Berechnet die Unterschiede (Deltas) zwischen diesem und dem, der Methode übergebenen, Graphelement (siehe Kapitel 3).

`IGraphEdge` – Dieses Interface repräsentiert eine Variable innerhalb des Speichergraphen. Diese Variable kann in einem Objekt (Attribut) oder Funktion (Parameter und lokale Funktionsvariablen) definiert sein und verweist auf Werte innerhalb des Programmspeichers. Somit kommen wir vom Ort der Definition (Quellknoten: engl. Sourcenode) durch den Zugriff auf die Variable zum Wert (Zielknoten: engl. Targetnode).

- `getTargetNode`: Gibt den Zielknoten zurück.
- `getSourceNode`: Gibt den Quellknoten zurück.
- `getName`: Gibt den Namen der Kante zurück. Der Name einer Kante muss ausgehend von dem jeweiligen Quellknoten eindeutig die repräsentierte Variable kennzeichnen (siehe Abschnitt 3.1).

`IGraphNode` – Der Graphknoten stellt einen Wert bzw. einen Typ innerhalb dem Speichergraphen dar. Dieser Wert kann von verschiedenen anderen Objekten oder Typen referenziert (direkter Vorgänger oder Eltern genannt: engl. Parents) bzw. verschiedene andere Werte referenzieren (direkter Nachfolger oder Kinder genannt: engl. Children).

- `unfoldSubGraph`: Entfaltet den Subgraphen, d.h. diesen Knoten und all seine potentiellen Nachfolgerknoten (siehe Abschnitt 2.6).
- `getChildren`: Gibt die Menge der direkten Nachfolger dieses Knotens zurück. Im Sinne des Programmspeichers sind dies alle Variablen die von dem Objekt bzw. Typ dieses Knotens definiert sind. Eine Ausnahme bilden Arrays. Arrays definieren keine Attribute. Die Kinder eines Arrays sind alle in den Arrayelementen referenzierten Werte.
- `getParents`: Gibt die Menge der direkten Vorgänger dieses Knotens zurück. Im Sinne des Programmspeichers sind dies alle Variablen, welche den Wert referenzieren, der durch diesen Knoten repräsentiert wird.

Eine zusätzliche Unterteilung des Interface `IValueNode` in primitive Werte, Objekte und Arrays entspricht den jeweiligen Subinterfaces von `IJavaValue` in Eclipse (siehe Abbildung 2.6). Außerdem ergänzen wir dieses, um den Knoten `INullValueNode`, welcher den Wert `null` repräsentiert. Der Wert `null` ist einer Objektvariable zugewiesen, sobald diese kein Objekt referenziert.

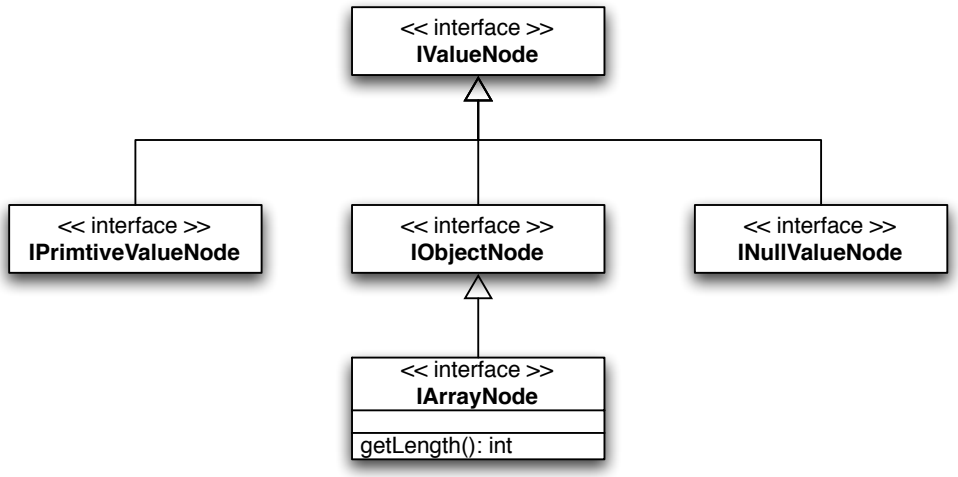


Abbildung 2.6: UML Klassendiagramm: `IGraphValueNode`

Nachdem wir die Knoten des Speichergraphen in Objekte, primitive Werte, etc. aufgeteilt haben, nehmen wir eine Klassifizierung der Kanten bzw. Variablen vor (siehe Abbildung 2.7). Wie wir in Abschnitt 2.2 gesehen haben, fügen wir dem Graphen zunächst die Basisvariablen hinzu. Diese unterteilen sich in lokale und globale Variablen. Die globalen Variablen können wir anhand der Klasse bzw. des Typs zu dem sie gehören gruppieren. Dies bedeutet ausgehend von dem Rootknoten gibt es Kanten für die lokalen Variablen (`ILocalVariableEdge`), sowie Kanten für die geladenen Klassen (`ITypeEdge`). Von jedem Typknoten, der eine in der JVM geladene Klasse repräsentiert gehen Kanten aus, welche die globalen Variablen referenzieren (`IGlobalVariableEdge`). Was wir außerdem beachten müssen, sind die Attribute von Objekten bzw. die Elemente von Arrays, welche durch die Kanten `IObjectFieldEdge` bzw. `IArrayElementEdge` mit den jeweiligen Objektknoten verknüpft sind.

Der objektorientierte Aufbau des Speichergraphen ist sehr komplex. Dieser ist allerdings notwendig, um die benötigten Aktionen wie das Vergleichen zweier Graphen, bzw. das Entfalten des Speichergraphen zu vereinfachen. Einen Überblick wie und an welcher Stelle die soeben eingeführten Interfaces bzw. deren Implementierungen in unserem Speichergraphen verwendet werden, soll die Graphik in Abbildung 2.8 verdeutlichen.

Aus der Übersicht geht hervor, dass von dem Rootknoten die Basisvariablen erreichbar sind. Dieses sind einerseits alle lokalen und globalen Variablen. Die globalen Variablen werden anhand ihres zugehörigen Typs gruppiert. Eine lokale bzw. globale Variable verweist auf einen Wert im Speicher. Dieser Wert wird durch einen Wertknoten repräsentiert. Als Spezialisierung eines Wertes unterscheiden wir zwischen Objekten, Arrays, primitiven Werten und dem Wert `null`. Objekte wiederum können durch ihrer Attribute weitere Werte referenzieren. Arrays, welche auch Objekte darstellen, definieren keine Attribute. Stattdessen können wir bei einem Array auf die Arrayelemente bzw. Felder zugreifen. Diese Felder sind Referenzen auf Werte. Zusätzlich können wir in der Abbildung erkennen, welcher Kantentyp (siehe Abbildung 2.7) die verschiede-

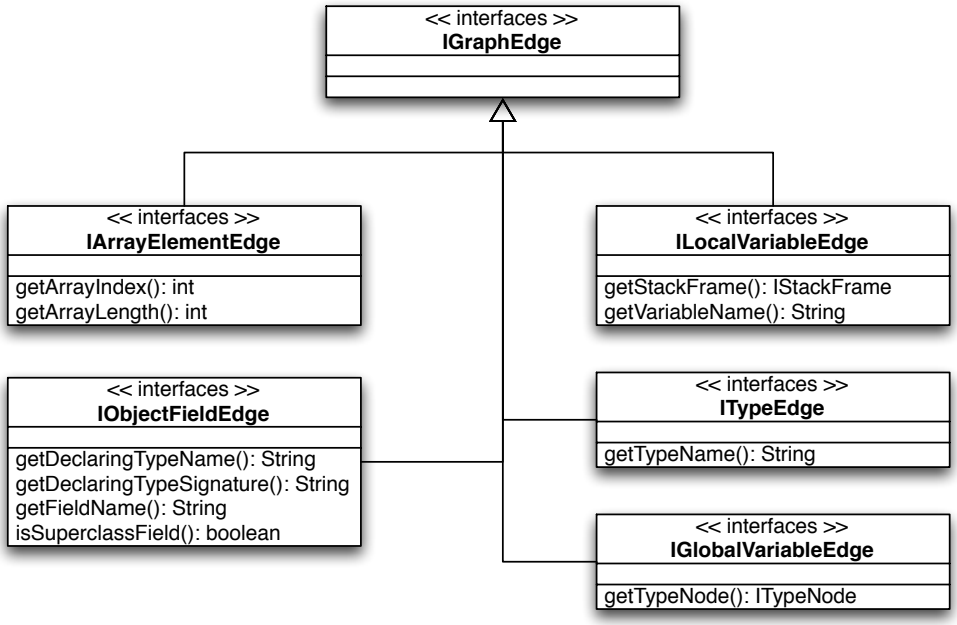


Abbildung 2.7: UML Klassendiagramm: `IGraphEdge`

nen Knoten miteinander verbindet. Die Farbgebung der einzelnen Knoten soll die verschiedenen Knotenarten besser unterscheiden und zur Lesbarkeit der Speichergraphen beitragen.

Wir haben das Objektmodell für den Speichergraphen erläutert und damit eine wohl definierte Basis geschaffen, um einen Programmzustand durch einen Graphen abzubilden. Um implementierungsunabhängig zu sein, haben wir zur Definition der Graphelemente Interfaces verwendet. Wenn wir die Implementierung eines Interface austauschen müssen, wäre es sinnvoll dies nur an einer Stelle im Programm zu tun. Die Konsequenz daraus ist, dass wir die Erstellung der konkreten Objekte an einer Stelle bzw. in einer Klasse im Programm bündeln. Deshalb haben wir in DDSTATE die Hilfsklasse `GraphElementUtil` bereitgestellt (siehe Abbildung 2.9). Alle Klassen, welche einen Knoten bzw. eine Kante des Speichergraphen erzeugen verwenden diese Hilfsklasse.

Die Funktion `createValueNode`, welche uns einen Knoten für einen Wert innerhalb des Speichers erstellt, ist eine Factory-Methode (siehe Metsker, 2002, Kapitel 16). Entsprechend dem übergebenen Wert, den die Variable `javaValue` besitzt, erzeugen wir ein Objekt vom Typ `IPrimitiveValueNode`, `INullValueNode`, `IObjectNode` oder `IArrayNode`. Zusätzlich stellt die Hilfsklasse Funktionen zur Verfügung, mit denen wir einen Knoten des Interfaces `ITypeNode` sowie die verschiedenen Kanten erzeugen können.

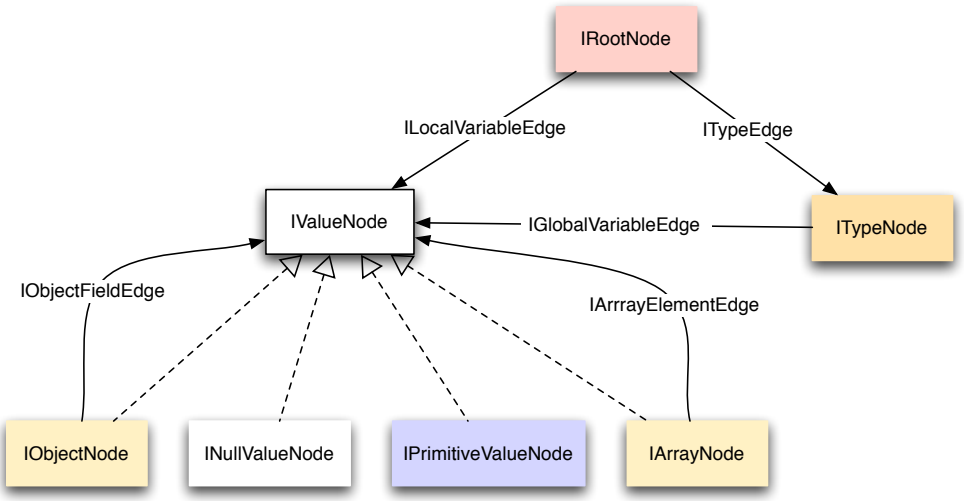


Abbildung 2.8: Verwendung der verschiedenen Graphelemente innerhalb des Speichergraphen

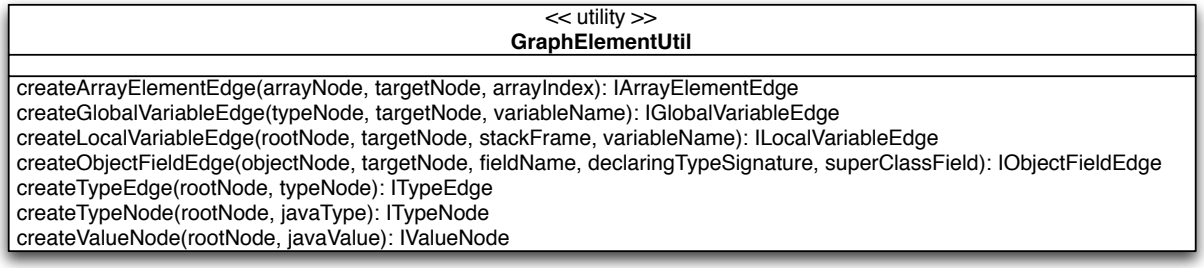


Abbildung 2.9: UML Klassendiagramm: `GraphElementUtil`

2.4 Zugriff auf den Programmspeicher

Um die Basisvariablen zu extrahieren, werden wir uns zunächst Zugriff auf den Programmspeicher des Debuggee verschaffen. Dazu verwenden wir das *Launching Framework* von Eclipse, welches Funktionen bereitstellt um Java Anwendungen zu starten und zu debuggen. Ganz in dem Sinne von Eclipse ist dieses Framework als Erweiterung bzw. Plugin in der Eclipse Plattform integriert. Szurszewski (2003) beschreibt die Funktionalitäten und Schnittstellen von diesem Framework. In dem Software Development Kit von Eclipse sind schon verschiedene so genannte *Launcher*, als konkrete Ausprägungen des Frameworks, integriert:

- Launcher zum Starten lokaler Java Anwendungen und Java Applets.
- Launcher zum Verbinden mit Java Anwendungen, welche auf einem anderen Rechner gestartet wurden (Remote Applikationen).

- Launcher zum Starten von JUnit Test Suites bzw. einzelner Tests innerhalb einer Suite.

Wright beschreibt weiterhin in Wright (2003), wie wir mit Hilfe dieses Frameworks Applikationen programmatisch starten können.

Mit Hilfe des Launching Framework können wir das Debuggee bzw. den JUnit Testfall starten. Um diesen innerhalb des Programmlaufs anhalten zu können, nutzen wir das Konzept der *Haltepunkte* (engl. breakpoints). Dabei gibt es verschiedene Arten von Haltepunkten: Z.B. Ausnahmehaltepunkte, Methodenhaltepunkte oder auch Zeilenhaltepunkte. Die zwei zuletzt genannten sind am interessantesten für uns.

Methodenhaltepunkte (engl. Method Entry Breakpoints) – Unterbrechen den Programmlauf beim Aufruf oder dem Verlassen einer bestimmten Methode. Um einen Methodenhaltepunkt *eindeutig* zu definieren benötigen wir die Angabe der Klasse, in der die Methode definiert wurde, sowie den Methodennamen und die JDI Signatur der Methode. Letztere wird benötigt für den Fall, dass in einer Klasse zwei Methoden mit gleichen Namen aber unterschiedlichen Signaturen spezifiziert sind.

Zeilenhaltepunkte (engl. Line Breakpoints) – Unterbrechen den Programmlauf *vor* dem direkten Aufruf einer bestimmten Zeile innerhalb einer Klasse. Mit Hilfe des so genannten *hit count* können wir die Anzahl bestimmen, wie oft die Zeile durchlaufen werden soll, bevor der Programmlauf angehalten wird. Damit ist der Zeilenhaltepunkt das technische Pendant zu dem Programmzeitpunkt (vgl. Definition 1).

Um auf das Erreichen eines Haltepunkts zu reagieren, können wir in Eclipse einen Listener gemäß dem Observer Pattern registrieren (Metsker, 2002, Kapitel 9). Die Funktionalität hierfür finden wir in der Klasse `JDIDebugModel` in dem Eclipse Plugin `Debug Core`. Diese Klasse verfügt weiterhin über Methoden, um einen Haltepunkt zu erzeugen. Um diesen aber zu registrieren, benötigen wir ein Objekt vom Typ `IBreakpointManager`. Den Breakpointmanager verwendet Eclipse, um Haltepunkte zu re- und deregistrieren. Eine Instanz können wir über die Klasse `DebugPlugin` erhalten.

Betrachten wir die Signatur der Methode `breakpointHit` des Listeners näher (siehe Abbildung 2.10): Zu dem Haltepunkt, der bei der Programmausführung erreicht wird, bekommen wir den aktuellen Thread, indem der Haltepunkt erreicht wird, übergeben. Über diesen Thread haben wir Zugriff auf den Aufrufstapel und die Basisvariablen. Er dient somit als Einstiegspunkt, um den Programmzustand auszulesen.

Wir wissen, welche Ressourcen bzw. Plugins wir in Eclipse verwenden können, um eine Applikation zu starten und diese gezielt an einer Stelle anzuhalten. Des Weiteren kommen wir über den Haltepunkt-Listener an den aktuellen Thread und damit an den Zustand des zu untersuchenden Programms. Als nächstes betrachten wir, wie wir einen Programmzustand an einer Stelle extrahieren, d.h. wie wir den Programmzustand in einen Speichergraphen abbilden. Dies geschieht zunächst durch Extrahieren der Basisvariablen und -objekte.

```
1  public interface IJavaBreakpointListener {
2
3    /**
4     * Notification that the given breakpoint has been hit
5     * in the specified thread.
6     * ...
7     */
8    public int breakpointHit(IJavaThread thread,
9      IJavaBreakpoint breakpoint);
10
11   ...
```

Abbildung 2.10: Implementierung des Interface `IJavaBreakpointListener`

2.5 Extrahieren der Basisvariablen

Nachdem wir die Struktur des Speichergraphen definiert und den Zugriff auf den Thread hergestellt haben, geht es darum, die Basisvariablen zu extrahieren. Dazu benötigen wir zwei Schritte:

- Zunächst müssen wir von allen Funktionen auf dem Aufrufstapel die lokalen Variablen extrahieren.
- Danach können wir von allen geladenen Klassen und Interfaces die globalen Variablen hinzufügen.

In Abschnitt 2.4 haben wir gesehen, wie wir uns Zugriff auf den Programmspeicher verschaffen, wenn wir den Programmlauf angehalten haben. Über den Thread, den wir über den Haltepunktlistener bekommen, haben wir Zugang zu allen Frames auf dem Aufrufstapel. Über die Frames können wir dann alle lokalen Variablen extrahieren, welche wir dem Rootknoten hinzufügen (siehe Abbildung 2.11). Um den Zugriff auf den Speicher zu erleichtern, haben wir eine Klasse `MemoryAccessUtil` eingeführt, die als reine Hilfsklasse fungiert. Die Funktion `getAllJavaVariables` extrahiert aus dem übergebenen Frame ein Array von Variablen, die von diesem Frame aus erreichbar sind. Dies sind einerseits die Parameter, der zu dem Frame gehörenden Funktion, und die innerhalb des Frames definierten lokalen Variablen.

Nachdem wir die lokalen Variablen erzeugt haben, fügen wir die globalen Variablen zu dem Rootknoten hinzu. Diese gruppieren wir nach dem Typ (Klasse oder Interface) in dem sie definiert wurden. Deshalb erweitern wir den Rootknoten zunächst um die Typen, welche statische (globale) Variablen definieren. Beim Entfalten der Typknoten werden die globalen Variablen dann den entsprechenden Typknoten hinzugefügt (siehe Abschnitt 2.6). Wir müssen also zunächst unter allen geladenen Klassen und Interfaces diejenigen herausfiltern, welche statische Variablen definieren. Dazu verwenden wir die Hilfsfunktion `getTypesWithStaticVariables`. Der Rootknoten wird dann um die extrahierten Typen ergänzt (siehe Abbildung 2.12).

```
1  private void extractLocalVariables(IJavaThread thread,
2    RootNode rootNode) {
3
4    IJavaStackFrame[] frames =
5      MemoryAccessUtil.getAllStackframes(thread);
6
7    for (int j = 0; j < frames.length; i++) {
8      IJavaVariable[] variables =
9        MemoryAccessUtil.getAllJavaVariables(frames[j]);
10
11     for (int i = 0; i < variables.length; i++) {
12       rootNode.addLocalVariable(variables[i], frame);
13     }
14   }
15 }
```

Abbildung 2.11: Algorithmus zum Extrahieren von lokalen Variablen

```
1  private void addTypes(IJavaThread thread, RootNode rootNode) {
2
3    IJavaType[] types =
4      MemoryAccessUtil.getTypesWithStaticVariables(thread);
5
6    rootNode.addType(type);
7  }
```

Abbildung 2.12: Algorithmus zum Extrahieren der Klassen, welche globale Variablen definieren

Wenn wir einen Knoten erzeugen müssen wir sicherstellen, dass zu dem Wert, für den wir den Knoten erstellen, nicht bereits ein Knoten erstellt wurde. Dies bezieht sich allerdings nur auf Werte, welche ein Objekt in dem Programmspeicher repräsentieren, da Objekte von mehreren Variablen referenziert werden können. Um zu vermeiden, dass zu einem Objekt im Programmspeicher mehrmals ein Knoten in dem Speichergraph erzeugt wird, registrieren wir den jeweils erzeugten Objektknoten mit dem dazugehörigen Objektwert. Einen geeigneten Platz für die Registrierung ist der Rootknoten, da er von allen anderen Knoten aus erreicht werden kann und genau einmal pro Speichergraph existiert. Eine Funktion, welche überprüft ob ein Knoten bereits existiert und in Abhängigkeit davon einen neuen Knoten erzeugt, haben wir in der Klasse `GraphNode` (Implementierung von `IGraphNode`) bereitgestellt, so dass jeder Graphknoten auf diese Funktionalität zurückgreifen kann (siehe Abbildung 2.13).

```
protected IGraphNode createAndRegisterNode(IJavaVariable variable) {

  IGraphNode result = null;
  IValue value = variable.getValue();

  if (value instanceof IJavaObject
      && this.getRootNode().existObjectNode((IJavaObject) value)) {
    result = this.getRootNode().getObjectNode((IJavaObject) value);

  } else {
    result = GraphElementUtil.createValueNode(this.getRootNode(),
      variable);

    if (result instanceof IObjectNode) {
      this.getRootNode().registerObjectNode((IObjectNode) result);
    }
  }

  return result;
}
```

Abbildung 2.13: Algorithmus zum Erzeugen und Registrieren von Speicherknoten

2.6 Entfalten von Objekten

Bisher haben wir gesehen, wie wir die lokalen Basisvariablen und geladenen Klassen extrahieren und dem Rootknoten hinzufügen. In diesem Abschnitt zeigen wir, wie wir die Basisvariablen entfalten um dadurch den kompletten Speichergraphen zu erhalten.

Ein Objekt besteht aus keinem, einem oder mehreren Attributen. Wenn es sich bei dem Objekt um ein Array handelt, sprechen wir von Elementen oder Feldern. Die Attribute eines Objektes werden in den jeweiligen Objektklassen definiert. Ein Objekt der Klasse c besteht somit aus allen Attributen die in c definiert wurden. Weiterhin enthält das Objekt alle Attribute die in den Oberklassen von c definiert wurden. In dem Speichergraph aus Abbildung 2.3 gibt es ein Objekt der Klasse `java.util.Vector`. Dieses Objekt besitzt fünf Attribute, wobei nur vier dieser Attribute in der Klassendefinition von `Vector` stehen. Das Attribut `modCount` ist in der Oberklasse `java.util.AbstractList` definiert. Da `Vector` eine Unterklasse von `AbstractList` ist, enthält das `Vector`-Objekt im Programmzustand auch das `modCount` Attribut.

Sei V_O der Knoten in dem Speichergraphen zu einem Objekt O im Programmzustand. Um V_O zu entfalten, müssen wir von allen Attributen bzw. Elementen von O neue Knoten erzeugen und diese V_O hinzufügen. Diese neuen Knoten werden dadurch direkte Nachfolger des Objektknotens. Wir wenden dieses Verfahren beginnend bei den Basisobjekten rekursiv auf jeden, noch

```
1  public void unfoldSubgraph() {
2
3    if (!this.isUnfolded) {
4      this.isUnfolded = true;
5      Iterator itChildren = this.getChildren().iterator();
6
7      while (itChildren.hasNext()) {
8        GraphEdge graphEdge = (GraphEdge) itChildren.next();
9        GraphNode targetNode = (GraphNode) graphEdge.getTargetNode();
10
11       if (!targetNode.isUnfolded) {
12         targetNode.createChildren();
13         targetNode.unfoldSubgraph(monitor);
14       }
15     }
16   }
17 }
```

Abbildung 2.14: Algorithmus zum Entfalten eines Speicherknotens

nicht entfaltenden Objektknoten an. Das Ergebnis ist ein Speichergraph, welcher den kompletten Programmzustand abbildet.

Wie die Objektknoten, definiert ein Typknoten Attribute. Dies sind die statisch definierten Klassen- bzw. Interfaceattribute. In Abschnitt 2.5 hatten wir beschrieben, dass wir die globalen Variablen nicht als direkte Nachfolger dem Rootknoten hinzufügen, sondern zuerst die jeweiligen Typknoten. Es ist in einem zweiten Schritt erforderlich den Typknoten, um die von der jeweiligen Klasse bzw. Interface global definierten Variablen zu erweitern.

Allgemein ist die Vorgehensweise diese: Wir betrachten von jedem Knoten in dem Speichergraphen, der noch nicht entfaltet ist, seine Nachfolgerknoten. Für den jeweiligen Nachfolgerknoten bestimmen wir wiederum seine Nachfolger, indem wir die Attribute, Arrayelemente oder globalen Variablen dem jeweiligen Knoten hinzufügen. Wenn wir Knoten erzeugen achten wir darauf, dass wir zu einem Objekt in dem Programmspeicher nicht einen Knoten doppelt erzeugen (siehe Abbildung 2.13). Sollte bereits der Knoten für einen Wert im Programmspeicher existieren, erstellen wir nur noch eine Kante, welche dann auf diesen Knoten verweist.

Um sicherzustellen, dass wir einen Knoten nicht mehrfach entfalten, nutzen wir ein Flag in jedem Knoten, dass gesetzt wird sobald wir anfangen den Knoten zu entfalten. Das Entfalten eines Knoten ist für jeden Graphknoten gleich und ist deshalb in der Funktion `unfoldSubgraph` in der Klasse `GraphNode` implementiert (siehe Abbildung 2.14). Wir beginnen das Entfalten des Speichergraphen mit dem Rootknoten, nachdem wir diesen um die Basisvariablen erweitert haben.

2.7 Details der Implementierung

Atomare Behandlung von String-Objekten

Ein String beschreibt eine Sammlung von Zeichen, die im Speicher geordnet abgelegt werden. Java sieht für die Repräsentation von Strings mehrere Klassen vor – eine dieser Klassen ist `String` aus dem Paket `java.lang`. Sie wird am Häufigsten für die Repräsentation von Zeichenketten in Java verwendet und stellt Zeichenketten dar, die sich nicht mehr verändern können (*immutable* Objekte). Neben der `String`-Klasse bezeichnet die Klasse `StringBuffer` so genannte dynamische Zeichenketten (siehe Ullenboom, 2006, Kapitel 4).

String-Objekte stellen in Java eine Besonderheit dar. Sie sind die einzigen Objekte, die direkt über ein Literal ohne konkreten Aufruf eines Konstruktors erzeugt werden können:

```
String test = "Hello World";
```

Diese String-Literale werden direkt als Objekt angelegt und können auch als solche genutzt werden. Das bedeutet auch, dass hinter dem String-Literal gleich ein Punkt für den Methodenaufruf stehen kann:

```
int x = "Hello World".length();
```

Durch die literale Repräsentation werden Strings oftmals wie primitive Datentypen angesehen. Betrachten wir den Programmzustand eines Strings (vgl. Abbildung 1.2), so wird das String-Objekt durch ein Character-Array sowie die Attribute `CASE_INSENSITIVE_ORDER`, `count`, `hash`, `offset`, `serialPersistentFields` und `serialVersionUID` repräsentiert. Um eine Vereinfachung bei der Extraktion und dem Vergleich von `String`-Objekten zu erreichen, fügen wir ein weiteres Subinterface `IStringObjectNode` der Schnittstelle `IObjectNode` zu dem Objektmodell des Speichergraphen hinzu. Da wir den kompletten Programmzustand extrahieren möchten und daher keine Informationen verlieren dürfen, merken wir uns die Zeichenkette des jeweiligen Strings in dem Stringknoten.

Von einem Stringknoten können keine weiteren Objekte referenziert werden. Daher müssen wir bei dem Entfalten eines Stringknotens auch nichts weiter tun. Der Stringknoten ähnelt sehr stark einem primitiven Werteknoten. Der entsprechende Wert ist die Zeichenkette, welche durch den Knoten repräsentiert wird. Allerdings gibt es zu dem Knoten vom Typ `IPrimitiveValueNode` einen großen Unterschied. Stringknoten können nämlich von mehreren Variablen referenziert werden, während ein primitiver Wert *genau von einer* primitiven Variable referenziert wird.

Durch die atomare Behandlung von `String`'s, wird die Repräsentation der `String`-Objekte innerhalb des Speichergraphens der literalen Verwendung dieser Objekte in Java angepasst. Wenn wir im Quellcode Strings verwenden, betrachten wir diese auch nicht als eine geordnete Sammlung von Zeichen (inklusive Offset, Hashwert etc.), sondern als komplette Einheit. Diese Betrachtungsweise wird sich auch in den weiteren Kapiteln nützlich erweisen und zur Verständlichkeit der Fehleranalyse beitragen. Da die gewählte Form der Repräsentation keine Notwendigkeit

darstellt (wir können den String auch in seine elementaren Attribute zerlegen), werden wir in den folgenden Kapiteln Knoten vom Typ `IStringObjectNode` nicht näher bei der Beschreibung der Algorithmen behandeln.

2.8 Fazit

In diesem Kapitel haben wir definiert, was der Programmzustand eines Java-Programms umfasst. Wir haben ein Objektmodell erstellt, um den Zustand in Form eines Speichergraphen abbilden zu können. Dazu mussten wir zunächst Zugriff zum Programmspeicher bekommen. Wir haben gesehen, wie wir die Basisvariablen extrahieren und danach rekursiv den kompletten Speichergraphen extrahieren. Durch die Abbildung des Programmzustands in einen Speichergraphen sind wir in der Lage Unterschiede zwischen zwei Graphen herauszufinden. Wie wir diese Unterschiede bestimmen betrachten wir in dem nächsten Kapitel.

3 Vergleichen von Speichergraphen

Im letzten Kapitel haben wir gesehen, wie wir zu einem gegebenen Programmzeitpunkt innerhalb der Programmausführung einen Speichergraphen extrahieren können. Dazu definierten wir die Struktur des Speichergraphen, den wir dann durch rekursiven Aufbau erstellt haben. Nachdem wir in der Lage sind Programmzustände mittels Speichergraphen abzubilden, wollen wir in diesem Kapitel untersuchen, wie wir zwei Speichergraphen miteinander vergleichen. Als Ergebnis des Vergleichs erhalten wir eine Menge von *Graphunterschieden*, so genannte *Deltas*. Diese Deltas repräsentieren die Unterschiede des Speichergraphen und somit die Unterschiede der Programmzustände.

Wir untersuchen die Speichergraphen für zwei verschiedene Programmläufe: Einen funktionierenden Lauf ($r_{✔}$) und einen fehlerhaften Lauf ($r_{✘}$). Die jeweiligen Speichergraphen $G_{✔}$ und $G_{✘}$ unterscheiden sich, entweder in ihrer Struktur und/oder in den jeweiligen Knoten. Was wäre die Konsequenz, wenn beide Speichergraphen identisch sind? Wir können einen Programmzustand als eine Eingabe betrachten, um den nächsten Programmzustand zu berechnen. Wenn wir zwei Programmläufe an der gleichen Stelle (identische Call Stacks) anhalten und ihnen die gleiche Eingabe geben (identischer Zustand), wird auch das gleiche Ergebnis (identischer Folgezustand) berechnet. Daraus folgt aber wiederum, dass beide Programmläufe zum gleichen Endergebnis führen. Dies widerspricht aber der Annahme, dass wir zwei Programmläufe betrachten, welche ein unterschiedliches Ergebnis produzieren.

Von allen Deltas zwischen zwei Graphen ist nur eine Teilmenge für den fehlerhaften Programmlauf verantwortlich. Mit Hilfe von Delta Debugging sind wir in der Lage, aus allen zuvor bestimmten Deltas, den bzw. die fehlerverursachenden Unterschiede zu berechnen. Doch wie berechnen wir die Unterschiede in zwei Speichergraphen? Die Idee ist, dass wir zunächst den größt möglichen Teilgraphen von $G_{✔}$ und $G_{✘}$ bestimmen. Alle Graphelemente welche sich nicht in diesem gemeinsamen Teilgraphen befinden repräsentieren Unterschiede zwischen den dazugehörigen Programmzuständen.

3.1 Finden des größten gemeinsamen Teilgraphen

In Zimmermann u. Zeller (2002) ist beschrieben, wie wir für einen Speichergraphen den größten gemeinsamen Teilgraphen (engl. Maximum Common Subgraph; kurz: MCS) finden. Der vorgestellte Algorithmus beruht auf den Algorithmen von Bron u. Kerbosch (1973) sowie Barrow u. Burstall (1976). Da das Problem den größten gemeinsamen Teilgraphen zu finden zu der Klasse

der NP-vollständigen Probleme zählt, wählen wir eine für uns pragmatische Alternative, indem wir einen möglichst großen Teilgraphen mit Hilfe der Tiefensuche berechnen. Die Komplexität der Tiefensuche ist proportional zu der Anzahl der Graphelemente und beträgt $O(|V| + |E|)$, wobei $|V|$ die Anzahl der Knoten und $|E|$ die Anzahl der Kanten darstellt.

Wir beginnen die Tiefensuche an den Rootknoten der Speichergraphen. Die Tiefensuche ist ein rekursives Verfahren. Die Rekursion terminiert sobald wir zwei Graphelemente miteinander vergleichen, welche per Definition nicht gleich sind, oder schon verglichen worden sind. Ob zwei Graphelemente gleich sind, wird für jedes Element durch die Funktion `graphEquals` bestimmt. Diese muss in jedem Graphelement implementiert sein, da sie in dem Interface `IGraphElement` deklariert wurde. Um die Information zu speichern, dass zwei Graphelemente in beiden Speichergraphen gleich sind, und somit zueinander gehören, verknüpfen wir die jeweiligen Graphelemente miteinander. Wir tun dies, indem wir Querverweise zwischen den jeweiligen Graphelementen erstellen.

Zunächst definieren wir, welche Voraussetzungen zwei Graphelemente mitbringen müssen, damit sie als gleich gewertet werden können. Dazu bestimmen wir die Gleichheit von Knoten, danach die von Kanten.

Knotengleichheit

Generell unterscheiden wir zwei Arten von Knoten: Knoten, welche einen konkreten Wert im Programmzustand beschreiben, und spezielle Knoten, die wir als »Hilfsknoten« zum Aufbau eines Speichergraphen nutzen. Wenn wir die in Abschnitt 2.3 definierten Knotenarten zu Grunde legen gehören zu der ersten Gruppe alle Knoten vom Typ `IValueNode` und deren Subinterfaces. Die Folge ist, dass `IRootNode` und `ITypeNode` spezielle Knoten sind, denen wir keinen expliziten Wert im Speicher zuordnen. Als generelle Voraussetzung für Knotengleichheit gilt, dass beide Graphknoten von dem gleichen Interface abgeleitet sind, d.h. die gleiche Knotenart repräsentieren.

Zwei Rootknoten sind per Definition gleich und dienen als Startknoten für die Tiefensuche. Wenn wir zwei Knoten vom Typ `ITypeNode` vergleichen, dann sind diese genau dann gleich, wenn der zugrunde liegende Typ, den sie repräsentieren, der Gleiche ist. Dies überprüfen wir, indem wir den *voll qualifizierten Typnamen* (Paket- und Klassenname) vergleichen.

Wir unterscheiden in einem Programmzustand insgesamt vier verschiedene Arten von Werten (Subinterfaces von `IValueNode`; Vgl. Abbildung 2.6):

Null-Werte – zwei Knoten, welche `null`-Werte im Programmspeicher beschreiben sind immer gleich.

Primitive Werte – zwei primitive Wertknoten sind genau dann gleich, wenn sie den gleichen Wert repräsentieren und beide Werte vom gleichen Typ sind.

Objekte und Arrays – zwei Objekte bzw. Arrays sind genau dann gleich, wenn der konkrete Typ des Objekts bzw. Arrays der Gleiche ist.

Existiert z.B. in beiden Programmzuständen eine Variable `telephonList` welche als `java.util.List` deklariert wurde: Wenn der konkrete Typ, des von `telephonList` referenzierten Objektes in $P_{✓}$ eine `LinkedList` und in $P_{✘}$ eine `ArrayList` darstellt, so sind die jeweiligen Objektknoten in den Speichergraphen nicht gleich.

Sobald wir zwei gleiche Knoten gefunden haben, verknüpfen wir diese miteinander, und vergleichen alle ausgehenden Kanten der beiden Knoten miteinander.

Kantengleichheit

Es gilt die Regel: Zwei Kanten sind genau dann gleich, wenn die Quellknoten und die Namen der beiden Kanten gleich sind. Dabei ist es wichtig, dass alle Kanten, ausgehend von einem Knoten, einen *eindeutigen* Namen besitzen. Bevor wir also zwei Kanten miteinander vergleichen können, müssen wir zunächst die Namen der Kanten definieren. Um die Eindeutigkeit der jeweiligen Kantennamen zu zeigen, definieren wir die Namen in Abhängigkeit von den verschiedenen Knotenarten, welche über ausgehende Kanten verfügen können (vergleiche Abbildung 2.8):

`IRootNode` – Von dem Rootknoten gibt es zwei Arten ausgehender Kanten: `ITypeEdge` und `ILocalVariableEdge`. Den Namen von `ITypeEdge` ist durch den vollständigen Namen des Typs, den der Zielknoten der jeweiligen Kante beschreibt, definiert. Da der Zielknoten einen Objekttypen, d.h. eine in der JVM geladene Klasse repräsentiert, ergibt sich der Kantenname wie folgt:

`Paketname.Klassenname`

Der zweite Kantentyp `ILocalVariableEdge` zeigt auf lokale Variablen. Diese können einem Frame im Aufrufstapel eindeutig zugewiesen werden. Daher setzt sich der Name einer Kante, welche eine lokale Variable repräsentiert, aus der Framenummer und dem eigentlichen Variablennamen wie folgt zusammen:

`frame[`*`Framenummer`*`].`*`Variablenname`*

Da die Framenummer eindeutig ist und der Variablenname innerhalb eines Frames (d.h. innerhalb der dem Frame zugrunde liegenden Methode) eindeutig sein muss, ist die gewählte Namensgebung ebenfalls eindeutig. Da die Paket- und Klassennamen sich nur aus Buchstaben zusammensetzen darf, und daher keine Klammern enthalten können, sind auch die Namen von `ILocalVariableEdge` und `ITypeEdge` untereinander eindeutig.

`ITypeNode` – Die statischen Variablen, die innerhalb einer Klasse bzw. eines Typs deklariert werden, müssen einen eindeutigen Namen tragen. Deshalb entspricht der Name von `ITypeEdge` dem Namen der jeweils zugrunde liegenden statischen Variable.

`IObjectNode` – Der vollständige Zustand einer Objektinstanz besteht aus den Variablen, die in der Klasse des konkreten Objekttyps definiert wurden, sowie allen Variablen die in Oberklassen definiert wurden (siehe Abschnitt 2.6). Da die Namensgebung der Attribute innerhalb einer Klassenhierarchie nicht eindeutig sein muss, kann es vorkommen, dass eine Objektinstanz mehrere Attribute mit dem gleichen Namen besitzt. Diese Attribute gleichen Namens müssen in *unterschiedlichen* Klassen definiert sein. Daher definieren wir die Namensgebung von Objektattributen wie folgt:

Attribute, welche in dem konkreten Typ der Objektinstanz definiert wurden, werden einzig durch den jeweiligen Attributnamen definiert:

`Attributname`

Wurde das Attribut in einer Oberklasse (Superklasse) definiert, so wird der Klassenname der Oberklasse zur eindeutigen Identifizierung in Klammern vorangestellt:

`(Klassenname)Attributname`

`IArrayNode` – Im Vergleich zu Objekten ist bei Arrays die Kantenbenennung wesentlich einfacher. Um ein Element innerhalb eines Arrays eindeutig zu identifizieren, genügt es den jeweiligen Index des Elements innerhalb des Arrays zu verwenden. Deshalb bilden wir den Namen einer Instanz von `IArrayElementEdge` wie folgt:

`[Arrayelementindex]`

Nachdem wir definiert haben, unter welchen Bedingungen zwei Graphknoten bzw. -kanten gleich sind, können wir mittels rekursiver Tiefensuche einen größtmöglichen Teilgraphen von zwei Speichergraphen bestimmen. Die Implementierung des Algorithmus finden wir in der Klasse `MCSComparator` (siehe Abbildung 3.1). Wir beginnen die Suche bei den beiden Rootknoten der Graphen. Mit Unterstützung der Hilfsklasse `GraphEdgeDiffUtil` bestimmen wir von zwei Knoten, die Menge aller ausgehenden Kanten mit gleichem Namen. Nachdem wir zwei gleiche Kanten gefunden haben, vergleichen wir die jeweiligen Zielknoten miteinander. Sind diese auch gleich, gehen wir in Rekursion.

Wenn wir zwei gleiche Graphelemente gefunden haben, können wir die Verknüpfung zwischen den beiden Graphelementen über die Funktion `setAssociatedElement` herstellen. Der gemeinsame Teilgraph besteht, nach dem Ausführen der Tiefensuche, aus allen verknüpften Graphelementen.

Werfen wir ein Blick auf die Speichergraphen, welche wir erhalten, wenn wir den fehlerhaften und funktionierenden Programmlauf des »Car-Configurator«-Beispiel vor dem jeweiligen Aufruf der `Assert`-Überprüfung anhalten (siehe Anhang A.2). In Abbildung 3.2 sehen wir den aus $G_{✓}$ und $G_{✘}$ ermittelten größten gemeinsamen Teilgraphen. Aus der Abbildung wird ersichtlich,

```
1  public void findMCS(IGraphNode firstGraph, IGraphNode secondGraph) {
2    GraphEdgeDiff diff = GraphEdgeDiffUtil.computeDiff(firstGraph,
3      secondGraph);
4
5    // look for equal edges in both nodes
6    Collection graphPairs = diff.getEdgesInBothGraph();
7
8    for (Iterator itGraphPairs = graphPairs.iterator(); itGraphPairs
9       .hasNext();) {
10
11     GraphEdgePair graphPair = (GraphEdgePair) itGraphPairs.next();
12
13     IGraphEdge firstSuccEdge = graphPair.getFirstEdge();
14     IGraphEdge secondSuccEdge = graphPair.getSecondEdge();
15
16     if (firstSuccEdge.graphEquals(secondSuccEdge)) {
17       // set the association between the two edges
18       firstSuccEdge.setAssociatedElement(secondSuccEdge);
19       secondSuccEdge.setAssociatedElement(firstSuccEdge);
20
21       IGraphNode firstTargetNode = firstSuccEdge.getTargetNode();
22       IGraphNode secondTargetNode = secondSuccEdge.getTargetNode();
23
24       if (firstTargetNode.graphEquals(secondTargetNode)) {
25
26         if (firstTargetNode.hasAssociatedElement()
27           || secondTargetNode.hasAssociatedElement())
28           return; // terminate recursion
29
30         // set the association between the two nodes
31         firstTargetNode.setAssociatedElement(secondTargetNode);
32         secondTargetNode.setAssociatedElement(firstTargetNode);
33
34         this.findMCS(firstTargetNode, secondTargetNode);
35       }
36     }
37   }
38 }
```

Abbildung 3.1: Algorithmus zum Finden eines großen gemeinsamen Teilgraphen mittels Tiefensuche

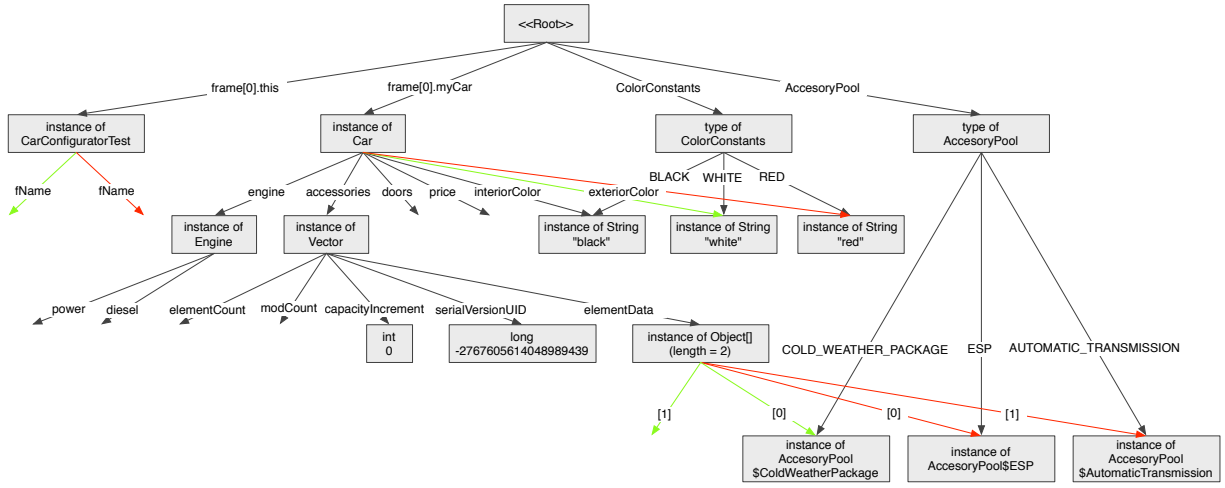


Abbildung 3.2: »Car-Configurator«-Beispiel: Größter gemeinsamer Teilgraph. Die farbigen Kanten sind in beiden Graphen vorhanden, zeigen aber auf jeweils unterschiedliche Knoten.

welche Knoten und Kanten in beiden Graphen vorhanden sind. So wurden z.B. die Klassen `ColorConstants` und `AccessoryPool` in beiden Programmläufen geladen. Die in den Klassen definierten Konstanten sind ebenfalls Teil des MCS.

Nicht Teil des gemeinsamen Graphen und damit unterschiedlich in ihren jeweiligen Werten, sind die Knoten der Variablen `power`, `diesel`, `elementCount`, `modCount`, `fName` sowie `price`. Die Kanten, welche in Grün bzw. Rot gezeichnet sind, bestehen im funktionierenden bzw. im fehlerhaften Lauf, zeigen allerdings auf unterschiedliche Wertknoten in den beiden Graphen. Wenn wir den Programmzustand später übertragen, werden wir diese Variablen entsprechend anpassen müssen. Dies bedeutet, dass wir die Referenzen, von den Werten, auf welche die grünen Kanten zeigen, auf die Werte, auf welche die roten Kanten zeigen, umbiegen müssen. Handelt es sich bei den Referenzen um Referenzen auf primitive Werte, so müssen wir die entsprechenden primitiven Werte anpassen.

3.2 Klassifizierung der Speichergraphunterschiede

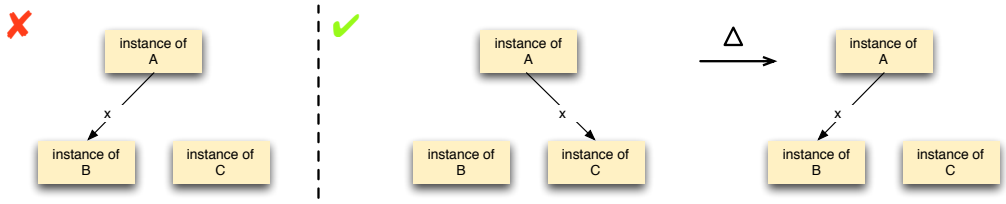
Bevor wir beschreiben, wie wir die Unterschiede zwischen zwei Graphen bestimmen, schauen wir uns an welche Unterschiede bzw. Deltas zwischen zwei Programmzuständen bestehen können. Wir werden untersuchen an welchen Stellen und wie sich zwei Programmzustände unterscheiden können. In Abschnitt 3.3 werden wir dann die verschiedenen Unterschiede in einem Objektmodell abbilden. In Abschnitt 3.4 beschreiben wir, wie wir die Unterschiede in konkreten

Deltas erfassen. Das Ergebnis, eine Menge von Deltas, repräsentiert alle Unterschiede zwischen zwei Speichergraphen.

Ein Programmzustand besteht aus Variablen bzw. Referenzen und deren Werte. Werte können wir in Objekte und primitive Werte klassifizieren. Ein Objekt wiederum besteht aus einer Menge von Referenzen auf weitere Objekte oder primitive Werte. Um einen Programmzustand in einen zweiten zu überführen können wir

- Objektreferenzen verbiegen,
- Objekte und Arrays erzeugen,
- Objektreferenzen löschen,
- Primitive Werte verändern und
- die Größe von Arrays anpassen.

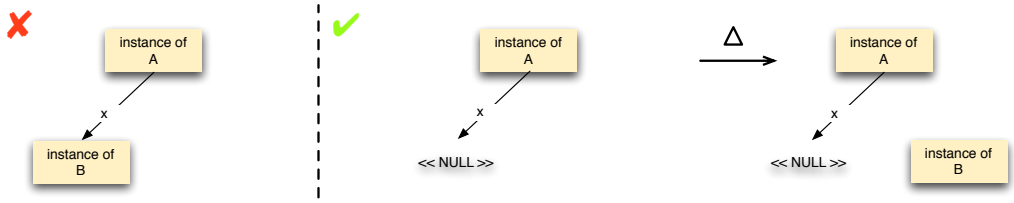
Verbiegen von Objektreferenzen



Jede Objektreferenz, und damit auch die `null`-Referenzen, können auf ein bestehendes Objekt im Speicher verwiesen werden. Hierzu verändern wir die Referenz, indem wir ihr den Wert des neuen Objekts zuweisen. Die Referenz wird damit auf die Stelle im Speicher verbogen, in dem sich der zugewiesene Wert befindet. Im Gegensatz zu C ist in Java ein direkter Zugriff auf den Programmspeicher nicht erlaubt. Um eine Objektreferenz auf ein bestehendes Objekt zu verweisen, benötigen wir immer die entsprechende Instanz des Objektes.

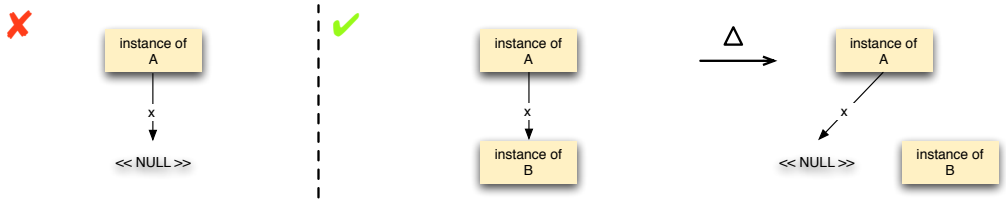
Bei dem Verbiegen der Objektreferenz muss folgende Voraussetzung erfüllt sein: Der *deklarierte Typ* der Objektreferenz muss dem *konkreten Typ* des Objekts bzw. dessen Interfaces, oder eines seiner Oberklassen bzw. deren Interfaces entsprechen. Wenn wir eine Objektreferenz verbiegen benötigen wir zwei Informationen: Welche *Variable* müssen wir auf welchen *Wert* verbiegen.

Erzeugen von Objekten



Existiert in $G_{\text{✘}}$ ein Objekt, welches in $G_{\text{✔}}$ nicht existiert, so müssen wir dieses erzeugen. Nachdem wir das Objekt erzeugt haben, können wir eine oder mehrere Variablen auf dieses Objekt verweisen. Um ein Objekt zu erzeugen müssen wir wissen, welcher *konkrete Typ* das neue Objekt haben soll. Handelt es sich bei dem Objekt um ein Array, benötigen wir zusätzlich die Angabe der *Arraygröße*.

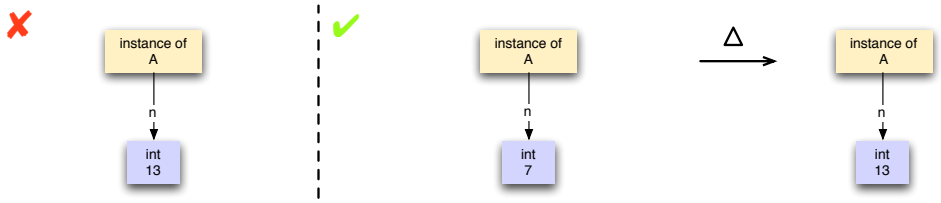
Löschen von Objektreferenzen



Objektreferenzen können wir löschen, indem wir der Referenz den Wert `null` zuweisen. Um eine Objektreferenz auf `null` zu setzen, genügt die Angabe der entsprechenden Variable.

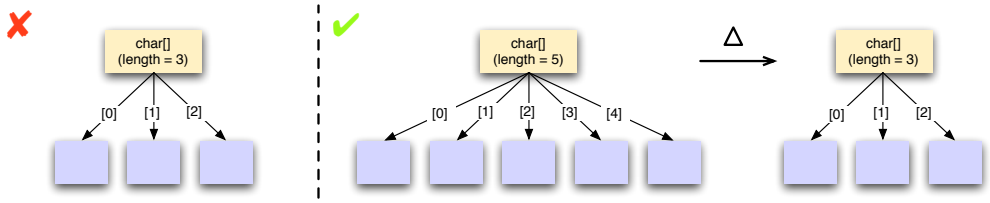
Wenn wir eine Objektreferenz auf `null` setzen, wird nicht das zuvor referenzierte Objekt aus dem Programmspeicher gelöscht. Das Löschen von Objekten aus dem Programmspeicher ist ganz und gar dem Speichermanagement von Java bzw. der Virtual Machine überlassen. Java setzt dabei das Verfahren der Automatischen Speicherbereinigung (engl. Garbage-Collection) ein. Damit die JVM ein Objekt aus dem Programmspeicher entfernen kann, müssen alle Referenzen auf das Objekt gelöst werden. Wann letztendlich das Objekt aus dem Programmspeicher entfernt wird, liegt rein in der Implementierung des Garbage-Collectors und kann sich von der jeweiligen Implementierung der Virtual Machine unterscheiden. Sollte es nötig sein ein Objekt aus dem Speicher zu löschen, geschieht dies, indem wir alle Objektreferenzen, welche auf die entsprechende Objektinstanz verweisen, auf `null` setzen.

Verändern von primitiven Variablen



Die primitiven Variablen sind das Gegenstück zu den Objektreferenzen. Durch Angabe der Variable und des neuen Wert, können wir jeder Zeit den Wert einer primitiven Variable ändern. Im Gegensatz zu Objektreferenzen können wir den Wert nicht löschen, indem wir den Wert `null` zuweisen. Primitive Variablen besitzen somit immer einen Wert, der dem deklarierten Typ der Variable entspricht. Wird in einer Klasse bei Initialisieren eines Objekts die primitive Variable nicht explizit mit einem Wert vorbelegt, so bekommt sie einen von Java definierten Standardwert zugewiesen (siehe Lindholm u. Yellin, 1999, Kapitel 2.5). Gleiches geschieht auch bei der Initialisierung eines primitiven Arrays.

Verändern der Arraygröße



Arrays als Spezialisierung von Objekten behandeln wir gesondert. Im Gegensatz zu zwei Objekten gleichen Typs, welche die gleichen Attribute definieren, wird bei Arrays vom gleichen Typ die Anzahl der Attribute durch die Größe des Arrays bestimmt. Dies bedeutet, wir müssen bei dem Vergleich zweier Arrays vom gleichen Typ, die Anzahl der Arrayelemente mit berücksichtigen und eventuell anpassen.

3.3 Das Objektmodell der Graphdeltas

Die Unterschiede auf primitiven Werten, Objektreferenzen und Objekten finden sich in dem Klassendiagramm des Interface `IMemoryGraphDelta` wieder (siehe Abbildung 3.3). Eine primäre Unterteilung der Zustandunterschiede erfolgt in die Subschnittstellen `ICreateDelta`,

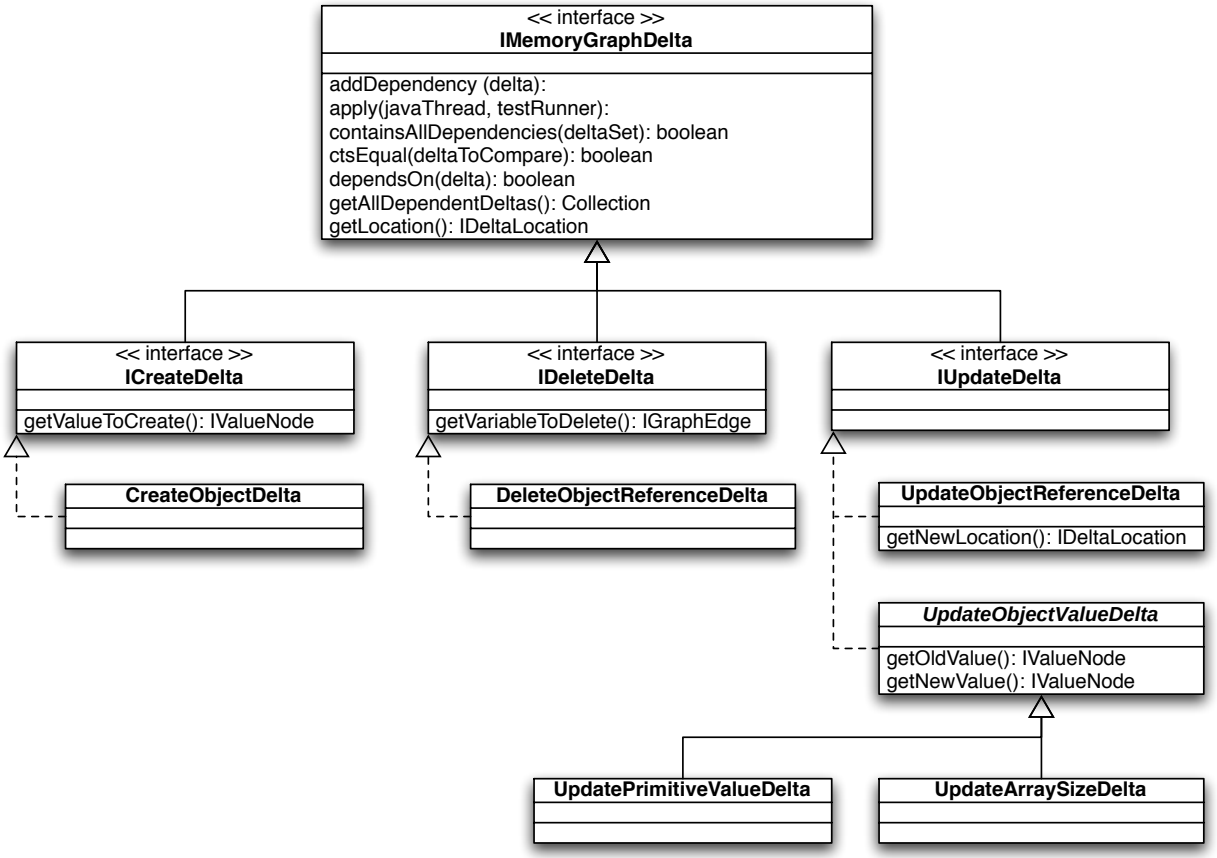


Abbildung 3.3: UML Klassendiagramm: `IMemoryGraphDelta`

`IDeleteDelta` und `IUpdateDelta` zum Erzeugen, Löschen und Verändern von Graphelementen bzw. deren Pendant im Speicher.

Das Interface `IMemoryGraphDelta` deklariert alle Funktionen, die jedes Graphdelta implementieren muss. Die drei Interessantesten seien hier erwähnt:

- `apply(IJavaThread, ITestRunner)`: Mit Hilfe der `apply`-Funktion wir das entsprechende Delta auf dem Programmzustand des übergebenen Java-Threads angewendet. Der *Testrunner* ist ein Proxy für den entsprechenden Testlauf. Mit Hilfe des Testrunners können wir einen Testlauf steuern und neue Objekte erzeugen (siehe Kapitel 4).

- `getLocation()`: Die *Location* beschreibt eindeutig den Wert, das Objekt oder die Variable innerhalb des Programmzustandes, an den das jeweilige Delta angewendet werden soll. Um ein Graphelement eindeutig im Speichergraphen wieder zu finden, suchen wir nach einem Pfad von dem Rootknoten bis zu dem Element. Durch diesen Pfad können wir das jeweilige Element und somit die Location des Deltas eindeutig spezifizieren.

- `ctsEqual(IMemoryGraphDelta)`: Diese Funktion überprüft, ob zwei Deltas gemäß dem Cause Transition Algorithmus gleich sind. Gleichheit zweier Deltas bedeutet, dass beide Deltas die gleiche Variable oder das gleiche Objekt im Speicher verändern oder erzeugen. Wir überprüfen dies, indem wir die Location beider Deltas miteinander vergleichen. Wir benötigen diese Funktion, um von zwei minimalen fehlerrelevanten Deltamengen entscheiden zu können, ob sie die gleichen Deltas definieren. Wenn die Deltamengen nicht gleich sind liegt ein Ursachenübergang vor (Cleve u. Zeller, 2005).

Die in `IMemoryGraphDelta` deklarierten Methoden, welche hier nicht näher erläutert wurden, sind Hilfsfunktionen und bieten die Möglichkeit Abhängigkeiten zwischen Deltas zu beschreiben (siehe Abschnitt 3.6). Diese Abhängigkeiten liefern uns Kontextinformationen, um beim Anwenden einer Teilmenge aller Deltas entscheiden zu können, ob die entsprechende Teilmenge gültig ist oder nicht. Somit können wir vor dem Ausführen eines Testlaufes entscheiden, ob die Anwendung aller Graphunterschiede einer Deltamenge möglich ist oder nicht.

3.4 Bestimmen der Speichergraphunterschiede

Nachdem wir gesehen haben, wie wir einen gemeinsamen Teilgraphen zwischen zwei Speichergraphen berechnen und wir die möglichen Deltas klassifiziert haben, müssen wir die Menge aller Deltas berechnen. Die Unterschiede dienen später dazu einen Programmzustand in einen anderen zu überführen. Ohne Beschränkung der Allgemeinheit der Algorithmen gehen wir bei den folgenden Beschreibungen davon aus, dass wir einen Programmzustand $P_{✔}$, des funktionierenden Laufes, in einen Programmzustand $P_{✘}$, des fehlerhaften Laufes, überführen.

Zur Berechnung der Deltas auf den extrahierten Speichergraphen verwenden wir die Funktion `computeDeltaSet`, welche in dem Interface `IGraphElement` definiert ist. Wir starten mit der Berechnung bei den Rootknoten der Speichergraphen. Rekursiv werden wir dann die Menge aller Unterschiede zwischen den beiden Speichergraphen extrahieren.

Damit wir die Graphelemente in den Speichergraphen nicht mehrfach vergleichen oder sogar in eine Endlosrekursion laufen, werden wir jedes Element nach dem Vergleich kennzeichnen. Dies geschieht über ein Markerattribut, welches in den Graphelementen gesetzt werden kann. Durch das Markerattribut werden wir das Ergebnis des Vergleiches abspeichern. Das Attribut und somit auch das damit verbundene Graphelement können folgende Zustände annehmen:

- `NOT_COMPARED`: Das Graphelement wurde noch nicht verglichen. Dies ist der initiale Zustand des Markerattributs. Nachdem wir die Unterschiede zwischen zwei Speichergraphen berechnet haben, darf in den Graphen kein Element als `NOT_COMPARED` markiert sein.

- `EQUAL`: Zwei Graphelemente, jeweils ein Element aus $G_{✔}$ und $G_{✘}$, werden als `EQUAL` markiert, genau dann wenn beide Elemente Teil des gemeinsamen Speichergraphen sind und sie miteinander verknüpft wurden.

- `ONLY_IN_FIRSTGRAPH`: In diesem Zustand befindet sich ein Graphelement, wenn das Element nicht Teil des gemeinsamen Teilgraphen ist und nur in $G_{\checkmark}$ existiert.
- `ONLY_IN_SECONDGRAPH`: Genau wie der Zustand `ONLY_IN_FIRSTGRAPH` nur für Elemente in $G_{\checkmark}$ gesetzt werden kann, können wir nur Elemente des zweiten Graphen mit `ONLY_IN_SECONDGRAPH` markieren. Nach dem Vergleich beider Graphen muss das Markerattribut jedes Graphelement aus $G_{\times}$, welches nicht Bestandteil des gemeinsamen Teilgraphen ist, auf `ONLY_IN_SECONDGRAPH` gesetzt worden sein.
- `DIFFERENT`: Graphelemente, welche als `DIFFERENT` markiert werden, bestehen in beiden Graphen, sind aber bezüglich der Funktion `graphEquals` nicht gleich (vgl. Abschnitt 3.1). Dieses können z.B. zwei primitive Graphknoten mit unterschiedlichem Werten sein, oder ein Array, welches in beiden Graphen besteht aber unterschiedlich viele Elemente enthält.

Um vollständig $P_{\checkmark}$ in $P_{\times}$ zu überführen müssen alle Elemente, die als `ONLY_IN_FIRSTGRAPH` markiert wurden, gelöscht werden. Weiterhin müssen wir alle Elemente, welche als `ONLY_IN_SECONDGRAPH` markiert wurden, erzeugen. Graphelemente, welche als `DIFFERENT` markiert wurden, müssen in $P_{\checkmark}$ angepasst werden.

Bevor wir uns der Manipulation von Programmzuständen widmen, welche in Kapitel 4 beschrieben ist, müssen wir die Zustandsunterschiede berechnen. Wie schon erwähnt geschieht dieses über die Funktion `computeDelatSet`. Im Folgenden schauen wir uns an, was wir bei dem Vergleich der verschiedenen Graphelemente beachten müssen. Voraussetzung für den Vergleich ist, dass wir nur Graphelemente vom gleichen Typ miteinander vergleichen. Für Kanten gilt zusätzlich, dass die Quellknoten der Kanten gleich sind und die Kanten den gleichen Namen tragen müssen.

Vergleich von Graphknoten

Bei dem Vergleich zweier Knoten betrachten wir, ob sich die Knoten in Wert bzw. Typ unterscheiden. Sind beide Knoten gleich, fahren wir mit dem Vergleich bei den Kindern der Knoten fort. Ansonsten erstellen wir ein Delta, welches den entsprechenden Unterschied repräsentiert.

Vergleich von Rootknoten

Zwei Rootknoten sind per Definition gleich und werden somit als `EQUAL` markiert. Die Kinder eines Rootknoten sind die Basisvariablen. Wie zuvor schon erwähnt, können in beiden Speichergraphen durchaus unterschiedliche Basisvariablen bestehen. Um dies zu verhindern, sollten wir die Programmzustände an Stellen mit identischen Callstack extrahieren. Da dies nicht immer möglich ist, werden wir nur die Basisvariablen vergleichen, die in beiden Graphen identische Namen tragen. Alle anderen Basisvariablen können wir nicht übertragen.

Vergleich von Typknoten

Der Vergleich zweier Typknoten ist sehr einfach. Die Vorraussetzung dabei ist, dass wir jeweils die Typknoten vergleichen, welche die gleiche Klasse bzw. das gleiche Interface repräsentieren. Unter dieser Prämisse sind zwei Typknoten immer gleich und zur Berechnung der Graphunterschiede betrachten wir die ausgehenden Kanten der beiden Knoten. Da beide Typknoten gleich sind, sind die in der Klasse bzw. in dem Interface definierten statischen Variablen auch die Gleichen.

Vergleich von Null-Knoten

Zwei Knoten vom Typ `INullValueNode` sind immer gleich und werden daher als `EQUAL` markiert. Die Rekursion endet bei dem Vergleich von `null`-Knoten, da von diesen keine weiteren Objekte oder primitiven Werte referenziert werden.

Vergleich von primitiven Knoten

In den bisher betrachteten Knotentypen haben wir noch keine Unterschiede festgestellt. Untersuchen wir zwei primitive Werte, so können sich diese unterscheiden. Wenn ein Unterschied besteht, erzeugen wir ein Delta (`UpdatePrimitiveValueDelta`).

Das Delta wird mit beiden primitiven Graphknoten initialisiert. Den Knoten aus $G_{\checkmark}$ benötigen wir, um die Location innerhalb von $P_{\checkmark}$ zu identifizieren. Die Location definiert die Variable deren Wert wir ändern müssen, um den Programmzustand zu übertragen. Der primitive Knoten aus $G_{\times}$ enthält die Information über den neuen Wert, welcher der primitiven Variable in $P_{\checkmark}$ zugewiesen werden soll. Unterscheiden sich die beiden Graphknoten in ihren Werten werden sie als `DIFFERENT` gekennzeichnet, andernfalls als `EQUAL`.

Vergleich von Objektknoten

Der Vergleich zweier Objektknoten setzt voraus, dass beide Objektknoten ein Objekt des gleichen Typs im Speicher repräsentieren. Dadurch ist sichergestellt, dass das Objekt die gleichen Attribute definiert. Die Objektknoten selbst werden somit mit `EQUAL` markiert, die Rekursion folgt mit der Berechnung der Unterschiede auf den ausgehenden Kanten.

Vergleich von Arrayknoten

Arrays als Spezialisierung von Objekten müssen wir getrennt betrachten. Wenn zwei Arrays den gleichen Typ besitzen ist nicht sichergestellt, dass sie sich nicht in der Anzahl ihrer Elemente unterscheiden. Betrachten wir dazu zwei Arrays vom gleichen Typ (z.B. `int[]`), so ist aus dem Typ nicht ersichtlich wie viele Elemente die beiden Arrays enthalten. Im Gegensatz zu Objekten, die wenn sie den gleichen Typ deklarieren auch die gleichen Attribute besitzen, müssen wir bei den Arrays auf die Anzahl der jeweiligen Elemente Rücksicht nehmen. Wir differenzieren daher beim Vergleich zweier Arrayknoten ohne Beschränkung der Allgemeinheit drei verschiedene Zustände:

- Die Größe beider Arrays sind gleich:

  Wenn dies der Fall ist, können wir die Arrays wie Objektknoten behandeln und die jeweiligen ausgehenden Kanten miteinander vergleichen. Dabei vergleichen wir jeweils die Kanten, welche den gleichen Namen tragen. Dies entspricht aber den Arrayelementen, welche in beiden Arrays den gleichen Index besitzen. An dem Array selbst müssen keine Änderungen vorgenommen werden, sie werden als `EQUAL` markiert.

- Das Array in $G_{✓}$ ist größer als in $G_{✘}$:

  Wir erstellen ein `UpdateArraySizeDelta`, das den Größenunterschied beschreibt und markieren beide Arrayknoten als `DIFFERENT`. Wenn wir das Delta anwenden sollte die Arraygröße von dem Array in $G_{✓}$ auf Größe des Arrays in $G_{✘}$ reduziert werden.

  Den Vergleich setzen wir fort, indem wir `computeDeltaSet` auf allen Nachfolgern, welche in beiden Graphen bestehen, aufrufen. Die Nachfolger, die nur in dem $G_{✓}$ vorhanden sind, werden als `ONLY_IN_FIRSTGRAPH` markiert. Wir müssen keine Deltas erzeugen, um diese Elemente zu löschen. Wenn das Array verkleinert wird, fallen die Referenzen auf die entsprechenden Werte oder Objekte weg.

- Das Array in $G_{✓}$ ist kleiner als in $G_{✘}$:

  Wir erstellen ein `UpdateArraySizeDelta` und suchen rekursiv nach Deltas in den Elementen beider Arrays. Zusätzlich müssen wir Deltas erzeugen, die uns die Arrayelemente erzeugen, welche ausschließlich in $G_{✘}$ vorhanden sind.

  Handelt es sich um ein primitives Array, wird durch die Vergrößerung des Arrays, alle Arrayelemente mit dem Defaultwert vorbelegt. Wir erstellen pro zusätzlichen Wert des Arrays in $G_{✘}$ ein `UpdatePrimitiveValueDelta` um die Werte des Arrays in $P_{✓}$ anzupassen. Außerdem vergleichen wir die Elemente, die in beiden Arrays vorkommen. Dieses geschieht durch den Aufruf von `computeDeltaSet` auf den jeweiligen ausgehenden Kanten der Arrayknoten (siehe Abbildung 3.4).

  Für primitive Arrays war der Vergleich ja noch recht einfach. Aber was müssen wir tun, wenn es sich bei dem Array um ein Objektarray handelt? Grundsätzlich bestehen die Elemente von Objektarrays aus Referenzen auf andere Objekte oder `null`-Referenzen. Die

Vorgehensweise ist daher zunächst wie bei den primitiven Arrays. Wir erstellen ein Delta um den Größenunterschied zwischen beiden Arrays zu beschreiben, danach vergleichen wir alle Nachfolger, welche in beiden Arrays vorkommen.

Aber wie behandeln wir die Referenzen, die in dem Array aus $P_{✔}$ nicht bestehen? Dazu schauen wir uns den Zielknoten der jeweiligen Kante an, welche die entsprechende Arrayreferenz in $G_{✘}$ repräsentiert. Wir unterscheiden drei Möglichkeiten:

- Die Kante zeigt auf einen `null`-Knoten. Wir müssen nichts Weiteres beachten, da beim Vergrößern des Objektarrays die zusätzlichen Referenzen mit dem Standardwert `null` vorbelegt werden.
- Die Kante zeigt auf einen Knoten, welcher Teil des gemeinsamen Graphen ist. Dies bedeutet zum Knoten in $G_{✘}$ gibt es einen verknüpften Knoten in $G_{✔}$. Wir erstellen dann ein `UpdateObjectReferenceDelta`, um die Referenz in dem Array auf den entsprechenden Knoten in $G_{✔}$ umzubiegen.
- Die Kante zeigt auf einen Knoten, welcher nur in $G_{✘}$ besteht. Es genügt nun nicht nur das entsprechende Objekt des Knoten zu erzeugen und zu referenzieren. Das Objekt kann weitere Objekte referenzieren, die ebenfalls in $G_{✔}$ nicht existieren. Deshalb müssen wir eine Menge von Deltas definieren, die den kompletten, in $G_{✔}$ nicht vorhandenen, Objektgraphen erzeugt. Die genaue Vorgehensweise ist in Abschnitt 3.5 beschrieben.

Vergleich von Kanten

Wir haben gesehen, wie wir die verschiedenen Knotenarten miteinander vergleichen. Wenn diese Knoten Nachfolger (z.B. Objektattribute, Arrayelemente) besitzen, müssen wir auch für diese die Unterschiede bestimmen. Dies geschieht durch den Aufruf der Funktion `computeDeltaSet` auf den Kanten, welche die entsprechenden Nachfolger verknüpfen. Voraussetzung zum Vergleich zweier Kanten ist, dass die Quellknoten der Kanten Teil des gemeinsamen Teilgraphen sind und die Kanten den gleichen Namen tragen.

Bei dem Vergleich zweier Kanten betrachten wir die Zielknoten, auf die beide Kanten zeigen. Wir unterscheiden drei Szenarien:

- Beide Zielknoten sind Teil des gemeinsamen Teilgraphen und somit miteinander verknüpft: Wir kennzeichnen die Kante als `EQUAL` und fahren mit der Berechnung der Graphunterschiede an den beiden Zielknoten fort. Dieses Szenario trifft immer zu, wenn die Zielknoten primitive oder Typknoten sind, oder wenn die Zielknoten Objektknoten vom gleichen Typ sind.
- Die Kante aus $G_{✘}$ verweist auf einen `INullValueNode` und die Kante aus $G_{✔}$ auf einen Objektknoten: Wir müssen der entsprechenden Objektreferenz in $G_{✔}$ `null` zuweisen. Den Unterschied beschreiben wir durch ein `DeleteObjectReferenceDelta`.

Betrachten wir den Zustand zweier primitiver Arrays mit der Variablenbelegung wie sie in folgender Tabelle dargestellt ist:

Programmzustand	Variable			Wert			
	Name	Typ	Größe	`a[0]`	`a[1]`	`a[2]`	`a[3]`
$P_{✔}$	`a`	`int[]`	2	3	1	-	-
$P_{✘}$	`a`	`int[]`	4	3	4	2	5

Um den Zustand von $P_{✔}$ nach $P_{✘}$ zu übertragen, müssen wir folgende Änderungen an $P_{✔}$ vornehmen:

Delta		Variable	Alter Wert	Neuer Wert
Δ_1	`UpdateArraySizeDelta`	`a`	2	4
Δ_2	`UpdatePrimitiveValueDelta`	`a[1]`	1	4
Δ_3	`UpdatePrimitiveValueDelta`	`a[2]`	-	2
Δ_4	`UpdatePrimitiveValueDelta`	`a[3]`	-	5

Zunächst erhöhen wir die Arraygröße von 2 auf 4. Danach müssen wir die Variable `a[1]` von 1 auf 4 ändern. Die Varialben `a[2]` und `a[3]` werden bei der Vergrößerung der Arraygröße automatisch mit 0 vorinitialisiert. Wir müssen sie danach auf die Werte 2 bzw. 5 setzen.

Abbildung 3.4: Beispiel: Übertragen des Zustands eines primitiven Arrays

- Die Kante aus $G_{✘}$ verweist auf ein Objektknoten, welcher nicht mit dem Zielknoten aus $G_{✔}$ verknüpft ist: Wir müssen unterscheiden, ob das Objekt, auf das die Kante in $G_{✘}$ verweist, Element des gemeinsamen Teilgraphen ist oder nicht. Wenn ja, können wir mittels `UpdateObjectReferenceDelta` die Kante in $G_{✔}$ auf das zu dem in $G_{✘}$ verknüpften Objekt umbiegen. Wenn nicht müssen wir das Objekt und gegebenenfalls all seine Unterobjekte erzeugen (siehe Abschnitt 3.5).

3.5 Bestimmen der Deltas zum Erstellen von Objektteilgraphen

Wir haben gesehen, wie wir Knoten und Kanten vergleichen können, die in beiden Speichergraphen existieren. Aber was geschieht mit Knoten, welche nur in dem fehlerhaften Graph bestehen. Um den kompletten Programmzustand zu übertragen müssen wir diese Knoten bzw. ihre Objekte erzeugen.

Beim Vergleich zweier Arrays und zweier Kanten haben wir gesehen, dass es nötig sein kann ein oder mehrere Objekte zu erzeugen. Doch was ist mit den Objekten, die von diesen Objek-

ten referenziert werden. Diese können bereits vorhanden sein, müssen aber nicht. Dies bedeutet, wenn wir ein Objekt erzeugen, betrachten wir alle von dem Objekt referenzierten Objekte und erzeugen diese wenn nötig auch. Es stellt sich also die Frage, wie generieren wir komplette Objektstrukturen, welche nur in $G_{✘}$ bestehen. Was wir berechnen möchten sind alle Unterschiede oder Deltas, um später die Objektstruktur in dem funktionierenden Programmzustand reproduzieren zu können.

Der Plan ist: Für jeden Objektknoten (inklusive der Arrayknoten), welche nur in $G_{✘}$ vorhanden sind, erstellen wir ein `CreateObjectDelta`. Weiterhin müssen wir uns um die Attribute der neu erzeugten Objekte kümmern. Wenn die Attribute primitive Werte darstellen, erzeugen wir dementsprechend für jedes Attribut ein `UpdatePrimitiveValueDelta`. Für jede Objektreferenz, die keine `null`-Referenz darstellt, erzeugen wir ein `UpdateObjectReferenceDelta`.

Betrachen wir uns einmal den $G_{✘}$ und $G_{✔}$ aus Abbildung 3.5. Wenn wir $G_{✔}$ komplett in $G_{✘}$ transformieren möchten, müssen wir zunächst jeweils ein Objekt vom Typ `Car` und `Engine` erstellen (1. `CreateObjectDelta`). Angenommen wir erstellen die Objekte im nicht initialisierten Zustand, d.h. keinem Objektattribut wurde ein Wert zugewiesen, so sind die Objektreferenzen `null`-Referenzen, die primitiven Variablen mit ihren jeweiligen Standardwerten vorbelegt (z.B. `false` für ein `boolean`-Attribut). Im zweiten Schritt ändern wir die primitiven Attribute, so dass sie mit den Objekten in $G_{✘}$ übereinstimmen (2. `UpdatePrimitiveValueDelta`). Erst im letzten Schritt weisen wir den entsprechenden Objektreferenzen, die neu erzeugten Objekte zu (3. `UpdateObjectReferenceDelta`).

3.6 Details der Implementierung

Abhängigkeiten zwischen Deltas

Zwischen allen berechneten Graphdeltas können Abhängigkeiten bestehen. Diese Abhängigkeiten haben wir zuvor nicht beachtet. Untersuchen wir die Abhängigkeiten an folgendem Beispiel:

Wir vergleichen die Variable `carengine` zweier Programmzustände. In $P_{✔}$ ist `carengine` eine `null`-Referenz, in $P_{✘}$ eine Referenz auf eine Objektinstanz der Klasse `Engine`. Diese Objektinstanz hat zwei Attribute `power` und `diesel` welche mit den Werten `150` bzw. `true` belegt sind:

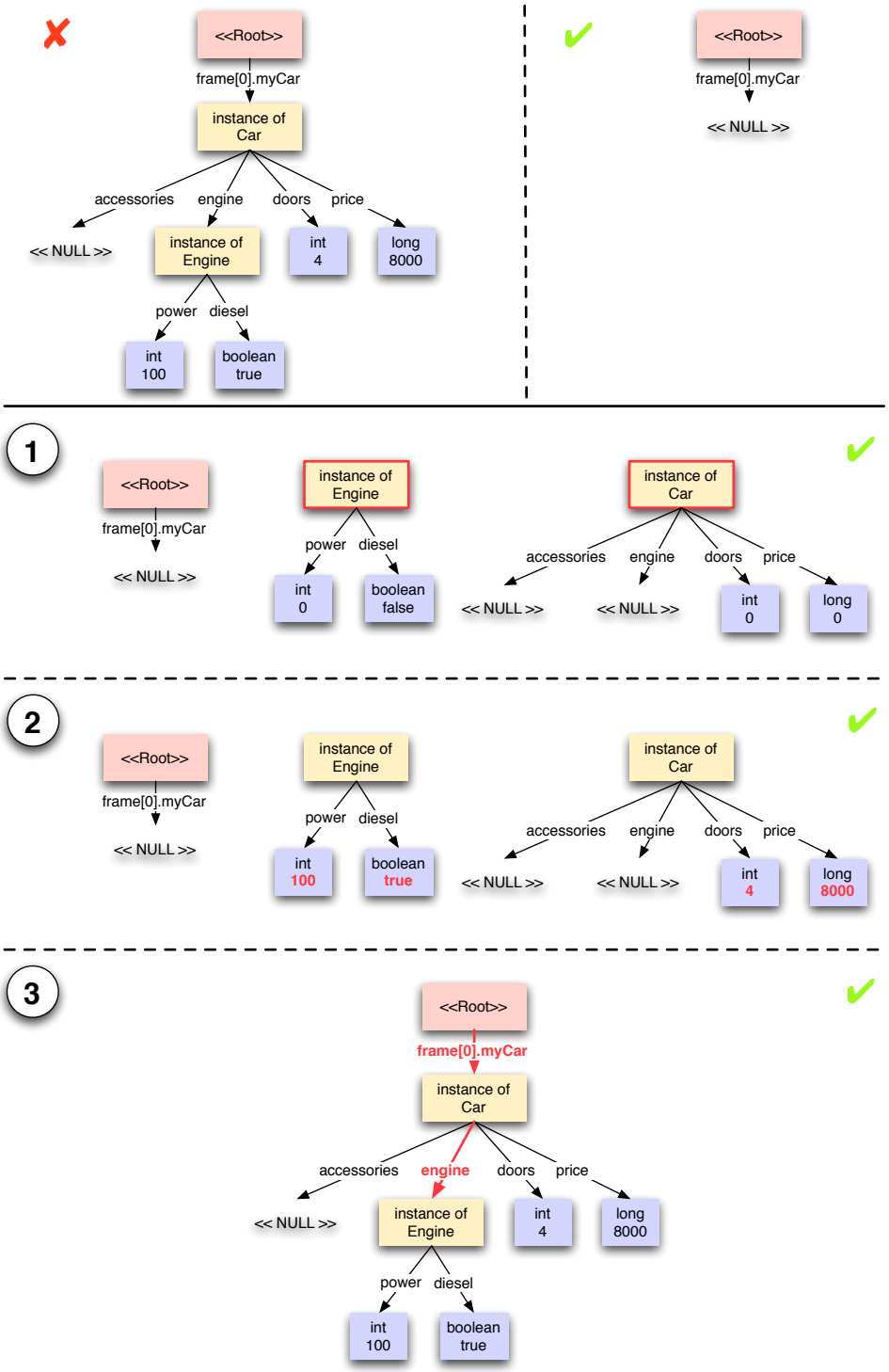


Abbildung 3.5: Beispiel: Sequentielles Erstellen eines Objektgraphen

Programmzustand	Variable	Typ	Wert
$P_{✔}$	`carengine`	`Engine`	`null`
$P_{✘}$	`carengine`	`Engine`	`64dc11`
	`carengine.power`	`int`	`150`
	`carengine.diesel`	`boolean`	`true`

Um den Programmzustand $P_{✔}$ in $P_{✘}$ zu überführen sind folgende Schritte notwendig:

Δ_1 Erstellen einer Objektinstanz der Klasse `Engine`:
`CreateObjectDelta`

Δ_2 Verweisen der Variable `carengine` auf die neue Objektinstanz:
`UpdateObjectReferenceDelta`

Δ_3 Setzen des Attributes `power` der neu erstellten Instanz mit dem Wert `150`:
`UpdatePrimitiveValueDelta`

Δ_4 Setzen des Attributes `diesel` der neu erstellten Instanz mit dem Wert `true`:
`UpdatePrimitiveValueDelta`

Für jeden dieser Schritte erstellen wir ein Delta. Die Abhängigkeiten sind klar ersichtlich. Wir können weder eines der Objektattribute setzen noch die Variable auf die Objektinstanz umbiegen, wenn wir das Objekt zuvor nicht erzeugt haben. Mit Hilfe der Methode `addDependency` des Interfaces `IMemoryGraphDelta` (siehe Abbildung 3.3) können wir Abhängigkeiten zwischen zwei Deltas definieren. Um zu bestimmen, dass Δ_2 nicht angewendet werden kann, ohne dass Δ_1 angewendet wurde, fügen wir Δ_1 zu der Liste der abhängigen Deltas von Δ_2 hinzu. Dieses geschieht über den Aufruf Δ_2`.addDependency(`Δ_1`).`

Bei der Berechnung der Deltas können wir an vielen Stellen strukturelle Abhängigkeiten zwischen den Deltas bestimmen. Zum Beispiel wenn wir ein Array vergrößern: Wir können die primitiven Werte oder Objektreferenzen, der neu erzeugten Arrayelemente erst verändern, wenn wir das Array vergrößert, d.h. das entsprechende `UpdateArraySizeDelta` ausgeführt haben. Weiterhin sind, wie an dem obigen Beispiel gesehen, die Abhängigkeiten hilfreich beim Erzeugen von Objektstrukturen. Attribute von neu erzeugten Objekten lassen sich erst verändern, wenn wir das `CreateObjectDelta` zum Erzeugen der Objekte angewendet haben.

Aber wozu benötigen wir die Abhängigkeiten zwischen den Deltas? Durch die Anwendung von Delta Debugging werden Teilmengen aller berechneten Zustandsunterschiede getestet, ob sie zu dem fehlschlagenden Lauf führen oder nicht. Dabei wird beim Bestimmen der Teilmengen nicht auf strukturelle Abhängigkeiten geachtet. Mit Hilfe der Definition von Abhängigkeiten zwischen den Deltas, können wir das Finden der fehlerrelevanten Unterschiede zweier Programmzustände

beschleunigen, indem wir die jeweilige zu testende Teilmenge der Deltas auf ihre Abhängigkeiten prüfen. Wenn nicht alle benötigten Deltas in der Teilmenge vorhanden sind müssen wir den entsprechenden Test gar nicht anstarten. Dadurch können wir die Gesamtzahl der auszuführenden Tests und damit die Ausführungszeit reduzieren.

3.7 Fazit

In diesem Kapitel haben wir die Unterschiede zwischen zwei Speichergraphen bestimmt. Diese so genannten Deltas beschreiben Unterschiede in der Struktur der beiden Graphen und Unterschiede in den jeweiligen Werten der Graphknoten. Gleichzeitig repräsentieren die Deltas die Unterschiede der Programmzustände, von denen wir die Speichergraphen extrahiert haben. Durch das Anwenden der Deltas auf den Programmzustand des funktionierden Laufs $P_{✔}$, d.h. durch Verändern von primitiven Werten, Umbiegen oder Löschen von Objektreferenzen und Erzeugen neuer Objekte, sind wir in der Lage $P_{✔}$ in den Programmzustand $P_{✘}$ des fehlerhaften Laufes zu überführen (siehe Kapitel 4). Allerdings gibt es auch Einschränkungen. Mit den gegebenen Mitteln eines Debuggers sind wir nicht in der Lage den kompletten Programmzustand nach belieben zu verändern:

- Wir können keine Veränderungen an der Struktur oder Anzahl der Basisvariablen vornehmen. Wie wir wissen, werden die Basisvariablen aus den Frames des angehaltenen Thread extrahiert. Es sind die Variablen, welche in den verschiedenen Funktionen der Frames definiert wurden. Wir können den Frames keine neuen Variablen hinzufügen oder bestehende löschen.

  D.h. die Basisvariablen in beiden Programmzuständen *müssen* die Gleichen sein (dies gilt nicht von den jeweiligen Basisvariablen referenzierten Werte). Dies erreichen wir, indem wir beide Programmläufe an Stellen anhalten, an denen die Callstacks identisch sind. Martin Mehlmann hat in seiner Diplomarbeit das Problem nicht identischer Callstacks und deren Auswirkungen auf die Fehlersuche zwischen zwei Programmzuständen näher untersucht (siehe Mehlmann, 2005). Das Ergebnis ist, dass es möglich ist auch bei nicht identischen Callstacks die fehlerrelevanten Unterschiede zu bestimmen. Allerdings müssen diese Callstacks gewisse Gleichheitskriterien erfüllen. Dies führt dazu, dass wir zusätzlich zu den identischen Callstacks zweier Programmläufe, nur eine geringe Anzahl nicht identischer Callstacks finden.

- Wenn wir zwei Programmzustände aus verschiedenen Programmläufen untersuchen ist es möglich, dass die von der JVM geladenen Klassen sich unterscheiden.

  Über den Klassenlader (engl. Classloader) sind wir in der Lage Klassen während der Programmausführung zu laden, aber nicht wieder zu »entladen«. Momentan berücksichtigen wir dieses Problem allerdings nicht, da es sicherlich nur selten Unterschiede in den von der JVM geladenen Klassen gibt. Wenn es Unterschiede gibt, haben diese nur in wenigen

Fällen Auswirkungen auf den Programmlauf und sind somit irrelevant für das Fehlschlagen des Programmlaufs. Beim Vergleich der Rootknoten werden wir Klassen, welche nur in einem der beiden Speichergraphen existieren ignorieren.

4 Manipulation von Programmzuständen

Wir können einen Programmzustand mit Hilfe eines Speichergraphen abbilden. Außerdem können wir zwei Speichergraphen miteinander vergleichen. Angenommen wir haben einen Programmzustand eines funktionierenden Laufs ($r_{✔}$) mit dem eines fehlerhaften Laufs ($r_{✘}$) verglichen. Welche Informationen enthalten die Unterschiede beider Programmzustände über den Fehler in $r_{✘}$? Die Antwort ist, dass die Unterschiede im Prinzip die Ursache für den Fehler darstellen. Was uns jedoch wirklich interessiert sind die *fehlerrelevanten Ursachen*[1], also die *minimalen* Unterschiede, die zu dem Fehler in $r_{✘}$ führen.

Diese minimalen Unterschiede werden durch Delta Debugging automatisch bestimmet (Zeller, 2005, Kapitel 13). Dazu benötigen wir:

- einen automatisierten Test der entscheidet, ob der Fehler auftritt oder nicht – hierfür verwenden wir den funktionierenden JUnit-Testfall.
- die Möglichkeit den funktionierenden Programmzustand durch die ermittelten Zustandsunterschiede teilweise in den fehlerhaften zu überführen.

Wir gehen dabei wie folgt vor. Wir starten den funktionierenden Programmlauf und lassen ihn bis an die Stelle laufen, an der wir den Programmzustand extrahiert haben. Danach manipulieren wir den Programmzustand, durch die *Anwendung* der Deltas. Wenn wir ein Delta *anwenden* bedeutet dies, dass wir die von dem Delta betroffenen Variablen, Werte oder Objekte bezüglich ihrer Konfiguration in dem funktionierenden Programmzustand in die Konfiguration des fehlerhaften Programmzustands überführen. Wir erzeugen somit einen *gemischten* Zustand mit Werten und Objekten sowohl aus $P_{✔}$ und $P_{✘}$. Anschließend wird der Programmlauf fortgeführt und überprüft, ob die Testmethode funktioniert (✔), fehlschlägt (✘) oder eventuell sogar ein unerwarteter Fehler auftritt (?). Wenn wir keine Deltas anwenden, bekommen wir den funktionierenden Lauf. Wenden wir alle Unterschiede an, erhalten wir den fehlerhaften Lauf. Der Delta Debugging Algorithmus minimiert, durch eine Art binäre Suche, von der Menge aller Zustandsunterschiede die Menge der fehlerrelevanten Unterschiede.

In diesem Kapitel beschreiben wir die Verfahren, um den Zustand anhand der ermittelten Deltas zu manipulieren. Entsprechend der in Kapitel 3 beschriebenen Deltas werden wir folgende Manipulationen im Programmzustand vornehmen:

[1] Eine ausführliche Beschreibung von *Ursachen* (engl. Causes), *Wirkungen* (engl. Effects) und die Zusammenhänge zwischen Ursachen und Wirkung findet sich in (Zeller, 2005, Kapitel 12)

- `UpdatePrimitiveValueDelta` – Zuweisung eines primitiven Werts zu einer primitiven Variable (Abschnitt 4.1)
- `DeleteObjectReferenceDelta` – »Löschen« einer Objektreferenz durch Zuweisung des Wert `null` (Abschnitt 4.2)
- `UpdateObjectReferenceDelta` – Verbiegen einer Objektreferenz auf im Programmzustand bestehende Objekte (Abschnitt 4.2)
- `CreateObjectDelta` – Erzeugen von einem Objekt (Abschnitt 4.3) oder Array (Abschnitt 4.4) eines beliebigen Typs
- `UpdateArraySizeDelta` – Verändern der Arraygröße (Abschnitt 4.4)

4.1 Manipulation primitiver Variablen

Primitive Variablen existieren an verschiedenen Stellen innerhalb des Programmzustands. Sie sind entweder Feldelemente eines Arrays, Attribute eines Objekts oder global definierte Klassenattribute. Wenn wir eine primitive Variable ändern, geschieht dies, indem wir ihr einen neuen Wert zuweisen. Doch bevor wir dies tun können, müssen wir die Variable, welche wir ändern möchten, im Programmzustand eindeutig spezifizieren. Betrachten wir dazu den jeweiligen Speichergraphen als Abstraktion des Programmzustands. Eine Variable im Programmzustand wird durch eine Kante im Speichergraphen abgebildet. Damit wir die Kante im Speichergraphen wiederfinden ist die Angabe des Kantennamens nicht ausreichend, da dieser nicht eindeutig ist.

Um die Kante im Speichergraphen eindeutig zu identifizieren, suchen wir einen Pfad zu der entsprechenden Kante. Dabei verwenden wir das Verfahren der Tiefensuche. Wir suchen einen Pfad zu dem Rootknoten und beginnen die Suche bei der entsprechenden Kante. Da alle Kanten und Knoten, ausgehend von dem Rootknoten extrahiert wurden, besteht mindestens ein Pfad, welcher den Rootknoten mit der Kante verbindet. Über diesen Pfad haben wir die Kante im Speichergraphen und somit die Variable im Programmzustand eindeutig spezifiziert.

Betrachten wir einen Ausschnitt des Speichergraphens von dem Fallbeispiel aus dem Einleitungskapitel (siehe Abbildung 4.1). Ein Unterschied in beiden Programmzuständen besteht z.B. in der Motorstärke der jeweiligen Fahrzeuge. Die Folge davon ist, dass wir bei dem Vergleich der beiden Programmzustände ein `UpdatePrimitiveValueDelta` (Δ_{power}) erhalten. Dieses Delta besagt, dass der Wert der Variable `power` in $P_{\checkmark}$ den Wert 100 und in $P_{\times}$ den Wert 150 beträgt. Da wir später den funktionierenden Lauf manipulieren möchten, suchen wir nach dem Pfad zu der Variable in $G_{\checkmark}$. Der einzige Pfad, beinhaltet, ausgehend von dem Rootknoten, die Kanten `frame[0].myCar`, `engine` und `power`. Die zu ändernde Variable wird dabei durch das letzte Glied in dem Pfad beschrieben.

Angenommen wir haben den funktionierenden Testfall neu gestartet und am gleichen Programmzeitpunkt angehalten, an dem wir zuvor $P_{\checkmark}$ extrahiert haben. Wir haben Δ_{power} und somit den

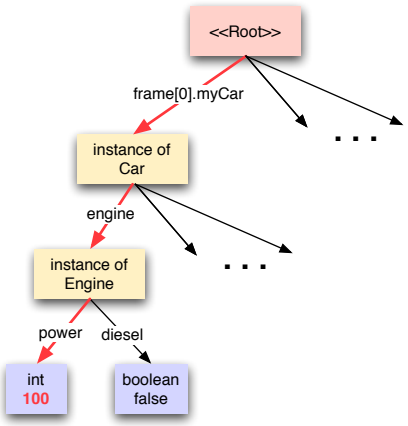


Abbildung 4.1: Auszug aus dem Speichergraph aus Abbildung 2.3. Die rot markierten Kanten beschreiben den Pfad im Graphen, um die Kante `power` im Speichergraphen eindeutig zu identifizieren.

Pfad zu der Variable berechnet. Wenn wir Δ_{power} anwenden, möchten wir der Variable den Wert aus dem fehlerhaften Programmzustand zuweisen. Zu diesem Zweck werden wir, ausgehend von dem angehaltenen Thread, die Variable `power` unter Verwendung des Pfads in dem Programmzustand wiederfinden (siehe Abbildung 4.2):

1. Die Kante `frame[0].myCar` enthält zwei Information: Über `frame[0]` wird eindeutig bestimmt, welchen Frame auf dem Callstack wir betrachten. Von dem Frame aus gelangen wir an die lokale Variable `myCar`. Durch den Funktionsaufruf `getValue` auf der Variable bekommen wir das Objekt, auf welches die Variable im Speicher zeigt (Zeile 1 – 6).

2. Unter Verwendung des konkreten `Car`-Objektes können wir auf die Feldvariable `engine` und deren Objektinstanz zugreifen (Zeile 8 – 10).

3. Zuletzt lesen wir die Variable `power` aus der `Engine`-Instanz und setzen den Wert auf `150`. Eclipse bietet die Möglichkeit primitive Werte zu setzen, indem wir sie als String übergeben (Zeile 12 – 16). Eclipse garantiert dabei die Typsicherheit, indem es überprüft, ob der String in den benötigten primitiven Typ (hier: `int`) umgewandelt werden kann.

An dem Beispiel haben wir gesehen, wie wir eine primitive Variable innerhalb des Programmzustands wiederfinden und verändern können. Fassen wir einmal kurz zusammen. Zum Verändern von primitiven Variablen wurde in Kapitel 3 das Delta `UpdatePrimitiveValueDelta` definiert. Über die Funktion `getLocation` wird eindeutig die Position der Variable, im Programmzustand bestimmt. Dazu suchen wir in dem Speichergraphen einen Pfad zu der entsprechenden Kante. Der Pfad wird bereits bei der Erstellung des Deltas berechnet.

Um das Delta anzuwenden, wird der Programmlauf an dem gleichen Programmzeitpunkt angehalten, an dem die Unterschiede bestimmt wurden. Mit Hilfe des Pfades können wir dann die

```
1  // get 0th stackframe
2  IJavaStackFrame stackFrame = thread.getStackFrames()[0];
3
4  // get variable "myCar" and value
5  IJavaVariable varMyCar = stackFrame.findVariable("myCar");
6  IJavaObject objMyCar = (IJavaObject) varMyCar.getValue();
7
8  // get variable "engine"
9  IJavaVariable varEngine = objMyCar.getField("engine");
10 IJavaObject objEngine = (IJavaObject) varEngine.getValue();
11
12 // get variable "power"
13 IJavaVariable varPower = objEngine.getField("power");
14
15 // set the value to 150
16 varPower.setValue("150");
```

Abbildung 4.2: Quellcode-Beispiel: Finden und Verändern der Variable `engine` innerhalb des Programmzustandes.

Variable im Programmzustand wiederfinden. Dazu nutzen wir die Informationen der Kanten auf dem Pfad. Diese beschreiben einen Weg über den Frame (oder auch die entsprechenden geladenen Klassen oder Interfaces) und die verschiedenen Variablen zu der gewünschten Variable. Haben wir die Variable erreicht, wird ihr der Wert aus $P_{\text{✘}}$ zugewiesen. Diese Information ist ebenfalls in dem Delta verfügbar und kann über die Funktion `getNewValue` abgerufen werden.

4.2 Manipulation von Objektreferenzen

Objektreferenzen sind das Gegenstück zu primitiven Variablen. Entweder verweisen die Referenzen auf Objekte im Programmspeicher oder auf `null`. Eine Manipulation von Objektreferenzen kann auf zweierlei Arten erfolgen. Wir können eine Objektreferenz löschen, d.h. wir setzen den Wert der Variable auf `null`, oder wir verbiegen die Referenz auf einen im Programmspeicher bestehenden Wert.

Das Löschen einer Objektreferenz wird durch das Delta `DeleteObjectReferenceDelta` repräsentiert. Wenn wir das Delta anwenden, ist die die Vorgehensweise identisch zu der Manipulation eines primitiven Werts. Bei der Erstellung des Delta berechnen wir den Pfad von dem Rootknoten zu der Kante, welche die Objektreferenz im Speichergraphen darstellt. Nachdem wir die Variable in dem entsprechenden Experimentallauf wieder gefunden haben, können wir dieser den Wert `null` zuweisen.

Eine Objektreferenz auf einen neuen Wert im Speicher zu verbiegen stellt auch kein Problem dar. Zusätzlich zu dem Pfad zu der Objektreferenz, welche wir verbiegen, merken wir uns den Pfad zu dem Wert, welcher der Objektreferenz zugewiesen werden soll. Dieser wird wie zuvor durch eine Abfolge von Kanten (Variablen) bestimmt, die zu dem entsprechenden Knoten (Objekt) im Speichergraphen führen. In Kapitel 3 wurde dazu das Delta `UpdateObjectReferenceDelta` deklariert (vgl. Abbildung 3.3). Mit Hilfe der Funktion `getNewLocation` erhalten wir den Pfad zu dem Wert, den wir der Objektreferenz zuweisen möchten.

4.3 Erzeugen von Objekten

Beim Vergleich zweier Programmzustände haben wir beobachtet, dass in $P_{\text{✘}}$ Objekte existieren können zu denen wir kein vergleichbares Objekt in $P_{\text{✔}}$ finden. Jedes dieser Objekte resultiert in einem Delta vom Typ `CreateObjectDelta`. Wenn wir das Delta anwenden bedeutet dies, dass wir das Objekt in $P_{\text{✔}}$ erstellen. Damit wir ein Objekt einer Klasse erzeugen können, muss die Klasse einen Konstruktor bereitstellen (siehe Ullenboom, 2006, Kapitel 6). Wird nicht mindestens ein Konstruktor im Quellcode definiert, so wird bei dem Compilieren der Klasse ein so genannter öffentlicher *Standard-Konstruktor* zur Verfügung gestellt. Unter einem Standard-Konstruktor versteht man einen Konstruktor ohne Argumente. Wird in dem Quellcode der Klasse ein Konstruktor definiert, so fügt der Compiler *keinen* Standard-Konstruktor hinzu, auch wenn es sich bei dem definierten Konstruktor *nicht* um einen Standard-Konstruktor handelt.

Weiterhin gilt, dass die erste Anweisung in jedem Konstruktor einer Klasse der Aufruf eines Konstruktors der Oberklasse sein muss[2]. Wenn wir einen Konstruktor definieren, können wir diesem Parameter mit übergeben, um die Klasse mit einem sinnvollen Anfangszustand zu initialisieren. Zusätzlich wird bei der Definition die Sichtbarkeit des Konstruktors festlegt. Java definiert hierfür vier Sichtbarkeitsklassen: `public`, `protected`, `package protected` und `private` (siehe Lindholm u. Yellin, 1999, Kapitel 2). Durch die Sichtbarkeit ist geregelt wer (welche Klasse) bzw. von wo aus (welches Package) wir den entsprechenden Konstruktor aufrufen können. Public-Konstruktoren können von jeder beliebigen Klasse aufgerufen werden, während Private-Konstruktoren nur innerhalb der eigenen Klasse verwendet werden können.

Problemstellung

Bei dem Vergleich zweier Speichergraphen, wird für jedes Objekt das nur in $P_{\text{✘}}$ existiert ein `CreateObjectDelta` angelegt. Wenn wir das Objekt aus $P_{\text{✘}}$ in $P_{\text{✔}}$ erzeugen möchten, sind wir mit verschiedenen Problemen konfrontiert:

[2] Es gibt genau eine Ausnahme und das ist die Klasse `java.lang.Object`. Dies ist die einzige Klasse, welche von keiner Oberklasse abgeleitet ist. Die `Object`-Klasse definiert genau einen Konstruktor und dies ist ein Standard-Konstruktor.

```
1  MyClass {
2
3    // initialize single instance of MyClass at class loading
4    private final static MyClass instance = new MyClass();
5
6    // private default constructor
7    private MyClass() {}
8
9    // factory method
10   public static MyClass getInstance() {
11     return MyClass.instance;
12   }
13
14   ...
```

Abbildung 4.3: Quellcode-Beispiel: Anwendung des Singleton-Designpatterns

- Die Klasse definiert keinen Standard-Konstruktor:

  Wie sollen wir ein Objekt instanziieren, dessen Klasse keinen Standard-Konstruktor definiert? Welche Argumente müssten wir einem parametrisierten Konstruktor übergeben, damit die Objektinstanz dem Objekt aus $P_{\text{✘}}$ entspricht? Welche Argumente sind überhaupt zulässig und führen nicht zu einer Exception bei dem Aufruf des Konstruktors? Und welchen Konstruktor rufen wir auf, wenn die Klasse verschiedene Konstruktoren definiert?

- Die Klasse definiert keinen Public-Konstruktor:

  In der Java-Spezifikation ist nicht vorgeschrieben, dass eine Klasse mindestens einen öffentlichen (als public deklarierten) Konstruktor bereitstellen muss. Ein konkreter Anwendungsfall, bei dem ein öffentlicher Konstruktor explizit nicht erwünscht ist, wird durch das Singleton-Designpattern repräsentiert (siehe Metsker, 2002, Kapitel 8). Bei diesem Designpattern wird gefordert, dass die Klasse einen privaten Standard-Konstruktor definiert und keine weiteren öffentlichen Konstruktoren definiert sein dürfen. Die Instanziierung erfolgt dann über eine statische Factory-Methode, die dem Aufrufer immer dieselbe Instanz des Objekts zurückliefert (siehe Abbildung 4.3). Doch wie sollen wir wissen, ob die Klasse eine Factory-Methode oder auch ganz andere Verfahren anbietet, um ein Objekt zu erzeugen?

- Die Instanziierung eines Objekts verändert den Programmzustand:

  Wenn wir ein Objekt über einen Konstruktoraufruf erzeugen, so wird Code ausgeführt, um das Objekt in einen gewünschten Initialzustand zu bringen. Einerseits werden die Anweisungen ausgeführt, die direkt in dem aufgerufenen Konstruktor implementiert sind, zusätzlich aber auch die Anweisungen, die in so genannten *Konstruktorblöcken* stehen. Konstruktorblöcke sind ein Hilfsmittel in Java, um Anweisungen, die bei der Erzeugung jedes

Konstruktors ausgeführt werden sollen, nur einmal zu implementieren. Weiterhin wird bei jedem Konstruktoraufruf die Initialisierung der Klassenattribute mit ihren Standardwerten vorgenommen.

Durch die ausgeführten Anweisungen kann der Programmzustand maßgeblich verändert werden. So können z.B. statische Variablen von Klassen bzw. Interfaces oder auch der Zustand von Objekten im Programmspeicher manipuliert werden. Wenn wir ein Objekt erzeugen muss sichergestellt sein, dass es durch diesen Vorgang zu keinen Veränderungen an anderen Objekten oder Klassen kommt. Dies ist wichtig, da wir sonst mit dem Erzeugen den Programmzustand nachhaltig ändern würden. Wenn wir jedoch ein `CreateObjectDelta` anwenden, sollte *nur* das entsprechende Objekt erzeugt werden. Sonstige Änderungen an dem Programmzustand könnten das Ergebnis der Fehlersuche beeinträchtigen.

- Das Laden einer Klasse hat Einfluss auf andere Objekte im Programmzustand:

  Wird ein Objekt einer Klasse zum ersten Mal erzeugt, wird zuvor die Klasse von dem Classloader geladen. Vergleichbar mit der Instanziierung eines Objekts, werden bei dem Klassenladen die globalen Variablen mit den im Quellcode definierten Werten belegt. Zusätzlich können im Quellcode ein oder mehrere statische Blöcke deklariert sein. Diese statischen Blöcke beinhalten Code, welcher beim Laden der Klasse ausgeführt wird. Wenn wir ein Objekt einer Klasse erzeugen, die noch nicht geladen wurde, so muss sichergestellt sein, dass der Code innerhalb der statischen Blöcke keinen Einfluss auf den Programmzustand hat.

Wie können wir ein Objekt einer *beliebigen* Klasse erzeugen, auch wenn sie keinen öffentlichen Konstruktor zur Verfügung stellt? Und wie können wir sicherstellen, wenn wir eine Klasse erzeugen, dass der Programmzustand nicht verändert wird? In den folgenden Abschnitten untersuchen wir, wie es uns gelingt ein beliebiges Objekt im Programmspeicher zu erzeugen, ohne den Programmzustand und somit andere Objekte zu verändern.

Ein öffentlicher Konstruktor für alle Klassen

Angenommen wir müssen ein Objekt einer Klasse erzeugen, welche keinen öffentlichen Konstruktor definiert. Wie können wir eine Instanz dieser Klasse erzeugen? Eine Lösung des Problems ist der entsprechenden Klasse einen zusätzlichen öffentlichen Konstruktor hinzuzufügen. Dabei nutzen wir Verfahren, um den Java-Bytecode einer Klasse zu verändern. Mit Hilfe der *Byte Code Engineering Library* (kurz BCEL) sind wir in der Lage Bytecode zu Analysieren, Manipulieren und somit neuen Bytecode zu erzeugen (Apa, 2005).

Mit BCEL können wir jeder Klasse einen neuen öffentlichen Konstruktor hinzufügen. Da wir von dem Debuggee den Quellcode zur Verfügung haben, ist die Frage berechtigt, wieso wir die Veränderungen nicht direkt an dem Quellcode vornehmen. Diese Vorgehensweise hat gegenüber der Bytecode-Manipulation mehrere Nachteile:

- Bibliotheken, die von dem Programm genutzt werden, könnten nur in Form von Bytecode vorhanden sein (z.B. die Java Bibliotheks- und Laufzeitklassen). Dies hat zur Folge, dass wir von den Klassen dieser Bibliotheken keine neuen Objekte erzeugen können.

- Die Anweisungen der Konstruktorblöcke würden beim Übersetzen mit in den neuen Konstruktor eingefügt und somit bei der Objekterzeugung ausgeführt.

Daher werden wir alle Klassen, die wir während der Ausführung des Programmlaufs benötigen, dahingehend manipulieren, dass wir ihnen einen leeren Konstruktor mit Hilfe von BCEL hinzufügen. Allerdings können wir nicht einfach einen Standard-Konstruktor hinzufügen, da dieser durch die Klasse selbst schon definiert sein könnte. Allgemein gilt, dass die Signatur einer Methode oder eines Konstruktors innerhalb der Klasse eindeutig sein muss. Zur Lösung des Problems erweitern wir den Konstruktor um einen Parameter:

```
public class MyClass {

  // new public constructor with unique signature
  public MyClass(org.deltadebugging.ddstate.ClassController cc) {
  }

  ...
```

Damit wir sicher sein können, dass der Konstruktor eindeutig ist, wird als Übergabeparameter ein Objekt der von uns definierten Klasse `ClassController` erwartet. Bei der Manipulation der Klassen müssen wir die Klasse *`java.lang.Object`* außen vor lassen. Die Erweiterung von `Object` durch einen Konstruktor führt zu einem Henne-Ei-Problem. Um eine Klasse zu laden, müssen all ihre abhängigen Klassen geladen werden. Durch einen neuen Konstruktor würden wir eine Abhängigkeit von `Object` zu der Klasse `ClassController` definieren. Da `ClassController` aber eine Unterklasse von `Object` ist besteht eine bilaterale Abhängigkeit. Die Klasse `Object` definiert jedoch selbst einen öffentlichen Standard-Konstruktor. Somit ist es kein Problem ein Objekt der Klasse zu erzeugen. Außerdem können wir beim Instanziieren eines `Object`-Objekt garantieren, dass dies keine Auswirkungen auf andere Objekte im Programmzustand hat.

Wie anfangs erwähnt, muss jeder Konstruktor, sofern es sich nicht um einen Konstruktor der Klasse `java.lang.Object` handelt, einen Konstruktor der Oberklasse aufrufen. Diesen Vorgaben werden wir Folge leisten, indem wir den eigens erstellten Konstruktor in der Oberklasse aufrufen:

```
public class MyClass extends UpperClass {

  public MyClass(org.deltadebugging.ddstate.ClassController cc) {
    super(cc); // call constructor of the upper class
  }

  ...
```

Handelt es sich bei der Oberklasse um die Klasse `java.lang.Object` rufen wir den Standard-Konstruktor auf. Durch den neuen öffentlichen Konstruktor haben wir sichergestellt, dass wir ein Objekt jeder Klasse erzeugen können. Da der Konstruktor, außer dem Aufruf der Superklasse, keine weiteren Anweisungen ausführt, bleibt der Programmzustand beim Instanziieren eines Objekts unverändert. Die in der Klasse definierten Attribute werden mit ihren Standardwerten vorbelegt (für Objektreferenzen ist dies z.B. der Wert `null`). Die Definition der Standardwerte für jeden Variablentyp ist in der Java-Spezifikation geregelt (siehe Lindholm u. Yellin, 1999, Kapitel 2.5).

Wir können jetzt durch den neu erstellten öffentlichen Konstruktor ein Objekt jeder beliebigen Klasse erzeugen. Der Programmzustand könnte dennoch ungewollt durch Ausführung der statischen Blöcke verändert werden, wenn die Klasse des zu erzeugenden Objekts zuvor noch nicht geladen wurde.

Kontrolliertes Ausführen des statischen Blocks

Die Initialisierung der statischen Variablen, sowie die Anweisungen aller in einer Klasse definierten statischen Blöcke werden im Bytecode zu *einem* statischen Block zusammengefasst. Wenn die Klasse durch den Classloader geladen wird, führt die JVM die Anweisungen innerhalb dieses Blocks aus. Da die Ausführung der Anweisungen ungewünschte Modifikationen an dem Programmzustand mit sich bringen können, versuchen wir das Ausführen des statischen Blocks zu kontrollieren. Dieses geschieht wiederum durch Instrumentation des Bytecodes unter zu Hilfenahme von BCEL.

Wir modifizieren den Code dahingehend, dass wir den statischen Block mit einer `IF`-Anweisung umschließen. Die Bedingung der Ausführung ist dabei an eine von uns definierte globale Variable geknüpft. Wir verwenden dazu wieder die Klasse `ClassController` und fügen dieser eine globale Variable `WITH_STATIC_BLOCK` hinzu. Ist der Wert der Variable auf `true` gesetzt wird der statische Block beim Laden der Klassen ausgeführt, andernfalls nicht.

```
public class ClassController {

  // variable to controll the execution of static blocks
  public static boolean WITH_STATIC_BLOCK = true;

}

public class MyClass {

  static {
    if (ClassController.WITH_STATIC_BLOCK) {
      ... // original static block
    }
  }

  ...
```

Erzeugen eines Objekts einer beliebigen Klasse

Wir haben gesehen, wie wir einer Klasse einen neuen Konstruktor hinzufügen und die Ausführung des statischen Blocks durch Einführung einer zusätzlichen Bedingung kontrollieren können. Wenn wir ein Objekt einer Klasse erzeugen, benötigen wir nur noch den vollständigen Klassennamen. Über den Aufruf Funktion `getValueToCreate` auf dem `CreateObjectDelta` bekommen wir den Knoten des entsprechenden Objekts (`IObjectNode`), welches erzeugt werden soll. Der Knoten selbst kennt den entsprechenden Klassennamen, der über die Funktion `getReferenceTypeName` abgerufen werden kann (vgl. Abbildung 2.5).

Wenn das Objekt erzeugt ist, kann es in dem Programmzustand Objektreferenzen zugewiesen werden und die entsprechenden Objektattribute können gesetzt werden (siehe Abschnitt 3.5). Dazu muss das Objekt in der gleichen Virtual Machine erzeugt werden, in der wir den Programmzustand manipulieren. Dieses ist jedoch nicht die VM, in der Eclipse und DDSTATE ausgeführt werden. Ein JUnit-Testfall wird in einer eigenen VM gestartet. Damit wir in dieser VM das Objekt erzeugen können, erweitern wir den *JUnit-Testrunner*. Der Testrunner ist die Klasse, welche die verschiedenen Tests eines Testcase ausführt und die Ergebnisse der Tests an Eclipse bzw. DDSTATE zurückliefert. Wir erweitern den Testrunner, um die Funktionalität, die wir benötigen, um den Programmzustand entsprechend den Deltas zu manipulieren.

Über die Funktion `createNewObject` können wir ein neues Objekt einer beliebigen Klasse erzeugen (siehe Abbildung 4.3). Bei der Ausführung der Funktion gehen wir wie folgt vor. Zunächst deaktivieren wir die Ausführung des statischen Blocks. Danach laden wir die Klasse durch den eigens bereitgestellten Klassenlader (`InstrumentationClassLoader`). Durch diesen werden, während des Klassenladens, für den Programmlauf transparent die Manipulationen an dem Bytecode vorgenommen. Mit Hilfe des `Class`-Objekts rufen wir den neu eingefügten

```
1  public Object createNewObject(String className) {
2    Object result = null;
3
4    try {
5      ClassController.WITH_STATIC_BLOCK = false;
6      Class c = new InstrumentationClassLoader().loadClass(className);
7
8      if (className.equals("java.lang.Object")) {
9        Constructor constr = c.getConstructor(new Class[] {});
10       result = constr.newInstance(new Object[] {});
11
12     } else {
13       Constructor constr =
14         c.getConstructor(new Class[] { ClassController.class });
15       result = constr.newInstance(new Object[] { null }));
16
17     }
18
19   } finally {
20     ClassController.WITH_STATIC_BLOCK = true;
21
22   }
23
24   return result;
25 }
```

Abbildung 4.4: Methode zum Instanziieren eines Objektes einer beliebigen Klasse in dem JUnit-Testrunner

Konstruktor auf. Dabei wird unterschieden, ob ein Objekt vom Typ `java.lang.Object` erzeugt werden soll, oder ein beliebiges anderes Objekt. Bei dem Erzeugen einer `Object`-Instanz wird der von `Object` bereitgestellte Standard-Konstruktor verwendet. Andernfalls rufen wir den eigens in der Klasse generierten Konstruktor auf. Wichtig ist, dass bevor wir den Test weiterlaufen lassen, das Flag `WITH_STATIC_BLOCK` auf `true` gesetzt wird, da sonst bei dem Laden jeder weiteren Klasse das Ausführen des statischen Blocks verhindert wird.

Nach dem Erzeugen des Objekts, wird dieses zunächst von keiner Variablen referenziert. Damit das Objekt nicht von dem Garbage Collector erfasst wird und wir es für spätere Modifikationen wiederfinden, werden wir es referenzieren. Dazu registrieren wir das Objekt in einem `Object`-Array innerhalb des JUnit-Testrunner. Wir nutzen das Array als Pool, um alle neu erzeugten Objekte zu speichern und sie für spätere Modifikationen (z.B. dem Ändern der Objektattribute) wieder zu finden. Bei dem Registrieren des Objektes innerhalb des Arrays merken wir uns den Arrayindex zu dem Objekt. Über diesen können wir das Objekt später wiederfinden.

Wir haben am Anfang dieses Kapitels gesehen, wie wir eine Variable oder ein Objekt in $P_{\checkmark}$ mittels eines Pfads wiederfinden. Wie finden wir aber neu erzeugte Objekte, um die Attribute entsprechend dem Objekt in $P_{\times}$ zu setzen, oder um sie bestehenden Objektvariablen zuzuweisen? Für Deltas, die sich auf neu erzeugte Objekte beziehen, können wir bei der Erstellung kein Pfad berechnen, da diese Objekte im Programmzustand und somit im Speichergraphen noch nicht existieren. Alle weiteren Deltas, die sich auf diese Objekte beziehen (vgl. Abschnitt 3.5), müssen den Pfad zu den jeweiligen Objekten zur Laufzeit berechnen. Deshalb wurde neben der *Pfad-Location* eine *Delta-Location* erstellt. Eine Delta-Location bezieht sich dabei auf ein Objekt, das durch ein `CreateObjectDelta` erzeugt wurde.

Nehmen wir z.B. an, dass ein Objekt vom Typ `Engine` erzeugt werden muss (Δ_{engine}) und danach das Attribut `power` auf 150 gesetzt werden soll (Δ_{power}). Da wir bei der Erstellung von Δ_{power} keinen Pfad zu der Variable `power` berechnen können, erzeugen wir eine Delta-Location. Diese verweist auf die Variable `power` des von Δ_{engine} erzeugten Objekts. Wir haben somit einen dynamischen Pfad, der bei der Erstellung der Deltas noch nicht feststeht. Die Verantwortlichkeit, welches Objekt geändert werden muss, liegt bei Δ_{engine}. Nachdem dieses Delta angewendet wurde, können wir den Pfad berechnen, da das entsprechende `Engine`-Objekt an einer bestimmten Stelle im JUnit-Testrunner registriert wurde.

4.4 Erzeugen und Manipulieren von Arrays

Den aufwendigen und komplizierten Prozess des Objekt-Erstellen's, haben wir in dem vorherigen Abschnitt näher untersucht. In diesem Abschnitt untersuchen wir, wie wir ein Array erzeugen (`CreateObjectDelta`) und die Größe eines Arrays verändern (`UpdateArraySizeDelta`). Ein Array ist ein Subtyp eines Objekts. Das Erstellen eines neuen Arrays ist aber wesentlich einfacher, da der jeweilige Typ von dem ein Array erzeugt werden soll keinen Konstruktor definieren muss. Vergleichen wir zwei Arrays mit gleichem Typ, so können sich diese in beiden Programmzuständen durch ihre Größe, d.h. die Anzahl ihrer Elemente unterscheiden.

Erzeugen eines Arrays von einer beliebigen Klasse

Arrays sind in jeder Hinsicht spezielle Objekte, auch wenn es darum geht ein Array zu erzeugen. Möchten wir z.B. ein Array vom Typ `Car` mit einer Größe von 10 Elementen erzeugen, so geschieht dies mit Hilfe des `new`-Operator, unter Angabe der Größe in eckigen Klammern:

```
Car[] myCars = new Car[10];
```

Hierbei spielt es keine Rolle, ob der Typ, von dem wir ein Array erzeugen möchten, einen öffentlichen Konstruktor deklariert oder nicht. Wir können Arrays von jedem beliebigen Typ erzeugen (so auch von primitiven Datentypen, wie z.B. `int`). Um ein Array mit einer bestimmten Anzahl von Elementen zu erstellen, erweitern wir den JUnit-Testrunner um die Funktion

```
1  public Object createNewArray(String typeName, int length) {
2    Object result = null;
3
4    try {
5      ClassController.WITH_STATIC_BLOCK = false;
6      // get the class of the type (e.g. int.class)
7      Class c = this.loadClassByName(typeName);
8
9    } finally {
10     ClassController.WITH_STATIC_BLOCK = true;
11
12   }
13
14   result = java.lang.reflect.Array.newInstance(c, length)
15
16 return result; }
```

Abbildung 4.5: Methode zum Instanziieren eines Arrays mit beliebigem Datentyp und Größe

`createNewArray` (siehe Abbildung 4.4). Diese Funktion bekommt den Typnamen des Arrays und die Anzahl der Elemente als Parameter übergeben. Bei dem Aufruf der Funktion wird folgendes getan: Zunächst wird die Ausführung des statischen Blocks ausgeschaltet. Danach laden wir die Klasse zu dem übergebenen Typnamen. Da primitive Datentypen nicht direkt über den Classloader geladen werden können, verwenden wir eine eigens definierte Funktion `loadClassByName(String)`, welche das dem übergebenen Typnamen entsprechende `Class`-Objekt zurückgibt. Das Erstellen eines Arrays geschieht dann über die `Array`-Klasse aus dem Reflektionpaket der Java Bibliotheksklassen. Durch die Angabe des `Class`-Objektes und der Anzahl der Elemente kann so dynamisch ein neues Array erzeugt werden. Die neu erzeugten Arrayobjekte werden, ebenso wie die neu erzeugten Objekte, in dem JUnit-Testrunner registriert.

Modifikation der Arraygröße

Angenommen wir haben ein Array aus $P_{\checkmark}$ mit n Elementen ($A^n_{\checkmark}$). Weiterhin haben wir ein Array aus $P_{\times}$ mit m Elementen ($A^m_{\times}$), wobei gilt: $m \neq n$. Wenn wir $A^n_{\checkmark}$ in $A^m_{\times}$ transformieren, müssen wir $A_{\checkmark}$ auf die Größe von $A_{\times}$ reduzieren bzw. erhöhen. Danach können wir alle notwendigen Änderungen an den entsprechenden Arrayelementen vornehmen.

Betrachten wir folgendes Beispiel: In $P_{\checkmark}$ existiert ein `char`-Array $A_{\checkmark}$ = `{'T', 'i', 'm'}`, während in $P_{\times}$ das entsprechende Array $A_{\times}$ die Zeichen `'T'`, `'o'`, `'m'`, `'a'` und `'s'` beinhaltet. Um $A_{\checkmark}$ in $A_{\times}$ zu transformieren, müssen wir zunächst die Arraygröße von $A_{\checkmark}$ auf 5 erhöhen. Danach können wir das erste Arrayelement in `'o'` ändern und das dritte und vierte Element auf `'a'` bzw. `'s'` setzen.

Doch wie können wir in Java die Größe eines Arrays verändern? Jedes Array definiert das Attribut `length` über das die Anzahl der Elemente einer Arrayinstanz ausgelesen werden kann. Das Attribut selbst ist aber als `final` deklariert und kann somit nicht neu gesetzt werden. Die Anzahl der Elemente eines Arrays wird in Java bei der Initialisierung des Arrays festgelegt und eine dynamische Veränderung der Größe ist nachträglich nicht mehr möglich. Wir müssen eine andere Lösung finden.

Um die Größe des Arrays anzupassen, verfahren wir in drei Schritten. Zuerst erzeugen wir ein neues Array des gleichen Typs mit der gewünschten Anzahl der Elemente. Danach verbiegen wir alle Referenzen von dem alten auf das soeben erzeugte Array. Zuletzt füllen wir die Arrayelemente des neuen Arrays mit den Werten aus dem alten Array. Für primitive Arrays bedeutet dies, dass wir die entsprechenden primitiven Werte kopieren. Für Arrays, welche Objektreferenzen beinhalten, müssen wir die Referenzen auf die entsprechenden Objekte verbiegen. Wenn wir die Arraygröße verkleinert haben kopieren bzw. verbiegen wir die Referenzen bis zu dem höchst möglichen Arrayindex.

In Abbildung 4.6 sind für das `char`-Array Beispiel die verschiedenen Schritte anhand der entsprechenden Speichergraphen abgebildet:

1. Es wird ein neues `char`-Array mit 5 Elementen erzeugt.
2. Alle Variablen, die $A_{\checkmark}$ referenziert haben, werden auf das neu erzeugte Array umgebogen.
3. Die Zeichen `'T'`, `'i'` und `'m'` werden unter Beibehaltung der ursprünglichen Reihenfolge in das neue Array kopiert.

Die Manipulation der einzelnen Arrayelemente, entsprechend dem Array aus $P_{\times}$, ist nicht mehr Aufgabe des `UpdateArraySizeDelta`. Dazu werden bei dem Vergleich beider Arrays weitere Deltas erzeugt (in dieses Fall vom Typ `UpdatePrimitiveValueDelta`).

4.5 Details der Implementierung

Modifikation der Java Laufzeitklassen

Um von jeder Klasse eine Objektinstanz zu erzeugen, fügen wir allen Klassen während des Ladens mit Hilfe von BCEL einen neuen Konstruktor hinzu. Dazu haben wir einen eigenen Klassenlader geschrieben, welcher die Änderungen transparent bei der Ausführung des JUnit-Tests vornimmt. Es gibt allerdings auch Klassen, die wir nicht während des Ladens modifizieren können – dieses sind beispielsweise die Java Laufzeitklassen (engl. Runtime Classes) (zu den Laufzeitklassen Klassen gehören z.B. alle Klassen aus dem Paket `java.lang`).

Das Laden der Klassen wird in der JVM durch Klassenlader kontrolliert. Es gibt einen so genannten *Boostrap-Klassenlader*, welcher fest in der JVM verankert und für das Laden der Java

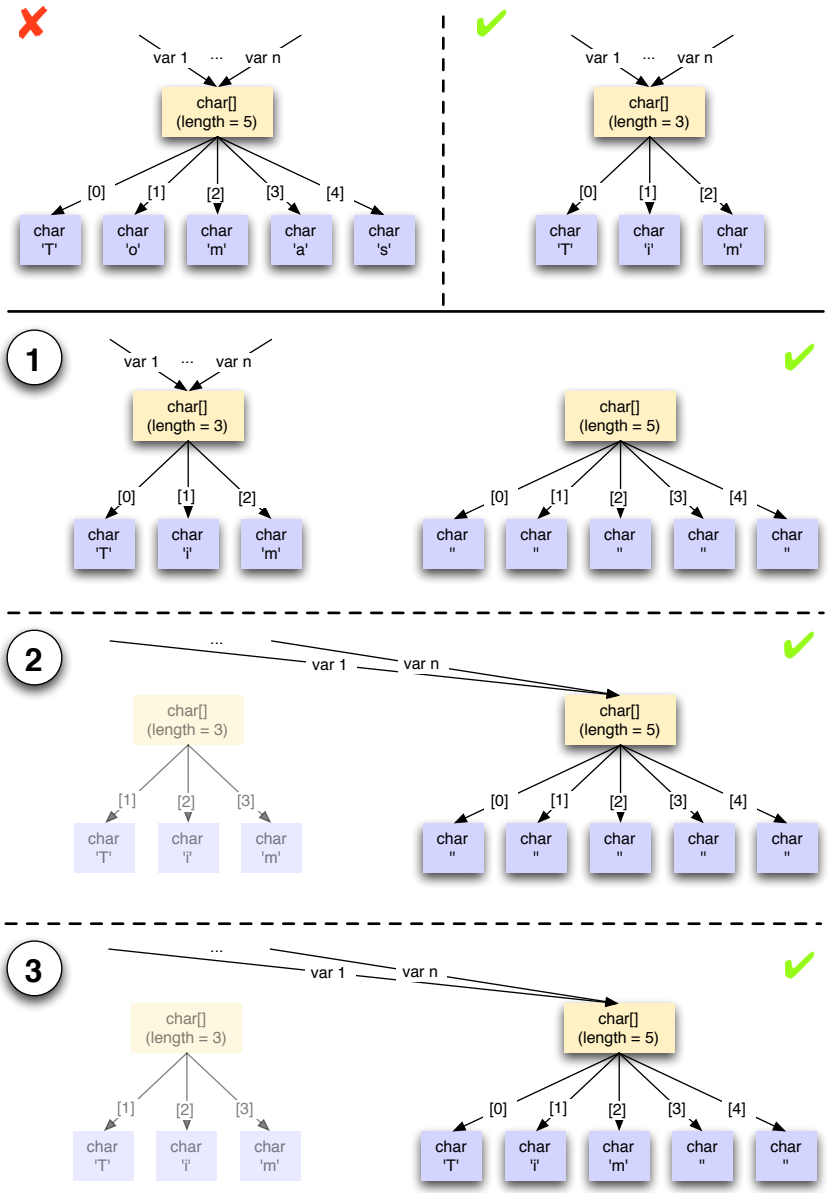


Abbildung 4.6: Beispiel: Vergrößerung der Arraygröße eines `char`-Arrays von 3 auf 5 Elemente.

Laufzeitklassen verantwortlich ist (Sosnoski, 2003). Dieser spezielle Klassenlader lädt nur Klassen, die sich im so genannten Boot-Classpath befinden. Dazu gehören auch die Java Laufzeitklassen.

Da wir Klassen, welche der Bootstrap-Klassenlader geladen hat, nicht mehr mit dem eigenen Klassenlader laden können, müssen wir die Veränderung an den Laufzeitklassen vornehmen, bevor wir den JUnit-Test starten. Wir werden deshalb vor der eigentlichen Ausführung des Tests die Laufzeitklassen aus ihren Archiven (JAR-Paketen) entpacken, modifizieren und eine modifizierte Version der Klassen speichern. Wenn wir den JUnit-Test starten, verwenden wir dann die modifizierten Klassen und sind somit in der Lage auch von jeder beliebigen Java Laufzeitklasse eine Instanz zu erzeugen.

4.6 Fazit

In Kapitel 3 haben wir die Deltas zwischen zwei Speichergraphen bestimmt. Diese Deltas beschreiben Zustandsunterschiede der jeweiligen Programmzustände, aus denen die Speichergraphen extrahiert wurden. Extrahieren wir die Speichergraphen aus einem funktionierenden und einem fehlerhaften Programmlauf, wissen wir, dass die Zustandsunterschiede Ursache für den fehlschlagenden Lauf sind.

In diesem Kapitel haben wir gesehen, wie wir die berechneten Deltas verwenden, um einen Programmzustand zu manipulieren. In einem Programmzustand können wir

- primitiven Variablen einen neuen Wert zuweisen,
- Objektreferenzen auf bestehende Objekte verbiegen,
- Objektreferenzen den Wert `null` zuweisen,
- neue Objekte und Arrays erzeugen und
- die Größe von bestehenden Arrays verändern.

Durch diese Operationen sind wir in der Lage den funktionierenden Programmzustand in den fehlerhaften zu überführen. Wenden wir nur ein Teil der berechneten Deltas an, so erzeugen wir einen Mischzustand. Wird der funktionierende Programmlauf nach der Manipulation fortgesetzt, können wir testen, ob der erzeugte Mischzustand und der fehlerhafte Programmlauf zu dem gleichen Fehler führen. Wenn ja, wissen wir, dass bereits eine Teilmenge der angewendeten Deltas für den Fehler relevant sein muss. Delta Debugging beschreibt ein Verfahren, um durch systematische Tests eine möglichst kleine Teilmenge von fehlerrelevanten Zustandsunterschieden zu bestimmen. In dem nächsten Kapitel sehen wir, wie DDSTATE unter Verwendung der Delta Debugging-Algorithmen aus allen berechneten Zustandsunterschieden, die fehlerrelevanten extrahiert.

5 Anwendung: DDstate

In den vorhergehenden Kapiteln haben wir Algorithmen betrachtet, die einen Programmzustand in Form eines Speichergraphen extrahieren. Aus zwei verschiedenen Speichergraphen können wir die Unterschiede zwischen beiden Graphen berechnen. Durch die Anwendung der aus den Unterschieden resultierenden Deltas sind wir in der Lage, einen Zustand zu manipulieren, um die fehlerrelevanten Unterschiede zu bestimmen. Die beschriebenen Verfahren wurden zusammen mit dem Delta Debugging Algorithmus in DDSTATE integriert. DDSTATE analysiert Programmzustände respektive Zustandsunterschiede, um den Anwender bei der Fehlersuche zu unterstützen. In diesem Kapitel werden wir an Fallbeispielen sehen, wie DDSTATE – voll integriert in die Java-IDE von Eclipse – verwendet werden kann, um fehlerrelevanten Variable (siehe Abschnitt 5.1) bzw. fehlerrelevante Codestellen (siehe Abschnitt 5.2) zu bestimmen.

Ein fehlschlagender Testfall ist prinzipiell eine Folge der Ausführung eines Defekts im Quellcode. Dieser Defekt provoziert eine Infektion im Programmzustand, welche weiterpropagiert bis ein für den Benutzer sichtbarer Fehler entsteht (beispielsweise verfärbt sich der Balken in der Benutzeroberfläche von JUnit rot). Unter zusätzlicher Verwendung eines funktionierenden Testfalls, können wir die Fehlerursache mittels zweier Verfahren lokalisieren:

- Die »Suche im Raum«, über einen kompletten Programmzustand mit eventuell mehreren tausend Variablen, hilft uns die fehlerrelevante(n) Variable(n) zu extrahieren.
- Durch die »Suche in der Zeit« sind wir in der Lage, aus einer Vielzahl von Programmzuständen die Stelle im Programmlauf zu ermitteln, an der die Infektion begonnen hat.

5.1 »Suche im Raum« – Bestimmen fehlerrelevanter Variablen

Dieser Abschnitt beschreibt, wie wir DDSTATE verwenden, um die fehlerverursachenden Variablen eines fehlschlagenden JUnit-Testfalls zu einem gegebenen Programmzeitpunkt zu bestimmen. Dazu betrachten wir die Testklasse `CarConfiguratorTest` aus Kapitel 1 (siehe Abbildung 1.1). Um die Fehlerursache zu bestimmen, benötigen wir neben dem fehlschlagenden Testlauf ($r_{✘}$) einen funktionierenden ($r_{✔}$). Fakt ist, dass ein oder mehrere Unterschiede zwischen den Programmzuständen aus $r_{✘}$ und $r_{✔}$, für das Fehlschlagen verantwortlich sind.

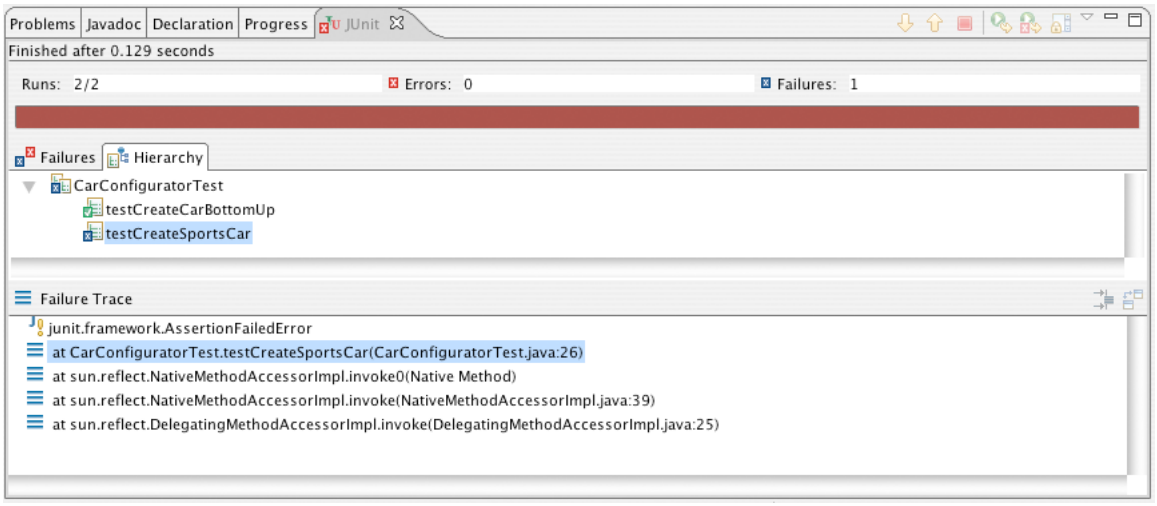


Abbildung 5.1: Ergebnis der Ausführung des JUnit-Testfall `CarConfiguratorTest`. In dem View wird in dem Register *Hierarchy* angezeigt, welche Testmethoden ausgeführt wurden, sowie das entsprechende Ergebnis.

In der Testklasse `CarConfiguratorTest` überprüfen wir die Gültigkeit der Konfigurationen von zwei `Car`-Objekten. Das Ergebnis der Testausführung des JUnit-Tests zeigt, dass die Konfiguration des Objekts innerhalb der Testmethode `testCreateCarBottomUp` gültig ($r_{✔}$), die Konfiguration von `testCreateSportsCar` ungültig ($r_{✘}$) ist (siehe Abbildung 5.1). Die Frage ist, welche Unterschiede, in den Konfigurationen der beiden Fahrzeuge, für das Fehlschlagen des Testlaufs verantwortlich sind.

Bevor wir die Unterschiede zweier Programmzustände bestimmen können, müssen wir zunächst die Programmzeitpunkte definieren, an denen wir die Zustände vergleichen möchten (siehe Abschnitt 2.1). Jeweils einen Zeitpunkt aus $r_{✔}$ und $r_{✘}$ nennen wir *Vergleichspunkt* (engl. Matching Point).

Auswahl der Vergleichspunkte

An welchen Stellen innerhalb der Testläufe vergleichen wir beide Läufe, damit wir einen Hinweis auf den Fehler bekommen? In Kapitel 3 haben wir festgelegt, dass beiden Vergleichspunkten ein identischer Callstack zu Grunde liegen sollte. Dies ist notwendig, da wir die Menge der lokalen Variablen nicht verändern können.

Wir benötigen zu der Ausführung von DDSTATE zwei verschiedene JUnit-Testläufe – einen der ohne Fehler läuft und einen der fehlschlägt. Diese Testläufe werden durch den Aufruf von zwei verschiedenen JUnit-Testmethoden gestartet. Daraus folgt aber, dass die Callstacks während der

Ausführung der Testläufe nie identisch sein können. Deshalb werden wir die gestellte Bedingung abschwächen indem wir fordern, dass beide Callstacks *bis* auf die Testmethode identisch sein müssen. Weiterhin gilt für beide Testmethoden, dass die Menge der lokalen Variablen übereinstimmen sollte. Dies bedeutet, dass zu jeder lokalen Variable in der funktionierenden Testmethode eine lokale Variable gleichen Namens und Typs in der fehlerhaften Testmethode existieren sollte. Variablen, die nicht in beiden Methoden bestehen, werden beim Vergleich der Zustände nicht berücksichtigt. In dem Car-Configurator-Beispiel wird in beiden Testmethoden jeweils die Variable `myCar` vom Typ `Car` deklariert. Damit haben wir sichergestellt, dass der Zustand der Variable bei dem Vergleich der Programmzustände mit berücksichtigt wird.

Damit das Verfahren erfolgreich angewendet werden kann, d.h. die fehlerrelevanten Unterschiede berechnet werden, muss *mindestens ein* Unterschied in beiden extrahierten Programmzuständen bestehen. Wenn wir den Programmzustand direkt nach dem Start der beiden Testfunktionen vergleichen, besteht noch kein Unterschied in den Zuständen. Der einzige Unterschied besteht dann nur in dem Callstack selbst (dieser unterscheidet sich in der jeweiligen Testmethode). Es hat sich herausgestellt, dass der Programmzeitpunkt kurz vor dem Auftreten des Fehlers einen guten Vergleichspunkt liefert. Dies ist in dem Beispiel für $r_{\text{✘}}$ die Zeile 26. Eine gute Stelle für den Vergleich in $r_{\text{✔}}$ liefert die Zeile 13:

- An beiden Stellen besteht der gleiche Callstack (bis auf die JUnit-Testmethoden), sowie die gleichen lokalen Variablen (`myCar`, etc.).
- An beiden Instanzen der Variable `myCar` wurden alle Konfigurationen vorgenommen. Die komplette Konfiguration der Fahrzeuge befindet sich damit im Programmspeicher. Ein Vergleich der Programmzustände bedeutet somit ein Vergleich der beiden Fahrzeugkonfigurationen. Da wir wissen, dass der Unterschied der Konfigurationen zum Fehler führt, können wir durch einen Vergleich der Programmzustände den Fehler innerhalb der Konfiguration finden.

Erstellen einer Startkonfiguration

Wir haben alle Informationen, die wir benötigen um DDSTATE zu starten. Ein Aufruf des Menüpunkt *Run* > *Debug...* startet den *Launch Configuration Manager*. Hier werden die Konfigurationen zum Starten und Debuggen von Java-Programm, Java-Applets, JUnit-Tests, etc. definiert. Nach der Installation von DDSTATE steht ein weiterer Konfigurationstyp »DDstate« zur Verfügung. Dort können die von DDSTATE benötigten Eingaben festlegt werden. Diese einmal festgelegte Konfiguration kann beliebig oft wieder verwendet werden.

Im Register *Test* geben wir das Projekt, die Testklasse, sowie den funktionierenden und fehlerhaften Testfall an (siehe Abbildung 5.2). Über das Register *Matchpoint* definieren wir die Vergleichspunkte, an denen DDSTATE den fehlerrelevanten Unterschied bestimmen soll (siehe Abbildung 5.3). Die Angabe der Vergleichspunkte erfolgt durch die Definition zweier Zeilenhaltepunkte.

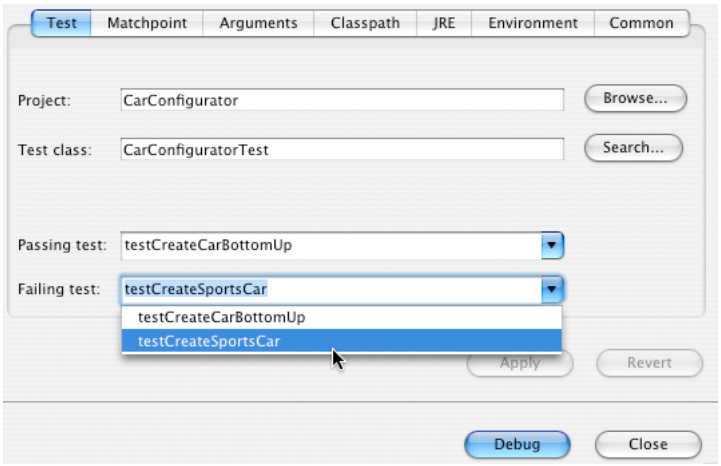


Abbildung 5.2: Definition der Startkonfiguration in DDSTATE: Auswahl des Eclipse-Projekts, der Testklasse sowie der Testmethoden des funktionierenden und fehlerhaften Testfall.

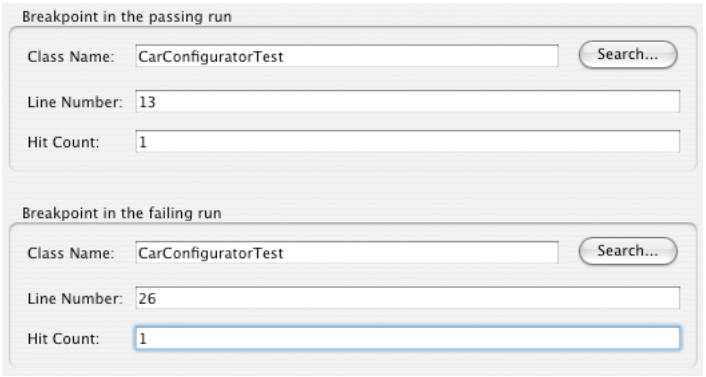


Abbildung 5.3: Definition der Vergleichspunkte in Form von Zeilenhaltepunkten. Die Zeilenhaltepunkte müssen bei der Ausführung der jeweiligen Testläufe erreicht werden. Sie bestimmen eindeutig die Zeitpunkte innerhalb der Testläufe, an denen die Programmzustände zum Vergleich extrahiert werden.

Nachdem wir alle Angaben in die Formularfelder eingetragen haben, sind wir nur ein Mausklick von der automatischen Analyse entfernt. Durch klick auf *Debug* startet DDSTATE.

Der Debugging-Lauf

Einmal gestartet berechnet DDSTATE die fehlerrelevanten Unterschiede vollautomatisch und erfordert keine weiteren Benutzerinteraktionen. DDSTATE überprüft zuerst, ob die für die Anwendung benötigten modifizierten Java Laufzeitklassen zur Verfügung stehen (siehe Abschnitt 4.5). Wenn nicht, wird automatisch eine Version des Runtime-JAR mit den in Abschnitt 4.3 beschriebenen Erweiterungen gebaut. Danach werden beide Testläufe bis zu den definierten Vergleichspunkten ausgeführt. Wenn diese erreicht sind, werden die Programmzustände extrahiert und die Unterschiede berechnet. Aus diesen wird schließlich unter Verwendung von Delta Debugging die fehlerrelevanten Variablen ermittelt.

Der ganze Prozess kann, ohne den Anwender bei seiner Arbeit zu behindern, im Hintergrund ausgeführt werden. Sobald die fehlerrelevanten Unterschiede berechnet wurden, werden diese in einem View dargestellt.

Ergebnis eines Testlaufs

Wenn DDSTATE den Debugging-Lauf beendet hat, werden die Ergebnisse in dem View »DDstate Run« angezeigt. Darin finden die Menge, der von DDSTATE bestimmten fehlerrelevanten Zustandsunterschiede. Der View ist in insgesamt vier Register aufgeteilt:

- *Minimized Deltas*: Dieser View listet tabellarisch die von DDSTATE berechneten fehlerrelevanten Unterschiede. Dies sind die Unterschiede zwischen den Programmzuständen der beiden Testläufe, die dafür verantwortlich, dass $r_{\text{✘}}$ fehlgeschlagen ist.
- *Computed Deltas*: Eine tabellarische Auflistung aller zwischen den Programmzuständen der Testläufe berechneten Unterschiede.
- *Test History*: Der Verlauf aller von DDSTATE durchgeführten einzelnen Test und deren Ergebnisse. Über diese Tests sehen wir, wie DDSTATE die fehlerrelevanten Variablen von der ursprünglichen Menge der Unterschiede berechnet hat.
- *Info*: Dieser View enthält zusätzliche Informationen über den Debugging-Lauf. Diese beinhalten die zuvor definierten Testmethoden sowie die ausgewählten Vergleichspunkte.

Im Register *Minimized Deltas* finden wir eine tabellarische Auflistung der durch DDSTATE berechneten fehlerrelevanten Zustandsunterschiede. Die Unterschiede entsprechen den Veränderungen zwischen $P_{\text{✓}}$ und $P_{\text{✘}}$, welche zu dem fehlschlagenden Testlauf geführt haben. In der

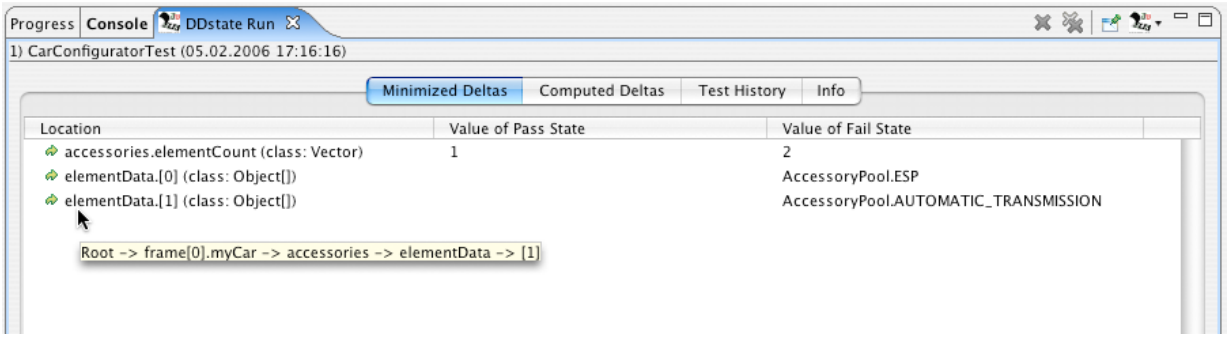


Abbildung 5.4: Ergebnis DDSTATE: Fehlerrelevante Zustandsunterschiede

Spalte *Location* werden die Namen der fehlerrelevanten Variablen angezeigt (inklusive der Position der Variable innerhalb des Programmzustands). Die vorangestellten Symbole sind dabei wie folgt zu deuten:

- Ein Objekt oder Array wurde neu erzeugt und dem Programmzustand hinzugefügt.
- Ein Wert, eine Objektreferenz bzw. die Größe eines Arrays wurden verändert.
- Einer Objektreferenz wurde der Wert `null` zugewiesen.

Um eine Variable innerhalb eines Programmzustands eindeutig zu bestimmen, benötigen wir einen Pfad vom Rootknoten zu der Kante der Variable (vgl. Kapitel 4). In der Spalte *Location* werden die letzten zwei Elemente des Pfads angezeigt (das letzte Element entspricht dem Variablennamen). Den vollständigen Pfad können wir uns mittels Tooltiptext ansehen. Zusätzlich zu dem Variablennamen bekommen wir die Information über den konkreten Typ des Objekts, zu dem die Variable gehört.

Neben dem Variablennamen sehen wir in den Spalten *Value of Pass State* bzw. *Value of Fail State* die Werte der Variable aus dem funktionierenden bzw. fehlerhaften Programmzustand.

Das Ergebnis des DDSTATE Debugging-Laufs zu dem Car-Configurator-Beispiel ist dabei wie folgt zu deuten: Die Ausführung von `testCreateSportsCar` schlug fehl, da die Variable `elementCount` des Objekts `accessories` (eine Instanz der Klasse `Vector`) von 1 auf 2 geändert wurde (Abbildung 5.4). Zusätzlich wurde dem ersten Arrayelement aus dem Array `elementData` das Objekt zugewiesen, welches durch die statisch definierte Variable `ESP` in der Klasse `AccessoryPool` referenziert wird. Außerdem wurde das Arrayelement 1 des Arrays `elementData` dem Objekt zugewiesen, welches von `AUTOMATIC_TRANSMISSION` in der Klasse `AccessoryPool` referenziert wird. Das Ergebnis besagt, dass $r_{\times}$ fehlgeschlagen ist, da der Programmzustand des fehlerhaften Laufs die beschriebenen Eigenschaften aufweist. Auf die Konfiguration des Fahrzeug bezogen bedeutet dies: Das Hinzufügen der Zubehörpakete ESP *und* Automatische Gangschaltung führte zu dem Fehler.

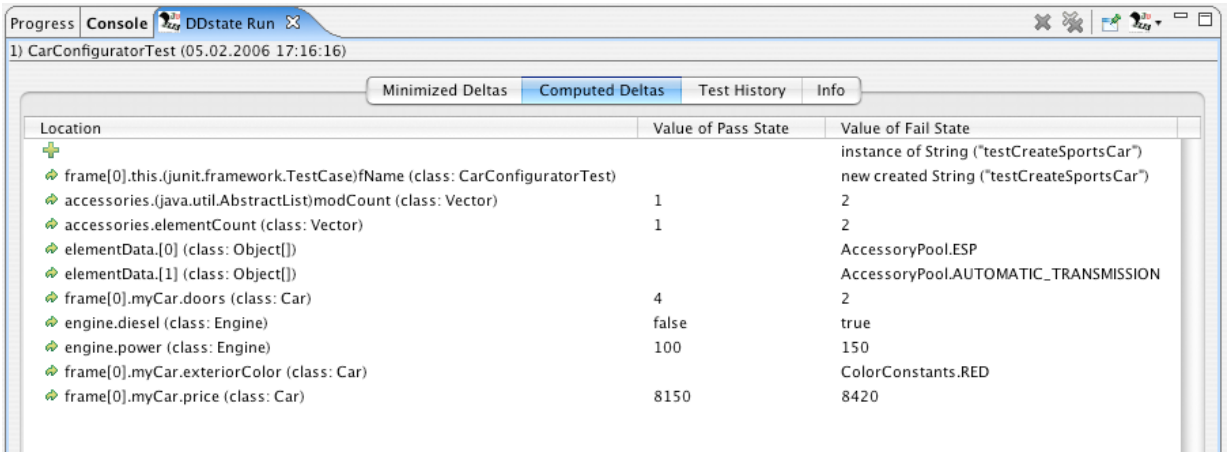


Abbildung 5.5: Ergebnis DDSTATE: Fehlerverursachende Zustandsunterschiede

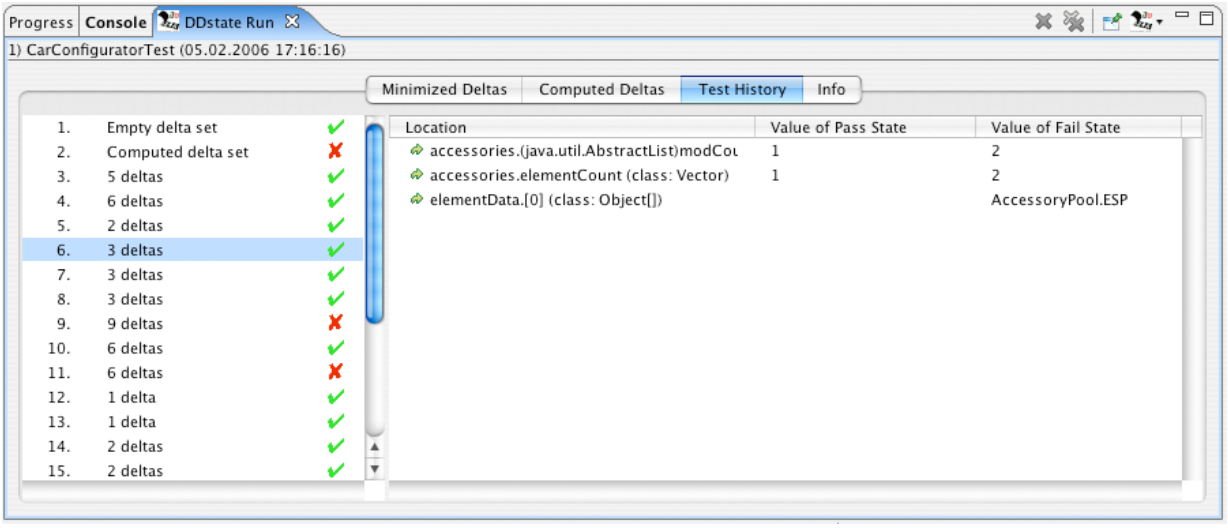


Abbildung 5.6: Ergebnis DDSTATE: Auflistung der verschiedenen von DDSTATE ausgeführten Testläufen mit den jeweiligen Ergebnissen in Form einer Testhistorie.

Neben den minimierten Zustandsunterschieden werden im Register *Computed Deltas* alle von DDSTATE berechneten Unterschiede angezeigt (Abbildung 5.5). Wir sehen, dass insgesamt elf Unterschiede berechnet wurden, von denen jedoch nur drei fehlerrelevant waren. Weiterhin wird ein chronologischer Verlauf der von DDSTATE durchgeführten Tests in dem Register *Test History* dargestellt. Durch die Auswahl eines dargestellten Testlaufs können wir uns die getestete Deltamenge dieses Testlaufs anzeigen lassen (Abbildung 5.6).

5.2 »Suche in der Zeit« – Isolation von fehlerrelevantem Programmcode

Durch die Anwendung von DDSTATE auf zwei Programmzustände können wir die Unterschiede des fehlerhaften Programmzustands auf die fehlerrelevanten Ursachen minimieren. Der Ursprung der fehlerrelevanten Variablen ist allerdings nicht immer so offensichtlich wie bei dem Car-Configurator-Beispiel. Die fehlerrelevanten Variablen stammen aus für den Fehler relevanten Berechnungen. In dem Beispiel haben wir diese Berechnungen in den Testmethoden durchgeführt, indem wir die Konfiguration der Fahrzeuge definiert haben und somit in dem `Car`-Objekt die verschiedenen Attribute gesetzt bzw. verändert haben. Für einen Entwickler wäre es interessant und sehr hilfreich, wenn er direkt die fehlerhafte Stelle innerhalb des Quellcodes gezeigt bekommt. Dazu müssen wir nach dem Defekt suchen, welcher das Fehlschlagen des JUnit-Testfalls verursacht. Wir nennen dies »Suche in der Zeit«, da wir über den kompletten Programmlauf den Moment suchen, an dem der Defekt im Code ausgeführt wird und somit eine Infektion – ein unerwünschter Programmzustand – entsteht.

In beiden JUnit-Testmethoden der Testklasse `CalculatePriceTest` aus Abbildung 5.7 haben wir den Fehler, welcher zu der ungültigen Konfiguration in `CarConfiguratorTest` führte, dahingehend korrigiert, dass das Zubehörpaket Automatische Gangschaltung entfernt wurde. Bei diesem Test überprüfen wir nicht mehr die Konfiguration, sondern die Korrektheit der Preisberechnung der Fahrzeuge. Dabei liegt folgendes vereinfachtes Preismodell zu Grunde: Ein Auto kostet 8000€. Jedes Zubehör kostet einen Aufpreis. Das Winterpaket kostet 150€, Automatikgangschaltung 300€ zusätzlich. Der Preis des elektronischen Stabilitätssystems ESP richtet sich nach dem Basispreis des Autos und beträgt 15 Prozent des Basispreises.

In der Testmethode `testCreateCarBottomUp` erzeugen wir ein Fahrzeug und fügen diesem das Zubehör Winterpaket hinzu. Der erwartete Preis beträgt daher 8150€. Nach der Ausführung von `testCreateSportsCar` sollte der Preis bei 9200€ liegen, das entspricht dem Basispreis inklusive 15 Prozent Preiserhöhung für das ESP. Führen wir den JUnit Test aus, so kommen wir zu folgendem Ergebnis: Die Funktion `testCreateCarBottomUp` läuft ohne Fehler ($r_{✓}$), während die Testmethode `testCreateSportsCar` ein `AssertionFailedError` wirft ($r_{✘}$). Der erwartete und der berechnete Preis stimmen nicht überein.

Verwenden wir wie beim ersten Testfall die »Suche im Raum« und ermitteln die fehlerrelevanten Zustandsunterschiede. Dazu vergleichen wir die Programmzustände beider Testmethoden an den

```
1  public class CalculatePriceTest extends junit.framework.TestCase {
2
3    public void testCreateCarBottomUp() {
4      Car myCar = new Car();
5      myCar.setEngine(new Engine(100, false));
6
7      myCar.setNumberOfDoors((short) 4);
8      myCar.setExteriorColor(ColorConstants.WHITE);
9      myCar.setInteriorColor(ColorConstants.BLACK);
10     myCar.addAccesory(AccessoryPool.COLD_WEATHER_PACKAGE);
11
12     junit.framework.Assert.assertEquals(8150, myCar.getPrice());
13   }
14
15   public void testCreateSportsCar() {
16     Car myCar = Car.createSportsCar();
17
18     myCar.setNumberOfDoors((short) 2);
19     myCar.setExteriorColor(ColorConstants.RED);
20     myCar.setInteriorColor(ColorConstants.BLACK);
21     myCar.addAccesory(AccessoryPool.ESP);
22
23     junit.framework.Assert.assertEquals(9200, myCar.getPrice());
24   }
25 }
```

Abbildung 5.7: JUnit-Testfall: Funktionstest der Preisberechnung des Car-Configurator

Zeilen 12 ($r_{✔}$) und 23 ($r_{✘}$), direkt vor der Überprüfung der Korrektheit des Preises durch JUnit. Die Ausführung von $r_{✘}$ schlägt fehl, da der Wert der Variable `price` des Objekts `myCar` in $P_{✘}$ 8120 betragen hat. Somit wissen wir nur, dass der Preis des Objekts `myCar` in $r_{✘}$ für den Fehler relevant ist. Diese Information hatten wir aber auch schon zuvor, da der JUnit-Testfall schließlich wegen des fehlerhaften Preises fehlschlagen ist. Uns interessiert viel mehr, an welcher Stelle während der Programmausführung die fehlerhafte Berechnung des Preises stattfand – wir möchten den Defekt im Programmcode lokalisieren.

DDSTATE bietet uns eine Möglichkeit den Defekt in dem Programmcode vollautomatisch einzugrenzen. Die Idee ist, nach Statements zu suchen, an den fehlerrelevanten Variablen berechnet oder genutzt werden. Dies bedeutet, wenn wir eine Fehlerursache im Programmzustand finden, suchen wir nach dem Code, durch den diese Ursache entstanden ist. Um die Ursachen im Programmcode zu finden, betrachten wir die Variablen, welche mit den fehlerhaften Deltas im Programmzustand verknüpft sind. Angenommen, es gibt eine Stelle im Programmlauf, an der eine Variable A aufhört, während eine Variable B beginnt fehlerrelevant zu werden. Eine solche Stelle nennen wir *Ursachenübergang* (engl. Cause Transition; kurz: CT). Dieser Ursachenübergang von

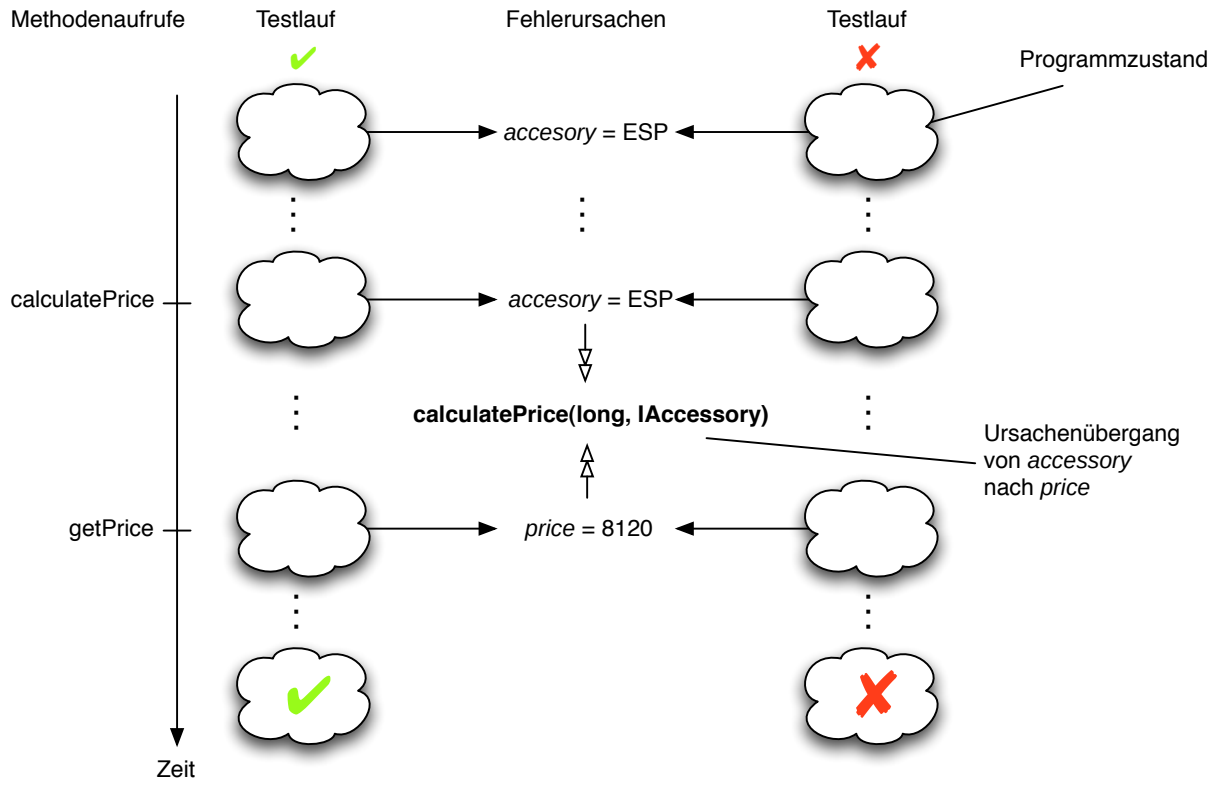


Abbildung 5.8: Eingrenzung des Ursachenübergangs durch Anwendung des CTS-Algorithmus. Zwischen dem Aufruf der Funktion `calculatePrice` und `getPrice` hört die Variable `accessory` auf fehlerrelevant zu sein, während die Variable `price` anfängt fehlerrelevant zu sein.

A nach B ist ein Anfangspunkt von B als Fehlerursache. Ein Ursachenübergang ist deshalb eine gute Stelle, um die Ursachewirkungskette zu unterbrechen und einen Fix in dem Programmcode vorzunehmen. Es hat sich gezeigt, dass der ermittelte Ursachenübergang oft nahe dem Defekt im Quellcode liegt (siehe Cleve u. Zeller, 2005). Für die Testklasse haben wir durch die »Suche im Raum« ermittelt, dass die Variable `price` in dem Zustand von $r_{\text{✘}}$, Fehlerursache ist. Wir suchen daher nach dem Ursachenübergang, an dem die Variable `price` beginnt fehlerrelevant zu werden.

Der CTS Algorithmus zum Finden eines Ursachenübergangs arbeitet nach dem Prinzip der binären Suche[1]. Abbildung 5.8 verdeutlicht dies für die Testklasse `CalculatePriceTest`. Direkt nach dem Aufruf von der Funktion `calculatePrice` war die lokale Variable `accessory` (Parameter der Funktion) fehlerrelevant. Bei dem Aufruf der Funktion `getPrice` war dann die Variable `price` Fehlerursache. Damit können wir sagen, dass der Code, welcher zwischen dem Aufruf der Funktion `calculatePrice` und `getPrice` ausgeführt wurde, für die fehlerhafte

[1] Eine formale Beschreibung der verwendeten Algorithmen findet sich in (Zeller, 2005, Anhang A.3).

Berechnung des Preises verantwortlich ist.

Damit wir Ursachenübergänge auf dem Programmlauf eingrenzen können, müssen wir zunächst mehrere Vergleichspunkte zwischen beiden Programmläufen finden. Zu diesem Zweck zeichnen wir alle Methodenaufrufe für jeden der beiden Programmläufe separat auf. Für jeden Aufruf wird ebenfalls das Ende der Methodenausführung protokolliert. Dabei muss darauf geachtet werden, dass eine Methode auf zwei Arten beendet werden kann – entweder durch eine `return`-Anweisung oder durch eine Exception. Durch das Ergebnis der Aufzeichnung sind wir in der Lage, zu jedem Methodenaufruf innerhalb des Programmlaufs den vollständigen Callstack zu erzeugen. Das Ergebnis ist für jeden Programmlauf eine Menge von chronologisch aufeinander folgenden Callstacks – wir bezeichnen dies als einen *Callstack Trace*. Unter Beachtung der Reihenfolge wird versucht, eine möglichst große Anzahl an Vergleichspunkten zwischen den beiden Callstack Traces zu finden. Dazu verwenden wir den *diff*-Algorithmus (siehe Myers, 1986). Über diesen finden wir die Bereiche an denen in beiden Callstack Traces die Callstacks gleich sind.

In der Startkonfiguration von DDSTATE können wir die Berechnung von Cause Transitions in dem Register *Matchpoint* aktivieren (eine manuelle Eingabe von Vergleichspunkten ist dann nicht mehr nötig). Wenden wir DDSTATE auf die Testklasse `CalculatePriceTest` an, so werden insgesamt drei Ursachenübergänge berechnet:

1. Direkt nach dem Start von $r_{\times}$ konnte keine fehlerrelevante Variable ermittelt werden. Bei dem Aufruf des Methode `addAccesory` in der Klasse `Car` war die lokale Variable `newAccesory` fehlerrelevant – sie wurde auf ESP geändert.

2. Beim ersten Aufruf von `calculatePrice` in der Klasse `AccessoryPool` war die lokale Variable `accessory` fehlerrelevant – sie wurde ebenfalls auf ESP geändert.

3. Beim ersten Aufruf von `getPrice` in der Klasse `Car` war die Variable `price` fehlerrelevant.

An dem Ursachenübergang 1. und 2. sehen wir, dass die falsche Preisberechnung durch die veränderte Zubehörkomponente in $r_{\times}$ verursacht wurde. Da bei dem Aufruf von `getPrice` der Preis Fehlerursache war, bei dem Aufruf von `calculatePrice` jedoch noch nicht, muss die Berechnung des Preis zwischen diesen beiden Methodenaufrufen stattgefunden haben. Wenn wir den Quellcode zu der Methode `calucaltePreise` untersuchen, sehen wir dass der Preis für das ESP anstatt der geforderten 15 Prozent nur um 1,5 Prozent erhöht wird:

```
public static long calculatPrice(long carPrice,
    IAccessory accessory) {
  long result = 0;

  if (accessory == COLD_WEATHER_PACKAGE)
    result = 150;                          // fix price - 150 Euro

  if (accessory == AUTOMATIC_TRANSMISSION)
    result = 300;                          // fix price - 300 Euro

  if (accessory == ESP)
    result = (long) (0.015 * carPrice);    // 15% of the car price

  return result;
}
```

Wir haben damit den Defekt über den kompletten Programmlauf auf einen Methodenaufruf eingeschränkt. Die durch DDSTATE berechneten Ursachenübergänge werden in Form einer Ursache-Wirkungskette, in dem View »DDstate Status« präsentiert (siehe Abbildung 5.9). Dabei wird zu den verschiedenen Vergleichspunkten die fehlerrelevanten Variablen aufgelistet. Durch Hyperlinks ist es dem Benutzer möglich direkt in die angegebenen Funktionen, Klassen oder zu der Deklaration der fehlerrelevanten Variablen zu navigieren.

5.3 Fazit

In diesem Kapitel haben gesehen, wie wir DDSTATE als Debugging-Tool integriert in die Eclipse Entwicklungsumgebung nutzen können. Anhand zweier Anwendungsfälle haben wir demonstriert, wie uns DDSTATE bei der Fehlersuche unterstützt. Durch die »Suche im Raum«, bei der wir die Programmzeitpunkte innerhalb der beiden Testläufe manuell wählen können, werden die fehlerrelevanten Variablen zwischen zwei Programmzuständen berechnet. Durch die Suche nach der Ursache der fehlerrelevanten Variablen, können wir den Fehler im Programm finden.

Durch die Anwendung der »Suche in der Zeit« werden mehrere Vergleichspunkte über die Testläufe automatisch bestimmt. Über diese Vergleichspunkte wird die »Suche im Raum« angewendet, um dadurch Ursachenübergänge innerhalb des fehlschlagenden Testlaufs zu bestimmen. Diese Ursachenübergänge helfen dem Anwender den Defekt im Programmcode einzugrenzen. Der Nachteil der »Suche in der Zeit« gegenüber der »Suche im Raum« ist, dass sie wesentlich aufwendiger ist, da an mehreren Vergleichspunkten die fehlerrelevanten Variablen bestimmt werden müssen.

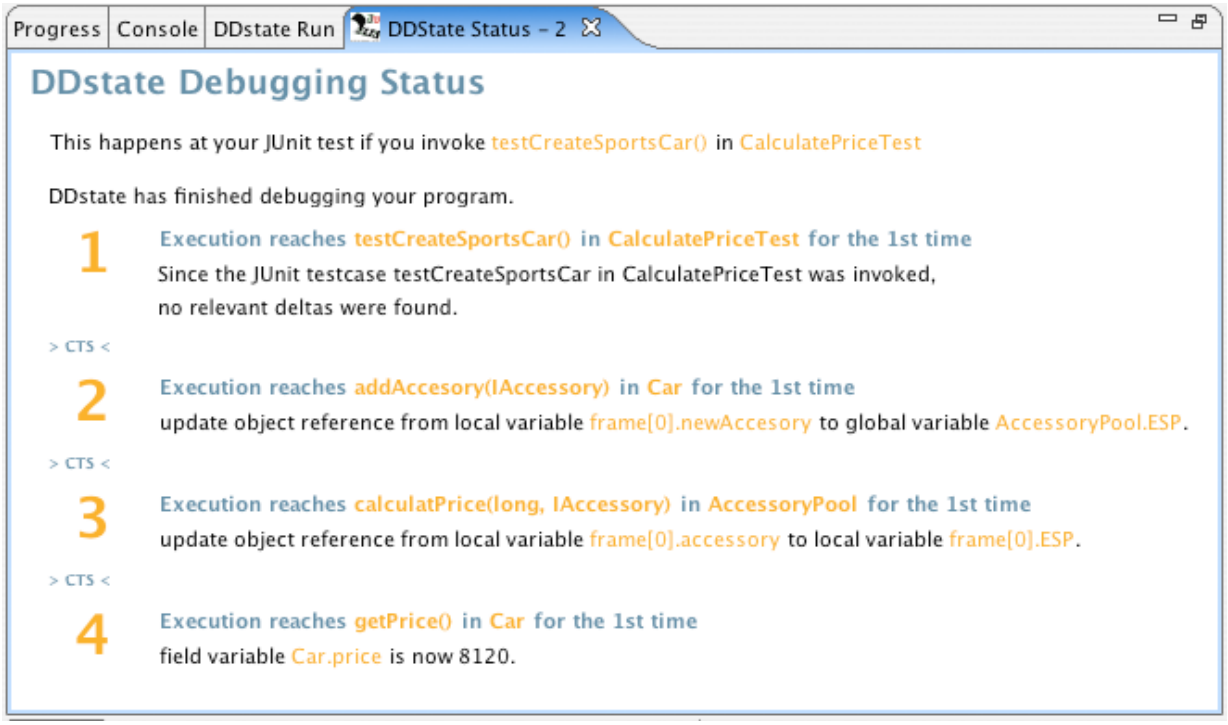


Abbildung 5.9: Die von DDSTATE berechneten Ursachenübergänge werden in Form einer Ursache-Wirkungskette in dem View *DDstate Status* präsentiert. Dieser enthält Informationen über die fehlerrelevanten Variablen, sowie den Zeitpunkt an dem diese berechnet wurden.

6 Grenzen der Anwendung und bekannte Probleme

6.1 Behandlung von multithreaded Anwendungen

DDSTATE unterstützt momentan keine Anwendungen, welche mit mehreren Threads arbeiten (engl. Multithreading). Eine Prämisse bei der Anwendung von DDSTATE ist, dass die Testfälle und somit ihre Ergebnisse reproduzierbar und damit mehrfach auf immer die gleiche Weise ausführbar sind. Da wir den Programmlauf während der Ausführung anhalten, um dort den Programmzustand zu begutachten, gilt aber auch, dass der Programmzustand an einer bestimmten Stelle innerhalb des Programmlaufs immer der Gleiche sein muss. Nur so können wir die, bei dem Zustandsvergleich extrahierten Deltas, erfolgreich anwenden.

Um DDSTATE auf Programmen mit mehreren Trheads anwenden zu können, sind folgende Maßnahmen zu treffen. Wenn wir den Programmzustand extrahieren, sind die Basisvariablen aller Threads zu verwenden. Dieses wäre sicherlich realisierbar, indem wir die bestehenden Algorithmen auf jeden in dem System existierenden Thread anwenden. Von dem Rootknoten müssten dann zunächst alle Threads aufgelöst werden, von dort aus dann die jeweiligen Basisvariablen. Momentan geschieht dies nicht, da wir nur die Basisvariablen des Thread extrahieren, an dem der Haltepunkt ausgelöst wurde. Bei dem Vergleich der Programmzustände müssten dann die Threads mit berücksichtigt werden. Dann stellt sich aber die Frage, wie man mit der Tatsache umgehen soll, dass in dem funktionierenden und fehlerhaften Programmlauf unterschiedliche Threads laufen bzw. aktiv sein können.

Betrachten wir die Definition von Programmzeitpunkt (siehe Definition 1), durch die wir eine Stelle innerhalb eines Programmlaufes eindeutig bestimmt haben, so gilt diese nicht für multithreaded Programme. Um hier eine Stelle innerhalb der Programmausführung eindeutig zu bestimmen, müssten wir für jeden Thread einen Haltepunkt definieren. Durch die Asynchronität bei der Ausführung der Threads kann aber immer noch nicht sichergestellt werden, dass bei mehrmaligem Ausführen des Programms der Programmzustand an einem bestimmten Programmzeitpunkt immer der Gleiche ist. Dieses könnten wir durch ein Capture-Replay Verfahren erreichen, bei dem wir einen Programmlauf aufzeichnen und immer in der gleichen Art und Weise ablaufen lassen.

6.2 Java SUN JRE/SDK 5.0

Seit dem 30. September 2004 gibt es den Nachfolger der Java-Version 1.4.2 – die Version 5.0 mit dem Codenamen Tiger. Neben den Erweiterungen und Änderungen der Java Bibliotheks- und Laufzeitklassen, enthält die neue Version Spracherweiterungen. Zusätzlich zu generischen Typen (so genannten Generics), können wir mittels Annotationen den Quellcode mit Metadaten versehen. Des Weiteren gab es Vereinfachungen bezüglich der Syntax, um über Collections, Maps oder Arrays zu iterieren. Eine vollständige Liste aller Erweiterung findet sich unter Sun (2005a).

Die volle Unterstützung des neuen Release durch Eclipse wurde erst in der Version 3.1 zugesichert, welche am 28. Juni 2005 fertig gestellt wurde. DDSTATE wurde mit und für Eclipse 3.0/3.1 programmiert und unterstützt die Java-Version 1.4.2. Bei Verwendung der neuen Java-Version 5.0 wäre zu prüfen, ob sich Änderungen bezüglich des Speichergraphen (eventuell Überarbeitung des Objektmodells) ergeben, was auch Änderungen beim Vergleich und dem Anwenden der Deltas zur Folge hat. Außerdem müsste geprüft werden, wie die Modifikationen an dem Bytecode vorgenommen werden, da die momentan aktuelle und von DDSTATE verwendete Version 5.1 von BCEL Java 5.0 nicht unterstützt. Einen Test der Modifikation des Bytecodes der Java 5.0 Laufzeitklassen mit Hilfe von BCEL, um den von uns verwendeten Spezialkonstruktor zur Verfügung zu stellen (vgl. Abschnitt 4.3), fiel negativ aus. Die verwendete Version von BCEL 5.1 erzeugt, bezüglich Java 5.0, keinen gültigen Bytecode.

6.3 Eclipse-Bug Nummer 101075

Betrachten wir den funktionierenden und fehlerhaften JUnit-Testlauf aus dem Beispiel von Abbildung 6.1. Es ist leicht ersichtlich, wieso die Funktion `testFailingRun` fehlschlägt. Um den Fehler zu beheben, müsste die lokale Variable `array` gleich `null` sein. Mit Hilfe von DDSTATE sollte es kein Problem sein, den fehlerrelevanten Unterschied zwischen den beiden Läufen festzustellen.

Bei der Überführung des Programmzustands aus dem funktionierenden Lauf in den Programmzustand des fehlerhaften müssen wir ein `Object`-Array (mit keinem Element) erzeugen und die lokale Variable `array` darauf verweisen. Wenn wir die komplette Transformation des funktionierenden in den fehlerhaften Programmzustand vornehmen bricht DDSTATE bei der Zuweisung der Variable `array` zu dem neu erzeugten `Object`-Array ab, da Eclipse folgende Fehlermeldung wirft:

```
com.sun.jdi.InvalidTypeException:
Type of the value not compatible with the expected type
```

Der Grund liegt darin, dass in der momentan aktuellen Version 3.1.1 der Eclipse-Workbench die Zuweisung eines Arrays zu einer als Objekt deklarierten Variable nicht erlaubt ist. Dieses

```
1  public class TestCreateArray extends TestCase {
2
3    public void testPassingRun() {
4      Object array = null;
5      Assert.assertNull(array); // <- compare program state here
6    }
7
8    public void testFailingRun() {
9      Object array = new Object[0];
10     Assert.assertNull(array); // <- compare program state here
11   }
12
13 }
```

Abbildung 6.1: JUnit-Testfall: Zuweisen eines Arrays zu einer als `Object` deklarierten Variable

ist ein Fehler, da nach der Java-Spezifikation Arrays Objekte sind (siehe Lindholm u. Yellin, 1999, Kapitel 2.15). Der Fehler wurde in dem Bugreportsystem von Eclipse (`https://bugs.eclipse.org`) unter der Fehlernummer 101075 eingetragen und sollte in der kommenden Version Eclipse 3.2 (voraussichtliches Erscheinungsdatum: Ende Juni 2006) behoben sein.

7 Fazit

In dieser Arbeit haben wir aufgezeigt, wie fehlerrelevante Ursachen eines fehlschlagenden JUnit-Testlaufs automatisch bestimmt werden können. Diese Automatisierung ist in dem Debugging-Tool DDSTATE implementiert. Durch die vollständige Automatisierung können wir den zeitintensiven und frustrierenden Prozess des Debugging für den Entwickler vereinfachen. Mit DDSTATE unterstützen wir ebenfalls die Berechnung von Ursachenübergängen, indem wir die Suche nach fehlerrelevanten Ursachen systematisch an verschiedenen Stellen im Testlauf anwenden. Die dabei erstellte Diagnose, in Form einer Ursachen-Wirkungskette, kann die Suche nach dem Defekt für den Anwender vereinfachen, da der Defekt oft in unmittelbarer Nähe zu einem Ursachenübergang liegt.

Zur Umsetzung des kompletten Debugging-Prozess auf Programmzuständen, erarbeiteten wir Lösungen zu verschiedenen Teilproblemen. Zum ersten Mal haben wir die für C-Programme bestehenden Verfahren *vollständig und erfolgreich* auf Java portiert. Um einen Programmzustand eines Java-Programms in Form eines Speichergraphen zu erfassen, definierten wir zunächst ein Objektmodell. Im Rahmen dieser Arbeit entwickelten wir Algorithmen, um einen Speichergraphen aus einem Programmzustand zu extrahieren. Durch den Vergleich von Speichergraphen eines funktionierenden und eines fehlerhaften Testlaufs ist es uns möglich, die Ursachen für den Fehler zu berechnen. Aus den berechneten Ursachen sind wir in der Lage, die fehlerrelevanten Ursachen mittels Delta Debugging zu bestimmen.

Anhand von zwei Beispielen haben wir demonstriert, wie DDSTATE verwendet wird, um fehlerrelevante Variablen in einem Programmzustand zu bestimmen, und die Ausführung des Defekts innerhalb eines fehlerhaften JUnit-Testlaufs zu lokalisieren. DDSTATE haben wir als Erweiterung zu Eclipse, einer der populärsten freien Java-Entwicklungsumgebungen, entwickelt. Durch die Implementierung und Integration von DDSTATE als Plugin in Eclipse, kann jeder Anwender seine Installation von Eclipse mit der Funktionalität eines Delta-Debuggers erweitern.

Die Thematik der Fehlersuche auf Programmzuständen ist umfangreich, und somit bietet diese Arbeit Möglichkeiten zur Verbesserung und Erweiterung der bestehenden Funktionalität:

- Diese Arbeit enthält keine Untersuchungen, ob die von DDSTATE erstellte Fehleranalyse Zeit und damit verbundene Kosten einsparen kann. Durch eine weitergehende Studie wären wir in der Lage diese These zu belegen oder widerlegen.
- Die Suche nach Ursachenübergängen beschränkt sich momentan auf Vergleichspunkte, die direkt an Methodenaufrufe geknüpft sind. Durch Verfeinerung der bereits gefundenen

Ursachenübergänge bis auf Codezeilen oder sogar einzelne Statements, würde der Defekt im Programmcode näher eingegrenzt.

- Für die Berechnung der fehlerrelevanten Ursachen benötigen wir, neben dem fehlerhaften Testfall, einen funktionierenden Testfall. Indem wir alle in der Testklasse implementierten Testmethoden betrachten, wären wir in der Lage den benötigten funktionierenden Testlauf automatisch zu bestimmen.
- Die Repräsentation der Ergebnisse mit Hilfe von Visualisierungen, würden die Ergebnisse verdeutlichen und damit für die Anwender intuitiver präsentieren. Denkbar wäre eine Abbildung der Knoten und Kanten innerhalb des Speichergraphens, an denen fehlerrelevante Änderungen bestimmt wurden.

A Anhang

A.1 Installation von DDstate

Die Installation von DDSTATE für Eclipse 3.1.x erfolgt über das Internet, unter der Verwendung von dem in Eclipse integrierten Installations- und Updatedmechanismus:

1. Öffnen des Installations-Wizard durch Klicken von *Help* > *Software Updates* > *Find and Install* in der Eclipse Workbench.
2. Auswahl des Buttons »Search for new features to install« und *Next* klicken.
3. Anlegen einer neuen »Remote Update Site« danach mit *Finish* bestätigen:
   - Name: `DDstate`
   - URL:

     `http://www.st.cs.uni-sb.de/eclipse/update-site/ddstate`
4. Expandieren der Update-Site:

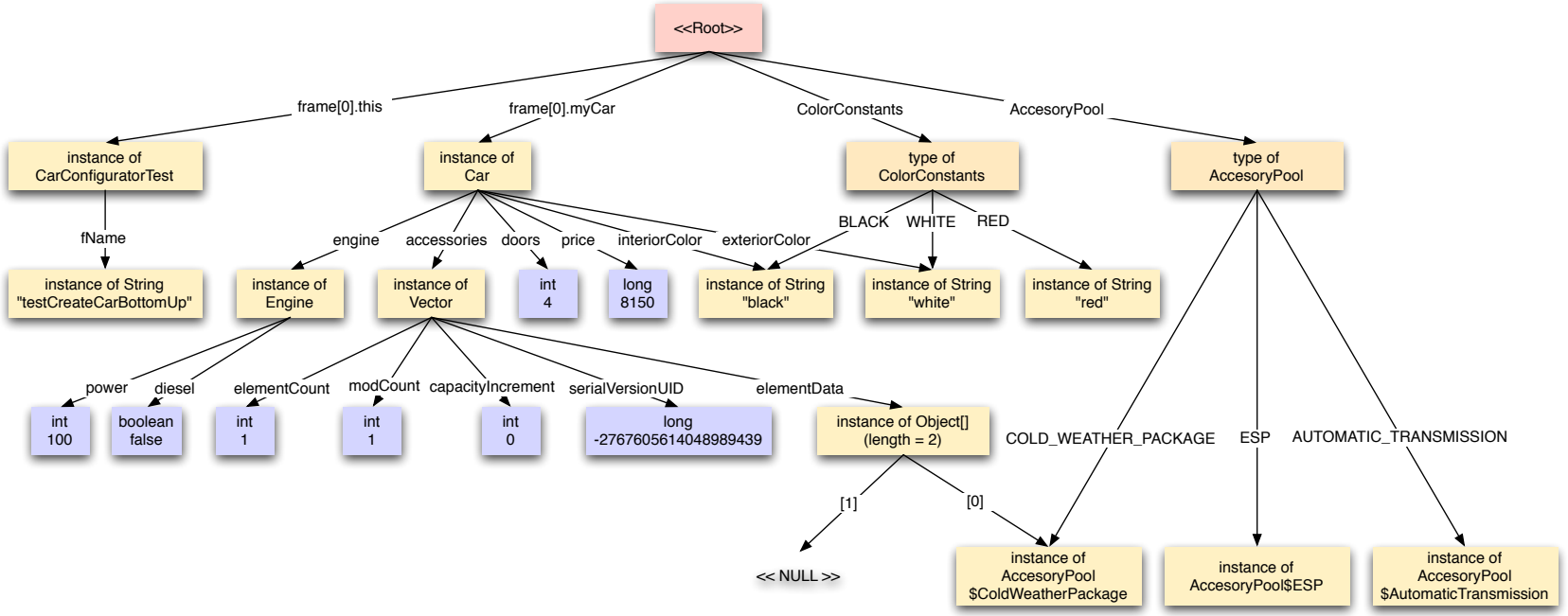
   Unter der Kategorie »Automated Debugging« stehen insgesamt zwei Features zur Verfügung:
   - »DDstate«: DDSTATE – Delta Debugger für Eclipse
   - »DDstate Visualizer«: eine Erweiterung zu DDSTATE zur Visualisierung von Speichergraphen[1].
5. Auswählen der gewünschten Features und Bestätigen der Auswahl mit *Next*. Danach den weiteren Anweisungen von Eclipse folgen.
6. DDSTATE wird automatisch heruntergeladen und installiert. Nach der Installation ist ein Neustart der Eclipse Anwendung nötig.

[1] DDSTATE VISUALIZER verwendet Grappa, ein Java-Framework für Graphviz (AT&, 2005). Graphviz wurde von AT&T entwickelt und ist ein Tool um automatisch Graphen zu visualisieren.

A.2 Speichergraphen

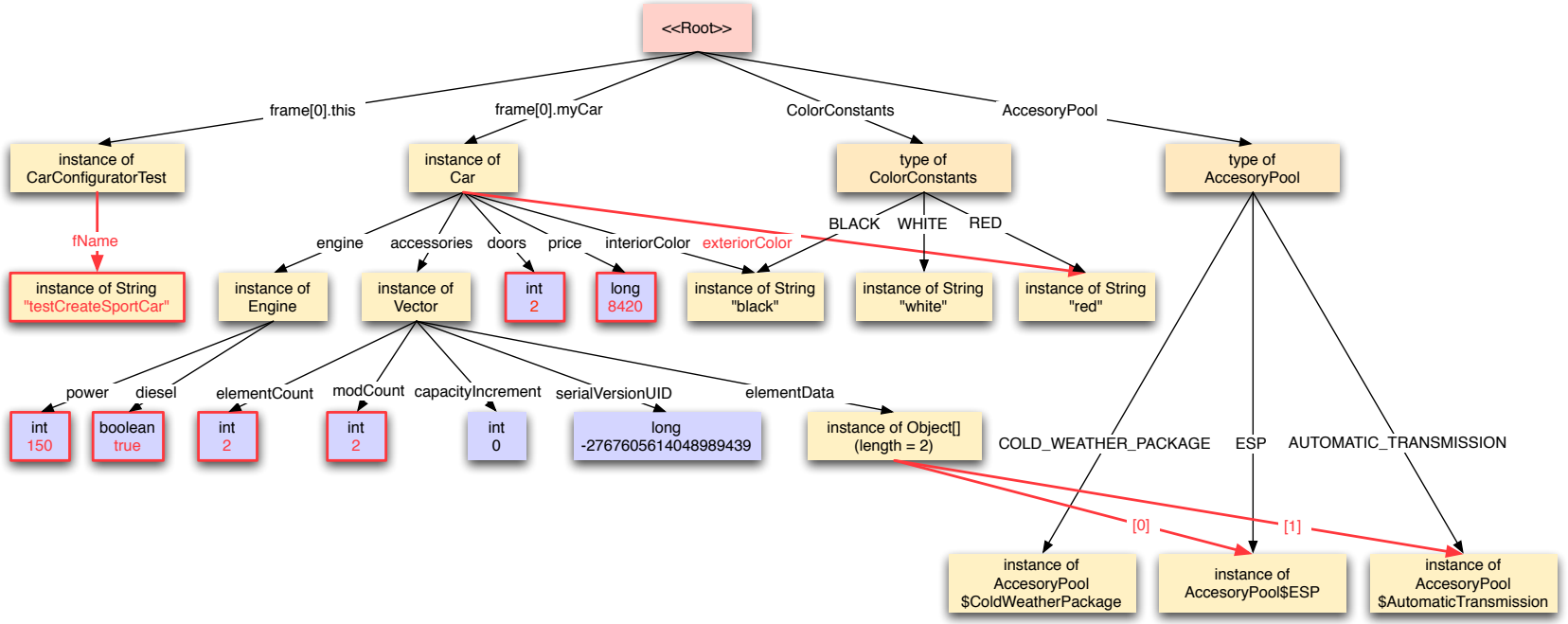
Die folgenden Abbildungen sind Speichergraphen, welche die Programmzustände aus dem »Car-Configurator «-Beispiel repräsentieren.

Speichergraph aus $P_{\checkmark}$



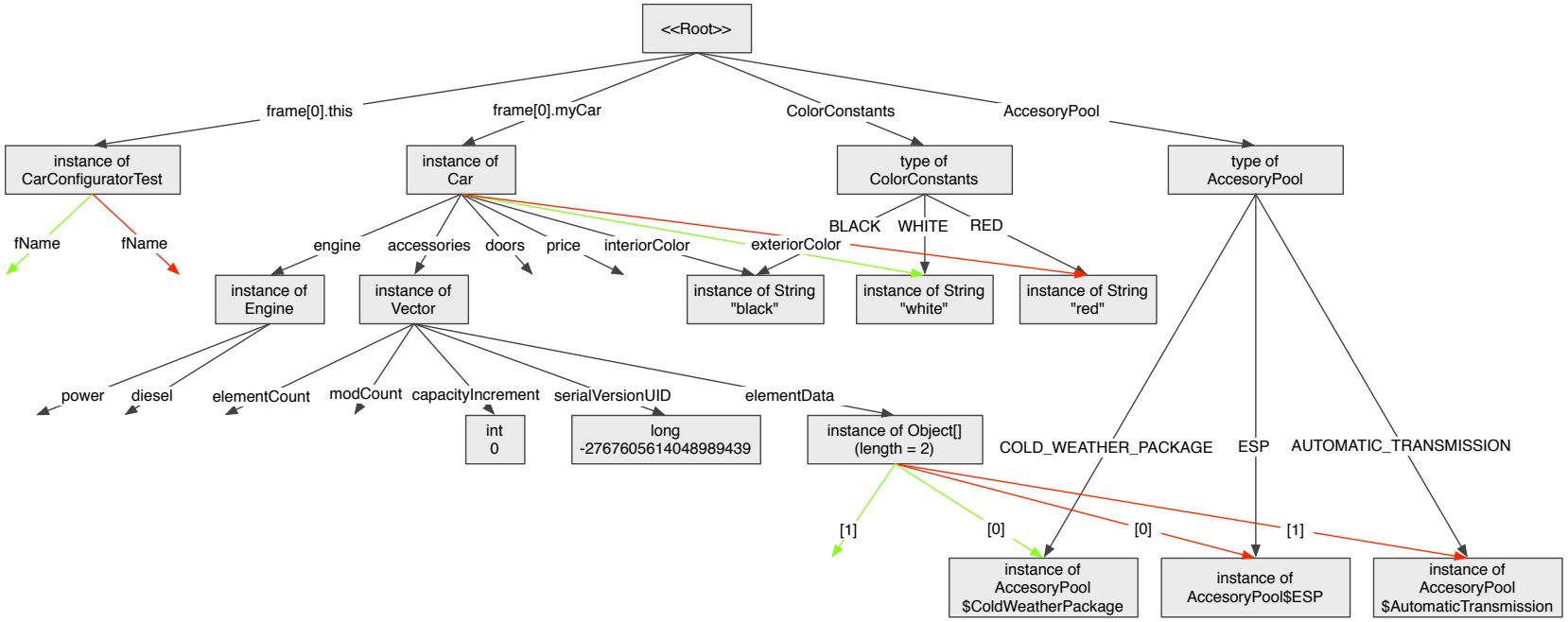
Speichergraph aus $P_{✘}$

Die rot markierten Kanten und Knoten repräsentieren die Unterschiede zu dem Speichergraphen aus dem funktionierenden Programmlauf. Rot markierte Kanten zeigen, gegenüber $G_{✔}$, auf unterschiedliche Objekte. Rot markierte Knoten haben unterschiedliche Werte bzw. existieren nur in $G_{✘}$.



Größter gemeinsamer Teilgraph

Dieser Graph repräsentiert den größten gemeinsamen Teilgraphen aus $G_{✔}$ und $G_{✘}$. Kanten die grün ($G_{✔}$) bzw. rot ($G_{✘}$) dargestellt sind, existieren in beiden Graphen, zeigen dort aber auf unterschiedliche Knoten.



A.3 Quelltexte

AccessoryPool

```
1  public class AccessoryPool {
2
3    /** Constants for the different car accessories */
4    public final static IAccessory AUTOMATIC_TRANSMISSION =
5      new AutomaticTransmission();
6    public final static IAccessory COLD_WEATHER_PACKAGE =
7      new ColdWeatherPackage();
8    public final static IAccessory ESP = new ESP();
9
10   /**
11    * @return calculate and return the price of the given accessory
12    *   based on the car price.
13    */
14   public static long calculatPrice(long carPrice,
15       IAccessory accessory) {
16     long result = 0;
17
18     if (accessory == COLD_WEATHER_PACKAGE)
19       result = 150;                        // fix price - 150 Euro
20
21     if (accessory == AUTOMATIC_TRANSMISSION)
22       result = 300;                        // fix price - 300 Euro
23
24     if (accessory == ESP)
25       result = (long) (0.015 * carPrice); // 15% of the car price
26
27     return result;
28   }
29
30   // inner classes for the different accessories of this pool
31   static class ColdWeatherPackage implements IAccessory {
32     public String getDescription() {
33       return "This includes heatable front seats and " +
34           "front heated windshield washer nozzles";
35     }
36   }
37
38   static class ESP implements IAccessory {
39     public String getDescription() {
40       return "Electronic Stabilization Program";
41     }
```

```
42    }
43
44    static class AutomaticTransmission implements IAccessory {
45      public String getDescription() {
46        return "4-Speed Automatic Transmission";
47      }
48    }
49
50  }
```

Car

```
1   import java.util.Vector;
2
3   public class Car {
4
5     // Engine of the car
6     private Engine engine = null;
7
8     // Color settings
9     private String exteriorColor = null;
10    private String interiorColor = null;
11
12    // Number of the doors of the car
13    private short doors = 3;
14
15    // Basis price in Euro
16    private long price = 8000;
17
18    // All extras of the car
19    public Vector accessories = new Vector(2);
20
21    public Car() {
22    }
23
24    /**
25     * @return a new car with a gas engine (60PS).
26     */
27    public static Car createBaseLine() {
28      Car result = new Car();
29      result.setEngine(new Engine(60, false));
30
31      return result;
32    }
33
```

```
34    /**
35     * @return a new car with a diesel engine (150PS).
36     */
37    public static Car createSportsCar() {
38      Car result = new Car();
39      result.setEngine(new Engine(150, true));
40
41      return result;
42    }
43
44    /**
45     * @return the engine of this car.
46     */
47    public Engine getEngine() {
48      return this.engine;
49    }
50
51    /**
52     * Set the engine of this car.
53     */
54    public void setEngine(Engine engine) {
55      this.engine = engine;
56    }
57
58    /**
59     * Add the given accesory to the list of accessories of this car.
60     */
61    public void addAccesory(IAccessory newAccesory) {
62      this.accessories.add(newAccesory);
63
64      long accessoryPrice =
65        AccessoryPool.calculatPrice(this.price, newAccesory);
66      this.price += accessoryPrice;
67    }
68
69    /**
70     * @return true if this car has the given accesory, otherwise false
71     */
72    public boolean hasAccesory(IAccessory accessory) {
73      return this.accessories.contains(accessory);
74    }
75
76    /**
77     * @return true if the configuration of this car is valid,
78     *  otherwise false.
```

```
79     */
80   public boolean isConfigurationValid() {
81     boolean result = true;
82
83     if (engine == null)
84       result = false;
85
86     if (this.hasAccesory(AccessoryPool.ESP) &&
87         this.hasAccesory(AccessoryPool.AUTOMATIC_TRANSMISSION))
88       result = false;
89
90     if (exteriorColor == null)
91       result = false;
92
93     if (interiorColor == null)
94       result = false;
95
96     return result;
97   }
98
99   /**
100   * @return the exterior color of this car.
101   */
102  public String getExteriorColor() {
103    return this.exteriorColor;
104  }
105
106  /**
107   * Set the exterior color of this car.
108   */
109  public void setExteriorColor(String exteriorColor) {
110    this.exteriorColor = exteriorColor;
111  }
112
113  /**
114   * @return the interior color of this car.
115   */
116  public String getInteriorColor() {
117    return this.interiorColor;
118  }
119
120  /**
121   * Set the interior color of this car.
122   */
123  public void setInteriorColor(String interiorColor) {
```

```
124     this.interiorColor = interiorColor;
125   }
126
127   /**
128    * @return the number of doors of this car.
129    */
130   public short getNumberOfDoors() {
131     return this.doors;
132   }
133
134   /**
135    * Set the number of doors of this car.
136    */
137   public void setNumberOfDoors(short numberOfDoors) {
138     this.doors = numberOfDoors;
139   }
140
141   /**
142    * @return the calculated price for this car.
143    */
144   public long getPrice() {
145     return this.price;
146   }
147
148 }
```

ColorConstants

```
1  /**
2   * This interface describes some color constants for the interior
3   * and exterior color of the car.
4   */
5  public class ColorConstants {
6
7    public static String BLACK = "black";
8    public final static String WHITE = "white";
9    public final static String RED = "red";
10 }
```

Engine

```
1  /**
2   * The class describes a car engine.
3   */
4  public class Engine {
```

```
5
6     private boolean diesel;
7     private int power;
8
9     public Engine() {}
10
11    public Engine(int power, boolean diesel) {
12      this.power = power;
13      this.diesel = diesel;
14    }
15
16    /**
17     * @return the kind of fuel for this engine.
18     */
19    public boolean isDiesel() {
20      return this.diesel;
21    }
22
23    /**
24     * @return the power of this engine.
25     */
26    public int getPower() {
27      return this.power;
28    }
29
30  }
```

IAccessory

```
1  /**
2   * This interface describes a car accessory.
3   */
4  public interface IAccessory {
5
6    /**
7     * @return a human description for this accessory.
8     */
9    public String getDescription();
10
11 }
```

Abbildungsverzeichnis

Literatur

[Apa 2005]
Apache Software Foundation: *BCEL - Byte Code Engineering Library - Homepage*. Dezember 2005. – http://jakarta.apache.org/bcel

[Arthorne u. Laffra 2004]
ARTHORNE, John ; LAFFRA, Chris: *Official Eclipse 3.0 Faq (Eclipse Series)*. Addison-Wesley Professional, 2004. – ISBN 0–321–26838–5

[AT& 2005]
AT&T Research: *Graphviz - Graph Visualization Software*. September 2005. – http://www.graphviz.org

[Barrow u. Burstall 1976]
BARROW, H.G. ; BURSTALL, R.M.: Subgraph isomorphism, matching relational structures and maximal cliques. In: *Information Processing Letters* 4 (1976), Nr. 4

[Bloch 2001]
BLOCH, Joshua: *Effective Java programming language guide*. Mountain View, CA, USA : Sun Microsystems, Inc., 2001. – ISBN 0–201–31005–8

[Boehm 1981]
BOEHM, Barry W.: *Software Engineering Economics*. Prentice-Hall, 1981

[Bouillon 2004]
BOUILLON, Philipp: *A Framework for Delta Debugging in Eclipse*, Saarland university, Diplomarbeit, Mai 2004

[Bron u. Kerbosch 1973]
BRON, Coen ; KERBOSCH, Joep: Algorithm 457: finding all cliques of an undirected graph. In: *Commun. ACM* 16 (1973), Nr. 9, S. 575–577. http://dx.doi.org/10.1145/362342.362367. – DOI 10.1145/362342.362367. – ISSN 0001–0782

[Cleve u. Zeller 2005]
CLEVE, Holger ; ZELLER, Andreas: Locating causes of program failures. In: *ICSE '05: Proceedings of the 27th international conference on Software engineering*. New York, NY, USA : ACM Press, 2005. – ISBN 1–59593–963–2, S. 342–351

[D'Anjou u. a. 2004]
D'ANJOU, Jim ; FAIRBROTHER, Scott ; KEHN, Dan ; KELLERMAN, John ; MCCARTHY, Pat: *Java(TM) Developer's Guide to Eclipse, The (2nd Edition)*. Addison-Wesley Professional, 2004. – ISBN 0321305027

[Diehl 2002]
DIEHL, Stephan (Hrsg.): *LNCS State-of-the-Art Survey*. Bd. 2269: *Software Visualization*. Springer Verlag, 2002

[Gamma u. Beck 2003]
GAMMA, Erich ; BECK, Kent: *Contributing to Eclipse: Principles, Patterns, and Plugins*. Redwood City, CA, USA : Addison Wesley Longman Publishing Co., Inc., 2003. – ISBN 0321205758

[IBM 2005]
IBM OTI Labs: *eclipse.org - Homepage*. September 2005. – http://www.eclipse.org

[JUn 2005]
JUnit: *junit.org - Homepage*. November 2005. – http://www.junit.org

[Lauer 2004]
LAUER, Christoph: *Memory Graphs for Java Programs*, Universität des Saarlandes, Diplomarbeit, April 2004

[Lindholm u. Yellin 1999]
LINDHOLM, Tim ; YELLIN, Frank: *The Java(TM) Virtual Machine Specification (2nd Edition)*. Addison-Wesley Professional, 1999. – http://java.sun.com/docs/books/vmspec. – ISBN 0201432943

[Mehlmann 2005]
MEHLMANN, Martin: *Bestimmen von Fehlerursachen in Programmläufen an Vergleichspunkten mit abweichendem Kontrollfluss*, Universität des Saarlandes, Diplomarbeit, November 2005

[Metsker 2002]
METSKER, Steven J.: *The Design Patterns Java Workbook*. Boston, MA, USA : Addison-Wesley Longman Publishing Co., Inc., 2002. – ISBN 0–201–74397–3

[Myers 1986]
MYERS, Eugene W.: An $O(ND)$ Difference Algorithm and Its Variations. In: *Algorithmica* 1 (1986), Nr. 2, S. 251–266

[Sof 2005a]
Software Engineering Chair - Saarland University: *AskIgor - Automated Debugging Service*. September 2005. – http://www.st.cs.uni-sb.de/askigor

[Sof 2005b]
Software Engineering Chair - Saarland University: *Debugging Tools for Eclipse*. Oktober 2005. – http://www.st.cs.uni-sb.de/eclipse

[Sosnoski 2003]
SOSNOSKI, Dennis: Java programming dynamics, Part 1: Classes and class loading. (2003), April. – http://www-128.ibm.com/developerworks/java/library/j-dyn0429

[Sun 2005a]
Sun Microsystems, Inc.: *Java™ - Homepage*. Dezember 2005. – http://java.sun.com

[Sun 2005b]
Sun Microsystems, Inc.: *Java™ Platform Debugger Architecture*. November 2005. – http://java.sun.com/j2se/1.4.2/docs/guide/jpda/architecture.html

[Szurszewski 2003]
SZURSZEWSKI, Joe: We Have Lift-off: The Launching Framework in Eclipse. (2003), Januar. – http://www.eclipse.org/articles/Article-Launch-Framework/launch.html

[Ullenboom 2006]
ULLENBOOM, Christian: *Java ist auch eine Insel - Programmieren mit der Java Standard Edition Version 5*. Galileo Computing, 2006. – http://www.galileocomputing.de/openbook/javainsel5. – ISBN 3–89842–747–1

[Wikipedia 2005]
WIKIPEDIA: *Programmfehler — Wikipedia, die freie Enzyklopädie*. 2005. – http://de.wikipedia.org/w/index.php?title=Programmfehler&oldid=10971579 [Online; abgerufen am 30.11.2005; Version vom 02:20, 21 November 2005]

[Wright 2003]
WRIGHT, Darin: Launching Java Applications Programmatically. (2003), August. – http://www.eclipse.org/articles/Article-Java-launch/launching-java.html

[Wright 2004]
WRIGHT, Darin: How to write an Eclipse debugger. (2004), August. – http://www.eclipse.org/articles/Article-Debugger/how-to.html

[Zeller 1999]
ZELLER, Andreas: Yesterday, my program worked. Today, it does not. Why? In: *Proceedings of Joint 7th European Software Engineering Conference (ESEC) and 7th ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE-7)* Bd. LNCS 1687, Springer Verlag, 1999

[Zeller 2002]
ZELLER, Andreas: Isolating Cause-Effect Chains from Computer Programs. In: *Proc. ACM SIGSOFT International Symposium on the Foundations of Software Engineering (FSE)*. Charleston, South Carolina, November 2002

[Zeller 2005]
ZELLER, Andreas: *Why Programs Fail: A Guide to Systematic Debugging*. San Francisco, CA, USA : Morgan Kaufmann Publishers Inc., 2005. – http://www.whyprogramsfail.com. – ISBN 1558608664

[Zeller u. Hildebrandt 2002]
ZELLER, Andreas ; HILDEBRANDT, Ralf: Simplifying and Isolating failure-inducing input. In: *IEEE Transactions on Software Engineering* 28 (2002), Februar, Nr. 2, S. 183–200

[Zimmermann u. Zeller 2002]
ZIMMERMANN, Thomas ; ZELLER, Andreas: Visualizing Memory Graphs. In: *Software Visualization Diehl (2002)*. Springer Verlag, 2002

Selbstständigkeitserklärung

Hiermit erkläre ich, dass ich die vorliegende Diplomarbeit zum Thema »Automatisches Bestimmen fehlerverursachender Programmzustände in Eclipse« vollkommen selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt habe. Des Weiteren habe ich sämtliche Zitate kenntlich gemacht.

St. Ingbert, 12. Februar 2006

. . . . . .
Karsten Lehmann