Dynamische Code-Evolution für Java

Thomas Würthinger

Institut für Systemsoftware
Johannes Kepler University Linz
wuerthinger@ssw.jku.at

Abstract: Dynamische Code-Evolution ermöglicht strukturelle Änderungen an laufenden Programmen. Das Programm wird temporär angehalten, der Programmierer verändert den Quelltext und dann wird die Ausführung mit der neuen Programm-Version fortgesetzt.

Diese Arbeit beschreibt einen neuartigen Algorithmus für die unlimitierte dynamische Neudefinition von Java-Klassen in einer virtuellen Maschine. Die unterstützten Änderungen beinhalten das Hinzufügen und Entfernen von Feldern und Methoden sowie Veränderungen der Klassenhierarchie. Der Zeitpunkt der Veränderung ist nicht beschränkt und die aktuell laufenden Ausführungen von alten Versionen einer Methode werden fortgesetzt. Mögliche Verletzungen der Typsicherheit werden erkannt und führen zu einem Abbruch der Neudefinition. Die entwickelten Techniken können die Entwicklung neuer Programme beschleunigen sowie den Versionswechsel ohne Abschaltpause von Server-Anwendungen ermöglichen. Die Arbeit präsentiert auch ein Programmiermodell für sichere dynamische Aktualisierungen und diskutiert nützliche Limitierungen, die es dem Programmierer ermöglichen, über die semantische Korrektheit einer Aktualisierung Schlussfolgerungen zu ziehen.

Alle Algorithmen sind in der Java HotSpot VM implementiert und es wird derzeit seitens Oracle an der Integration in die offizielle Java-Version gearbeitet. Die Evaluierung zeigt, dass die neuen Fähigkeiten weder vor noch nach einer dynamischen Veränderung einen negativen Einfluss auf die Spitzenleistung der virtuellen Maschine haben.

1 Einführung

Der Eingriff in das Verhalten eines laufenden Programms wurde bereits früh in der Geschichte der Informatik bearbeitet [Fab76]. Die Forschung fokussierte dabei auf prozedurale Programmiersprachen: Bei einem Eingriff ersetzte man die Definitionen von Funktionen und es gab eigene Methoden zur Umwandlung der Daten. Mit der Einführung von objekt-orientierten Programmiersprachen wurden Klassendefinitionen und Subtypbeziehungen ein wichtiger Teil eines Programms. Die dynamische Veränderung eines Programms muss in diesem Kontext auch das Layout von existierenden Objekten sowie die Semantik von Methodenaufrufen aufgrund der aktuellen Klassenhierarchie berücksichtigen.

Die Benutzung einer virtuellen Maschine (VM) zur Ausführung von Programmen hilft beim Lösen dieser neuen Herausforderungen: Eine VM erhöht die Möglichkeiten führ dynamische Code-Evolution aufgrund der zusätzlichen Abstraktionsschicht zwischen dem

ausgeführten Programm und der Hardware. Die Hauptaufgaben dieser Zwischenschicht sind automatische Speicherverwaltung, dynamisches Klassenladen und Programmverifikation. Die in dieser Arbeit präsentierten Algorithmen für die dynamische Veränderung von Klassendefinitionen verwenden die existierende Infrastruktur der VM.

Die dynamische Veränderung von Java-Programmen ist derzeit unter dem Namen "hotswapping" bekannt, da sie auf das Austauschen von Methodendefinitionen beschränkt ist. Die Verbesserung dieser Funktionalität um zusätzliche Veränderungsmöglichkeiten ist von hoher Priorität für viele Java-Programmierer. Dies zeigt sich unter anderem auf der Oracle-Webseite für "requests for enhancements" wo diese Verbesserung zu den am meisten unterstützten Anfragen gehöhrt (Bug ID: 4910812) [Ora11].

2 Stufen der Code-Evolution

Aufgrund unserer Erfahrungen bei der Implementierung der Prototyp-VM schlagen wir die Unterscheidung der folgenden vier Stufen der Code-Evolution vor, die sich durch die Komplexität der Implementierung in einer VM unterscheiden:

Austauschen von Methodendefinitionen: Die einfachste mögliche Veränderung ist das Austauschen der Bytecodes einer Methode. Diese Stufe ist bereits in aktuellen Produkt-VMs implementiert und wird "hotswapping" genannt.

Hinzufügen und Entfernen von Methoden: Die VM verwaltet eine Tabelle mit Zeigern auf die virtuellen Methoden für jede Klasse. In einem objekt-orientierten Programm kann das Hinzufügen oder Entfernen einer Methode zu einer Veränderung der Tabelle in der betroffenen Klasse oder in Subklassen führen. Weiters muss Maschinencode mit Referenzen auf betroffene Methoden invalidiert oder neu berechnet werden.

Hinzufügen und Entfernen von Feldern: Bis zu dieser Stufe haben die Veränderungen nur die Metadaten der VM beeinflusst. Jetzt müssen auch die aktuell existierenden Objektinstanzen des laufenden Programms modifiziert werden. Auch Maschinencode der vom Layout von Klassen abhängig ist muss invalidiert werden.

Hinzufügen und Entfernen von Supertypen: Diese Stufe ist die komplexeste mögliche Veränderung für objekt-orientierte Sprachen. Die möglichen Auswirkungen auf die virtuelle Maschinen beeinhalten zusätzlich zu den Auswirkungen aller vorherigen Stufen noch die Möglichkeit, dass das Typsystem verletzt ist.

Die Veränderung eines Java-Programms kann auch danach klassifiziert werden, ob sie binär kompatibel ist [Dmi01]. Die hellgrauen Bereiche in Abbildung 1 stellen binär kompatible Veränderungen dar, die dunkelgrauen Bereiche binär inkompatible Veränderungen. Wir beschreiben Lösungen für das Problem von binär inkompatiblen Veränderungen in Abschnitt 5. Die im Rahmen der Arbeit entwickelte Prototyp-VM ist die erste VM, die alle vorgestellten Stufen der Code-Evolution unterstützt.

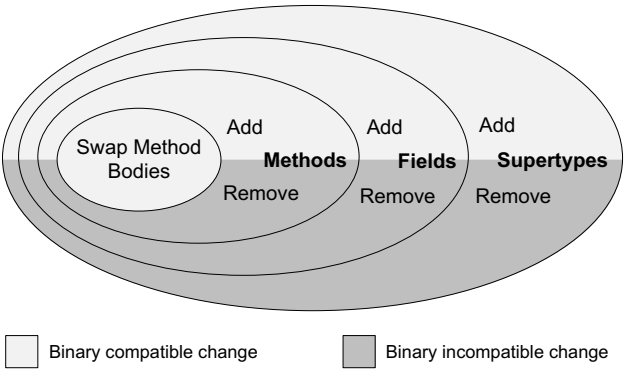

Abbildung 1: Stufen der Code-Evolution.

3 Anwendungsbereiche

Dynamische Code-Evolution kann in verschiedenen Bereichen eingesetzt werden, die jeweils ihre eigenen speziellen Anforderungen mitbringen. Wir unterscheiden vier Hauptanwendungsbereiche:

Beschleunigte Programmentwicklung. Wenn ein Programmierer häufig kleine Veränderungen einer Applikation mit einer langen Startzeit macht, hilft dynamische Code-Evolution die Produktivität bei der Entwicklung zu erhöhen. Anstatt das Programm jedesmal neu zu starten, kann der Programmierer sofort nach der Veränderung das geänderte Verhalten der Anwendung beobachten.

Langlebige Server-Anwendungen. Kritische Server-Anwendungen, die nicht heruntergefahren werden dürfen, können nur mit dynamischer Code-Evolution verändert werden. Für diese Anwendung ist es wichtig, dass das Programm im Normalbetrieb nicht langsamer läuft und ein besonderes Augenmerk liegt auf der Korrektheit einer Veränderung.

Dynamische Sprachen. Dynamische Veränderungen zur Laufzeit sind ein fixer Bestandteil vieler dynamischer Sprachen und die Unterstützung von dynamische Code-Evolution auf VM-Ebene kann die Implementierung dieser dynamischer Sprachen vereinfachen.

Dynamische Aspekt-Orientierte Programmierung. Code-Evolution ist auch relevant für aspekt-orientierte Programmierung (AOP). Es gibt verschiedene dynamische AOP-Werkzeuge deren Limitierungen sich in den beschränkten Möglichkeiten zur Code-Evolution begründen [CST03, VBAM09]. Diese Werkzeuge profitieren sofort von den neuen Code-Evolution-Algorithmen.

Der Fokus unserer Implementierung ist die Unterstützung einer beschleunigten Programmentwicklung, da dies der populärste Anwendungsfall von Code-Evolution ist. Im Rahmen

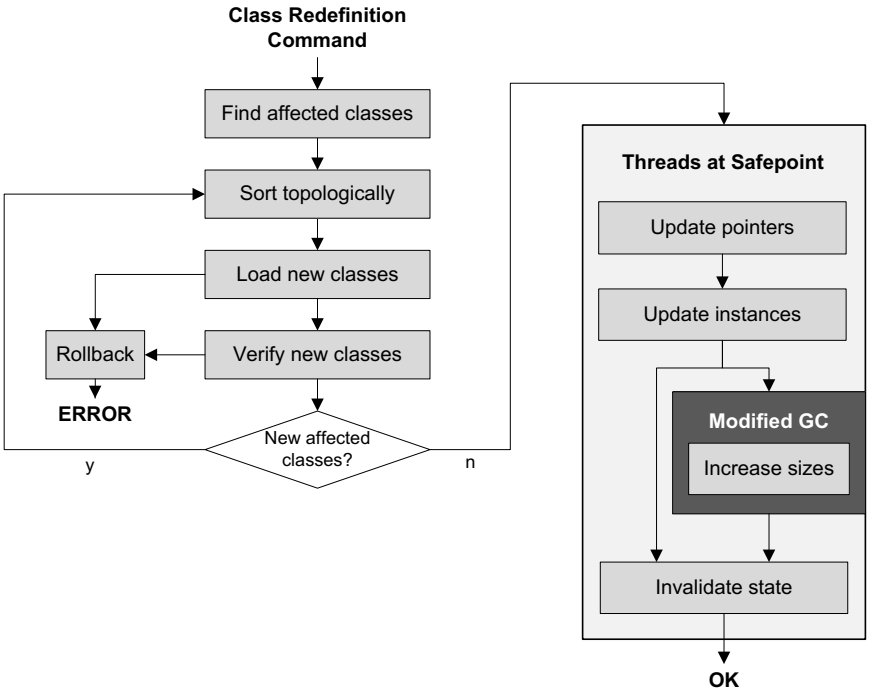

Abbildung 2: Schritte des Algorithmus zur Klassenredefinition.

der Arbeit gehen wir aber insbesondere auch auf die Herausforderungen in bezug auf Korrektheit einer Veränderung ein, die bei langlebigen Server-Anwendungen eine große Rolle spielt (siehe Abschnitt 6).

4 Algorithmus

Unser Algorithmus für unlimitierte Klassenredefinition ist als Modifikation einer Java VM implementiert. Die in dieser Arbeit vorgestellten generellen Konzepte sind jedoch generell auf virtuelle Maschinen anwendbar, die statisch typisierte und objekt-orientierte Programme ausführen. Für unseren Prototyp haben wir eine Produkt-VM anstatt einer Forschungs-VM gewählt, um sicherzugehen, dass die Evaluierungsergebnisse in einer Produkt-Umgebung gültig sind und die Algorithmen auch in einer hochoptimierten VM anwendbar sind.

Abbildung 2 gibt eine Übersicht der Schritte, die von unserem Algorithmus zur Klassenredefinition ausgeführt werden. Als ersten Schritt erzeugt der Algorithmus eine Liste der von der Redefinition betroffenen Klassen, die auch die Subklassen von redefinierten Klassen enthält. Diese Liste wird topologisch auf Basis der Subtyp-Beziehungen sortiert

und definiert im weiteren Verlauf die Reihenfolge in der die Klassen verarbeitet werden. Anschließend werden die neuen Klassen geladen und zum Typ-Universum der VM hinzugefügt. Sie bilden ein Neben-Universum, da die alten Klassen zu diesem Zeitpunkt noch im System verbleiben. Dieser Ansatz ermöglicht es auch, dass die Ausführung des alten Java-Programms zu diesem Zeitpunkt noch parallel zur Klassenredefinition erfolgen kann. Für das Laden und die Verifikation der neuen Klassen werden die normalen in der VM bereits implementierten Mechanismen verwendet. Falls die Verifikation einer Klasse fehlschlägt wird die Redefinition abgebrochen und der ursprüngliche Systemzustand wiederhergestellt. Bevor die Redefinition endgültig durchgeführt wird, muss noch einmal geprüft werden ob neue betroffene Klassen von der parallel laufenden Java-Applikation geladen wurden. In diesem Fall müssen auch sie redefiniert werden.

Zur Durchführung der Veränderung werden alle laufenden Java-Threads angehalten. Zusätzlich werden globale Locks benutzt um parallele Kompilierung oder paralleles Laden von Klassen zu verhindern. Anschließend werden in einer vollständigen Iteration über den Speicherbereich der VM alle Zeiger auf alte Klassen durch Zeiger auf neue Klassen ersetzt. Im Rahmen dieser Iteration wird auch das Layout von Objekten verändert deren Größe gleich geblieben oder sich verringert hat. Für die Anpassung von vergrößerten Objekten ist ein leicht modifizierter Speicherbereinigungs-Lauf durchgeführt, in dessen Rahmen die Objekte auf die neue Größe angepasst werden. Im nächsten Schritt werden innerhalb der VM in verschiedenen Bereichen Zustände invalidiert, die nicht mehr konsistent mit den neuen Klassendefinitionen sind. Zum Schluss werden alle Locks wieder freigegeben und die Ausführung der Java-Threads setzt mit der neuen Programm-Version fort.

5 Binär Inkompatible Veränderungen

Damit eine Veränderung binär kompatibel ist, muss jedes vorher gültige Klassenelement auch nach der Änderung weiterhin gültig sein. Weiters müssen alle Subtyp-Beziehungen erhalten bleiben. Binär kompatible Veränderungen einer Klasse sind: Das Hinzufügen von Feldern, Methoden oder implementierten Interfaces. Binär inkompatible Veränderungen einer Klasse sind: Das Entfernen von Feldern, Methoden oder Subtyp-Beziehungen. Die VM muss sich um eine sichere Lösung für binär inkompatible Veränderungen kümmern, da alter Code nach einer Versionsveränderung weiterhin ausgeführt werden kann. Dies ist möglich wenn zum Zeitpunkt der Veränderung alte Methoden noch aktiv sind.

Wir unterscheiden vier Lösungen für das Problem von entfernten Klassenelementen:

Statische Überprüfung: Eine statische Erreichbarkeitsanalyse überprüft ob die fortgeführte Programmausführung eine Instruktion erreichen kann, die auf ein entferntes Klassenelement verweist. In diesem Fall wird die Redefinition abgelehnt.

Dynamische Überprüfung: Die Redefinition wird immer durchgeführt und es wird sichergestellt, dass alte Methoden immer im Interpreter ausgeführt werden. Dieser überprüft zur Laufzeit ob gerade eine entsprechende in der neuen System-Version ungültige Instruktion ausgeführt wird und wirft dann eine Ausnahme.

Zugriff auf entfernte Klassenelemente: In dieser Lösung wird statt der Ausnahme auf das in der neuen Programmversion entfernte aber noch im System verfügbare Klassenelement zugegriffen. Diese Möglichkeit steht nicht für Instanzfelder zur Verfügung, da diese unwiederbringlich gelöscht werden.

Zugriff auf alte Klassenelemente: In allen bisherigen Konfigurationen wird ein Methodenaufruf immer auf die neueste Version einer Methode weitergeleitet. Dies kann jedoch zu Problemen führen wenn die alte Methode nicht mit der neuen aufgerufenen Methode kompatibel ist. Als mögliche Lösung unterstützt die DCE VM auch noch die dynamische Suche nach der exakt richtigen Version einer Methode aufgrund der Version der aufrufenden Methode.

Es ist eine notwendige Invariante einer statisch typisierten VM, dass zu jedem Zeitpunkt der Typ eines Werts ein Subtyp des statisch deklarierten Typs des den Wert beinhaltenden Felds besitzt. Wenn das nicht mehr der Fall ist, kann Typsicherheit nicht mehr garantiert werden, was höchstwahrscheinlich zu einem Absturz der VM führt, wenn der Wert das nächste Mal benutzt wird. Bei der Entfernung einer Subtyp-Beziehung durch eine Klassenredefinition kann ein derartiger Fall auftreten. Um dennoch das Entfernen von Subtyp-Beziehungen nicht vollständig zu verbieten, setzt die DCE VM einen Algorithmus ein, der in einer Iteration über alle Werte die Einhaltung der Invariante überprüft. Sollte die Invariante nicht garantiert sein, wird die Klassenredefinition abgelehnt und der ursprüngliche Systemzustand wiederhergestellt.

6 Sicherer Versionswechsel

Die Techniken zur Klassenredefinition, die in den vorhergehenden Abschnitten beschrieben wurden, definieren die Veränderung eines Programms auf der Ebene von Feldern, Methoden und Supertypen. In diesem Abschnitt gehen wir einen Schritt weiter und betrachten die detailierten Unterschiede zwischen zwei Methodendefinitionen auf Bytecode-Ebene. Wir stellen eine neue Lösung für das Verändern von Java-Methoden, die zum Zeitpunkt der Redefinition aktiv sind, vor. Das ist insbesondere für Methoden mit langlaufenden oder endlosen Schleifen hilfreich, die ansonsten nie verändert werden könnten.

Wir definieren ein Programmiermodell, das uns den Wechsel zwischen einem Basisprogramm und einem erweiterten Programm erlaubt. Das erweiterte Programm muss vom Basisprogramm abgeleitet sein: Es darf sich durch hinzugefügte Klassenelemente, neue Klassen und hinzugefügte Bytecode-Abschnitte vom Basisprogramm unterscheiden. Der Wechsel vom Basisprogramm zum erweiterten Programm kann zu einem beliebigen Zeitpunkt erfolgen. Beim Wechsel zurück zum Basisprogramm stellen wir sicher, dass sich alle ausführenden Threads außerhalb der hinzugefügten Bytecode-Abschnitte befinden.

Die vielen Restriktionen der Unterschiede zwischen dem Basisprogramm und dem erweiterten Programm verhindern viele mögliche Versionswechsel. Es ist jedoch möglich sowohl das aktuell ausführende Programm `X` als auch das neue Programm `Y` als erweiterte Programme eines virtuellen gemeinsamen Basisprogramms `B` zu betrachten. Auf diese

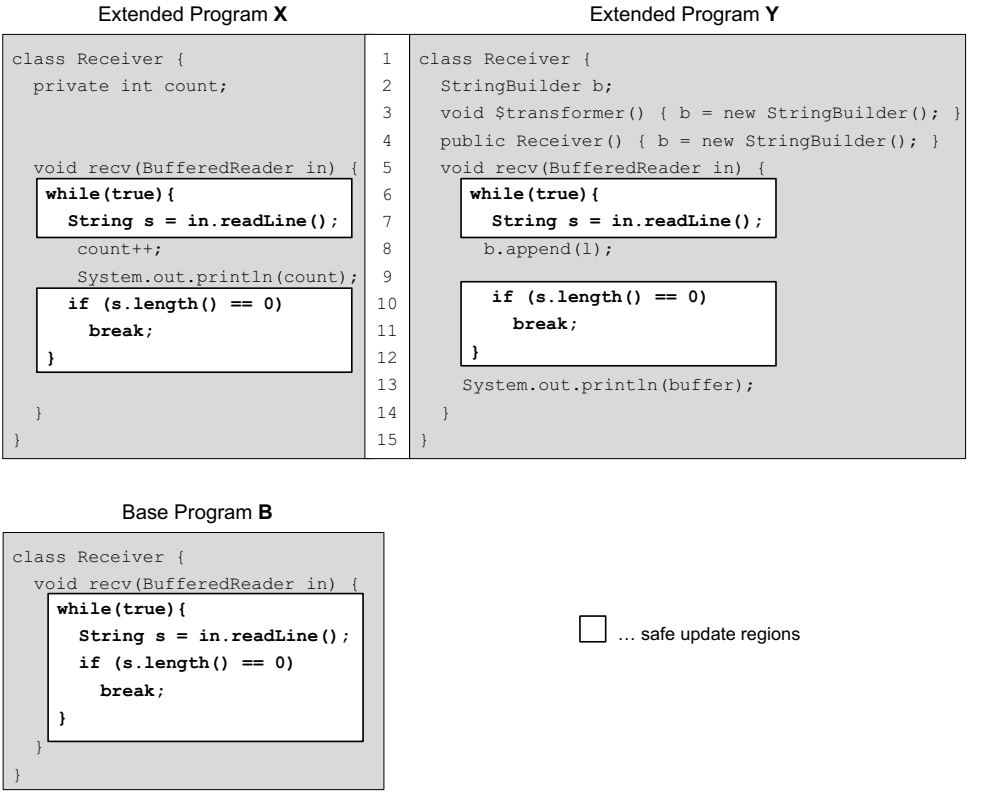

Abbildung 3: Wechsel zwischen zwei Programmversionen.

Weise kann dann die Transformation von `X` nach `Y` als eine Transformation von `X` nach `B` und von `B` nach `Y` angesehen werden.

Abbildung 3 zeigt zwei verschiedene Programme mit einer Schleife, in der Daten empfangen werden. Das Programm `X` auf der linken Seite zählt die Anzahl der Textzeilen und das Programm `Y` auf der rechten Seite speichert die Textzeilen in einem `StringBuilder`-Objekt. Das gemeinsame Basisprogramm ist weiß markiert. Alle notwendigen Restriktionen zwischen den beiden erweiterten Programmen und dem Basisprogramm werden befolgt, deshalb kann auf sichere Art und Weise zwischen den beiden Programmversionen gewechselt werden. Das Basis-Programm wird selbst nie ausgeführt, es dient lediglich als Definition für die sicheren Code-Regionen in denen der Wechsel stattfinden darf.

	DYMOS	Ginseng	Upstare	CLOS	Smalltalk	JDrums	DVM	JVolve	Hotswap	DCE VM
Austauschen von Methoden	X	X	X	X	X	X	X	X	X	X
Methoden Hinzufügen/Entfernen	X	X	X	X	X	X	X	X		X
Felder Hinzufügen/Entfernen	X	X	X	X	X	X	X	X		X
Klassenhierarchie-Veränderungen				X	X					X
Atomare Klassenredefinition								X	X	X
Veränderung Aktiver Methoden			X							X
Kein Performance-Verlust				X	X			X	X	X
Entfernung Virtueller Methoden										X
Literatur-Referenz	[CL83]	[NHSO06]	[MB09]	[Ste90]	[GR83]	[AR00]	[MPG+00]	[SHM09]	[Dmi01]	

Tabelle 1: Feature comparison of systems that allow dynamic changes to running programs.

7 System-Vergleich

Tabelle 1 vergleicht die Funktionalitäten verschiedener dynamischer Code-Evolutions-Systeme mit der DCE VM. Die meisten Systeme erlauben die Veränderung von Methoden und Feldern eines Programms. Im Fall von prozeduralen Systemen wie zum Beispiel DYMOS werden anstelle von Feldern globale Datenbereiche verändert.

Veränderungen der Klassenhierarchie sind nur in Systemen erlaubt, die mit dem Konzept von Metaklassen-Definitionen ausgestattet sind (z.B., CLOS und Smalltalk). Diese Systeme unterstützen jedoch keine atomare Redefinition von mehreren Klassen wie die DCE VM. Die Möglichkeit zum Verändern von Methoden, die gerade aktiv sind, bietet Gingseng. Während das System von Gingseng flexibler in der Art der Veränderung ist, bietet es keine Möglichkeit, Aussagen über die semantische Korrektheit einer Veränderung zu machen und unterstützt auch nicht das Konzept von sicheren Update-Regionen.

Dynamische Code-Evolution kann ohne Performance-Verlust unterstützt werden, wenn ein Programm in einer VM ausgeführt wird (z.B., CLOS, Smalltalk, JVolve, Hotswap und DCE VM). Andere Techniken resultieren in teilweise in signifikanten Performance-Verlusten (z.B., 38.5% für Ginseng). Die Möglichkeit, bereits entfernte Methoden aufgrund der Version der derzeit ausgeführten Methode und des Aufrufadressaten auszuführen, ist nur in der DCE VM verügbar.

8 Zusammenfassung

Die vorliegende Dissertation enthält folgende wissenschaftliche Beiträge:

- Wir beschreiben einen neuen Algorithmus zur Klassenredefinition in einer Java VM (siehe Abschnitt 4).
- Wir erlauben das Hinzufügen und Entfernen von Feldern und Methoden zur Laufzeit und unterstützen auch die Veränderung der Klassenhierarchie.
- Wir diskutieren die möglichen Probleme, die durch binär inkompatible Veränderungen ausgelöst werden.
- Wir beschreiben eine Lösung für das Problem von gelöschten Klassenelementen (siehe Abschnitt 5).
- Wir schlagen einen Algorithmus zur Prüfung der Typsicherheit im Fall von entfernten Supertypen vor (siehe Abschnitt 5).
- Wir präsentieren ein eingeschränktes Programmiermodell für sichere dynamische Updates von Java-Programmen (siehe Abschnitt 6).
- Wir beschreiben drei verschiedene Fallstudien, die mögliche Anwendungsbereiche aufzeigen.
- Wir zeigen, dass unser Ansatz keinen Performance-Verlust für das ausgeführte Java-Programm vor oder nach der Klassenredefinition bedeutet.
- Wir evaluieren die Performance des Algorithmus zur Veränderung von Objektinstanzen an ausgewählten Micro-Benchmarks.

Der Hauptbeitrag dieser Arbeit ist die Dynamic Code Evolution VM (DCE VM). Nach unserem besten Wissen ist die DCE VM die erste VM für eine statisch typisiert objektorientierte Sprache, die unlimitierte Unterstützung von Klassenredefinitionen bietet, ohne die Ausführungsgeschwindigkeit zu beeinträchtigen. Die DCE VM hat signifikantes Interesse unter Java-Entwicklern hervorgerufen. Es gibt auch bereits Pläne, die Modifikationen in den Quelltext der HotSpot VM zu übernehmen. Dies würde die unlimitierte Code-Evolution mehreren Millionen Java-Entwicklern verfügbar machen. Ein Prototyp der DCE VM mit Binärdaten und Quelltext können von `http://ssw.jku.at/dcevm/` heruntergeladen werden.

Literatur

[AR00] Jesper Andersson und Tobias Ritzau. Dynamic code update in JDrums. In *Workshop on Software Engineering for Wearable and Pervasive Computing*, 2000.

[CL83] Robert P. Cook und Insup Lee. DYMOS: A Dynamic Modification System. *SIGSOFT Software Engineering Notes*, 8:201–202, March 1983.

[CST03] Shigeru Chiba, Yoshiki Sato und Michiaki Tatsubori. Using HotSwap for Implementing Dynamic AOP Systems. In *Proceedings of the Workshop on Advancing the State-of-the-Art in Run-time Inspection*, 2003.

[Dmi01] Mikhail Dmitriev. *Safe Class and Data Evolution in Large and Long-Lived Java Applications*. Dissertation, University of Glasgow, 2001.

[Fab76] Robert S. Fabry. How to Design a System in Which Modules Can Be Changed on the Fly. In *Proceedings of the International Conference on Software Engineering*, Seiten 470–476. IEEE Computer Society, 1976.

[GR83] Adele Goldberg und David Robson. *Smalltalk-80: the Language and its Implementation*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1983.

[MB09] Kristis Makris und Rida Bazzi. Immediate Multi-threaded Dynamic Software Updates Using Stack Reconstruction. In *Proceedings of the USENIX Annual Technical Conference*. USENIX Association, 2009.

[MPG+00] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl Barr und J. Fritz Barnes. Runtime Support for Type-Safe Dynamic Java Classes. In *Proceedings of the European Conference on Object-Oriented Programming*, Seiten 337–361. Springer-Verlag, 2000.

[NHSO06] Iulian Neamtiu, Michael Hicks, Gareth Stoyle und Manuel Oriol. Practical Dynamic Software Updating for C. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*. ACM Press, 2006.

[Ora11] Oracle Corporation. *Top 25 RFEs (Requests for Enhancements)*, 2011. `http://bugs.sun.com/top25_rfes.do`.

[SHM09] Suriya Subramanian, Michael Hicks und Kathryn S. McKinley. Dynamic Software Updates: a VM-Centric Approach. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, Seiten 1–12. ACM Press, 2009.

[Ste90] Guy L. Steele, Jr. *Common LISP: the Language*. Digital Press, second. Auflage, 1990.

[VBAM09] Alex Villazón, Walter Binder, Danilo Ansaloni und Philippe Moret. Advanced Runtime Adaptation for Java. In *Proceedings of the International Conference on Generative Programming and Component Engineering*, Seiten 85–94. ACM Press, Oktober 2009.

Thomas Würthinger wurde am 1. September 1986 geboren. Er studierte Informatik an der Johannes Kepler Universität und absolvierte sein Doktoratsstudium 2011. Die Promotion erfolgte unter den Auspizien des österreichischen Bundespräsidenten. Während seines Studiums absolvierte er Auslandspraktika bei Sun Microsystems, Oracle und Google. Seit April 2011 arbeitet er im Oracle-Forschungslabor. Seine vorrangigen Forschungsgebiete sind Übersetzerbau, dynamische Programmanalyse und virtuelle Maschinen. Er ist Leiter des OpenJDK-Projekts "Graal".