Synthese komponentenbasierter Konfigurationen

Alexander Stuckenholz
Lehrgebiet Datenverarbeitungstechnik
FernUniversität in Hagen
Alexander.Stuckenholz@FernUni-Hagen.de

Abstract: Der Aufwand und somit die Kosten für die Wartung komponentenbasierter Softwarearchitekturen werden traditionell unterschätzt. Inkompatibilitäten sind bei ungeplanter Softwareevolution an der Tagesordnung. Klassische Verfahren zur Untersuchung von Austauschbarkeit und Integrationstests stoßen jedoch schnell an ihre Grenzen. Es werden daher im Folgenden Verfahren vorgestellt, die in der Lage sind, Konfigurationsvorschläge automatisiert zu generieren und somit einen globalen Blick auf Systemkonfigurationen und ihre Evolution erlauben.

1 Motivation

Software unterliegt einem Alterungsprozess. Dieser wird zwar nicht durch den Verfall von Bits und Bytes hervorgerufen, sondern durch die schnelle Änderung von Rahmenbedingungen (Plattformen und Kommunikationsstrukturen) und wechselnde Anforderungen an die Softwaresysteme. Software, die vor 25 Jahren eine Aufgabe perfekt lösen konnte, würde dies zwar heute auch noch tun, allerdings ist die Wahrscheinlichkeit gering, die notwendige Systemumgebung (Compiler, Betriebssystem, Hardware) dafür zur Verfügung stellen zu können [Par94].

Das Allheilmittel der Softwareindustrie gegen den Alterungsprozess von Softwaresystemen liegt darin, auch nach der Fertigstellung der Systeme Updates zu produzieren und so im Nachhinein Software an gegebene Änderungen anzupassen. Der Aufwand für die Wartung und die Kosten, die in der Wartungsphase eines Softwareentwicklungsprojektes anfallen, werden allerdings traditionell unterschätzt. Untersuchungen zeigen, dass bis zu zwei Dritteln der Gesamtkosten in der Wartungsphase eines Entwicklungsprojektes anfallen [Boe81, WPC00]. Zudem ergeben sich zum Teil gänzlich neue Anforderungen für die Wartung von komponentenbasierten Softwarearchitekturen, so dass bewährte Verfahren wie die Analyse der Auswirkungen von Änderungen (engl. *impact analysis*) und Integrationstests nur bedingt erfolgreich angewendet werden können. Voas formuliert dies in [Voa98] sehr treffend:

> Although maintaining unfamiliar code is a common maintenance dilemma, maintaining systems filled with black boxes adds a new level of difficulty.

1.1 Never run a changing system!

Die Probleme, die durch nachträgliche Änderungen in Form von Komponentenupdates entstehen können, sind vielfältig: Häufig führen sog. *interface refactorings* dazu, dass Schnittstellen nicht mehr zusammenpassen, Methodenaufufe ins Leere führen und benötigte Dienste versagen [Dig05]. Aber auch auf höheren Vertragsebenen führen veränderte Vor- oder Nachbedingungen, neue Protokoll- und Synchronisationsbedingungen mitunter zu Inkompatibilitäten.

Um Inkompatibilität im Vorfeld auszuschließen, werden in der Regel Austauschbarkeitsuntersuchungen angestrengt, die mehr oder weniger viele Vertragsebenen in die Untersuchungen mit einbeziehen. Dabei ist es das Ziel, festzustellen, ob eine neue Version einer Komponente einen Subtyp zu einer im Betrieb befindlichen Komponente darstellt. Eine Übersicht über derartige Mechanismen gibt [Stu05].

Wie die Marktforschung in [Stu06] zeigt, handelt es sich allerdings bei gut einem Drittel der untersuchten Updates kommerzieller Softwarekomponenten um sog. Major-Releases, bei denen wahrscheinlich keine Abwärtskompatibilität zu einer Vorversion mehr gegeben ist. Aufgrund neuer Abhängigkeiten und fehlender oder geänderter Schnittstellen muss also mit Schwierigkeiten bei der Integration solcher Komponenten in ein Softwaresystem gerechnet werden. Austauschbarkeitsuntersuchungen können zwar den Quell der Inkompatibilität feststellen, tragen aber ansonsten nicht zur Lösung des Problems, der Konstruktion eines ausgewogen konfigurierten Systems, bei.

1.2 Automatische Systemsynthese

Eine Lösung stellt die automatische Systemsynthese dar. Basierend auf der Annahme, dass über die Zeit hinweg eine große Menge an Softwarekomponenten zur Verfügung steht, kann ein automatisiertes Verfahren durch geschickte Kombination der Komponenten zu einer ausgewogenen, also fehlerfreien, Konfiguration gelangen. Dabei werden Abhängigkeiten aufgelöst, fehlende Komponenten nachinstalliert oder zumindest ein Konfigurationsvorschlag generiert, der das Ziel, die sonst inkompatiblen Komponenten zu integrieren, weitestgehend erreicht. Darüber hinaus können Nebenbedingungen formuliert werden, die die Anzahl der verwendeten Komponenten minimiert, das System auf dem aktuellsten Stand hält oder die Verteilungsstruktur optimiert.

Die Leistungsfähigkeit eines solchen Ansatzes kann im Bereich der Linux-Betriebssysteme wie Debian oder Suse Linux begutachtet werden. Dort existieren seit Jahren entsprechende Paketverwaltungsverfahren, um Konfigurationen aktuell und valide zu halten [Bai97]. Da im Open-Source-Bereich die Evolutionsgeschwindigkeit der Softwaresysteme weit höher ist als in den übrigen Bereichen, spielen derartige Mechanismen dort eine besondere Rolle [BP03].

Trotz dieser Ansätze bleibt das Mittel der automatischen Systemsynthese weitestgehend unerforscht. Neben [DDL+06] existieren kaum wissenschaftliche Arbeiten in diesem Bereich, welche die Möglichkeiten dieser Methode analysieren oder auf neue Anwendungs-

felder übertragen.

2 Boolesche Systemsynthese

In [SO06] wird das Problem der Synthese komponentenbasierter Softwarekonfigurationen als boolesches Optimierungsproblem formuliert. Für jede zur Verfügung stehende Komponentenversion wird dabei eine Entscheidungsvariable vorgesehen, die am Ende eines Suchprozesses entweder den Wert 1 (Komponentenversion ist Teil der Konfiguration) oder den Wert 0 (die Komponentenversion ist nicht Teil der Konfiguration) annehmen kann. Außerdem werden Nebenbedingungen formuliert, die zu einem ausgewogenen, aktuellen und in der Anzahl verwendeter Komponenten minimalen System führen sollen.

Beginnend mit einer kleinen Menge von zu integrierenden Komponentenversionen wird mit Hilfe der linearen Relaxation des Optimierungsproblems und des Branch-and-Bound-Algorithmus entsprechend der Nebenbedingungen eine Konfiguration erzeugt. Sofern eine Lösung existiert, kann diese nach einer endlichen Anzahl von Iterationsschritten erzeugt werden. Wie [Sch90, S. 360-363] zeigt, garantiert das Verfahren zudem, die global optimale Konfiguration aufzufinden.

Die boolesche Systemsynthese wurde in einem prototypischen Werkzeug namens **Componentor** zur Administration komponentenbasierter Softwaresysteme implementiert und in einer Fallstudie evaluiert. Trotz der guten Ergebnisse bezüglich der gefundenen Systemkonfigurationen zeigten sich im Alltag einige Nachteile dieses Ansatzes.

Bevor die boolesche Suche mit der Konstruktion einer Systemkonfiguration beginnen kann, müssen mit Hilfe von Signaturvergleichen alle Komponentenabhängigkeiten aufgelöst werden. Die errechneten Abhängigkeiten bilden dann die Wissensbasis der booleschen Suche. Dieser Prozess ist immer dann durchzuführen, wenn eine neue Komponentenversion im Repository veröffentlicht wird, also bei jedem Komponentenupdate. Diese Arbeit ist sehr zeitaufwändig, da jede exportierte Schnittstellensignatur mit jeder importierten Signatur abgeglichen werden muss.

Die boolesche Systemsynthese selbst ist, wegen der Evaluierung des gesamten Suchraums, eher unperformant und skaliert, im Vergleich zur Anzahl berücksichtigter Abhängigkeiten und der Anzahl von Komponentenversionen, vergleichsweise schlecht [Stu07, S. 159-183]. Das Verfahren ist daher ungeeignet, in einem reaktiven System zur online-Konstruktion von Systemkonfigurationen eingesetzt zu werden, bietet aber eine wertvolle Vergleichsmöglichkeit bei der Konstruktion eines performanteren Verfahrens.

3 Heuristische Systemsynthese

Die Erfahrungen und das domänenspezifische Wissen, das in der praktischen Anwendung der booleschen Systemsynthese gewonnen werden konnte, wurde anschließend erfolgreich in der Konstruktion eines heuristischen Verfahrens angewandt.

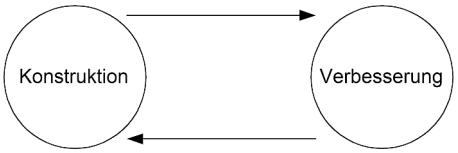

Abbildung 1: Regelkreis der heuristischen Systemsynthese.

Das resultierende heuristische Verfahren besteht aus einem zweistufigen Regelkreis, aus einem konstruktiven und einem verbessernden Teil (siehe Abbildung 1). Die erste Stufe basiert auf einem Greedy-Algorithmus, welcher beginnend mit einer oder mehreren Komponentenversionen anhand von Signaturabgleichen versucht, eine Konfiguration zu erzeugen, in der zu jeder importierten Schnittstelle eine gleichnamige exportierte Schnittstelle vorhanden ist. Dabei werden möglichst aktuelle Komponentenversionen herangezogen.

Die zweite Stufe des Verfahrens basiert auf einer lokalen Suche. Nach einer Bewertung der aus der ersten Stufe resultierenden Konfiguration werden einzelne Komponentenversionen so lange gegen ältere Versionen getauscht, bis die Anzahl der Konfigurationsfehler minimiert ist. Sollten bei dieser Tauschung neue Abhängigkeiten hinzukommen, wird wieder die erste Stufe der Suche herangezogen.

Abbildung 2 zeigt das Verhalten der booleschen und der heuristischen Systemsynthese im Vergleich. Dabei wird der Wert der Zielfunktion des Optimierungsproblems in jedem Iterationsschritt einer beispielhaften Suche der Fallstudie dargestellt. Es ist deutlich erkennbar, dass die heuristische Suche den Zielwert auf fast direktem Weg erreicht. Die boolesche Suche hingegen erzeugt durch das Verwerfen einzelner Suchpfade mit Hilfe des Branch-and-Bound-Algorithmus einen sägezahnänlichen Funktionsverlauf.

4 Fallstudie

In einer Fallstudie konnten wichtige Erkenntnisse für die praktische Anwendbarkeit der Verfahren gesammelt werden. Hierzu wurde das Administrationswerkzeug **Componentor**, das beide Verfahren realisiert, in dem Open-Source-Projekt Calimero.CMS[1] über einen Zeitraum von über einem Jahr eingesetzt. Hierfür musste jedoch der PHP-Quellcode der dort eingesetzten PEAR-Komponenten [MÖ5] mit Hilfe spezieller Javadoc-Anweisungen mit Typinformationen angereichert werden, so dass ein automatisierter Schnittstellenabgleich erst ermöglicht wurde. Die damit verbundene Mehrarbeit der Entwickler wurde aber durch den Umstand kompensiert, dass durch die Bewertung einer Konfiguration mit Hilfe des Schnittstellenabgleiches Fehler aufgedeckt werden konnten, die sonst erst zur Laufzeit aufgetreten wären.

Bei der Anwendung der Syntheseverfahren im Falle von Komponentenupdates zeigte sich,

[1] http://www.calimero-cms.de

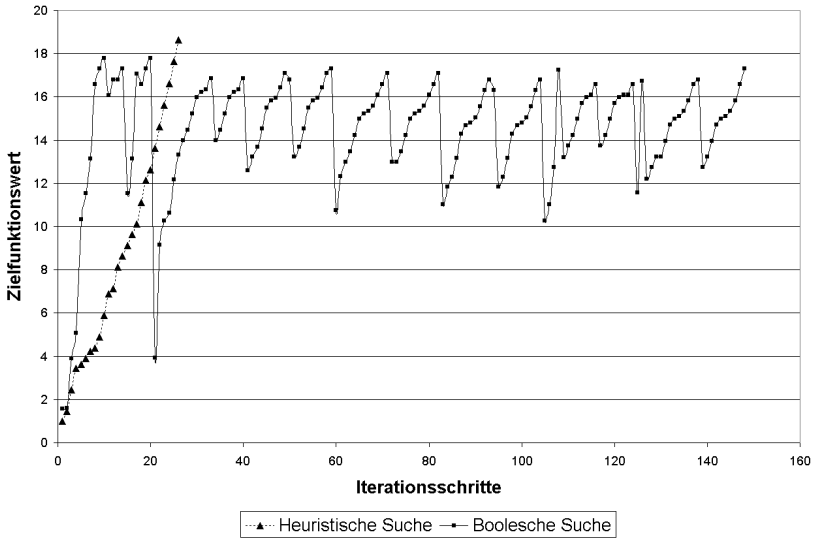

Abbildung 2: Verhalten der beiden Syntheseverfahren während der Suche.

dass das heuristische Verfahren sowohl sehr viel performanter ist, und zudem bezogen auf den Wert der Zielfunktion nur geringfügig schlechtere Ergebnisse als die boolesche Systemsynthese erzielen konnte. Das heuristische Verfahren eignet sich daher sehr viel eher für die Anwendung in einem Administrationswerkzeug.

Die Extraktion der Schnittstellenspezifikationen wurde in der Fallstudie immer dann automatisch angestoßen, wenn Änderungen am Quellcode einer Komponente im Versionierungssystem, in diesem Fall Subversion, veröffentlicht wurden. Dies ermöglichte die sanfte Integration der Verfahren in den Entwicklungsprozess. Zudem wurden für sog. Nightly-Builds die Konfigurationsvorschläge der Syntheseverfahren herangezogen.

Durch den Einsatz der Systemsynthese konnten sonst langwierige und personalintensive Integrationstests reduziert oder ganz vermieden werden. Konnte die Systemsynthese keine ausgewogene Konfiguration erzeugen oder entsprach die erzeugte Konfiguration nicht den Erwartungen, so deutete dies auf einen Fehler in der Implementierung einer oder mehrerer Komponenten hin. Fehler konnten dadurch automatisch lokalisiert und anschließend behoben werden.

5 Fazit

Die Systemsynthese stellt, quasi aus der Vogelperspektive, eine globale Sicht auf komponentenbasierte Softwaresysteme und ihre Evolution zur Verfügung. Neben der Bewertung gegebener Konfigurationen ist vor allem die automatisierte Generierung von Konfigurati-

onsvorschlägen ein Argument für diesen Ansatz. Durch die Reduktion von Integrationstests, die frühe Entdeckung von Fehlerquellen und das Aufzeigen von Konfigurationsalternativen sinken die Wartungskosten drastisch. Die Systemsynthese kann somit einen wichtigen Beitrag zur sicheren Softwareevolution und der Qualitätssicherung in komponentenbasierten Softwareprojekten leisten.

Literatur

[Bai97] Edward C. Bailey. *Maximum RPM*. Sams, 1997.

[Boe81] Barry Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[BP03] Andreas Bauer und Markus Pizka. The Contribution of Free Software to Software Evolution. In *Proceedings of the International Workshop on Principles of Software Evolution (IWPSE)*, Helsinki, Finland, 2003. IEEE Computer Society.

[DDL+06] Roberto Di Cosmo, Berke Durak, Xavier Leroy, Fabio Mancinelli und Jérôme Vouillon. Maintaining large software distributions: new challenges from the FOSS era. In *Proceedings of the FRCSS 2006 workshop*, 2006.

[Dig05] Danny Dig. Using refactorings to automatically update component-based applications. In *OOPSLA '05: Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, Seiten 228–230, New York, NY, USA, 2005. ACM Press.

[MÖ5] Carsten Möhrke. *PHP PEAR - Anwendung und Entwicklung - PEAR und PECL zur PHP-Programmierung nutzen*. Galileo Press, 2005.

[Par94] David Lorge Parnas. Software aging. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, Seiten 279–287, Los Alamitos, CA, USA, 1994. IEEE Computer Society Press.

[Sch90] Alexandeer Schrijver. *Theory of linear and integer programming*. John Wiley & Sons, Inc., New York, NY, USA, 1990.

[SO06] Alexander Stuckenholz und Andre Osterloh. Safe component updates. In *GPCE*, Seiten 39–48, 2006.

[Stu05] Alexander Stuckenholz. Component Evolution and Versioning - State of the Art. *SIGSOFT Softw. Eng. Notes*, 30(1):7, January 2005.

[Stu06] Alexander Stuckenholz. Softwarekomponenten und ihre Update-Zyklen: Eine Marktanalyse. *Praxis der Information und Kommunikation*, 29(2):92–99, 2006.

[Stu07] Alexander Stuckenholz. *Sichere Komponentenupdates*. Dissertation, FernUniversität in Hagen, 2007. Noch nicht veröffentlicht.

[Voa98] Jeffrey Voas. Maintaining Component-Based Systems. *IEEE Softw.*, 15(4):22–27, 1998.

[WPC00] Ye Wu, Dai Pan und Mei-Hwa Chen. Techniques of Maintaining Evolving Component-Based Software. *International Conference on Software Maintenance (ICSM'00)*, 00:236, 2000.