Evolutionsabkommen

Ausarbeitung zum Vortrag
im Blockseminar
Evolution von Softwaresystemen

Anne Groß
Matrikelnummer 2015501
Email: agross@studcs.uni-sb.de

Universität des Saarlandes
Lehrstuhl für Programmiersprachen und Übersetzerbau
Prof. Reinhard Wilhelm

Betreuer: Dr. Stephan Diehl

1 Überblick

Im Rahmen des Blockseminars „Evolution von Softwaresystemen“ befaßt sich das Thema „Evolutionsabkommen“ mit Softwareevolution in Komponentenbasierten Systemen. In Bezug zu den anderen Vorträgen, die, neben einer Fallstudie, zumeist praktische Anwendungen zur Unterstützung von Softwareevolution vorstellten (wie etwa Visualisierungsmethoden, Versionierung mit CVS, Konfigurationsmanagement), handelt es sich hierbei um einen eher theoretischen Ansatz - mit dem Ziel, Evolution in Komponentenbasierten Systemen kontrollieren zu können.
Hierzu wird das Konzept sogenannter „Requirements-/Assurances Contracts“ (kurz: “R/A-Contracts”) vorgestellt. Dabei handelt es sich um eine Beschreibungsmethodik, welche Komponentensysteme, oder genauer gesagt, Abhängigkeiten zwischen Komponenten spezifizieren kann.
Diese Spezifikation ermöglicht es letztendlich, die Abhängigkeiten nach erfolgter Evolu-

tion dahingehend zu überprüfen, ob diese immer noch Gültigkeit haben und die durchgeführte Evolution nicht zu einem Fehlverhalten des Systems führt.
Bevor die Beschreibungsmethodik eingeführt wird, wird zunächst untersucht, wie sich Evolution in Komponentenbasierten Systemen definieren lässt und es werden einige Evolutionsarten vorgestellt.
Um schliesslich zu zeigen, daß das Prinzip der R/A-Contracts wirklich auf solche Systeme angewendet werden kann, wird im letzten Teil der Ausarbeitung versucht, ein Mathematisches Modell vorzustellen, das dazu dienen soll, das Verhalten der Komponentensysteme im Laufe der Evolution analysieren und spezifizieren zu können.
Hierbei wird sich jedoch zeigen, daß der Ansatz, des dieser Ausarbeitung zugrunde liegenden Papieres, nicht vollkommen korrekt ist.

2 Evolution in Komponentenbasierten Systemen

Um Evolution in Komponentenbasierten Systemen definieren zu können, werden im Folgenden zunächst drei Funktionstypen vorgestellt:

2.1 Abstrakte Beschreibungsfunktion

Es handelt sich hierbei um eine Funktion, die einem gegebenen Komponententyp die Prädikatenmenge zuordnet, die für Instanzen dieses Typs wahr ist, d.h. für eine Komponente werden sozusagen ihre Eigenschaften zurückgegeben. Die Funktion ist wie folgt definert:

$$DESCRIPTION =_{def} TYPE \rightarrow \mathcal{P}(PREDICATE) \tag{1}$$

2.2 Konkrete Beschreibungsfunktion

Wendet man diese Art von Beschreibungsfunktion auf einen Komponententyp an, so wird für diesen Typ genau die Menge von Dokumenten zurückgegeben, die die Komponente beschreiben. [1]

$$DESCRIBED\_BY =_{def} TYPE \rightarrow \mathcal{P}(DOC) \tag{2}$$

[1] Hierbei ist anzumerken, daß ein System bzw. eine Komponente zu jedem Entwicklungszeitpunkt mittels verschiedener Dokumentarten beschrieben werden kann. Hierzu zählen z.B. *Strukturelle Dokumente*, die den Aufbau einer Systemes bzw. einer Komponente mit möglichen Unterkomponenten und Schnittstellen modellieren (wie etwa UML Klassendiagramme) oder aber auch *Implemetierungsdokumente*, die den Programmcode enthalten.

2.3 Semantikfunktion

Die Semantikfunktion läßt sich definieren als eine Abbildung aus einer das System beschreibenden Dokumentenmenge in eine Menge von Prädikaten und hat formal folgendes Aussehen:

$$SEM =_{def} \mathcal{P}(DOC) \rightarrow \mathcal{P}(PREDICATE) \tag{3}$$

Somit werden also einer gegebenen Dokumentenmenge, die ein System beschreibt, die Eigenschaften berechnet, die das jeweilige System charakterisieren. Darüberhinaus hat die Semantikfunktion die folgende Korrektheitsbedingung zu erfüllen:

$$sem_s(described\_by_s(t)) = description_s(t) \tag{4}$$

D.h. die Semantikfunktion ist korrekt, wenn die Prädikatenmenge, die die Semantikfunktion liefert, übereinstimmt mit der Prädikatenmenge, die man mittels (1) erhält.

2.4 Evolutionsfunktion und Arten von Evolution

Nun sind wir in der Lage, eine Evolutionsfunktion anzugeben und mit Hilfe der oben beschriebenen Semantikfunktion (3) verschiedene Evolutionsarten zu definieren:

$$EVOLVE =_{def} \mathcal{P}(DOC) \rightarrow \mathcal{P}(DOC) \tag{5}$$

Eine Evolution ist somit nichts anderes, als eine Veränderung einer Dokumentenmenge innerhalb einer gewissen Zeitspanne. Man unterscheidet folgende Evolutionsarten:

Verfeinerung Hierbei muss folgende Bedingung erfüllt sein:

$$sem_s(evolve_s(doc_s)) \supseteq sem_s(doc_s) \tag{6}$$

Mit anderen Worten: Die Menge der Eigenschaften, die die Semantikfunktion für die Dokumente nach erfolgter Evolution liefert, enthält alle Eigenschaften, die das System vor der Evolution besaß und noch weitere mehr.

Abstraktion Hierbei liefert die Semantikfunktion der Dokumentenmenge nach der Evolution nur noch Teile der vorherigen Systemeigenschaften und auch keine weiteren mehr, was durch folgende Bedingung verdeutlicht wird:

$$sem_s(doc_s) \supseteq sem_s(evolve_s(doc_s)) \tag{7}$$

Totale Evolution Bei einer totalen Evolution haben die Systemeigenschaften vor und nach der Evoultion keine Gemeinsamkeiten mehr, d.h. die jeweiligen Prädikatenmengen sind disjunkt.

$$sem_s(doc_s) \cap sem_s(evolve_s(doc_s)) = \emptyset \tag{8}$$

Strikte Evolution Bei dieser Art von Evolution handelt es sich weder um eine Verfeinerung (6), noch um eine Abstraktion(7), noch um eine Totale Evolution (8), d.h. die Menge der Eigenschaften nach der Evolution ist weder eine Obermenge noch eine Untermenge der ursprünglichen Prädikatenmenge, noch sind beide disjunkt.

$$sem_s(doc_s) \not\supseteq sem_s(evolve_s(doc_s)) \tag{9}$$
$$\wedge \quad sem_s(evolve_s(doc_s)) \not\supseteq sem_s(doc_s) \tag{10}$$
$$\wedge \quad sem_s(doc_s) \cap sem_s(evolve_s(doc_s)) \neq \emptyset \tag{11}$$

3 R/A-Contracts - Eine Einführung

Betrachtet man Evolution in einem Komponentenbasierten System, so stellt sich folgendes Problem: Wie kann man nach der Änderung eines (oder mehrerer) Dokumente nachvollziehen, welche Auswirkungen die Evolution auf andere Dokumente hat, die von den abgeänderten Dokumenten abhängig sind? Mit anderen Worten: Kann man nachprüfen, ob ein Evolutionsschritt die korrekte Funktionsweise des Systems immer noch garantiert? Um dieses Ziel der Korrektheitsprüfung zu erreichen, benötigt man eine Methodik bzw. Beschreibungstechnik, die die Abhängigkeiten der Dokumente innerhalb eines Systemes modelliert und nachvollziehbar macht. Dies gelingt durch sogenannte „Requirements-/Assurances-Contracts“.

4 R/A-Contracts - Formalisierung

Um Abhängigkeiten zwischen Dokumenten genau spezifizieren zu können, benötigt man die folgenden Funktionstypen:

$$\text{REQUIRES} =_{\text{def}} \quad (\text{COMPONENT\_TYPE} \rightarrow \mathcal{P}(\text{DOC})) \rightarrow \mathcal{P}(\text{PREDICATE}) \tag{12}$$
$$\text{ASSURES} =_{\text{def}} \quad (\text{COMPONENT\_TYPE} \rightarrow \mathcal{P}(\text{DOC})) \rightarrow \mathcal{P}(\text{PREDICATE}) \tag{13}$$

Hierbei berechnet (12) zunächst mittels einer konkreten Beschreibungsfunktion (siehe (2) eine das System beschreibende Dokumentenmenge. Für diese Dokumentenmenge wird (vergleichbar mit (3) eine Menge an Eigenschaften berechnet, und zwar genau die Eigenschaften, die der Komponententyp von der Umgebung - sprich: von anderen Komponenten - benötigt, um korrekt zu funktionieren (Requirements).
Analog dazu berechnet(13) ebenfalls eine Prädikatenmenge, wobei es sich hierbei um die Eigenschaften handelt, die der Komponententyp der Umgebung zur Verfügung stellt, so dass andere Komponenten diese benutzen können (Assurances).
Die Idee der R/A-Contracts liegt nun darin, eine Art „Vertrag“ zwischen den einzelnen

Komponenten abzuschliessen, und zwar derart, dass ein solcher Vertrag zwischen zwei Komponenten A und B dann zustande kommt, wenn B die Eigenschaften zur Verfügung stellt, die die Komponente A benötigt. Anders ausgedrückt: die Menge (oder auch nur eine Teilmenge) an Eigenschaften, die REQUIRES für A liefert, muss in der Prädikatenmenge enthalten sein, die ASSURES für die Komponente B liefert.
Formal handelt es sich bei einem Vertrag um ein 4-Tupel:

$$\begin{aligned} \text{CONTRACT} =_{\text{def}} \ & \text{COMPONENT\_TYPE} \\ & \times \text{PREDICATE} \\ & \times \text{COMPONENT\_TYPE} \\ & \times \text{PREDICATE} \end{aligned}$$

Hat man nun für jede Komponente die Requirements (mittels 12) und Assurances (mittels 13) bestimmt, und die benötigten „Verträge“ zwischen den Komponenten abgeschlossen, so muss nach einem erfolgten Evolutionsschritt für jede Komponente überprüft werden, ob die von ihr benötigten Eigenschaften immer noch bereitgestellt werden, so dass die Komponente weiterhin korrekt funktioniert. Dies kann mit Hilfe des folgenden Prädikats erreicht werden:

$$\text{FULFILLED} =_{\text{def}} \text{COMPONENT\_TYPE} \rightarrow (\mathcal{P}(\text{PREDICATE}) \rightarrow \text{BOOLEAN}) \quad (14)$$

wobei gilt:

$$\begin{aligned} & \mathit{fulfilled}_s(ct)(\mathit{requires}_s(ct)(\mathit{described\_by}_s(ct))) \\ \Leftrightarrow \ & \{p|(ct, r, x, p) \in \mathit{contract}_s \ \wedge \ r \in \mathit{requires}_s(ct)(\mathit{described\_by}_s(ct))\} \\ & \subseteq \ \{q|q \in \mathit{assures}_s(x)(\mathit{described\_by}_s(x))\} \end{aligned}$$

Die korrekte Funktionsweise einer Komponente ist genau dann gewährleistet, wenn das Prädikat (14) für diese Komponente den Wert “true” liefert d.h. für die gegebene Komponente ct und ihre Requirements r (berechnet mittels 12) muss ein “Vertrag” (ct, r, x, p) mit einer anderen Komponente x existieren, so daß die Requirements r der Komponente ct durch entsprechende Assurances p der Komponente x zur Verfügung gestellt werden.

5 Korrektheitsbeweiß für die R/A-Contracts

In den vorangegangenen Abschnitten wurde Evolution in Komponentenbasierten Systemen definiert und die Idee der R/A-Contracts vorgestellt. Nun bleibt noch zu zeigen, dass das vorgestellte Konzept auch tatsächlich anwendbar ist. Hierzu muss man nun ein mathematisches Konzept finden, um genau definieren zu können, wie sich ein Komponentenbasiertes System im Laufe seiner Entwicklung verändert.

5.1 Dynamische Änderungen

Betrachtet man ein System das aus einzelnen Komponenten besteht, so stellt man fest, daß sich das Verhalten eines solchen Systems nicht nur beschränkt auf die Kommunikation zwischen Paaren von Komponenten, sondern vielmehr muss die Veränderung bezüglich der kompletten Struktur betrachtet werden, die sich während der Laufzeit, z.B. durch Einführung neuer Variablen, ständig ändern kann.
Im Folgenden wird nun versucht, das Verhalten eines Komponentensystemes zu beschreiben, indem zunächst definiert wird, wie ein Systemzustand zu einem beliebigen Zeitpunkt dargestellt werden kann, um danach Rückschlüsse auf das gesamte Systemverhalten mittels einer Übergangsfunktion von einem Zustand in einen anderen, ziehen zu können.

5.2 Systemzustände und Verhalten

In dem Versuch, dass Verhalten des Systemes bzgl. seiner Struktur zu beschreiben, liegt bereits eine der Schwachstellen des Papiers. Dr. Rausch definiert die Basiselemente oder auch Einheiten eines Systemes so, dass er bereits Komponenten als Bestandteile solcher Einheiten mit einbezieht. D.h. ohne genau zu definieren, was eine einzelne Komponente ist oder vielmehr, wie sie sich verhält, baut er seine Verhaltensuntersuchung auf. In einem sogenannten „Schnappschuss“ (SNAPSHOT) versucht Rausch, einen Zustand eines Komponentensystems zu einem beliebigen Entwicklungszeitpunkt zu erfassen und ist im Folgenden darum bemüht, eine Verhaltensfunktion zu definieren, die es praktisch ermöglichen soll, einen Übergang von einem SNAPSHOT zu einem anderen berechnen zu können. [2]

$$BEHAVIOUR =_{\text{def}} \text{SNAPSHOT} \rightarrow \text{SNAPSHOT} \tag{15}$$

[2] dieser Aspekt ist übertragbar auf das Konzept Endlicher Automaten, wobei SNAPSHOT einem Zustand des EA entspricht und BEHAVIOUR der entsprechenden Übergangsfunktion δ

Doch auch dieser Ansatz schlägt in der folgenden Definition fehl:

$$behaviour_s^t =_{\text{def}} \bigcup_{c \in \text{COMPONENT}:\ alive_s^t(c)=true} behaviour_c \tag{16}$$

Dr. Rauschs Überlegung, die Übergangsfunktion für ein gesamtes Komponentensystem als die Vereinigung aller Übergänge von Komponenten, die zu einem Zeitpunkt t in dem System vorhanden sind [3], zu definieren, ist nicht korrekt.
Dies liegt daran, dass die Komponenten untereinander in Verbindung stehen und somit abhängig voneinander sind (wie aus den R/A-Contracts hervorgeht). Würde jede Komponente für sich alleine existieren und würde man für jede Komponente eine Übergangsfunktion bestimmen können, wäre die obige Definition sicher richtig. [4]

6 Zusammenfassung

Im Laufe dieser Ausarbeitung wurde zunächst anhand einiger Funktionstypen Evolution in Komponentenbasierten Systemen definiert und verschiedene Evolutionsarten wurden vorgestellt.
Mittels sogenannter "Requirements/Assurances-Contracts" wurde ein Lösungsansatz herausgearbeitet, der auf das Problem der kontrollierten Evolution solcher Systeme eingeht.
Der letzte Teil befaßte sich schließlich damit, ein Modell vorzustellen, welches es ermöglichen sollte, das Verhalten von Komponentenbasierten Systemen zu spezifizieren. Leider konnte letzteres Konzept nicht korrekt vorgeführt werden, da das Papier, das dieser Ausarbeitung zugrunde lag, an dieser Stelle einige Fehler aufweißt.

[3] das Vorhandensein einer Komponente wird mittels der Funktion ALIVE errechnet

[4] Auch dieser Sachverhalt läßt sich auf Endliche Automaten übertragen: Jede Komponente ist als ein Endlicher Automat realisierbar, wobei die Menge aller SNAPSHOTS den Zuständen, und BEHAVIOUR der jeweiligen Übergangsfunktion entspricht. Wäre jede Komponente isoliert, so könnte man einen EA für das ganze System definieren, in dem man einen neuen Startzustand definiert und diesen mit ϵ-Übergängen mit den Startzuständen der Endlichen Automaten für jede Komponente verbindet.

Literatur

Rausch, A. (2000a). Software evolution in componentware - a practical approach. In *Proceedings of the 2000 Australian Software Engineering Conference (ASWEC).*

Rausch, A. (2000b). Software evolution in componentware using requirements/assurances contracts. In *Proceedings of the 22th International Conference on Software Engineering (ICSE22).*