Optimale Integrationsreihenfolgen

Mario Winter

Institut für Informatik
Fachhochschule Köln, Campus Gummersbach
Steinmüllerallee 1
51643 Gummersbach
mario.winter@fh-koeln.de

Abstract: Der Integrationstest prüft das Zusammenspiel der Bausteine eines Softwaresystems. Hierbei bestimmt die gewählte Integrationsstrategie die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden. Zugunsten einer einfacheren Fehlerlokalisierung fügen schrittweise Integrationsstrategien immer nur eine begrenzte Anzahl weiterer Bausteine zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzu. Weist hierbei ein Baustein eine Abhängigkeit zu einem noch nicht in dieser Menge befindlichen Baustein auf, wird letzterer durch einen extra für den Test zu erstellenden Stellvertreter (stub) ersetzt. Wird ein neu hinzugenommener Baustein von noch nicht integrierten Bausteinen genutzt, so sind diese durch Treiber (driver) zu ersetzen. Gegenstand dieses Beitrages ist die Ermittlung einer optimalen Integrationsreihenfolge mit dem Ziel, den Aufwand zur Erstellung von Test-Stellvertretern und -Treibern zu minimieren. Dazu wird die Problemstellung als ganzzahliges Optimierungsproblem formuliert, welches mit Verfahren der dynamischen Programmierung gelöst werden kann. Zwei Experimente zeigen die Leistungsfähigkeit des vorgestellten Ansatzes auf.

1. Einleitung

Testen dient der Bewertung eines Softwareprodukts und dazugehöriger Arbeitsergebnisse mit dem Ziel, deren Anforderungserfüllung und Eignung festzustellen sowie ggf. Defekte zu finden. Dies kann einerseits mit statischen Analysen erfolgen, welche den zu untersuchenden bzw. zu prüfenden Gegenstand (Teil-, Zwischen- oder Endprodukt, im Folgenden als Testobjekt bezeichnet) als solches analysieren und somit auf alle Entwicklungsprodukte wie z. B. Anforderungs- und Entwurfsspezifikationen sowie Programmcode „als Text“ anwendbar sind. Im Gegensatz dazu führt man bei dynamischen Tests – oft einfach als „Testen“ bezeichnet –, „ausführbare“ Testobjekte, d. h. Programmteile, ganze Programme oder Systeme, unter kontrollierten Bedingungen mit dem Ziel aus, die korrekte Umsetzung der Anforderungen nachzuweisen, Fehlerwirkungen aufzudecken und das „Vertrauen“ in das Testobjekt zu erhöhen (vgl. [GTB10]).

Bei dynamischen Tests werden vor der Testausführung für ausgewählte Eingaben anhand der Spezifikation des Prüflings (der „Testbasis“) die erwarteten Ergebnisse

ermittelt. Nach der Ausführung wird das jeweilige tatsächliche Ergebnis der Ausführung mit dem vorher ermittelten erwarteten Ergebnis verglichen. Ausführbare Testobjekte im dynamischen Test sind z. B. einzelne Funktionen bzw. Prozeduren, die Instanzen einer Klasse, einzelne Teilsysteme oder das vollständige System.

Je nach der „Konstruktionsphase“, in der die Testbasis angelegt wurde, und der Granularität der jeweiligen Testobjekte ergeben sich die korrespondierenden Teststufen Modul- bzw. Komponententest (unit test), Integrationstest, Systemtest und Abnahmetest (vgl. [GTB10]). Der Integrationstest ist als Teststufe zwischen Modul- bzw. Komponententest und Systemtest gelagert. Dabei werden Klassen, Module, Komponenten, Teilsysteme oder auch ganze Systeme zusammengefügt und in ihrem Zusammenspiel geprüft. Zur Vermeidung von Begriffskonflikten wird im Weiteren der allgemeine Begriff „Baustein“ verwendet.

Wenn Bausteine zusammenspielen oder aufeinander aufbauen bzw. verweisen, besteht eine Abhängigkeit zwischen den Bausteinen. Jungmayr definiert Abhängigkeiten in seiner Dissertation zum Thema Testbarkeit (testability) folgendermaßen: *A dependency is a directed relationship between two entities where changes in one entity may cause changes in the other (depending) entity* ([Ju03], S. 9). Wichtig dabei ist, dass jede Abhängigkeit immer zwischen genau zwei Bausteinen besteht. Im Rahmen der Abhängigkeit spielen beide Bausteine unterschiedliche Rollen: Es gibt einen abhängigen und einen unabhängigen Baustein, wobei die Abhängigkeit vom abhängigen hin zum unabhängigen Baustein zeigt. Abb. 1 (a) skizziert z. B. eine Abhängigkeit von Baustein `A` zu Baustein `B`, die als `A_B` bezeichnet ist. Der abhängige Baustein `A` benötigt den unabhängigen Baustein `B` für seine korrekte Funktionsweise. Insofern wird der unabhängige Baustein auch oft als „benötigter Baustein“ bezeichnet. Abb. 1 (b) zeigt zwei gegenseitig voneinander abhänge Bausteine und damit einen Zyklus im Abhängigkeitsgraphen. Solche Zyklen können auch mehrere Bausteine umfassen und erschweren die Ermittlung einer Integrationsreihenfolge (s. U.).

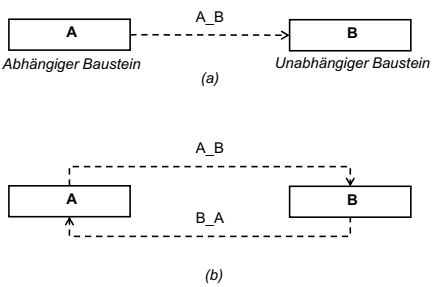

Abbildung 1: Abhängigkeiten zwischen Bausteinen

Der Integrationstest prüft das Zusammenspiel voneinander abhängiger Bausteine eines (Software-) Systems. Dazu werden Testfälle entworfen, deren Eingaben die zu integrierenden Bausteine stimulieren und deren erwartete Ergebnisse die Interaktionen zwischen diesen Bausteinen sowie deren Parameter- und Rückgabewerte umfassen. Hierbei bestimmt die gewählte Integrationsstrategie die Reihenfolge, in der die Bausteine integriert und in ihrem Zusammenspiel getestet werden. Man unterscheidet

schrittweise Integrationsstrategien von der sog. „Big-Bang-Strategie“, welche alle Bausteine eines Systems auf einmal integriert und dann (i.d.R. mit Systemtests) testet.

Zugunsten einer einfacheren Fehlerlokalisierung fügen schrittweise Integrationsstrategien immer nur eine begrenzte Anzahl weiterer Bausteine zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzu. Weist hierbei ein Baustein eine Abhängigkeit zu einem noch nicht in dieser Menge befindlichen Baustein auf, wird letzterer durch einen extra für den Test zu erstellenden Stellvertreter (*stub*) ersetzt. Dieser stellt für die im Test geprüfte Nutzung eine rudimentäre Funktionalität sowie ggf. Protokollierungs- und Profiling-Funktionen bereit. Wird ein neu hinzugenommener Baustein von noch nicht integrierten Bausteinen genutzt, so sind letztere durch Treiber (*driver*) zu vertreten, wenn Testfälle für deren Nutzung des aktuell integrierten Bausteins notwendig sind. Dies ist dann der Fall, wenn der zu integrierende Baustein im Rahmen dieser Nutzungen seinerseits weitere Bausteine nutzt.

Schrittweise Integrationsstrategien lassen sich weiter unterteilen in von der Softwarestruktur abhängige und von ihr unabhängige Strategien. Für strukturabhängige Strategien wird oft der Abhängigkeitsgraph herangezogen, dessen Knoten die Bausteine und dessen Kanten ihre Abhängigkeiten darstellen (s. Abb. 1). Prominente Vertreter sind z.B. die Bottom-Up und die Top-Down Strategie, welche von vollkommen unabhängigen (also nur „genutzten“) Bausteinen hin zu vollkommen abhängigen (also nur „nutzenden“) Bausteinen bzw. umgekehrt integrieren. Als Mischformen sind auch die Inside-Out und die Outside-In bzw. Sandwich-Strategie bekannt. Strukturunabhängige Strategien wählen den bzw. die als nächstes zu integrierenden Bausteine z.B. nach deren Risiko, Komplexität oder Verfügbarkeit aus.

Strukturabhängige Strategien sind nur dann unmittelbar z.B. durch eine topologische Sortierung umsetzbar, wenn der Abhängigkeitsgraph keine Zyklen aufweist. Dies ist jedoch bei den interaktions- bzw. nutzungsbasierten Abhängigkeiten in objektorientierten Systemen i.d.R. nicht der Fall ([Ov94][Ku95][BLW03], [Wi00]), so dass „objektorientierte Strategien“ versuchen, z.B. durch das Aufbrechen von Zyklen eine Reihenfolge für die Integration der Klassen zu ermitteln (*class integration test order*, CITO). Allerdings ist dort neben den interaktions- bzw. nutzungsbasierten Abhängigkeiten auch die Vererbung bzw. Generalisierung zu berücksichtigen.

Gegenstand dieses Beitrags ist die Ermittlung einer optimalen Integrationsreihenfolge im Rahmen schrittweiser strukturabhängiger Integrationsstrategien mit dem Ziel, den Aufwand zur Erstellung von Test-Stellvertretern und -Treibern zu minimieren. Im Gegensatz zu anderen Arbeiten wird hierbei keine Heuristik verwendet, sondern das Problem der Ermittlung einer optimalen Integrationsreihenfolge als ganzzahliges Optimierungsproblem modelliert, welches mit Algorithmen der kombinatorischen Optimierung bzw. der dynamischen Programmierung wie z.B. dem Branch- und Bound-Verfahren gelöst werden kann. Das Verfahren wurde in einem kommerziellen Optimierungsprogramm implementiert, um Experimente zu seiner Leistungsfähigkeit durchführen zu können. Der Beitrag stellt das Verfahren selbst sowie die Ergebnisse zweier Experimente vor.

Der Beitrag ist wie folgt strukturiert. Kapitel 2 beschreibt relevante andere Arbeiten aus der zahlreichen vorhandenen Literatur zur Ermittlung von Integrationsreihenfolgen. In Kapitel 3 wird die Problemstellung dann formalisiert, als Optimierungsproblem modelliert und in einem kommerziellen Optimierungs-Werkzeug implementiert. Kapitel 4 fasst die Ergebnisse der beiden Experimente zusammen. Kapitel 5 beendet den Beitrag mit einer Bewertung des Ansatzes und einem Ausblick auf geplante weitere Arbeiten.

2. Andere Arbeiten

Die Arbeiten zur Ermittlung von Integrationsreihenfolgen lassen sich grob in solche klassifizieren, die auf deterministischen graph-basierten Verfahren aufbauen, und solche, die heuristische Ansätze wie z.B. Suchverfahren oder genetische Algorithmen verwenden. Aus Platzgründen werden nur einige wenige repräsentative Arbeiten näher beleuchtet, jeweils stellvertretend für ihre Klasse. Umfassendere Ausführungen hierzu finden sich z.B. in [BLW03], [Ba09], [BP09] oder [Wi12].

2.1 Graph-basierte Ansätze

In einer der ersten Arbeiten zum Integrationstest für objektorientierte Software gibt Overbeck ein Verfahren zur Ermittlung einer Integrationsstrategie auf der Grundlage des Klassendiagramms an ([Ov94]). Das Verfahren orientiert sich primär an den Generalisierungsbeziehungen Top-Down, also von Basisklassen zu abgeleiteten Klassen, und sekundär an den Interaktions- bzw. Nutzungsabhängigkeiten. Die mit diesem Verfahren ermittelten Integrationsstrategien minimieren die Anzahl der für den Test benötigten Testtreiber und -stellvertreter. Probleme, die aus zyklischen Nutzungsbeziehungen resultieren, sollen durch „Aufbrechen“ der Zyklen anhand einer Tiefensuche mit entsprechenden Teststellvertretern behandelt werden. Ebenso durch Aufbrechen zyklischer Abhängigkeiten und nachfolgende topologische Sortierung wird die Integrationsreihenfolge von Kung et Al. ermittelt ([Ku95]).

[Je99] und [Wi00] stellen unabhängig voneinander eine Integrationsstrategie vor, welche die Betrachtung von ganzen Klassen auf deren konstituierende Bausteine verfeinert. Die Knoten der resultierenden Abhängigkeitsgraphen modellieren also Member der Klassen (Methoden, in [Wi00] auch Variablen), die Kanten stellen Aufrufe zwischen Operationen dar, in [Wi00] darüber hinaus auch Redefinitionen von Operationen sowie die Verwendung der Member-Variablen (def, use). Die feingranulare Betrachtung führt dazu, dass viele auf Klassenebene zu beobachtende zyklischen Abhängigkeiten auf der Ebene von Methoden in azyklische Teilgraphen bzw. Wälder zerfallen und somit oft eine „kanonische“ Integrationsreihenfolge gefunden werden kann.

Abdurazik und Offutt beschreiben ein graph-basiertes Verfahren, das sowohl die Knoten als auch die Kanten des Abhängigkeitsgraphen mit Gewichten belegt, welche durch unterschiedliche Code-Metriken ermittelt werden ([AO09]). Unter Berücksichtigung beider Arten von Gewichten erreichen sie in den Experimenten bessere Ergebnisse als vergleichbare Arbeiten.

2.2 Such-basierte Ansätze

Le Hanh et Al. sowie Briand et Al. experimentieren mit evolutionären Algorithmen zur Ermittlung optimaler Integrationsreihenfolgen ([Le01], [BFL02]). Als primäres Ziel- bzw. „Fitness"-Kriterium wird die Kopplungs-Komplexität zwischen Klassen sowie – in [Le01] – auch zwischen Methoden betrachtet. [Le01] verwenden als sekundäres Zielkriterium auch die Komplexität der einzelnen Klassen selbst. Experimentelle Vergleiche verschiedener evolutionärer und graph-basierter Algorithmen ergaben für erstere tw. bessere Ergebnisse, während letztere besser für große Systeme skalierten.

Assunção et Al. betrachten die verschiedenen Arten von Abhängigkeiten separat und formulieren ein Optimierungsproblem mit mehrfacher Zielsetzung ([As11]). Zur Lösung verwenden sie evolutionäre Mehrziel-Optimier-Algorithmen. Experimente mit zwei unterschiedlichen Algorithmen hinsichtlich vier Zielkriterien (Attribute, Methoden, Anzahl unterschiedlicher Typen der Parameter und Rückgabewerte) zeigen die Eignung des Ansatzes anhand vier realer Systeme.

3. Formulierung der Problemstellung und Optimierungsansatz

In diesem Kapitel formulieren wir zunächst die Problemstellung und überführen sie dann in ein ganzzahliges Optimierungsproblem, welches für ein kommerzielles Optimierungswerkzeug implementiert wird.

3.1 Problemformulierung

Seien BM die Menge der zu integrierenden Bausteine mit $|BM| = n$ und $N: BM \rightarrow \{1..n\}$ eine Nummerierung der Bausteine. Wir engen die Problemstellung auf schrittweise Integrationsstrategien ein, welche immer nur genau einen Baustein zur Menge bereits integrierter und integrationsgetesteter Bausteine hinzufügen. Lösungsraum für solche Integrationsstrategien ist somit die Menge aller Permutationen der Elemente von $\{1..n\}$, die im weiteren mit σ bezeichnet wird. Für ein $o \in \sigma$ bedeutet $o(i) = k$ mit $i, k \in \{1..n\}$, dass Baustein i als k-ter Baustein integriert wird.

Weist ein zu integrierender Baustein eine Nutzungs- oder eine Generalisierungs-Abhängigkeit zu einem noch nicht integrierten Baustein auf, ist letzterer durch einen Test-Stellvertreter zu ersetzen. Kann ein neu hinzugenommener Baustein nicht über bereits integrierte Bausteine angesteuert werden, so ist für jede entsprechende Aufruf-Abhängigkeit ein Treiber zu erstellen. Zusätzlich fällt immer ein Treiber für die Ansteuerung des Systems nach dem letzten Integrationsschritt an.

Die Aufwände hierfür lassen sich in drei Matrizen KS, KD und $KG \in Mat_{n,n}(\mathbb{R})$ festhalten. *KS(i, j)* bzw. *KG(i, j)* beziffern den Aufwand, wenn für eine Nutzungs- bzw.

Generalisierungs[1]-Abhängigkeit von Baustein *i* zu Baustein *j* ein Stellvertreter als Ersatz für *j* erstellt werden muss. *KD(i, j)* gibt den Aufwand an, für eine Abhängigkeit von *i* nach *j* anstelle des aufrufenden Bausteins *i* einen Treiber einzusetzen. .

Zur Ermittlung des geschätzten Aufwands für die Erstellung von Test-Stellvertretern und –Treibern können unterschiedliche Komplexitätsmetriken wie z.B. bei einer Nutzungsabhängigkeit die Anzahl der unterschiedlichen aus *i* aufgerufenen Methoden von *j* sowie deren Parameter und Rückgabewerte oder bei einer Generalisierungsabhängigkeit die Anzahl der von *i* aus *j* geerbten und verwendeten bzw. ggf. redefinierten Methoden etc. dienen (vgl. z.B. [As11], [BFL02] und [AO09]). Im Weiteren werden o.B.d.A. keine konkreten Aufwände ermittelt, sondern für eine Nutzungs- bzw. Generalisierungs-Abhängigkeit von Baustein *i* zu Baustein *j* die konstanten Werte *KS(i, j) = KG(i, j)* = 2 und *KD(j, i)* = 1 angesetzt. Für eine Integrationsreihenfolge $o \in \sigma$ fallen diese Aufwände natürlich nur dann an, wenn $o(i) < o(j)$ ist, Baustein *i* also bzgl. *o* vor Baustein *j* integriert wird.

Die gesamten Kosten *K* für eine bestimmte Integrationsreihenfolge $o \in \sigma$ lassen sich dann berechnen als

$$K(o) = \left(\sum_{i,j=1}^{n} \begin{cases} KS(i,j) + KG(i,j) + KD(i,j), & falls\ o(i) < o(j) \\ 0, & sonst \end{cases}\right) + 1 \qquad (1)$$

Für den in Abbildung 2 gezeigten einfachen „generalisierungsfreien“ Abhängigkeitsgraphen sind z.B. *KS(1, 2)* = 2, *KD(1, 2)* = 0 und *KD(2, 1)* = 1. Die „Kosten“ für den ersten Schritt der „Top-Down“ Integrationsreihenfolge o_{TD} = <1, 2, 3, 4, 5, 6, 7> betragen 7 (ein Treiber für Komponente 1 und drei Stellvertreter für die Komponenten 2 bis 4, also 1+3*2=7). Als Gesamtkosten ergeben sich $K(o_{TD})$ = 19. Besser ist die „Bottom-Up“ Reihenfolge o_{BU} = <7, 6, 5, 4, 3, 2, 1> mit den Gesamtkosten $K(o_{BU})$ = 10.

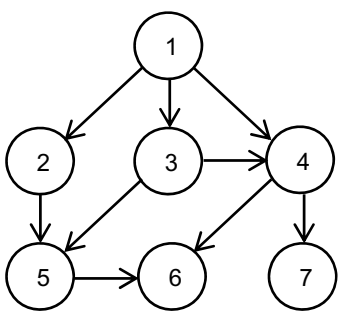

Abbildung 2: Einfacher (azyklischer) Abhängigkeitsgraph

[1] Generalisierungsabhängigkeiten (und Kompositionen) werden i.d.R. gesondert betrachtet, da diese einen azyklischen Graph ergeben (müssen), während dies bei Nutzungsabhängigkeiten (und Assoziationen) nicht der Fall ist.

3.2. Optimierungsansatz

Die Ermittlung einer optimalen Integrationsreihenfolge lässt sich nun folgendermaßen formulieren: Suche ein $o \in \sigma$ für das $K(o)$ ein Minimum ist, kurz: *Minimiere* $K(o)$.

Die vollständige Enumeration aller $n!$ Permutationen verbietet sich schon für recht kleines n. In Anlehnung an Formulierungen bekannter Reihenfolgeprobleme wie z.B. dem Rundreiseproblem (traveling salesman problem, TSP) oder dem Arbeitsplanproblem (job shop scheduling, JSS) wird das Integrationsreihenfolge-Problem als ganzzahliges Optimierungsproblem angesetzt, welches Methoden der dynamischen Programmierung zugänglich ist ([PS82]).

Eine Reihenfolge-Matrix $R \in Mat_{n,n}(\{0,1\})$ eliminiert hierbei die kleiner-Abfrage bzgl. der Reihenfolge bei der Berechnung der Integrationskosten in (1). Ist $R(i, j) = 1$, dann wird Baustein i vor Baustein j integriert. Darüber hinaus fassen wir die Einzelkosten-Matrizen KS, KD und KG in einer einzigen Kostenmatrix $C \in Mat_{n,n}(\mathbb{R})$ zusammen, mit $C(i, j) = KS(i, j) + KG(i, j) + KD(i, j)$. Der konstante Faktor eins in (1) wird vernachlässigt, da er bei allen Reihenfolgen für den Treiber des Systems anfällt.

Nun müssen noch die Bedingungen angegeben werden, unter denen die Reihenfolge-Matrix R tatsächlich eine Permutation der n Bausteine widerspiegelt. Zunächst ist für je zwei Bausteine i und j entweder i vor j oder aber j vor i zu integrieren:

$$\forall i, j \in \{1..n\}, i < j: R(i,j) + R(j,i) = 1 \tag{2}$$

Für die Permutation $o \in \sigma$ muss dann gelten (vgl. [PS82] S. 311):

$$\forall i, j \in \{1..n\}, i \neq j: \big(o(i) - o(j) + 1 \leq n \cdot R(j,i)\big) \tag{3}$$

Die Bedingung in (3) sagt zunächst aus, dass sich die in R kodierte Reihenfolge auch in o wiederfindet. Darüber hinaus erzwingt (3), dass keine Differenz je zweier Werte in o größer als $n - 1$ ist, o also genau alle Werte von 1 bis n umfasst. Es ergibt sich die folgende Optimierungsvorschrift:

$$Min \sum_{i,j=1}^{n} C(i,j) \cdot R(i,j) \quad \text{s.t. } (2) \wedge (3) \tag{4}$$

3.3 Implementierung

Im IBM ILOG CPLEX Optimization Studio ([IBM12]) lässt sich das obige Optimierungsproblem in Form der in Abbildung 3 gezeigten Zielfunktion und Beschränkungsmenge implementieren. Die Variable `n` gibt die Kardinalität der Bausteinmenge an, die Variable `edges` mit den Komponenten `<i,j>` umfasst die Kanten des in Listenform kodierten Abhängigkeitsgraphen. Für jede Kante `<i,j>` gibt der Eintrag `cost[<i,j>]` die Einzelkosten bei der Integration von Baustein `i` vor Baustein `j` an. `R` ist die Reihenfolgematrix, `o` die aktuelle Permutation.

```
// Objective
minimize sum (<i,j> in edges) cost[<i,j>]*R[i,j];
subject to {

        forall ( i in 1..n )
          forall ( j in i+1..n ) {
            R[i,j] + R[j,i] == 1;
          }

        forall ( i in 1..n )
          forall ( j in 1..n ) {
            if ( i != j ) {
              [i] - o[j] + 1 <= n * R[j,i];
        }
```

Abbildung 3: Model-Code des Ziels und der Beschränkungen für den Solver

4. Erste Experimente

In diesem Abschnitt werden zwei Experimente beschrieben, die mit der in Abbildung 3 gezeigten Implementierung des Optimierungsmodells durchgeführt wurden.

4.1 Project Planning Tool

Das Beispielsystem dieses Experiments stammt aus der Dissertation von Overbeck ([Ov94]). Der in Abbildung 4 gezeigte Abhängigkeitsgraph zeigt 8 Bausteine und ihre 26 Nutzungsabhängigkeiten. Zusätzlich ist die Nummerierung der Bausteine angegeben.

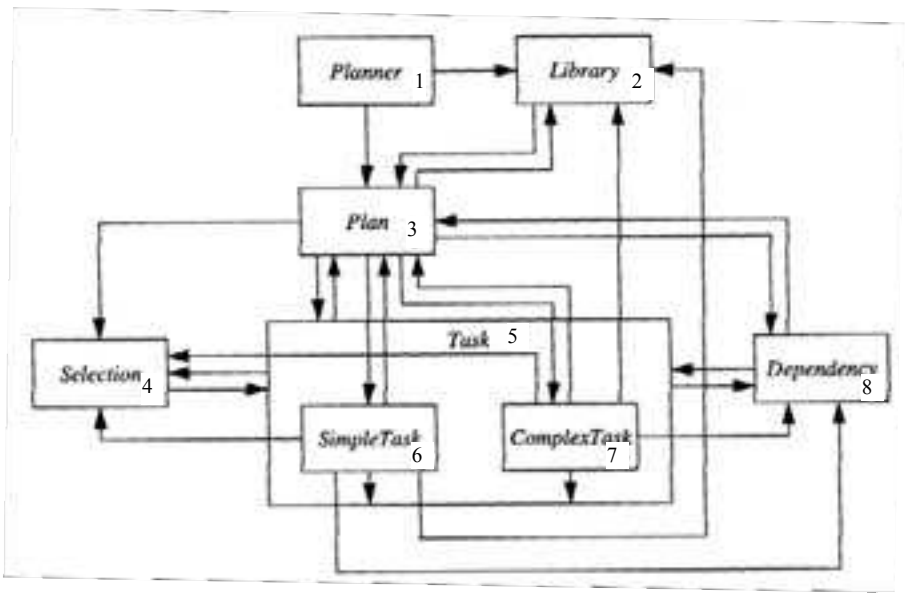

Abbildung 4: Project Planning Tool (PPT, aus [Ov94])

Die beiden Bausteine `SimpleTask` und `ComplexTask` stehen in einer Generalisierungsabhängigkeit zum Baustein `Task`, was in Abbildung 4 durch das „Enthaltensein“ in sowie Nutzungsbeziehungen zu dem Baustein-Rahmen des Letzteren symbolisiert ist. Die Kodierung des Problems zu seiner Optimierung umfasst also 26 Kanten mit den jeweiligen Kostenfaktoren.

Abbildung 5 zeigt das Setup-Log des Solvers nach der Modell-Kompilierung. Das Modell führte zu einem Gleichungssystem mit 72 Variablen und 85 Beschränkungen.

```
! ------------------------------------------------------------------------
! Minimization problem - 72 variables, 85 constraints
! Initial process time : 0,00s (0,00s extraction + 0,00s propagation)
!  . Log search space  : 88,0 (before), 88,0 (after)
!  . Memory usage      : 411,5 kB (before), 427,5 kB (after)
! Using parallel search with 2 workers.
! ------------------------------------------------------------------------
```

Abbildung 5: PPT - Setup-Log des Solvers

Abbildung 6 zeigt im Ergebnis-Log des Solvers, dass zwei Lösungen gefunden wurden, deren Beste den Wert 32 liefert. Dafür wurden 88.115 Zweige in 2,21 Sekunden eruiert.

```
! ------------------------------------------------------------------------
! Search terminated normally, 2 solutions found.
! Best objective         : 32 (optimal - effective tol. is 0)
! Number of branches     : 88.115
! Number of fails        : 43.844
! Total memory usage     : 1,8 MB (1,5 MB CP Optimizer + 0,2 MB Concert)
! Time spent in solve    : 2,21s (2,21s engine + 0,00s extraction)
! Search speed (br. / s) : 39.713,8
! ------------------------------------------------------------------------
```

Abbildung 6: PPT Ergebnis-Log des Solvers

Die optimale Lösung des Solvers mit Kosten von 32 ist in Abbildung 7 gelistet. In der ersten Zeile wird der erreichte Zielwert angegeben. Darunter sind die Reihenfolge-Matrix *R* und die optimale Reihenfolge *o* der acht Bausteine aufgeführt. Hierbei bedeutet $o(5) = 1$, dass die Integration mit Baustein 5 (`Task`) beginnt. Zuletzt wird Baustein 7 (`ComplexTask`) integriert ($o(7) = 8$). Im Vergleich dazu erfordert die von Overbeck angegebene Reihenfolge <8, 5, 6, 2, 1, 7, 4, 3> die Kosten von 34 ([Ov94], S. 110ff.).

```
// solution with objective 32
R =  [[0 0 0 0 0 1 1 0]
      [1 0 1 0 0 1 1 0]
      [1 0 0 0 0 1 1 0]
      [1 1 1 0 0 1 1 1]
      [1 1 1 1 0 1 1 1]
      [0 0 0 0 0 0 1 0]
      [0 0 0 0 0 0 0 0]
      [1 1 1 0 0 1 1 0]];
o = [6 4 5 2 1 7 8 3];
```

Abbildung 7: PPT Lösung des Solvers

4.2 ATM Simulator

Das zweite Experiment verwendet die in Abbildung 8 dargestellten Abhängigkeiten der Automated Teller Machine (ATM) Simulation aus [BFL02], App. A, S. 38ff. Das System umfasst 21 Klassen, deren Abhängigkeiten durch Generalisierungs- bzw. Vererbungsbeziehungen (*inheritance*) sowie durch Kompositions- (*composition*), Assoziations- und Nutzungsbeziehungen zustande kommen.

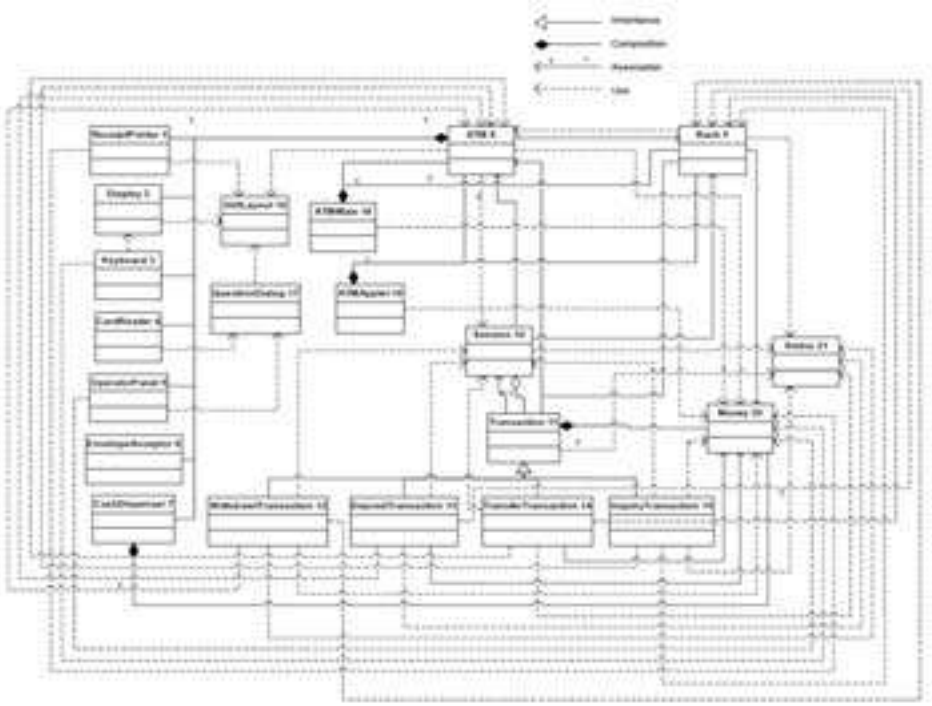

Abbildung 8: ATM-Simulator (ATM, aus [BFL02])

Bei den von Briand et Al. in [BFL02] beschriebenen Experimenten mit genetischen Algorithmen wurde festgelegt, dass die durch die Generalisierungs- sowie durch Kompositionsbeziehungen vorgegebenen Präzedenzen nicht durchbrochen werden dürfen. Im Optimierungsmodell wurden die entsprechenden Kanten daher mit den Maximalkosten belegt. Die übrigen Abhängigkeits-Kanten wurden in diesem Experiment mit den Werten der in [BFL02], A.3 auf S. 41 angegebenen Attribut-Kopplungsmatrix gewichtet.

Abbildung 9 und 10 zeigen das Setup- und das Ergebnis-Log des Solvers für die optimale Integrationsreihenfolge unter der Metrik *Attribut-Kopplung* (A, vgl. [BFL02]). In 145.943 Läufen wurden 13 optimale Reihenfolgen mit dem Zielfunktionswert 39 gefunden (aus max. 21! $\approx 5{,}1*10^{19}$ Möglichkeiten). Dieser wird auch von allen 10 in [BFL02] angegebenen Reihenfolgen erreicht. Eine der optimalen Reihenfolgen ist <20, 16, 17, 21, 5, 7, 2, 1, 6, 3, 4, 8, 9, 11, 10, 19, 18, 15, 13, 12, 14>.

```
! ------------------------------------------------------------------------
! Minimization problem - 462 variables, 631 constraints
! Initial process time : 0,00s (0,00s extraction + 0,00s propagation)
!  . Log search space  : 533,2 (before), 533,2 (after)
!  . Memory usage      : 0,9 MB (before), 1,1 MB (after)
! Using parallel search with 2 workers.
! ------------------------------------------------------------------------
```

Abbildung 9: ATM Setup-Log des Solvers (Attribute Coupling)

```
! ------------------------------------------------------------------------
! Search terminated normally, 13 solutions found.
! Best objective         : 39 (optimal - effective tol. is 0)
! Number of branches     : 145.943
! Number of fails        : 68.034
! Total memory usage     : 5,3 MB (5,0 MB CP Optimizer + 0,3 MB Concert)
! Time spent in solve    : 10,39s (10,39s engine + 0,00s extraction)
! Search speed (br. / s) : 14.045,6
! ------------------------------------------------------------------------
```

Abbildung 10: ATM Ergebnis-Log des Solvers (Attribute Coupling)

Die Ergebnisse zeigen, dass das einfache Optimierungsmodell aus (4) in der Lage ist, die mit den in der Literatur angegebenen Verfahren ermittelten Integrations-Reihenfolgen als optimal zu bestätigen oder sogar zu verbessern.

5. Zusammenfassung, Bewertung und Ausblick

Der vorgestellte Ansatz ermittelt eine optimale Integrationsreihenfolge mit dem Ziel, den Aufwand zur Erstellung von Test-Stellvertretern und -Treibern zu minimieren. Dazu wird die Problemstellung als ganzzahliges Optimierungsproblem formuliert, welches mit Verfahren der dynamischen Programmierung gelöst werden kann. Zwei Experimente zeigten die Leistungsfähigkeit des vorgestellten Ansatzes auf.

Die ganzzahlige Optimierung ist ein NP-hartes Problem, so dass die z.Zt. bekannten Lösungsverfahren im schlechtesten Fall exponentielle Laufzeit aufweisen ([PS82]). Die aktuellen Algorithmen zur Lösung konkreter Problemstellungen haben jedoch aufgrund der verwendeten Verfahren der linearen Programmierung in Verbindung z.B. mit Cutting-Plane-Algorithmen und Branch&Bound-Verfahren ([PS82]) zur Bestimmung ganzzahliger Lösungen in der Praxis oft eine erstaunlich hohe Effizienz ([IBM12]). Das Verfahren kann somit zumindest für kleine bis mittlere Systeme unmittelbar praktisch eingesetzt werden. Es ermöglicht aber auch die Ermittlung optimaler "Referenzreihenfolgen", anhand derer heuristische Verfahren bewertet und evaluiert werden können.

Unsere aktuellen Arbeiten loten die Grenzen des Ansatzes hinsichtlich Größe und Komplexität der zu integrierenden Software-(Teil-)Systeme aus. Darüber hinaus experimentieren wir mit unterschiedlichen Komplexitätsmetriken zur Ermittlung der Kanten-Gewichte, also des geschätzten „Aufwands“ für die Erstellung von Test-Stellvertretern und –Treibern.

Literaturverzeichnis

[AO09] Abdurazik, A. & Offutt, J.: Using Coupling-Based Weights for the Class Integration and Test Order Problem. Comput. J., Oxford University Press, 2009; S. 557-570.

[As11] Assunção, W. K. G.; Colanzi, T. E.; Pozo, A. T. R. & Vergilio, S. R.: Establishing integration test orders of classes with several coupling measures. Proc. 13th Conf. on Genetic and evolutionary computation, ACM, 2011; S. 1867-1874.

[Ba09] Bansal, P.; Sabharwal, S. & Sidhu, P.: An investigation of strategies for finding test order during integration testing of object-oriented applications. Proc. Int. Conf. on Methods and Models in Computer Science (ICM2CS 09), 2009; S. 1-8.

[BFL02] Briand, L. C.; Feng, J. & Labiche, Y.: Experimenting with Genetic Algorithms to Devise Optimal Integration Test Orders. Technical Report SCE-02-03, Carleton University, 2002.

[BLW03] Briand, L. C.; Labiche, Y. & Wang, Y.: An Investigation of Graph-Based Class Integration Test Order Strategies. IEEE Transactions on Software Engineering, Vol. 29, IEEE Computer Society, 2003; S. 594-607.

[BP09] Borner, L. & Paech, B.: Integration Test Order Strategies to Consider Test Focus and Simulation Effort. Proc. 1st Int. Conf. on Advances in System Testing and Validation Lifecycle (VALID 09), 2009; S. 80-85.

[GTB10] ISTQB/GTB Standardglossar der Testbegriffe, Version 2.1, Deutsch/Englisch, ISTQB/GTB, 2010.

[IBM12] Web-Seiten des IBM ILOG CPLEX Optimization Studio *http://www-01.ibm.com/software/integration/optimization/cplex-optimization-studio/* (zuletzt besucht am 10.10.2012)

[Je99] Jeron, T.; Jezequel, J.-M.; Le Traon, Y. & Morel, P.: Efficient strategies for integration and regression testing of OO systems. Proc. 10th Int. Symp. onSoftware Reliability Engineering, IEEE, 1999; S. 260-269.

[Ju03] Jungmayr, S.: Improving Testability of Object-Oriented Systems. Dissertation, Fernuniversität Hagen, 2003.

[Ku95] Kung, D., Gao, J., Hsia, P., Toyoshima, Y. & Chen, C.: A test strategy for object-oriented programs. Proc. 19th Computer Software and Applications Conf. (COMPSAC 95), IEEE Computer Society Press, 1995; S. 239-244.

[Le01] Le Hanh, V.; Akif, K.; Le Traon, Y. & Jézéque, J.-M.: Selecting an Efficient OO Integration Testing Strategy: An Experimental Comparison of Actual Strategies. In: Knudsen, J. (Ed.): Proc. ECOOP 2001, LNCS 2072, Springer, Berlin & Heidelberg, 2001; S. 381-401.

[Ov94] Overbeck, J.: Integration Testing for Object-Oriented Software. PhD Dissertation, Vienna University of Technology, 1994.

[PS82] Papadimitriou, C. H. & Steiglitz, K.: Combinatorial optimization - Algorithms and Complexity. Prentice-Hall, Englewood Cliffs, N.J., 1982.

[Ve10] Vergilio, S.; Pozo, A.; Arias, J.; da Veiga Cabral, R. & Nobre, T.: Multi-objective optimization algorithms applied to the class integration and test order problem. International Journal on Software Tools for Technology Transfer (STTT), Springer Berlin & Heidelberg, 2010; S. 1-15.

[Wi00] Winter, M.: Ein interaktionsbasiertes Modell für den objektorientierten Integrations- und Regressionstest. Informatik - Forschung und Entwicklung, Bd. 15, Springer Berlin / Heidelberg, 2000; S. 121-132.

[Wi12] Winter, M.; Ekssir-Monfared, M.; Sneed, H.M.; Seidl, R.; Borner; L.: Der Integrationstest – Von Entwurf und Architektur zur Komponenten- und Systemintegration. Hanser Verlag, München, 2012.