Softwaremodernisierung durch werkzeugunterstütztes Verschieben von Codeblöcken

Marvin Ferber, Sascha Hunold, Thomas Rauber
Lehrstuhl für Angewandte Informatik II, Universität Bayreuth
{marvin.ferber, hunold, rauber}@uni-bayreuth.de

Björn Krellner, Thomas Reichel, Gudula Rünger
Professur Praktische Informatik, Technische Universität Chemnitz
{bjoern.krellner, thomas.reichel, ruenger}@informatik.tu-chemnitz.de

Abstract: Das Zerlegen und Verschieben von Funktionen innerhalb einer Software ist ein wichtiges Mittel zur Umstrukturierung von Legacy-Anwendungen. Damit können sowohl die Komplexität einzelner Klassen reduziert als auch Komponenten durch Neuimplementierungen ersetzt werden. In dem vorliegenden Artikel wird beschrieben, wie und unter welchen Voraussetzungen zusammenhängende Teile von Klassen verschoben werden können. Dazu werden *MemberGroups* betrachtet, die einen zusammengehörenden Teil von Klassen darstellen und vom Rest der Klasse unabhängig sind. Das vorgestellte Verfahren wird anhand eines Codebeispiels erläutert.

1 Einleitung

Wartung und Weiterentwicklung großer Softwaresysteme ist ein weitreichendes Problem in der Softwareindustrie. Die Größe und Komplexität einzelner Softwaresysteme ist für einen einzelnen Programmierer kaum noch zu überblicken. Trotzdem unterliegt die Software ständig neuen Anforderungen und Kundenwünschen, die es notwendig machen, Teile des Gesamtsystems anzupassen oder neue Komponenten zu integrieren. Dadurch muss die Architektur und das Design verschiedener Komponenten angepasst werden. Aufgrund bestehender Strukturen können diese Änderungen oftmals nur behelfsmäßig in die Software eingearbeitet werden [Mas05]. Diese und weitere Probleme von Legacy-Software werden auch als *Code Decay* bezeichnet [EGK+01]. Eine ökonomisch tragbare Weiterentwicklung der über Jahre gewachsenen Software ist oft nur durch eine Reorganisation der Gesamtarchitektur möglich.

Das Projekt TransBS[1] beschäftigt sich mit der Legacy-Problematik von monolithischer Geschäftssoftware (z. B. ERP-Systeme). In [HKK+08] wird das favorisierte Vorgehen zur Modernisierung derartiger Softwaresysteme beschrieben. Mit dem

[1]Das diesem Bericht zugrundeliegende Vorhaben wurde mit Mitteln des Bundesministeriums für Bildung, und Forschung unter dem Förderkennzeichen 01 ISF 10 A gefördert. Die Verantwortung für den Inhalt dieser Veröffentlichung liegt beim Autor.

darin beschriebenen Werkzeug TransFormr kann ein sprachunabhängiges Softwaremodell aus abstrakten Syntaxbäumen (ASTs) des Sourcecodes extrahiert werden. Ausgehend von diesem Modell können Visualisierungen, Analysen (Metriken) oder auch Transformationen (Refactorings) erfolgen. In dem vorliegenden Artikel betrachten wir die Möglichkeiten und Bedingungen eine *MemberGroup* zwischen Klassen einer Legacy-Software zu verschieben, die in einer objektorientierten Programmiersprache geschrieben ist. Zu einer *MemberGroup* einer Klasse gehören alle Member-Variablen und -Methoden, die voneinander abhängig sind.

Das Verschieben von *MemberGroups* stellt eine komplexe Transformation dar, die aus gängigen Refactorings zusammengesetzt sein kann [Fow99]. Perkins et al. beschreiben, wie Programme mit Hilfe von Refactorings automatisiert an neue Bibliotheksfunktionen angepasst werden können [Per05]. Ein weiteres Framework zur werkzeugunterstützten Reorganisation von Software wird in [TK02] erläutert. Um die darin beschriebenen Refactorings anzuwenden, werden sowohl funktionale als auch nicht-funktionale Maße (z. B. Wartbarkeit) verwendet. In [FTC07] werden Distanzmetriken ausgenutzt, um die Zugehörigkeit einer Methode zu einer Klasse zu überprüfen. Ist die berechnete Distanz zu einer anderen Klasse geringer als zur aktuellen Klasse, wird sie verschoben. Weitere graphbasierte Ansätze zur Transformation von Software werden in [MDJ02] vorgestellt.

Der Artikel ist wie folgt gegliedert: In Abschnitt 2 werden sogenannte *MemberGroups* anhand des Abhängigkeitsgraphen eines Softwaresystems vorgestellt. Abschnitt 3 beschreibt, an welche Bedingungen das Verschieben einer *MemberGroup* geknüpft ist und wie potenzielle Zielklassen bestimmt werden können. An einem Beispiel wird in Abschnitt 4 das Verschieben einer *MemberGroup* verdeutlicht. Die Ergebnisse der Betrachtungen werden in Abschnitt 5 resümiert.

2 Definition der MemberGroup

Zur Definition von zusammenhängenden Codeblöcken beschreiben wir ein Softwaresystem als Graph. In Tabelle 1 sind die verwendeten Knotenmengen und Kantentypen erläutert, die als Basis der weiteren Definitionen dienen. Existiert eine Abhängigkeit zwischen zwei Knoten (Member) u und v, wird im Weiteren kurz $u \rightarrow v$ geschrieben.

Die Abhängigkeiten zwischen den einzelnen Membern des Projekts bilden einen Abhängigkeitsgraph. Die *dep*-Relation zwischen zwei beliebigen Membern M_1 und M_2 ($M_1 \neq M_2$) repräsentiert eine Abhängigkeit zwischen beliebigen Membern im Sourcecode. Demzufolge gibt $dep(M_1, M_2)$ dann *true* zurück, wenn $M_1 \rightarrow M_2$ oder $M_2 \rightarrow M_1$ gilt.

Speziell für Refactoring-Transformationen sind transitive Abhängigkeiten zwischen Variablen und Methoden zu berücksichtigen. Aus diesem Grund wird die $dep^*_{\mathcal{U}}(M_1, M_2)$-Relation eingeführt, die dann *true* liefert, wenn eine transitive Abhängigkeit zwischen den Membern M_1 und M_2 existiert. Die Richtung der Ab-

Tabelle 1: Knotenmengen und Kanten eines Softwaregraphen.

Beschreibung	Definition
Menge aller Klassen	$\Delta = \{C_0, \ldots, C_n\}$
Menge aller Konstruktoren einer Klasse C	$Con(C)$
Menge aller Methoden einer Klasse C	$Meth(C)$
Menge aller Member-Variablen der Klasse C	$Var(C)$
Menge aller Member der Klasse C	$M(C) = Meth(C) \cup Var(C) \cup Con(C)$
Menge aller Member des Projekts	$\mathcal{M} = \bigcup_{C \in \Delta} M(C)$
Ein Member u greift lesend auf Variable v zu	$u \xrightarrow{r} v$
Ein Member u schreibt Variable v	$u \xrightarrow{w} v$
In Methode u wird Methode v aufgerufen	$u \xrightarrow{call} v$

hängigkeit wird dabei wie bei der *dep*-Relation nicht beachtet (entspricht einem ungerichteten Pfad). Der Bezeichner $\mathcal{U}$ limitiert die Suche nach Abhängigkeiten auf eine Teilmenge $\mathcal{U} \subseteq \mathcal{M}$ aller Member. Mit Hilfe der $dep^*_{\mathcal{U}}$-Relation können logisch zusammengehörende Codeblöcke ermittelt werden.

Für einen Member $m \in \{Meth(C) \cup Var(C)\}$ der Klasse C wird die *MemberGroup* $MG(m)$ definiert:

$$\begin{aligned} MG(m) = \{m\} & \\ & \cup \{v \in \{Meth(C) \cup Var(C)\} \mid dep^*_{Var(C) \cup Meth(C)}(m, v) = 1\} \\ & \cup \ \{u \in Con(C) \mid (\exists t \in MG(m) \text{ mit } dep(u, t) = 1) \text{ und } u \neq m\}. \end{aligned} \tag{1}$$

In einer *MemberGroup* ist demnach der Member m selbst enthalten. Außerdem werden alle Methoden und Member-Variablen hinzugefügt, die eine transitive Abhängigkeit zu m innerhalb der Klasse C aufweisen. Zusätzlich werden noch die Konstruktoren der Klasse C aufgenommen, die eine direkte Abhängigkeit zu den Member-Variablen und *MemberGroup*-Methoden besitzen.

MemberGroups stellen somit funktional unabhängige Teilgraphen innerhalb des Abhängigkeitsgraphen einer Klasse dar. Konstruktoren $Con(C)$ einer Klasse C können in jedem dieser Teilgraphen vorkommen, um Abhängigkeiten aus Initialisierungen von Member-Variablen gesondert zu betrachten.

Bei der Verschiebung einer *MemberGroup* können auch Abhängigkeiten durch Realisierungen und Generalisierungen zu Einschränkungen führen. Da die Beschreibung dieser Abhängigkeiten den Umfang des Artikels übersteigt, beschränken wir uns im Folgenden auf die Betrachtung der definierten *dep*-Beziehungen.

Tabelle 2: Verwendete Bezeichner zum Verschieben von *MemberGroups*.

Bezeichner	Erläuterung
$M_{has}(S)$	Menge der Klassen mit Variable(n) vom Typ S.
$M_{ref}(\Psi, S)$	Menge der Klassen aus $\Delta \setminus \{S\}$, die in allen Klassen aus Ψ referenziert werden.
$M_{Ta}(S)$	Menge der möglichen Zielklassen (target classes).
$type(v)$	Datentyp (Klasse) einer Variable v.

3 Verschieben von MemberGroups

Bei der Umstrukturierung von Legacy-Software ist es oft notwendig, Klassen aufzuspalten. Neben einer Reduzierung der Komplexität der Klasse können so auch Abhängigkeiten zwischen Klassen beseitigt werden, was zu einem weniger gekoppelten Sourcecode führt. Eine mögliche Trennung von Zuständigkeiten stellt das Verschieben einer *MemberGroup MG* aus einer Klasse S in eine andere Klasse T dar. Im Weiteren wird betrachtet, welche Voraussetzungen erfüllt sein müssen, um eine *MemberGroup* zu verschieben.

3.1 Zielklassenbestimmung

Soll eine *MemberGroup MG* aus S verschoben werden, so müssen die möglichen Zielklassen in der Menge Δ bestimmt werden. Dazu werden zunächst die Klassen betrachtet, die eine Member-Variable vom Typ S besitzen. Diese Menge wird als $M_{has}(S)$ bezeichnet und ist wie folgt definiert:

$$M_{has}(S) = \{c \in \Delta \mid \exists x \in Var(c) \text{ mit } type(x) = S\}\,. \tag{2}$$

Will man die *MemberGroup* von einer Klasse S in eine Klasse T verschieben, muss sichergestellt werden, dass die Klassen, die S benutzen, auch Zugriff auf eine Instanz von T haben. Deshalb betrachten wir es in dieser Arbeit als notwendige Bedingung, dass die Klassen, die eine Member-Variable vom Typ S besitzen, auch eine Member-Variable der Zielklasse T besitzen. Aus diesem Grund werden die Klassen ermittelt, die als Datentyp einer Member-Variable in allen Klassen einer Menge $\Psi \subseteq \Delta$ verwendet werden. Diese Menge sei als M_{ref} bezeichnet und wird wie folgt bestimmt:

$$M_{ref}(\Psi, S) = \{t \in \Delta \mid t \neq S \text{ und } (\forall v \in \Psi\ \exists x \in Var(v) \text{ mit } type(x) = t)\}\,. \tag{3}$$

Damit eine *MemberGroup* verschoben werden kann, müssen demnach die Klassen ermittelt werden, die eine Variable vom Typ S besitzen (Menge $M_{has}(S)$). Danach werden in den Klassen aus M_{has} die Klassen gesucht, die in allen Klassen aus M_{has} referenziert werden und nicht S selbst sind. Mit den Definitionen der Mengen M_{has}

```
class A                    class B                    class C
{                          {                          {
 S s;                       S s;                       S s;
 T t;                       List<T> tlist;             T t;
}                           X x;                       X x;
                           }                          }
```

Abbildung 1: Beispiel zur Bestimmung der Zielklassen (Menge M_{Ta}).

und M_{ref} lässt sich die Menge der möglichen Zielklassen M_{Ta} für das Verschieben einer *MemberGroup* wie folgt angeben:

$$M_{Ta}(S) = M_{ref}(M_{has}(S), S). \tag{4}$$

Die Menge $M_{Ta}(S)$ enthält alle die Klassen des Projekts, die in den Klassen benutzt werden, die auch eine Member-Variable vom Typ S benutzen. Dies sei an dem Beispiel in Abbildung 1 verdeutlicht. Die Klassen A, B und C besitzen jeweils eine Variable vom Typ S und gehören somit zur Menge $M_{has}(S)$. Sucht man in dieser Menge nach möglichen Zielklassen, findet man die Klasse T, da sie von allen drei Klassen referenziert wird. Hingegen wird die Klasse X nicht in der Klasse A benutzt und kann deshalb nicht als Zielklasse in Frage kommen.

Die Selektion einer Zielklasse $T \in M_{Ta}(S)$ ist von der *MemberGroup* und von den Klassen, die sowohl S als auch T referenzieren, abhängig. Das bedeutet, dass die Menge der Zielklassen für den konkreten Fall nochmals überprüft und ggf. verkleinert werden muss.

3.2 Analyse der Beziehungen zwischen Quell- und Zielklasse

Nachdem die Zielklasse $T \in M_{Ta}(S)$ bestimmt ist, müssen die Abhängigkeiten zwischen Quelle und Ziel in allen Klassen analysiert werden, die S und T verwenden. Diese Klassen werden im Folgenden als $\xi \subseteq M_{has}(S)$ bezeichnet, d. h. jede Klasse $z \in \xi$ besitzt mindestens eine Member-Variable vom Typ S und eine vom Typ T:

$$\begin{aligned}\xi(S,T) = \{t \in \Delta \mid \exists x \in \mathit{Var}(t)\ \exists y \in \mathit{Var}(t) \\ \text{mit } (\, S = \mathit{type}(x) \text{ und } T = \mathit{type}(y)\,)\}.\end{aligned} \tag{5}$$

Eine Klasse $z \in \xi(S,T)$ kann unterschiedlich viele Variablen vom Typ S und vom Typ T aufweisen. Dabei können vier Fälle unterschieden werden, wobei $\mathcal{V}(i{:}j)$ angibt, wie viele Instanzvariablen vom Typ S (i) und wie viele vom Typ T (j) deklariert werden: $\mathcal{V}(1{:}1)$, $\mathcal{V}(1{:}n)$, $\mathcal{V}(n{:}1)$ und $\mathcal{V}(m{:}n)$. Für die drei Fälle, in denen n Variablen eines Typs verwendet werden, können nur im konkreten Fall Verschieberegeln definiert werden. Als Beispiel für den Fall $\mathcal{V}(1{:}n)$ sei eine Liste von Variablen des Typs T genannt, deren Länge erst zur Laufzeit bestimmt wird. Da für diese

```
s = new S();
t = new T();
s.foo();
s.bar();
s.a();
t = new T();
s.a();
```

```
s = new S();
v = new V();
t = new T();
t.setV(v);
s.foo();
s.bar();
t.a();  // intern v.a()  (Delegation)
t = new T();
t.setV(v);
t.a();
```

Abbildung 2: Beispiel einer Codetransformation für eine $\mathcal{I}(1{:}n)$-Beziehung (links: Ursprungscode, rechts: transformierter Code). Die neue Klassen V kapselt den Zustand der Ursprungklasse S.

Fälle auch noch die Anzahl der Instanziierungen eines Listeneintrags eine Rolle spielt, beschränken wir unsere weiteren Betrachtungen auf den Fall $\mathcal{V}(1{:}1)$.

Jedoch müssen auch in diesem Fall vier Unterfälle unterschieden werden, die analog zu den vorangegangenen Fällen definiert sind. Statt der Anzahl der deklarierten Variablen werden hierzu die Anzahl der Instanziierungen der Klassen S und T in z betrachtet. Existieren in einer Klasse z zwei Member-Variablen s und t vom Typ S respektive T, dann können jeder Variable im Verlauf der Ausführung unterschiedlich viele Instanzen des jeweiligen Typs zugewiesen werden. Mit $\mathcal{I}(k{:}l)$ sei der Fall bezeichnet, bei dem k Instanzen der Variable vom Typ S und l Instanzen der Variable vom Typ T zugewiesen werden. Der Fall $\mathcal{I}(m{:}n)$, für beliebige m und n, wird an dieser Stelle nicht weiter betrachtet. In diesem Fall müssen komplexe Codeanalysen herangezogen werden, die über eine einfache Transformationsregel hinausgehen.

Für $\mathcal{I}(1{:}n)$ und $\mathcal{I}(n{:}1)$ können im konkreten Fall, also mit Kenntnis der *MemberGroup*, Transformationsregeln gefunden werden. Eine mögliche Transformation für den Fall $\mathcal{I}(1{:}n)$ ist in Abbildung 2 veranschaulicht. Das dargestellte Problem besteht in der Beibehaltung des ursprünglichen internen Zustands der *MemberGroup*. Da im Ausgangscode, auf der linken Seite der Abbildung, nur *ein* S instanziiert wird, muss dieser *eine* Zustand auch in der transformierten Variante gerettet werden. Eine denkbare Lösung wäre die Nutzung einer zusätzlichen Klasse V, die die Methoden und Variablen der *MemberGroup* kapselt. Um in jeder Instanz von T auf dasselbe V zuzugreifen, muss sichergestellt werden, dass alle Instanzen eine Referenz zu V erhalten. Eine solche Transformation ist an verschiedene Bedingungen des Sourcecodes geknüpft. Im betrachteten Szenario muss z. B. sichergestellt werden, dass ein T im Ursprungscode instanziiert wird, bevor ein Aufruf einer Methode der *MemberGroup* von S stattfindet.

Die Transformationsregeln für $\mathcal{I}(1{:}n)$ und $\mathcal{I}(n{:}1)$ hängen vom Code der Klassen aus ξ und von der *MemberGroup* ab. In vielen Fällen kann eine statische Codeanalyse nicht alle Instanziierungen ermitteln. Deshalb betrachten wir im folgenden Beispiel den Fall $\mathcal{I}(1{:}1)$.

```
public class Z
{
 S s;
 T t;
 public Z() {
  this.s = new S();
  this.t = new T();
 }
 public void proc() {
  s.func();
  ...
 }
}

class S
{
 private int a;
 public S() {
  this.a = 0;
 }
 public void func() {
  this.a++;
 }
}

class T
{
}
```

```
public class Z
{
 S s;
 T t;
 public Z() {
  this.s = new S();
  this.t = new T();
 }
 public void proc() {
  t.func();
  ...
 }
}

class S
{
}

class T
{
 private int a;
 public T() {
  this.a = 0;
 }
 public void func() {
  this.a++;
 }
}
```

Abbildung 3: Verschiebung einer *MemberGroup* aus der Klasse S in die Klasse T (links: Ursprungscode, rechts: transformierter Code).

4 Exemplarisches Verschieben einer MemberGroup

Im folgenden Beispiel wird ein Softwaresystem mit den drei Klassen S, T und Z betrachtet. Abbildung 3 (links) zeigt einen Codeausschnitt der Ausgangssituation vor dem Verschieben. In der Klasse S ist die *MemberGroup* $MG(\texttt{func()}) = \{\texttt{func()}, \texttt{a}\}$ enthalten, welche im Weiteren als MG abgekürzt wird.

Mit dem Werkzeug TransFormr [HKK+08] können der Aufbau und die Abhängigkeiten in der Klassenstruktur in die zuvor definierte Graphstruktur überführt werden. Sie besteht aus einem vereinfachten abstrakten Syntaxbaum, welcher Abhängigkeiten (z. B. Funktionsaufrufe) als ausgezeichnete Kanten zwischen den Knoten abbildet. Mit Hilfe dieser Informationen kann ermittelt werden, wie viele Member-Variablen von S und T in Z existieren und welche Klassen die MG verwenden. Im Beispiel symbolisiert die Kante $\texttt{proc()} \overset{call}{\rightarrow} \texttt{func()}$ den Aufruf von `proc()` durch `func()`. In gleicher Weise kann ermittelt werden, welche Instanzen von S und T erzeugt werden. Somit lassen sich die Beziehungen zwischen den Klassen automatisch bestimmen.

Nach Gleichung (4) enthält die Menge der Zielklassen $M_{Ta}(S)$ nur die Klasse T. Falls es mehrere mögliche Zielklassen gibt, werden Codemetriken einbezogen, um

z. B. die Abhängigkeiten innerhalb der Software durch eine Transformation zu reduzieren.

Das vorliegende Codebeispiel enthält eine $\mathcal{V}$(1:1)-Beziehung von Variablen, da genau eine Variable jedes Typs deklariert wurde. Außerdem handelt es sich um eine $\mathcal{I}$(1:1)-Beziehung zwischen den Variablen, da beide Member-Variablen einmal im Konstruktor instanziiert werden. Für diese Art von Beziehung zwischen S und T lässt sich die *MemberGroup* wie folgt verschieben:

```
1 resolveNameClashes();
2 moveMemberGroup();
3 replaceMemberGroupReferences();
```

Die Methode `resolveNameClashes()` löst Kollisionen zwischen übereinstimmenden Bezeichnern in MG und $M(T)$ auf. Danach wird der Teilgraph der *MemberGroup* mit allen Variablen und Methoden von S nach T verschoben. Dies wird von `moveMemberGroup()` realisiert und umfasst auch Statements, die Variablen von MG im Konstruktor von S initialisieren. Diese Statements werden an das Ende des Konstruktors von T verschoben werden. Abschließend werden durch `replaceMemberGroupReferences()` alle Aufrufe der Methoden von MG in der Klasse Z durch die Referenz zur Zielklasse T ersetzt. In Abbildung 3 (Codeausschnitt rechts) sind die Klassen S, T und Z nach der Verschiebung dargestellt.

5 Zusammenfassung

Die Modernisierung von Legacy-Software erfordert vor allem eine Modularisierung, die mit einer Reduzierung der Komplexität des Gesamtsystems einhergeht. Aus diesem Grund müssen Refactoring-Methoden gefunden werden, die den Entwicklern in Werkzeugen (wie IDEs) die Möglichkeit bieten, verschiedene Umstrukturierungen automatisch durchzuführen. In diesem Artikel wird das Verschieben von *MemberGroups* zwischen zwei Klassen einer Software betrachtet, die in einer objektorientierten Programmiersprache geschrieben wurde. Es wird gezeigt, welche Voraussetzungen gegeben sein müssen, damit eine solche Aufgabe automatisch ausgeführt werden kann. Außerdem wird beschrieben, wie die Klassen anzupassen sind, die die Quell- und Zielklasse der Transformation verwenden. Abschließend wird an einem Softwarebeispiel das Verschieben einer *MemberGroup* demonstriert.

Literatur

[EGK+01] Stephen G. Eick, Todd L. Graves, Alan F. Karr, James S. Marron und Audris Mockus. Does Code Decay? Assessing the Evidence from Change Management Data. *IEEE Trans. Softw. Eng.*, 27(1):1–12, 2001.

[Fow99] Martin Fowler. *Refactoring: Improving the Design of Existing Code.* Addison-Wesley, Boston, MA, USA, 1999.

[FTC07] Marios Fokaefs, Nikolaos Tsantalis und Alexander Chatzigeorgiou. JDeodorant: Identification and Removal of Feature Envy Bad Smells. *23rd IEEE Int. Conf. Softw. Maint. (ICSM 2007)*, Seiten 519–520, October 2007.

[HKK+08] S. Hunold, M. Korch, B. Krellner, T. Rauber, T. Reichel und G. Rünger. Transformation of Legacy Software into Client/Server Applications through Pattern-Based Rearchitecturing. In *Proc. 32nd IEEE Int. Comp. Softw. and Appl. Conf.*, Seiten 303–310, Los Alamitos, CA, USA, 2008. IEEE Comp. Soc.

[Mas05] Dieter Masak. *Legacysoftware: Das lange Leben der Altsysteme*. Springer, Berlin, 2005.

[MDJ02] Tom Mens, Serge Demeyer und Dirk Janssens. Formalising Behaviour Preserving Program Transformations. In *ICGT '02: Proc. 1st Int. Conf. Graph Transformation*, Seiten 286–301, London, UK, 2002. Springer-Verlag.

[Per05] Jeff H. Perkins. Automatically Generating Refactorings to Support API Evolution. Seiten 111–114, New York, NY, USA, September 2005. ACM.

[TK02] Ladan Tahvildari und Kostas Kontogiannis. A Software Transformation Framework for Quality-Driven Object-Oriented Re-engineering. *Proc. Int. Conf. Softw. Maint. (ICSM'02)*, Seiten 596–605, 2002.