Optimierte Suche von Modellinstanzen UML/OCL-Beschreibungen in USE

Lars Hamann, Fabian Buettner, Mirco Kuhlmann, Martin Gogolla

Optimierte Suche von Modellinstanzen für UML/OCL-Beschreibungen in USE

Lars Hamann
AG Datenbanksysteme
Universität Bremen
lhamann@tzi.de

Fabian Büttner*
AtlanMod, INRIA
Ecole des Mines de Nantes
fabian.buettner@inria.fr

Mirco Kuhlmann
AG Datenbanksysteme
Universität Bremen
mk@tzi.de

Martin Gogolla
AG Datenbanksysteme
Universität Bremen
gogolla@tzi.de

Abstract: Konzeptuelle Modelle sind ein wichtiges Element modellgetriebener Softwareentwicklung, sowohl in der Beschreibung von Systemen als auch in der Metamodellierung domänenspezifischer Sprachen. Zu ihrer Beschreibung haben sich UML und OCL (und angelehnte Sprachen) als ein de facto Standard durchgesetzt. Validierung und Verifikation der Modelle sind hierbei wichtige Instrumente zur Sicherstellung der Modellqualität. Die Sprache ASSL (A Snapshot Sequence Language) bietet die Möglichkeit durch imperative Programmierung auf Modellebene und Backtracking konforme Instanzen systematisch zu erzeugen. Der White-Box-Ansatz ASSL ergänzt Black-Box-Ansätze, welche die Modellinstanziierung durch Abbildung auf (bspw.) ein Problem der relationalen Logik lösen. Dieser Beitrag beschreibt, wie die durch ASSL-Programme aufgespannten Suchräume durch Ausnutzung der Modellabdeckung der OCL-Constraints und der Modellstruktur erheblich verkleinert werden können und gibt einen Ausblick darauf, wie bestehende Black-Box-Ansätze in ASSL integriert werden können, um innerhalb eines imperativen Rahmens Teilinstanziierungen deklarativ beschreiben zu können.

1 Einleitung

Sauber definierte konzeptuelle Modelle sind ein wichtiges Element modellgetriebener Softwareentwicklung. Neben der direkten Verwendung in der Beschreibung von Softwaresystemen werden sie immer häufiger auch als Metamodelle zur Definition domänenspezifischer Sprachen benötigt. Zur Beschreibung dieser Art von Modellen hat sich die Verwendung von Klassendiagrammen und OCL-Constraints [OMG10] als ein de facto Standard durchgesetzt, wobei die Beschreibung der Klassenstrukturen in UML [OMG11b] oder in für diesen Bereich analogen Sprachen wie MOF [OMG11a], Ecore [DSM08] oder KM3 [JB06] erfolgen kann.

Validierung und Verifikation von Modellen sind Instrumente zur Sicherstellung der Mo-

*Diese Arbeit wurde teilweise durch das europäische Projekt CESAR unterstützt

dellqualität. Das Werkzeug USE (UML-based Specification Environment) [GBR07, USE] unterstützt den Entwickler[1] in diesen Aktivitäten bei der Erstellung von UML/OCL-Modellen.

Neben der Validierung von Modellinstanzen gegen ihr Modell bietet USE mit der Sprache ASSL (A Snapshot Sequence Language) [GBR05] die Möglichkeit, innerhalb gegebener Grenzen, zum Modell konforme Instanzen systematisch zu erzeugen, wobei auch weitere erwünschte Eigenschaften berücksichtigt werden können. Auf diesem Wege können Eigenschaften wie Redundanz oder Unerfüllbarkeit von Constraints gezeigt werden. Bei der Validierung des Modells durch ASSL handelt es sich aus Anwendersicht um einen „White-Box"-Ansatz. Die Beschreibung der Modell-Suchprozedur erfordert Kenntnis des Modells und seiner Constraints, um den Suchraum, welcher durch das Backtracking in ASSL entsteht, nicht zu groß werden zu lassen. Gleichzeitig bietet ASSL dem Anwender damit aber auch die Möglichkeit, den Suchraum nach Bedarf zu beeinflussen und Instanzen mit Eigenschaften zu erzeugen, die prozedural einfacher als in deklarativer Weise zu beschreiben sind.

Demgegenüber stehen "Black-Box"-Ansätze zur Instanziierung, welche die Aufgabe automatisiert in Formalismen wie die relationale Logik (Alloy [ABGR10], Kodkod [KHG11]), das Constraint Programming [CCR07] oder die genetische Programmierung [AIAB11] überführen, für welche ausgereifte Entscheidungsprozeduren existieren. Diese Verfahren haben den Vorteil, dass damit Instanzen für UML/OCL-Modelle gesucht und partielle Instanzen vervollständigt werden können, ohne dass der Anwender die Suche spezifisch beschreiben muss. Obwohl mit diesen Black-Box-Ansätzen effiziente Werkzeuge bereitstehen, hat die operational beschriebe Instanziierung von Modellen durch ASSL weiterhin ihre Berechtigung. Über die Fälle hinaus, in denen die gewünschten Instanzeigenschaften auf diese Weise einfacher beschrieben werden können, trifft dies insbesondere auf Instanzen und Modelle zu, die für die Abbildung auf z. B. die relationale Logik ungeeignet sind, weil sie Rekursion enthalten oder die Wertebereiche zu groß sind (trotz aller Effizienz der o.g. Ansätze ist das zugrundeliegende Problem im Allgemeinen NP-vollständig), vgl. dazu u.a. [BGF+10, AIAB11].

Auch wenn ASSL aufgrund seines enumerativen Charakters grundsätzlich schlechter skaliert als die o. g. Ansätze, lassen sich mit ASSL auch für diese Fälle Lösungen finden, wenngleich die Prozeduren in diesen Fälle spezifisch zu dem Modell und seinen Constraints erstellt werden müssen und nur noch wenige Backtracking-Punkte enthalten dürfen. Unsere aktuelle Arbeit hat das Ziel diese Einschränkung abzumildern und verbessert die Leistungsfähigkeit von ASSL durch zwei Veränderungen.

1. ein effektiveres Abschneiden von nicht zur Lösung führenden Zweigen im Backtracking auf Grundlage der Modellabdeckung der Constraints sowie der Modellstrukur;
2. eine Integration der Ergebnisse aus [KHG11] in ASSL, so dass auch innerhalb von ASSL-Prozeduren Teilinstanziierungen mit Hilfe der Abbildung auf die relationale Logik von Kodkod gesucht werden können, während die Gesamtsuche weiterhin prozedural durch ASSL beschrieben wird.

[1] Im Artikel wird nur die männliche Form verwendet, die aber stellvertretend für beide Geschlechter steht.

Dieser Beitrag stellt die Änderungen vor und bewertet die erreichte Verkleinung des Suchraums für (1.). Dieser Beitrag ist wie folgt gegliedert: Abschnitt 2 stellt die bisher vorhandenen Suchansätze (ASSL und Instanziierung durch Kodkod) vor. In Abschnitt 3 erläutern wir, wie das Backtracking in ASSL durch Ausnutzen der Modellstruktur und von sogenannten Constraint-Abdeckungen sowie über die Instanziierung von Teilmodellen über Kodkod verringert werden sollen. Ein laufendes Beispiel dient zur Veranschaulichung der Änderungen und zur Evaluation der Verbesserungen. Abschnitt 4 stellt unsere Ergebnisse in Bezug zu verwandten Arbeiten. Abschnitt 5 nimmt ein Fazit vor und erläutert die zukünftige Arbeit.

2 Modellinstanzsuche in USE

In USE kann mit Hilfe eines Generators für Objektdiagramme und der dazugehörigen Sprache ASSL (A Snapshot Sequence Language) nach Modellinstanzen gesucht werden. Die Suche nach Modellinstanzen kann dabei sehr detailliert beschrieben werden [GBR05]. ASSL ist eine einfache prozedurale Sprache, die zusätzlich zu normalen Befehlen zur Zustandsmanipulation (Erzeugen von Instanzen und Links, Setzen von Attributwerten) spezielle Befehle mit einem integrierten Backtracking Mechanismus bereitstellt.

2.1 Generator und ASSL

Die verfügbaren Anweisungen in der Generatorsprache ASSL lassen sich in zwei Kategorien einteilen. Die erste Kategorie ist vergleichbar mit gängigen atomaren Answeisungen in einer Programmiersprache. Mit Anweisungen dieser Kategorie können unter anderem Objektinstanzen und Links erzeugt werden und es stehen einfache Kontrollstrukturen zur Verfügung. Die zweite Kategorie von Anweisungen spannt die Zustandsräume auf und hebt sich durch ein integriertes Backtracking von der ersten Kategorie ab.

Als atomare Anweisung stehen Anweisungen zur Modifikation der Modellinstanz zur Verfügung, die imperativ die Instanz verändern. Dazu gehören Anweisungen zur Erzeugung von Instanzen (`Create, CreateN`) und Links (`Insert` bzw. `Create` auf Assoziationsklassen). Weiterhin können Attribut- und Variablenzuweisungen durchgeführt werden (`:=`). Zur Steuerung des Kontrollflusses stehen die Anweisungen `if` und `for` zur Verfügung. Alle hier genannten Anweisungen unterscheiden sich von Anweisungen aus bekannten Programmiersprachen nur durch die Möglichkeit, dass ihre Auswirkungen automatisch rückgängig gemacht werden können. Dieser Mechanismus wird für die im Folgenden beschriebenen Backtracking-Anweisungen benötigt.

2.1.1 Try-Anweisungen

Alle Anweisungen zum Aufspannen des Suchraums sind in ASSL mit `Try` benannt und unterscheiden sich nur anhand ihrer Parameter. Allen Try-Anweisungen gemein ist, dass

sie auf Mengen von Instanzen oder Werten arbeiten. Beim Backtracking innerhalb des Generators werden bei Bedarf alle Wertekombinationen eines Try-Befehls durchsucht. Bei mehreren aufeinander folgenden Try-Befehlen geschieht dies mit sämtlichen Kombinationen der jeweiligen Try-Befehle. Das Backtracking setzt ein, sobald eine ASSL-Prozedur bis zum Ende durchlaufen wurde und dieser Ausführungspfad zu keinem gültigen Zustand geführt hat. Der aktuell geprüfte Zustand kann aufgrund einer Strukturverletzung oder durch die Verletzung von Invarianten ungültig sein. Strukturverletzungen beziehen sich auf in der UML spezifizierbare Einschränkungen die z. B. durch Verletzung der Multiplizitäten oder der Existenz eines Zyklus innerhalb einer Kompositionsstruktur [HGK] auftreten können. Die Verletzung von Invarianten resultiert entweder daraus, dass diese nicht erfüllt werden oder, bei abgeschalteter Strukturprüfung, aus fehlerhaften Navigationsergebnissen falls z. B. zwei Instanzen über 0..1 Assoziationsende erreichbar sind.

Die Anzahl der Kombinationen, die ein Try-Befehl im schlechtesten Fall maximal auswertet hängt von der Art des Trys ab. Nachfolgend werden die Arten von Try-Anweisungen, die in ASSL zur Verfügung stehen beschrieben. Dabei wird auch die Formel zur Berechnung der maximal auszuwertenden Kombinationen ($\mathcal{K}$) angegeben. Die Anzahl der Elemente innerhalb einer Collection wird durch die Betragszeichen $|$*collection*$|$ dargestellt.

`Try(values:Sequence)` Liefert sukzessive ein Element der übergebenen Sequenz in jedem Backtracking-Schritt. Der Rückgabewert kann anschließend beliebig weiterverwendet werden. $\mathcal{K} = |\texttt{values}|$

`Try(association, seq`$_1$**`:Sequence, ..., seq`**$_n$**`:Sequence)`** Testet sukzessive alle Kombinationen von Linkmengen zwischen den Sequenzen $\texttt{Sequence}_1$ bis $\texttt{Sequence}_n$, wobei n gleich der Anzahl der Assoziationsenden von `association` ist. $\mathcal{K} = 2^{|\texttt{seq}_1|*\ldots*|\texttt{seq}_n|}$

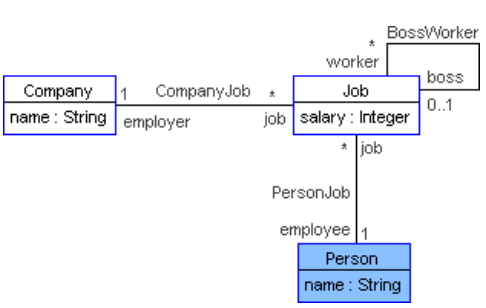
`Try(objects:Collection, attributeName, values:Sequence)` Testet sukzessive alle möglichen Attributwertkombinationen für die in `objects` enthaltenen Instanzen, die sich aus der übergebenen Wertesequenz `values` ergeben. $\mathcal{K} = |\texttt{objects}|^{|\texttt{values}|}$

2.1.2 Suchraum und Suchbaum

Im ungünstigsten Fall muss der Generator jede Kombination für jedes Try probieren. Die Anzahl der Blätter des sich dabei aufspannenden Suchbaumes ergibt dabei die zu prüfenden Zustände, also die Suchraumgröße ($\mathcal{S}$). Betrachtet man nur Try-Anweisungen, dann stellt eine Try-Anweisung einen Knoten dar, der sich $\mathcal{K}$ mal zur nächsten Try-Anweisung verzweigt. Die Gesamtzahl der Blätter berechnet sich durch das Produkt der jeweiligen maximalen Kombinationen der verwendeten Try-Anweisungen:

$$\mathcal{S} = \prod_{i=1}^{|Try|} \mathcal{K}(Try_i)$$

Der gesamte Suchraum einer ASSL-Prozedur lässt sich in den meisten Fällen einfach berechnen. Da an verschiedenen Stellen aber auch beliebige OCL Ausdrücke verwendet wer-




```
context Company inv companyNamesAreUnique:
  Company.allInstances->forAll(c2|
    self.name=c2.name implies self=c2)
context Person inv personNamesAreUnique:
  Person.allInstances->forAll(p2|
    self.name=p2.name implies self=p2)
context Job inv bossWorkerSameEmployer:
  self.worker->forAll(j|
    j.employer=self.employer)
context Job inv bossWorkerIsHierarchy:
  Set{self}->closure(boss)
    ->excludes(self)
```

Abbildung 1: Beispielmodell und -constraints

den können, die unter anderem durch rekursive Funktionen definiert sein können, kann die Suchraumgröße nicht mit einer allgemeinen Berechnungsvorschrift bestimmt werden.

Die in Listing 1 gezeigte generische ASSL-Prozedur versucht für das Beispiel aus Abbildung 1 einen Zustand zu finden, der für die einzelnen Klassen jeweils *numX* Instanzen enthält. Dabei werden die Namensattribute gesetzt und es werden Linkkombinationen auf den verschiedenen Asssoziationen ausprobiert.

Listing 1: Beispiel ASSL-Prozedur

```
1  procedure buildState(numCompanies:Integer,
2                       numJobs:Integer,
3                       numPersons:Integer,)
4  var
5    companies:Sequence(Company), persons:Sequence(Person), jobs:Sequence(Job),
6    personNames:Sequence(String), companyNames:Sequence(String);
7  begin
8    personNames  := [Sequence{'Ada','Bel','Cam','Day','Ali','Bob','Cyd','Dan'}];
9    companyNames := [Sequence{'All Inc.','Brew Ltd.','Cup Coop.','Duff Ltd.'}];
10
11   companies := CreateN(Company, [numCompanies]);
12   persons   := CreateN(Person,  [numPersons]);
13   jobs      := CreateN(Job,     [numJobs]);
14
15   Try([Person.allInstances()], name, [personNames]);
16   Try([Company.allInstances()], name, [companyNames]);
17
18   Try(CompanyJob, [companies], [jobs]);
19   Try(BossWorker, [jobs],      [jobs]);
20   Try(PersonJob,  [persons],   [jobs]);
21 end;
```

Der gesamte Suchraum berechnet sich bei dieser Prozedur anhand folgender Formel (unterhalb der jeweiligen Faktoren sind die Zeilennummern der entsprechenden Anweisungen aus Listing 1 angegeben):

$$\mathcal{S} = \underbrace{p^8}_{Z.15} * \underbrace{c^4}_{Z.16} * \underbrace{2^{(c*j)}}_{Z.18} * \underbrace{2^{(j*j)}}_{Z.19} * \underbrace{2^{(p*j)}}_{Z.20}$$

$$\text{mit } c = \texttt{numCompanies}, p = \texttt{numPersons}, j = \texttt{numJobs}$$

Eine effiziente Suche ist durch die kombinatorische Explosion schon bei kleinen Anzahlen von zu suchenden Objekten mit diesem Ansatz also nicht mehr möglich. Schon bei

jeweils 3 zu suchenden Objekten ergibt sich eine Suchraumgröße von 71.328.803.586.048 Zuständen.

Um dieser Zustandsexplosion entgegen zu wirken kann der Benutzer die ASSL-Prozedur mit zusätzlichem Wissen ausstatten, indem er verschiedene Optimierungsmöglichkeiten nutzt. Dabei muss allerdings beachtet werden, dass Zustände, die auf eine fehlerhafte Modellspezifikation hinweisen, möglicherweise ausgeschlossen werden, falls der konstruierte Suchraum diese nicht berücksichtigt. Daher sollten zusätzliche Maßnahmen zur Verbesserung bzw. Sicherung der Modellqualität, z. B. Unit-Tests für Modelle [HG10], eingesetzt werden.

Aufgrund der prozeduralen Möglichkeiten in ASSL kann die beschriebene generische Methode an bestimmten Stellen durch den Benutzer verbessert werden. Für die gezeigte Prozedur bietet es sich an, die Belegung der Namen prozedural zu beschreiben, da die Invarianten alleine bereits verifiziert werden können. Weiterhin kann ASSL dazu verwendet werden, die zu findende Modellinstanz aussagekräftiger zu gestalten. Im folgenden Beispiel wird dies durch die Zuweisung von männlichen und weiblichen Vornamen an Personen gezeigt. Kombiniert mit einem Attribut `geschlecht` ergibt dies eine sinnvollere Belegung innerhalb der gefundenen Modellinstanzen. Das Beispiellisting 2 ersetzt die Try-Anweisungen zur Attributzuweisung aus dem vorherigen Beispiel durch eine rein prozedurale Umsetzung, so dass kein Backtracking mehr benötigt wird. Da das Backtracking für die Namen entfällt reduziert sich der Suchraum für jeweils drei Instanzen auf 134.217.728, da die Faktoren c^8 und c^4 wegfallen.

Listing 2: Ausschnitt der geänderten ASSL-Prozedur

```
1  personNamesFemale := [Sequence{'Ada','Bel','Cam','Day'}];
2  personNamesMale   := [Sequence{'Ali','Bob','Cyd','Dan'}];
3  persons := CreateN(Person, [numPersons]);
4
5  numFemales := [(numPersons / 2).round()];
6
7  for i:Integer in [Sequence{1..numFemales}]
8  begin
9    [persons->at(numFemales + i)].name := [personNamesMale->at(i)];
10 end;
11
12 for i:Integer in [Sequence{numFemales + 1 .. numPersons}]
13 begin
14   [persons->at(i)].name := [personNamesMale->at(i)];
15 end;
```

2.2 USE-Model-Validator

Der Model-Validator stellt ein Framework zur Nutzung effizienter SAT-Techniken für eine leichtgewichtige und automatisierte Validierung und Verifikation von UML- und OCL-Modellen dar, der ebenso wie der USE-ASSL-Generator Zustandsräume zum Nachweis von Modelleigenschaften durchsucht [KHG11, KSG11]. Realisiert ist er als ein Plugin für das USE-System, sodass die Konfiguration der benutzerdefinierten Validierungs- und Verifikationsaufgaben mit Hilfe einer grafischen Oberfläche geschehen kann. Der Model-Validator stellt verschiedene Konfigurationsmöglichkeiten in Hinblick auf die zu suchen-

den Modellinstanzen zur Verfügung. Beispielsweise können die Anzahl der Objekte, Links und Attributwerte für jede Klasse, Assoziation und jedes Attribut vorgegeben, oder konkrete Domänen von möglichen Werten definiert werden. Insbesondere ist die Ausprägung von Konfigurationen durch die Angabe von partiellen Zuständen möglich, die der Model-Validator automatisiert vervollständigen kann.

Die Verbindung der UML- und OCL-Ebene mit der Ebene des SAT-Solving geschieht durch eine Transformation von UML-Klassendiagramm- und OCL-Konzepten in Konstrukte und Formeln der relationalen Logik. Diese werden wiederum durch den Constraint-Solver Kodkod in boolesche Formeln übersetzt, welche von SAT-Solvern gelöst und von Kodkod zurück in konkrete Belegungen auf der relationalen Ebene übersetzt werden können. Relationale Belegungen repräsentieren gefundene Zustände, die den Nutzern schließlich in Form von Objektdiagrammen präsentiert werden.

3 Reduzieren der Suchpfade

Wie bei der Beschreibung des Generators gezeigt, vergrößert sich der Suchraum einer ASSL-Prozedur ohne ein Eingreifen des Benutzers rasant. Es wurde gezeigt, wie der ASSL-Entwickler manuell den Suchraum einschränken kann. Mit Hilfe von verschiedenen Techniken, die Informationen aus dem Modell und der ASSL-Prozedur berücksichtigen können einige dieser Techniken automatisiert angewendet werden, so dass Fehlerquellen bei der Entwicklung von ASSL-Prozeduren verringert werden können.

In USE werden aktuell zwei Arten von Informationen bei der Suche berücksichtigt:

1. Die strukturellen Modelleinschränkungen und
2. die Modellabdeckungen der Constraints und der ASSL-Prozeduren.

Beide Arten versuchen die Anzahl der zu überprüfenden Zustände zu reduzieren, indem versucht wird, Teilbäume des Suchbaumes zu identifizieren die nicht betrachtet werden müssen. Der Unterschied zwischen beiden Ansätzen ist, dass die Berücksichtigung struktureller Einschränkungen vorausschauend funktioniert, da überhaupt nur zielführende Verzweigungen erzeugt werden, während die Berücksichtigung der Abdeckung rückwärts gerichtet funktioniert und die bisher konstruierte Modellinstanz dahingehend überprüft, ob überhaupt noch ein gültiger Zustand erreicht werden kann. Abbildung 2 zeigt den Suchbaum für die in Listing 1 gezeigte ASSL-Prozedur mit den in diesem Abschnitt beschriebenen Reduzierungen bis zum Try in der Zeile 18. Die entsprechenden Einzelheiten (hervorgehoben durch grau hinterlegte Knoten) werden in den folgenden Abschnitten im Detail eingeführt. Der Suchraum ist in der Abbildung komprimiert dargestellt. Die Teilbäume unterhalb einer Attributbelegung (z. B. `p1.name := 'Ada',...`) sind identisch und daher zusammengefasst. Dies ist erkennbar an den gestrichelten Pfeilen. Diese Darstellung legt nahe, dass der Suchraum in ASSL durch eine Symmetrieerkennung reduziert werden könnte. Da in ASSL aber verschiedene Befehle die aktuell gewählte Belegung zurückgeben und diese beliebig weiter verarbeitet werden können, müssen auch identische Teilbäume durchsucht werden.

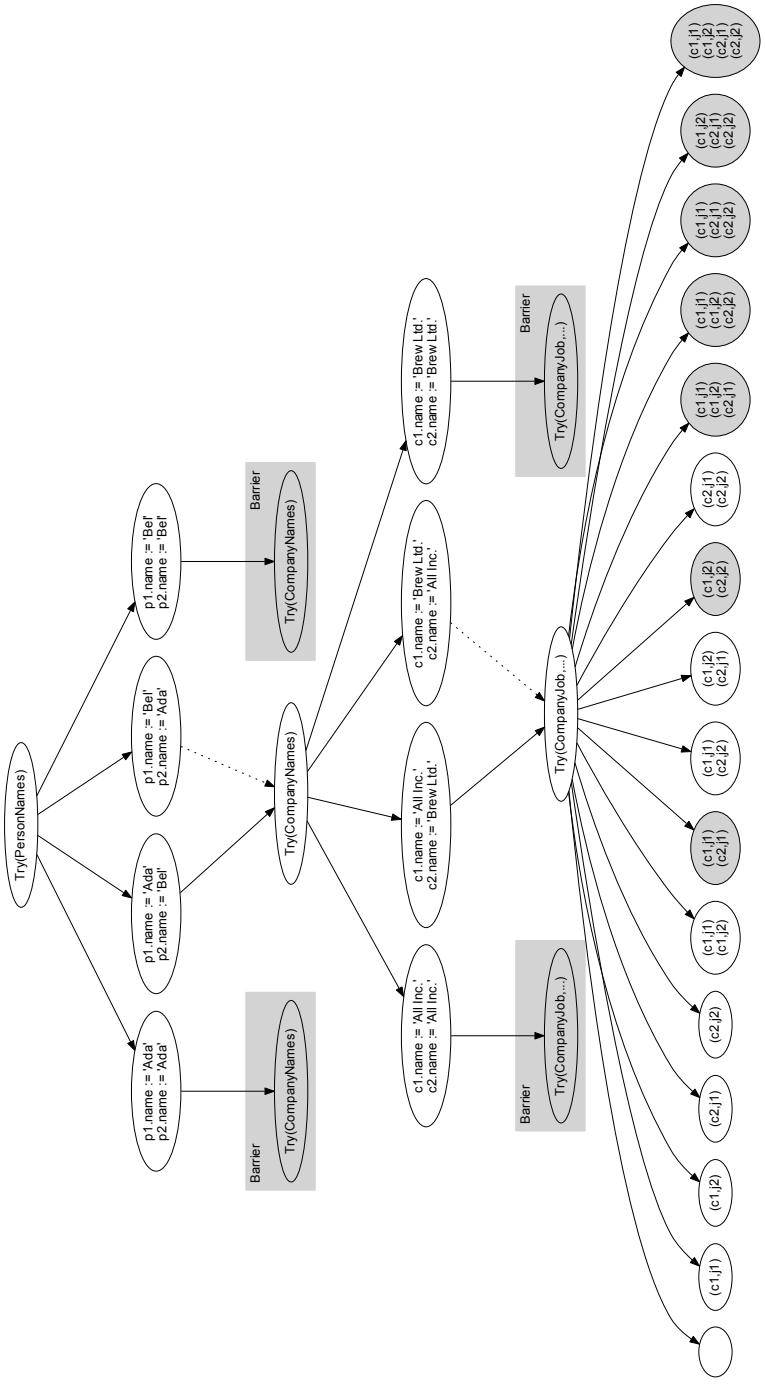


Abbildung 2: Reduzierung des Suchraums

3.1 Berücksichtigen von strukturellen Einschränkungen

Als strukturelle Einschränkungen in der UML bezeichnen wir in dieser Arbeit Einschränkungen, die im UML-Metamodell [OMG11b] definiert sind und die möglichen Instanzen eines Modells einschränken. Die wohl bekanntesten Einschränkungen sind die Multiplizitäten von Assoziationsenden. Weitere Beispiele sind Eigenschaften (Attribute oder Assoziationsenden), die mit `{unique}` markiert sind oder Kompositionsstrukturen.

Solche Einschränkungen können aufgrund der Monotonie von ASSL (es können nur Objekte erzeugt und nicht zerstört werden) bei der Suche "vorausschauend" verwendet werden. Dieses Vorgehen arbeitet nur mit dem aktuell aufgebauten Zustand und der gerade betrachteten spezifizierten Einschränkung. Wissen über nachfolgende Anweisungen oder über den Gültigkeitszustand des Modells ist nicht erforderlich. Am Beispiel der Multiplizitäten lässt sich dies veranschaulichen:

Wird ein Try auf einer Assoziation angewendet, die an mindestens einem Ende eine obere Schranke von 1 hat, lässt sich der Suchraum reduzieren[2]. Dies geschieht, indem nur Linkkombinationen erzeugt werden, die die entsprechend eingeschränkten Instanzen nur einmal enthalten. Im in Abbildung 2 gezeigten Suchbaum ist dies an den Blättern[3] zu erkennen. Alle 16 möglichen Linkkombinationen mit jeweils zwei möglichen Instanzen an den Assoziationenden der Assoziation `CompanyJob` sind auf der untersten Ebene des Baumes dargestellt. Diese müssten für eine $\star:\star$ Assoziation erzeugt werden. Da es sich bei der Assoziation `CompanyJob` aber um eine $1:\star$ Assoziation handelt, d. h. jede Instanz von `Job` darf nur mit genau einer Instanz von `Company` verbunden sein, können sämtliche Linkkombinationen, die eine Job-Instanz doppelt enthalten ignoriert werden. Die ignorierten Knoten sind in der Abbildung grau dargestellt. Für die Berechnung der Suchraumgröße ergibt dies eine Änderung der allgemeinen Formel für binäre $1:\star$ Assoziationen von $\mathcal{K} = 2^{n*m}$ zu $\mathcal{K} = (1+n)^m$ mit $n =$ *Anzahl der möglichen Instanzen am 0..1 Ende* und $m =$ *Anzahl der möglichen Instanzen am 0..* Ende*.

3.2 Berücksichtigen der Abdeckungen

Während das im vorherigen Abschnitt beschriebene Vorgehen die Menge der Verzweigungen, die generiert werden, reduziert, verhindert das in diesem Abschnitt beschriebene Vorgehen ein Fortsetzen der Suche in Teilbäumen des gesamten Suchbaums. Diese blockierenden Stellen bezeichnen wir als Barrieren. Die Position der einzelnen Barrieren kann dabei automatisch bestimmt werden.

Die Berechnung der Positionen für Barrieren berücksichtigt sowohl die Modellabdeckung von Invarianten, d. h. die von einer Invariante verwendeten Modellelemente, als auch die Modellabdeckung von ASSL-Prozeduren. Daher werden diese in den nächsten Abschnitten eingeführt, um anschließend die Positionierung der Barrieren beschreiben zu können.

[2] Entsprechend gilt dies auch für alle anderen oberen Schranken $\neq *$.

[3] Die in der Abbildung zu sehenden Blätter stellen nicht das jeweilige Ende eines Suchpfades dar, da aus Gründen der Übersicht nicht der gesamte Suchbaum gezeigt wird.

Die in Abbildung 1 gezeigte Invariante `PersonNamesAreUnique` fordert, dass alle Personen im Beispielmodell eindeutige Namen haben müssen. Das gezeigte Klassendiagramm zeigt weiterhin die Abdeckung der Invariante. Alle dunkel hinterlegten Klassen und Attribute werden von der Invariante benutzt. Alle weiß dargestellten Klassen und Attribute sind von ihr nicht berücksichtigt. Sie deckt also ausschließlich die Klasse `Person` und dessen Attribut `name` ab.

Für ein vereinfachtes Modell $\mathcal{M}$ mit den Mengen CLASS für die definierten Klassen, ATTRIB für die Attribute der Klassen, ASSOC für die Assoziationen und INV für die Menge der definierten Invarianten ist die Abdeckung $\mathcal{C}_{inv}(i)$ einer Invariante $i \in$ INV ein Tripel:

$$\mathcal{C}(inv) = (cls, att, ass), \text{ mit } cls \subseteq \text{CLASS}, att \subseteq \text{ATTR}, ass \subseteq \text{ASSOC}$$

Die Abdeckung der Invariante `PersonNamesAreUnique` ist demnach:

$$\mathcal{C}(\text{PersonNamesAreUnique}) = (\{\text{Person}\}, \{\text{Person::name}\}, \{\})$$

Die Invariante `BossWorkerSameEmployer` des Beispielmodells, die fordert, dass alle Stellen (`Job`) einer Stellen-Hierarchie zur gleichen Firma (`Company`) gehören besitzt folgende Abdeckung:

$$\mathcal{C}(\text{BossWorkerSameEmployer}) = (\{\text{Company}, \text{Job}\}, \{\}, \{\text{CompanyJob}\})$$

Die Abdeckung kann statisch durch eine Analyse der Ausdrücke berechnet werden. Beim Durchlaufen des abstrakten Syntaxbaums einer Invariante werden für die nachfolgend aufgeführten Ausdrucksarten die beschriebenen Modellelemente in die Abdeckung aufgenommen. Eine detaillierte Auseinandersetzung mit der Abdeckungsberechnung findet sich in [CT09]. Im Gegensatz zur Berechnung der allgemeinen Abdeckung ist die hier beschriebene Berechnung für die Verwendung im Zusammenspiel mit ASSL bereits optimiert. Dadurch, dass in ASSL ein Systemzustand nur vergrößert werden kann, müssen nur die Unterklassen berücksichtigt werden. Würde ASSL es erlauben, dass auch Instanzen zerstört werden könnten, dann könnte eine syntaktisch auf einer Oberklassse basierender Ausdruck Instanzen einer Unterklasse löschen und so die betrachtete Invariante beeinflussen. Für diesen Fall müssten auch die Oberklassen in die Abdeckung aufgenommen werden.

- Bei Attributzugriffen werden das Attribut und die definierende Klasse sowie ihre Unterklassen aus der Klassenhierarchie aufgenommen.
- Bei Navigationsausdrücken wird die beteiligte Assoziation und die beteiligten Klassen mit deren Unterklassen aufgenommen.
- Bei der in OCL integrierten Operation `allInstances()` wird die Klasse deren Instanzen ausgewählt werden, sowie deren Unterklassen aufgenommen.

Mit der beschriebenen Abdeckung einer Invariante kann festgelegt werden, wann eine Invariante innerhalb einer Suche als *stabil* angesehen werden kann, d. h. ihr Zustand wird sich im Laufe der Suche nicht mehr ändern. Für die Invariante PersonNamesAreUnique

ist dies der Fall, wenn weder neue Personeninstanzen erzeugt werden noch Zuweisungen an das Attribut `name` erfolgen. Die Instanziierung einer neuen Person kann dazu führen, dass zwei Personennamen den undefinierten Wert besitzen und die Invariante dadurch nicht mehr erfüllt ist. Eine nachfolgende Zuweisung kann beide Zustandsänderungen der Invariante bewirken.

Die durch eine ASSL-Prozedur veränderten Modellelemente können ähnlich zu den Invarianten über die Berechnung der Abdeckung bestimmt werden. So ändert z. B. der Zuweisungsoperator `object.attribute := stmt` den Attributwert einer einzelnen Instanz und deckt damit die Klasse des Objektes und das Attribut ab. Für einen ASSL-Ausdruck (`stmt`) lässt sich die Abdeckung $\mathcal{C}_{\mathcal{A}}(stmt)$ auch als Tripel von Modellelementen darstellen:

$$\mathcal{C}_{\mathcal{A}}(stmt) = (cls, att, ass), \text{ mit } cls \subseteq \textsc{Class}, att \subseteq \textsc{Attr}, ass \subseteq \textsc{Assoc}$$

Für eine Liste von ASSL-Statements $(stmt_1, \ldots, stmt_n),\ 1 < n$ lässt sich die Gesamtabdeckung durch die Vereinigung der einzelnen Abdeckungen berechnen.

$$\mathcal{C}_{\mathcal{A}}(stmt_1, \ldots, stmt_n) = \bigcup_{i=1}^{|n|} \mathcal{C}_{\mathcal{A}}(stmt_i)$$

Mit Hilfe der eingeführten Abdeckungen kann in einer ASSL-Prozedur nur durch syntaktische Analyse einer Invariante $\mathcal{I} \in \textsc{Inv}$ und der ASSL-Prozedur

$$proc = (stmt_1, \ldots, stmt_j, \ldots, stmt_n), 1 < j < n$$

eine Barriere $\mathcal{B}_{\mathcal{I}}$ für $\mathcal{I}$ vor dem Ausdruck $stmt_j$ eingeführt werden, wenn $\mathcal{C}_{inv}(\mathcal{I})$ und $\mathcal{C}_{\mathcal{A}}(stmt_j, \ldots, stmt_n)$ komponentenweise disjunkt sind. Die Barriere $\mathcal{B}_{\mathcal{I}}$ prüft vor dem Ausführen der ASSL-Anweisung $stmt_j$ die Invariante $\mathcal{I}$ und führt diese nur aus, falls die Invariante gültig ist. Andernfalls wird die Suche am vorherigen Backtrackingpunkt fortgesetzt.

Der in Listing 3 gezeigte ASSL-Prozedurrumpf der generischen Prozedur aus Listing 1 ist um Kommentare erweitert, die vor jeder ASSL-Anweisung die restliche Abdeckung der ASSL-Prozedur ($\mathcal{C}_{\mathcal{A}}$) sowie die implizit erzeugten Barrieren für die einzelnen Invarianten aufzeigen. Vor den Barrieren ist die Abdeckung der entsprechenden Invariante ($\mathcal{C}_{inv}$) angegeben. Die Auswirkungen der Barrieren auf den Suchraum sind in Abbildung 2 anhand von grauen Rechtecken dargestellt. Die eigentlich folgenden Teilbäume (identisch zu dem gezeigten Teilbaum in der Mitte) werden nicht evaluiert, da die entsprechenden Invarianten nicht mehr erfüllt werden können.

Die Reduzierung des Suchraums hängt von den jeweiligen Invarianten ab und kann allgemein nicht angegeben werden. Bei der hier dargestellten Prozedur ist dies aufgrund der einfachen Invarianten allerdings möglich, da sich die Werteauswahl bei den Namen von einer Permutation mit Wiederholung auf eine Permutation ohne Wiederholung (eindeutige Namen) reduziert, was zu den Faktoren $c!$ und $p!$ anstatt c^c und p^p führt.

Listing 3: Erweiterte ASSL-Prozedur

```
1  begin
2    /* C_A = ({Company,Person,Job}, {CompanyJob, PersonJob, BossWorker}, {Person::name, Company::name}) */
3    personNames := [Sequence{
4       'Ada','Bel','Cam','Day'
5       'Ali','Bob','Cyd','Dan'}->subSequence(1, numPersons)];
6
7    /* C_A = ({Company,Person,Job}, {CompanyJob, PersonJob, BossWorker}, {Person::name, Company::name}) */
8    companyNames := [Sequence{'All␣Inc.', 'Brew␣Ltd.',
9    'Cup␣Coop.', 'Duff␣Ltd.'}->subSequence(1, numCompanies)];
10
11   /* C_A = ({Company,Person,Job}, {CompanyJob, PersonJob, BossWorker}, {Person::name, Company::name}) */
12   companies := CreateN(Company, [numCompanies]);
13
14   /* C_A = ({Person,Job}, {CompanyJob, PersonJob, BossWorker}, {Person::name, Company::name}) */
15   persons := CreateN(Person, [numPersons]);
16
17   /* C_A = ({Job}, {CompanyJob, PersonJob, BossWorker}, {Person::name, Company::name}) */
18   jobs := CreateN(Job, [numJobs]);
19
20   /* C_A = ({}, {CompanyJob, PersonJob, BossWorker}, {Person::name, Company::name}) */
21   Try([Person.allInstances()], name, [personNames]);
22   /* C_inv = ({Person}, {}, {Person::name}) */
23   /* Barrier(Person::personNamesAreUnique); */
24
25   /* C_A = ({}, {CompanyJob, PersonJob, BossWorker}, {Company::name}) */
26   Try([Company.allInstances()], name, [companyNames]);
27   /* C_inv = ({Company}, {}, {Company::name}) */
28   /* Barrier(Company::companyNamesAreUnique); */
29
30   /* C_A = ({}, {CompanyJob, PersonJob, BossWorker}, {}) */
31   Try(CompanyJob, [companies], [jobs]);
32
33   /* C_A = ({}, {PersonJob, BossWorker}, {}) */
34   Try(BossWorker, [jobs], [jobs]);
35   /* C_inv = ({Job, Company}, {CompanyJob}, {}) */
36   /* Barrier(Job::bossWorkerSameEmployer); */
37
38   /* C_A = ({}, {PersonJob}, {}) */
39   Try(PersonJob, [persons], [jobs]);
40
41   /* C_A = ({}, {}, {}) */
42 end;
```

3.3 Ergebnisse

Die Auswirkungen der in den vorherigen Abschnitten gezeigten Optimierungen lassen sich anhand des Beispiels aus Listing 3 zeigen. Ohne die gezeigten Optimierungen (ASSL) und für die verschiedenen Kombinationen der Optimierungen (ASSL$^+$ steht hierbei für die Berücksichtigung von strukturellen Einschränkungen) ergeben sich folgende Berechnungsformeln für die Suchraumgröße:

$$
\begin{array}{lrrrrr}
S_{\text{ASSL}} & = c^c * & p^p * & 2^{c*j} * & 2^{j*j} * & 2^{p*j} \\
S_{\text{ASSL + Barrieren}} & = c! * & p! * & 2^{c*j} * & 2^{j*j} * & 2^{p*j} \\
S_{\text{ASSL}^+} & = c^c * & p^p * & (c+1)^j * & (j+1)^j * & (p+1)^j \\
S_{\text{ASSL}^+ \text{ + Barrieren}} & = c! * & p! * & (c+1)^j * & (j+1)^j * & (p+1)^j
\end{array}
$$

Tabelle 1 zeigt für eine unterschiedliche Anzahl von Instanzen den jeweiligen Suchraum für die verschiedenen Kombinationen der Optimierungen. Da die benötigte Zeit stark von den definierten Invarianten und der Geschwindigkeit des verwendeten Prozessors abhängt wird diese nicht aufgeführt. Ein Richtwert für das verwendete Beispiel sind ca. 30.000 Zustände pro Sekunde auf einem 2,5 Ghz Dual-Core Notebook.

c	j	p	ASSL	ASSL + B	ASSL$^+$	ASSL$^+$ + B
2	2	2	65.536	16.384	11.664	2.916
2	2	3	1.769.472	196.608	139.968	15.552
2	3	2	33.554.432	8.388.608	746.496	186.624
3	3	3	97.844.723.712	4.831.838.208	191.102.976	9.437.184

Tabelle 1: Suchraumgröße der verschiedenen Optimierungen

3.4 Kombination mit dem Model-Validator

Wie die Ergebnisse im vorherigen Abschnitt gezeigt haben, ermöglichen die hier vorgestellten Optimierungen eine Suche nach größeren Modellinstanzen, da der Suchraum langsamer anwächst. Das Wachstum ist aber weiterhin exponentiell. Aufbauend auf den hier und in [KHG11] vorgestellten Arbeiten zum Thema Abdeckung und relationaler Logik untersuchen wir derzeit eine Kombination aus beiden Ansätzen. Ziel dabei ist es, an bestimmten Stellen der ASSL-Prozeduren den Model-Validator Teilmodelle finden zu lassen, die die Anzahl der zu prüfenden Zustände reduziert.

Anhand der Invarianten- und ASSL-Abdeckungen können diese Stellen identifiziert werden. Da die Übersetzungszeiten des Model-Validators (von UML/OCL hin zur relationalen Logik und anschließend nach SAT) ein entscheidender Faktor bei der Effizienz des Ansatzes sind muss versucht werden die Anzahl der Übersetzungen zu reduzieren. So können aufeinander folgende Try-Anweisungen, die nicht durch prozedurale Kontrollstrukturen (if, for) unterbrochen werden zu einem Übersetzungslauf zusammengefasst werden. Weiterhin kann nach der ersten Instanzsuche im Model-Validator auf ein erneutes Übersetzen in die relationale Logik und SAT verzichtet werden, da die SAT-Übersetzung um das gefundene Modell dahingehend erweitert wird, dass dieses als Lösung ausgeschlossen wird.

Gegenüber der alleinigen Verwendung des Model-Validators bietet ein hybrider Ansatz mehrere Vorteile. Da auch die Übersetzung in die relationale Logik ab einer bestimmten Modellgröße an Grenzen stößt, bietet sich ein Aufteilen der Modellsuche in mehrere durch ASSL definierte Teilschritte, die jeweils in sich gültige Teile des Modells erzeugen, an. Diese Aufteilung kann die Menge der verarbeitbaren Modelle erweitern. Diese Teilmodelle können, falls vorhanden, mit Hilfe der Abdeckungen automatisch gefunden werden, so dass im Idealfall nicht alle Anweisungen einer ASSL-Prozedur zu einer Multiplikation bei der Suchraumgröße führen, sondern zu einer Addition. Weiterhin können bestimmte Arten von Invarianten, z. B. rekursiv definierte, nicht in die relationale Logik übersetzt werden. Enthält ein Modell nicht übersetzbare Invarianten kann der Model-Validator diese nicht verarbeiten. Bei einer Integration des Model-Validators in ASSL können diese Inva-

rianten vom Model-Validator ignoriert und durch die verfügbaren Mechanismen in USE ausgewertet werden. Sollte die gefundene Instanz die ignorierten Invarianten verletzen, dann wird die nächste vom Model-Validator gefundende Instanz geprüft, bis dieser keine gültigen Instanzen mehr findet oder eine Modellinstanz allen Invarianten genügt.

4 Verwandte Arbeiten

Es gibt eine Reihe verwandter Arbeiten im Bereich der Instanzgenerierung für UML-Modelle mit Constraints. [ABGR10] übersetzt die Modelle in Alloy und damit letztlich ebenfalls in Kodkod und kann mit dem in USE integrierten Ansatz [KHG11] verglichen werden, unterstützt aber aufgrund des Zwischenschritts einige OCL-Konstrukte wie beispielsweise verschachtelte Kollektionen nicht. [CCR07] übersetzt das Modell in Constraint Programming (Constraint Propagierung und Backtracking), allerdings ohne Einflussmöglichkeit durch den Anwender auf das Backtracking. [AIAB11] beschreibt ein (unvollständiges) heuristisches Verfahren auf Grundlage genetischer Algorithmen um Testfälle für große (industrielle) Modelle zu finden. Die genannten Ansätze sind als Black-Box-Verfahren anzusehen. Eine Einordnung der Grenzen der Skalierbarkeit dieser Ansätze findet sich in [BGF+10].

Auf Seiten der White-Box-Ansätze gibt eine Reihe von Sprachen, die zum Aufbau von Modellinstanzen mit imperativen Programmen geeignet sind, z. B. EOL [KPP06], Kermeta [MFJ05] oder ImperativeOCL. ASSL ist hierbei der einzige Ansatz, der über eine reine imperative Programmierung für OCL-annotierte Modelle hinaus einen Backtrackingmechanismus bietet. Die hier vorgeschlagene Berücksichtigung der Modellabdeckung von Constraints zur Optimierung des Backtrackings ähnelt dem in [CT09] vorgestellten Verfahren zur inkrementellen Auswertung von Constraints für UML-Modelle.

[KFdB+05, BW08, BHS07, CEdD09] adressieren das werkzeuggestützte logische Schließen über OCL-annotierte UML-Modelle durch Übersetzung in formale Theorien bzw. Prädikatenlogik.

5 Fazit und Ausblick

In dieser Arbeit wurden Verfahren vorgestellt, die der imperativen Modellinstanzsuche mit ASSL den zu durchsuchenden Zustandsraum erheblich verkleinern können. Dabei wurde gezeigt, wie strukturelle Einschränkungen des Modells berücksichtigt werden können.

Diese Art der Optimierung benötigt im Allgemeinen wenig Wissen über die Modellstruktur und den weiteren Suchprozeß. Weiterhin verringert die Berücksichtigung von strukturellen Einschränkungen die Anzahl der erzeugten Knoten des Suchbaumes, da sie vorausschauend arbeiten kann.

Die zweite gezeigte Technik zur Reduktion der Suchraumgröße arbeitet mit mehr Wissen über das Modell und berücksichtigt die Modellabdeckung der Invarianten und der Such-

prozedur. Dabei können Positionen innerhalb des Suchbaums bestimmt werden, an denen bestimmte Invarianten eines Modell bereits erfüllt sein müssen, damit der Suchvorgang insgesamt erfolgreich sein kann. Durch diese sogenannten Barrieren müssen bestimmte Teilbäume bei der Suche nicht berücksichtigt werden, da sie nicht zielführend sind.

Weiterhin wurde ein kombinierter Ansatz für die Instanzsuche beschrieben, der ASSL und den Model-Validator verwendet. Durch die Kombination beider Verfahren sollen die jeweiligen Nachteile möglichst ausgeglichen werden, so dass komplexere Modelle und größere Modellinstanzen gesucht werden können. Eine geplante detaillierte empirische Analyse dieses Ansatzes soll die Wirksamkeit dieser Kombination überprüfen und ist als zukünftige Arbeit vorgesehen. Weitere zukünftige Arbeiten umfassen die Aufnahme weiterer struktureller Einschränkungen in den Suchprozeß, so dass dieser bereits ohne den Model-Finder effizienter wird. Die Berücksichtigung der Abdeckung soll in Zukunft dahingehend erweitert werden, dass Invarianten, die sich nur auf einzelne Instanzen oder Teilmengen einer Instanzenmenge auswirken [CT09], berücksichtigt werden.

Literatur

[ABGR10] Kyriakos Anastasakis, Behzad Bordbar, Geri Georg und Indrakshi Ray. On challenges of model transformation from UML to Alloy. *Software and System Modeling*, 2010.

[AIAB11] S. Ali, M.Z. Iqbal, A. Arcuri und L. Briand. A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In *11th International Conference on Quality Software (QSIC)*, Seiten 41 –50, Juli 2011.

[BGF+10] Benoit Baudry, Sudipto Ghosh, Franck Fleurey, Robert France, Yves Le Traon und Jean-Marie Mottu. Barriers to systematic model transformation testing. *Commun. ACM*, 53:139–143, Juni 2010.

[BHS07] Bernhard Beckert, Reiner Hähnle und Peter Schmitt, Hrsg. *Verification of Object-Oriented Software. The KeY Approach*, LNCS 4334. Springer, 2007.

[BW08] Achim D. Brucker und Burkhart Wolff. HOL-OCL: A Formal Proof Environment for UML/OCL. In *Fundamental Approaches to Software Engineering, 11th International Conference, FASE 2008, Proceedings*, LNCS 4961. Springer, 2008.

[CCR07] Jordi Cabot, Robert Clarisó und Daniel Riera. UMLtoCSP: a tool for the formal verification of UML/OCL models using constraint programming. In *Automated Software Engineering, 22nd IEEE/ACM International Conference, ASE 2007*. ACM, 2007.

[CEdD09] Manuel Clavel, Marina Egea und Miguel Angel García de Dios. Checking Unsatisfiability for OCL Constraints. *ECEASST*, 24, 2009.

[CT09] Jordi Cabot und Ernest Teniente. Incremental integrity checking of UML/OCL conceptual schemas. *Journal of Systems and Software*, 82(9):1459 – 1478, 2009.

[DSM08] Marcelo Paternostro Dave Steinberg, Frank Budinsky und Ed Merks. *EMF: Eclipse Modeling Framework*. Addison Wesley Professional, 2nd. Auflage, 2008.

[GBR05] Martin Gogolla, Jörn Bohling und Mark Richters. Validating UML and OCL Models in USE by Automatic Snapshot Generation. *Journal on Software and System Modeling*, 4(4):386–398, 2005.

[GBR07] Martin Gogolla, Fabian Büttner und Mark Richters. USE: A UML-Based Specification Environment for Validating UML and OCL. *Science of Computer Programming*, 69:27–34, 2007.

[HG10] Lars Hamann und Martin Gogolla. Improving Model Quality by Validating Constraints with Model Unit Tests. In *2010 Workshop on Model-Driven Engineering, Verification, and Validation (MoDeVVa)*, Seiten 49 –54, Oktober 2010.

[HGK] Lars Hamann, Martin Gogolla und Mirco Kuhlmann. Zur Validierung von Kompositionsstrukturen in UML mit USE. In *Modellierung 2010*, LNI 161, Seiten 169–177.

[JB06] Frédéric Jouault und Jean Bézivin. KM3: A DSL for Metamodel Specification. In Roberto Gorrieri und Heike Wehrheim, Hrsg., *Formal Methods for Open Object-Based Distributed Systems*, LNCS 4037, Seiten 171–185. Springer, 2006.

[KFdB+05] Marcel Kyas, Harald Fecher, Frank S. de Boer, Joost Jacob, Jozef Hooman, Mark van der Zwaag, Tamarah Arons und Hillel Kugler. Formalizing UML Models and OCL Constraints in PVS. *Electr. Notes Theor. Comput. Sci.*, 115:39–47, 2005.

[KHG11] Mirco Kuhlmann, Lars Hamann und Martin Gogolla. Extensive Validation of OCL Models by Integrating SAT Solving into USE. In *TOOLS 2011*, LNCS 6705, Seiten 290–306. Springer, 2011.

[KPP06] Dimitrios S. Kolovos, Richard F. Paige und Fiona Polack. The Epsilon Object Language (EOL). In *Model Driven Architecture - Foundations and Applications, ECMDA-FA*, LNCS 4066, Seiten 128–142. Springer, 2006.

[KSG11] Mirco Kuhlmann, Karsten Sohr und Martin Gogolla. Comprehensive Two-level Analysis of Static and Dynamic RBAC Constraints with UML and OCL. In *Fifth International Conference on Secure Software Integration and Reliability Improvement, SSIRI 2011*. IEEE Computer Society, 2011.

[MFJ05] Pierre-Alain Muller, Franck Fleurey und Jean-Marc Jézéquel. Weaving Executability into Object-Oriented Meta-languages. In *Model Driven Engineering Languages and Systems, 8th International Conference*, LNCS 3713, Seiten 264–278. Springer, 2005.

[OMG10] *OMG Object Constraint Language (OCL) Specification Version 2.2*, Februar 2010. http://www.omg.org/spec/OCL/2.2.

[OMG11a] *OMG Meta Object Facility (MOF) Core Specification 2.4.1*, August 2011. http://www.omg.org/spec/MOF/2.4.1/PDF/.

[OMG11b] *OMG Unified Modeling Language (UML), Superstructure*, August 2011. http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/.

[USE] A UML-based Specification Environment. http://sourceforge.net/projects/useocl/.