Evaluierung von Möglichkeiten zur Implementierung von Semantischen Analysen für Domänenspezifische Sprachen

Christian Berg und Wolf Zimmermann
christian.berg@informatik.uni-halle.de wolf.zimmermann@informatik.uni-halle.de

Institut für Informatik
Martin-Luther-Universität
Halle-Wittenberg

Abstract: Domänen-spezifische Sprachen dienen der Spezifikation von Modellen einer Anwendungsdomäne. Neben der Syntaxprüfung ist es zunehmend erforderlich auch semantische Prüfungen der Modelle durchzuführen. In den klassischen Werkzeugen zur Entwicklung Domänen-spezifischer Sprachen aus dem Gebiet der Modell-basierten Entwicklung wird häufig OCL zur Spezifikation solcher Bedingungen im Meta-Modell eingesetzt. Dem gegenüber stehen Technologien aus dem Gebiet des Übersetzerbaus wie beispielsweise attributierte Grammatiken, die ebenfalls die Spezifikation semantischer Bedingungen ermöglichen. In diesem Beitrag stellen wir diese Technologien gegenüber und untersuchen deren Performanz der syntaktischen und semantischen Analyse und die Qualität von Fehlermeldungen.

1 Einleitung

Domänen-spezifische Sprachen[1] (DSLs) erleichtern die Entwicklung von Produkten oder Produktteilen. Mit steigender Komplexität der Produkte steigt auch die Komplexität der Domänen-spezifischen Beschreibungen und somit auch die Relevanz semantischer Prüfungen. Zusammen mit unseren Projektpartnern aus der Pumpendomäne entwickeln wir eine Domänen-spezifische Sprache, u.a. zur Anforderungsmodellierung. Pumpen selbst sind einer der größten Energieverbraucher in industriellen Prozessen [OL07]. Um ein Produkt, wie auch eine Pumpe oder Pumpensteuerung, bedarfsgerecht zu entwickeln, bedarf es Anforderungen an dieses Produkt[Par10, NE00]. Um eine Anforderung umsetzen zu können, kann es notwendig sein weitere, untergeordnete Anforderungen umzusetzen. So muss, damit die Anforderung „Druck und Durchfluss für eine industrielle Anlage“ umgesetzt werden kann, mindestens auch Sensorik und Aktorik sowie die Leistungskurve der Pumpe bekannt und ggf. umgesetzt sein. In der Literatur wird diese Abhängigkeitsstruktur auch als Anforderungshierarchie bezeichnet[KS00]. Ist eine Anforderung von einer anderen abhängig, wie „Druck und Durchfluss“ von der Sensorik, so kann diese nicht erfüllt werden, bevor diese Abhängigkeiten nicht erfüllt sind: Sensorik und Aktorik sind also zwingend notwendig bevor „Druck und Durchfluss für eine industrielle Anlage“ erfüllt sein kann. In der Domäne der Pumpenentwicklung hat diese Hierarchie, aufgrund der Entwicklung auf Basis von Kundenanforderungen, eine Wurzel: zum Beispiel die obige Anforderung „Druck und Durchfluss für eine industrielle Anlage“ zu erzeugen. Um diese einfache Kundenanforderung umzusetzen bedarf es vieler weiterer Anforderungen, deren Anzahl im industriellen Kontext durchaus 3-4 stellig sein kann. Die Umsetzung in ein Domänen-spezifisches Format ist bei unserem Projektpartner derzeit in Umsetzung, aber die Experimente sind für diesen Teil repräsentativ.

Um die Probleme der Komplexität der Produktentwicklung zu verringern werden immer wieder Domänen-spezifische Sprachen, wie Yakindu Requirements[wsY] zur Anforderungsmodellierung oder reqT.org[Reg13] zur Featuremodellierung, entwickelt. Allen solchen DSLs ist gemein, dass Elemente – Bezeichner mit Domäneninformationen – definiert und verwendet werden; so auch bei anderen Arten von Sprachen, wie der Electronic Device Description Language (EDDL)[wsE] oder dot[EGK+02]. Auch bei Typhierarchien werden Elemente (Typen) mittels Bezeichnern definiert und in der Hierarchie verwendet[BKWA11]; bei Aktionsbeschreibungen

[1] engl. domain-specific language

(z.B. Makefiles[2]) werden Elemente („Aktionspunkte“) definiert und Elemente benutzt, die zur Durchführung der Aktion verwendet werden. Nutzen und Aufwand der Entwicklung von Domänen-spezifischen Sprachen ist gut verstanden[MHS05]. DSLs und dazugehörige Editoren lassen sich aus Spezifikationen generieren. Zur Prüfung der statischen Semantik von DSLs, wie zum Beispiel, dass alle verwendeten Bezeichner definiert werden, kommen im Übersetzerbau oft Attributgrammatiken[ALSU06, Knu68], im modell-basierten Umfeld häufig die Object Constraint Language (OCL), zum Einsatz[KER99, Pan11]. Beiden gemein ist, dass zu einem Kontext Vor- und Nachbedingungen bzw. Invarianten spezifiziert werden können; diese können als Spezifikation der Semantik verstanden werden.

Basierend auf unseren Erfahrungen aus der Anforderungsmodellierung in der Pumpendomäne untersuchen wir anhand eines fiktiven Beispiels die Beziehung von Definition und Verwendung von Elementen (hier: Anforderungen). Diese Beziehung muss azyklisch sein, damit die Anforderungen eine Hierarchie statt eines beliebigen Graph bilden. Erst durch die Hierarchie können den Anforderungen Personen und Zeiten zugeordnet werden. Bei der Anforderungsmodellierung ebenfalls wichtig ist, dass Anforderungen nicht ausserhalb der Hierarchie liegen, denn dies deutet darauf hin, dass eine Anforderung nicht analysiert wurde oder gar nicht benötigt wird.

Wir vergleichen die Prüfung eben genannter Problematik – Prüfung auf Azyklizität, Finden nicht-benötigter Anforderungen und Ermitteln nicht vorhandener Anforderungen – am Beispiel der Abhängigkeitsstruktur von Anforderungen in der Domäne Pumpenentwicklung anhand zweier Implementierungen. Wir stellen die Methoden des Übersetzerbaus der Verwendung von OCL in Modell-basierten Werkzeugen gegenüber. Unser Hauptaugenmerk liegt auf der Fehlergenauigkeit und der Geschwindigkeit dieser Analysen. Ist eine Analyse langsam, kann dies eine Produktivitätssenkung aufgrund von Ablenkung, wie in [DL85] beschrieben, verursachen. Soll dies bei einer langsamen Analyse vermieden werden, so muss diese „über Nacht“ laufen, benötigt dann aber am nächsten Morgen eine sehr detaillierte Fehlerausgabe. Aufgrund der langen Laufzeit muss diese Fehlerausgabe dann auch Fehler erkennen, die üblicherweise erst nach der Behebung von Fehlern gefunden werden andernfalls würde die Produktivität weiter sinken, da eine erneute Analyse (über Nacht) notwendig wäre.

Die Vorstellung des laufenden Beispiels und damit einhergehender Kriterien und Abstraktionen ist Gegenstand von Abschnitt 2. Abschnitt 3 gibt eine kurze Einführung in die zum Verständnis der nachfolgenden Abschnitte notwendigen Definitionen. Die Umsetzung mittels Werkzeugen des Übersetzerbaus und mittels OCL im modell-basierten Umfeld beschreiben wir in Abschnitt 4. Wir vergleichen die Laufzeiten und die Fehlerausgaben der beiden Implementierungen in Abschnitt 5. In Abschnitt 6 diskutieren wird die Möglichkeiten im Modell-basierten Umfeld die statische Semantik von Sprachen zu spezifizieren und zeigen, dass bisher noch keine Gegenüberstellung der Möglichkeiten mit den Methoden des Übersetzerbaus erfolgt ist. Eine Zusammenfassung und offene Fragen finden sich in Abschnitt 7.

2 Anforderungen und Abhängigkeiten

Wie in Abschnitt 1 ausgeführt, basieren unsere Beispiele auf der Domäne *„Pumpenentwicklung“* und dabei auf dem Teilgebiet der Anforderungsmodellierung. Folgendes, laufendes Beispiel stellt fiktive Anforderungen dar[3].

Anforderungsnummer	**Beschreibung**	**benötigt:**
1	Es sollen Druck und Durchfluss für Anlage A erzeugt werden	2, 3, 4, 5, 8
2	Es muss der Druck ermittelt werden	4
3	Es muss der Druck erzeugt werden	4
4	Eine Anbindung an die Anlage muss existieren	2, 3
5	Es wird ein Antrieb benötigt	-
6	Die Temperatur muss gemessen werden	-
7	Es soll ein GPS-Navigationssystem zur Verfügung stehen	

Beispiel 1 – Die Anforderung mit der Nummer 1 ist die sog. Wurzelanforderung – die Anforderung, die umgesetzt werden soll. Anforderung 8 wird benötigt, aber ist undefiniert.

[2] z.B. via GNU make, http://www.gnu.org/software/make

[3] Echte Beispiele eignen sich aufgrund aktueller Geschäftsrelevanz und Komplexität nicht für eine Vorstellung im wissenschaftlichen Rahmen.

In dem Beispiel der Druckerzeugung für Anlage A sind wesentliche Fehler enthalten, die es einem Projektmanager unmöglich machen diese Anforderungen so zu verteilen, dass diese darin münden, dass die Wurzelanforderung umgesetzt ist: Druckermittelung (4), Druckerzeugung (3) und Anlagenanbindung[4] (2) sind zyklisch, von den Anforderungen 6 und 7 hängt keine Anforderung ab, es ist also egal, ob diese umgesetzt werden oder nicht. Wenn eine oder beide der Anforderungen 6 oder 7 umgesetzt werden, so wird Arbeitszeit verschwendet. Die Anforderung 8 wird zwar benötigt, ist aber nirgendwo spezifiziert: ein Projektmanager kann gar nicht festlegen wer diese Anforderung umsetzen soll. Die Anforderung mit der Nummer 7 ist keine Anforderung um ein Pumpe zu erstellen, sondern stammt aus einer anderen Domäne.

Abbildung 1 stellt die Abhängigkeiten aus Beispiel 1 dar, der undefinierte Knoten 8 ist explizit markiert.

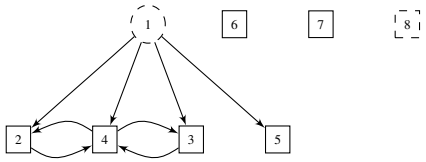


Abbildung 1 – Abhängigkeiten von Beispiel 1
Wurzel = gestrichelt, rund; *benutzt aber undefiniert* = gestrichelt, eckig; *definiert* = durchgezogen, eckig

In den folgenden Abschnitten zeigen wir, wie die obigen Ausführungen in statischen Analysen münden und wie diese geprüft werden können. Beispiel 1 dient hierbei der Veranschaulichung der Ergebnisse und wird in weiteren Abschnitten verwendet um die Implementierungen geeignet gegenüber zu stellen.

3 Grundlagen und Begriffe

In diesem Abschnitt werden die für diesen Beitrag notwendigen Grundlagen und Begriffe eingeführt.

Zur Spezifikation der Implementierung Domänen-spezifischer Sprachen werden im Modell-basierten Umfeld Meta-Modelle genutzt; im Übersetzerbau werden die abstrakte Syntax, sowie als Darstellungsform zum Programmierer die konkrete Syntax verwendet.

Definition 1. *Eine kontextfreie Grammatik ist ein Tupel* $\mathsf{G} \triangleq (T, N, P, Z)$*, wobei*

- N *ein endliches Alphabet der Nicht-Terminalsymbole,*
- T *ein endliches Alphabet der Terminalsymbole ist,*
- $P \subseteq N^* \times (T \cup N)^*$ *die Menge der Produktionsregeln und*
- $Z \in N$ *ein ausgezeichnetes Startsymbol ist.*

Wenn die durch G definierte Sprache $L(\mathsf{G})$ die gültige Folge von Grundsymbolen angibt, dann bezeichnen wir G als konkrete Syntax; definiert die Grammatik Baumaufbaukonstruktoren, so bezeichnen wir G als abstrakte Syntax. Sei $p \in P$, dann nutzen wir zur Darstellung der Regel $\rightarrow$ für die konkrete Syntax und `::=` für die abstrakte Syntax. Seien $X, Y \in T \cup N$ Symbole der rechten Seite einer Regel und $A \in N$, dann ist $A \rightarrow X|Y$ die Kurzschreibweise für $A \rightarrow X$, $A \rightarrow Y$; $A \rightarrow X^+$ Kurzschreibweise für $A \rightarrow Xs\ X$, $Xs \rightarrow X|Xs\ X$ und X^* die Kurzschreibweise für $A \rightarrow Xs|\varepsilon$, wobei ε für das leere Wort steht; analoges gilt für die abstrakte Syntax. Nichtterminale $nt \in N$ werden mittels $\langle \mathtt{nt} \rangle$ markiert.

Im Modell-basierten Umfeld ist das Vokabular ein Anderes, doch den Prinzipien des Übersetzerbaus sehr ähnlich[Zim13]. Für das Verständnis dieser Arbeit sind folgende Ausführungen ausreichend: wird bei Domänenspezifischen Sprachen von Programmen oder Beschreibungen gesprochen, wird dies als Instanz eines Modells im Modell-basierten Umfeld bezeichnet. Wird im Übersetzerbau von der abstrakten Syntax gesprochen, ist im Modell-basierten Umfeld Meta-Modell gemeint. Da dieses Meta-Modell, genauso wie Spezifikationen, auch eine Domänen-spezifische Sprache sind, wird deren Implementierung bzw. Spezifikation als Meta-Metamodell bezeichnet[SVEH07, KWB03, CSW08].

[4] Vom Querschnitt dieser Anbindung sind Eingangsdruck und Antriebsleistung abhängig.

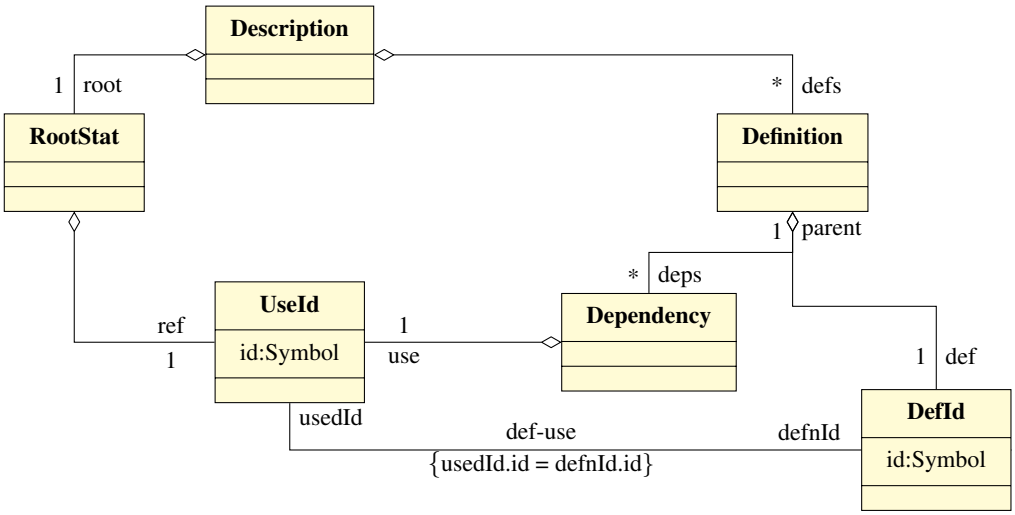


Abbildung 2 – Beispiel eines Klassendiagramms zur Darstellung eines Meta-Modells
Klassen entsprechen den Knoten der abstrakten Syntax (siehe Abbildung 3)

Da es keine einheitliche Definition für Modell und Meta-Modell gibt[SVEH07, KWB03, CSW08, Par10], geben wir hier nur eine kurze Definition für Meta-Modell wieder, entnommen aus [CSW08], die für diesen Beitrag ausreicht.

Definition 2. *Ein **Modell** abstrahiert von einem System, Produkt oder Eigenschaften. Ein **Meta-Modell** ist ein Modell, das die konkrete und abstrakte Syntax einer Modellierungssprache sowie die Semantik dieser Sprache definiert. Ein **Objekt** ist eine Instanz eines Modells und muss* Modellkonform *sein, d.h. der durch das Meta-Modell beschriebenen Syntax und Semantik des Modells genügen.*

Ein Meta-Modell muss Teil einer Meta-Modell-Architektur sein, in [CSW08] wird dies als Meta-Metamodell bezeichnet. Zur Darstellung der Beziehungen eines Modells oder Meta-Modells ist es üblich Klassendiagramme oder Entity-Relationship-Diagramme zu nutzen[Par10, CSW08]. Ein Beispiel eines solchen Metamodells zeigt Abbildung 2.

Um semantische Eigenschaften, spezifiziert als Vor- und Nachbedingungen bzw. Invarianten, zu prüfen, kommen im Übersetzerbau oft Attributgrammatiken zum Einsatz[ALSU06, Knu68, WG84], im Modell-basierten Umfeld wird häufig OCL verwendet[KER99, HJK+11].

Definition 3. *Eine **attributierte Grammatik** ist ein Tupel* $AG \triangleq (\mathsf{G}, A, R, B)$ *wobei*

- $\mathsf{G} \triangleq (T, N, P, Z)$ *eine Grammatik ist, die eine abstrakte Syntax definiert,*
- $A \triangleq \uplus_{X \in (T \cup N)} A(X)$ *eine endliche Menge von Attributen (für jedes Symbol* X *der Grammatik) ist,*
- $R \triangleq \uplus_{p \in P} R(p)$ *eine endliche Menge von Attributierungsregeln ist und*
- $B \triangleq \uplus_{p \in P} B(p)$ *eine endliche Menge von Bedingungen ist.*

***Notation**: Wir schreiben* $X.a$ *für ein Attribut* $a \in A(X)$ *für ein Symbol* $X \in (T \cup N)$*. Für eine Produktion* $p \in P$*, geschrieben* $p : X_0 ::= X_1 \cdots X_n$*, hat eine Attributierungsregel dann die Form* $X_i \leftarrow f(X_j, \ldots, X_k)$ *für* $i, j, k \in [0, n]$ *und* f *eine Funktion. Wir verzichten auf das Schreiben der Funktion, wenn* f *eine konstante Funktion oder die Identitätsfunktion ist (d.h.* $X_i \leftarrow id(X_j)$ *wird zu* $X_i \leftarrow X_j$*). Operationen, die in der Mathematik via Infixoperation geschrieben werden, werden auch bei uns so geschrieben. Eine Bedingung für eine Produktion* p *in obiger Form schreiben wir* $\varphi(X_j, \ldots, X_k)$*, wobei* φ *ein Prädikat ist.*

Attributierte Grammatiken wurden so erstmals von Knuth in [Knu68] vorgestellt, die hier verwendete Definition entstammt [WG84].

Quelltext 1 zeigt ein Beispiel zur Attributierung zweier Produktionsregeln, die Pfeilnotation (→{**error** ⟨`error_arguments`⟩}) drückt aus, dass wenn die Bedingung nicht eingehalten wird, eine Fehlermeldung ausgegeben werden soll. Durch ⟨`error_arguments`⟩ werden die aufgelisteten Parameter angegeben.

```
1  rule Dependency ::= UseId
2  attr Dependency.sym ← UseId.sym
3  cond UseId.sym ∈ Dependency.env → {error "UNKNOWN SYMBOL" UseId.sym }
```

Quelltext 1 – Prüfung, dass benutzte Bezeichner definiert sind

In vielen Programmen zur Generierung der Attributauswertung ist das Attribut `sym` vordefiniert und erlaubt den Zugriff auf das lexikalische Ergebnis für ein Terminalsymbol. Eine explizite Attributierung ist dafür nicht notwendig und wird von uns daher auch nicht durchgeführt.

Ist ein Modell oder ein Meta-Modell in einem UML-Klassendiagramm gegeben, kann mittels OCL die Semantik dieses Modells oder Meta-Modells angegeben werden. OCL ist eine typisierte Sprache zur Formulierung logischer Ausdrücke. Im folgenden werden einige Konstrukte von OCL beschrieben.

Über **Context** ⟨*Klassenname*⟩ **inv** ⟨*OptName*⟩ **:** ⟨*OCL-Ausdruck*⟩ wird eine Klasseninvariante für die Klasse mit dem Namen *Klassenname* erzeugt. Die Invariante kann einen Namen *OptName* haben. Durch **`def`** ⟨ *name*⟩ =⟨*expr*⟩ wird ein Attribut oder eine Abfragefunktion mit dem Namen *name* erzeugt. In einem Klassendiagramm wird mittels des Punkt-Operators auf Elemente und Rollen einer Klasse und Assoziation zugegriffen, mit dem Operator −> auf Elemente als Menge oder auf Funktionen. Ausgehend von dem Kontext Klassenname kann mit diesen Operationen zu anderen Elementen des Klassendiagramms navigiert werden. In dieser Arbeit werden die folgenden Funktionen benutzt:

`closure`$(name)$ Es wird der transitive Abschluss über die Rolle mit dem Namen $name$ gebildet. Das Ergebnis ist eine Menge.

set**`->includesAll`**(set') Prüft ob $set' \subseteq set$ ist.

set**`->forAll`**$(c|OCL - Ausdruck)$ Für jedes Element $c \in set$ wird der Wahrheitswert der Auswertung des Ausdrucks $OCL - Ausdruck$ bestimmt. Ist dieser Wert für alle Elemente c „wahr“, dann ist das Ergebnis dieses Aufrufs ebenso „wahr“, andernfalls „falsch“.

set**`->excludes`**(c) Prüft ob $c \notin set$ ist.

Ein *OCL-Ausdruck* ist ein logischer Ausdruck, der (u.a.) obige Funktionen und Navigation verwenden kann, d.h. der Ausdruck `self->defs.def.id->asBag() = self->defs.def.id->asSet()` zur Prüfung eindeutiger (nicht mehrfach vorkommender) Definitionen ist ein gültiger OCL-Ausdruck.

4 Umsetzung der DSLs

Wie bereits in Abschnitt 1 erwähnt, werden in vielen DSLs Bezeichner definiert und verwendet, daher abstrahieren wir nun von der konkreten Problematik – Anforderungsmodellierung in der Pumpendomäne – zu Darstellungen, die nur aus Definition und Nutzung bestehen, ab. Die Abstraktion ist in Abbildung 3 in Form einer abstrakten Syntax abgebildet; hierbei ist `id` ein Bezeichner. Abbildung 2 (vorheriger Abschnitt) zeigt das zur abstrakten Syntax zugehörige Meta-Modell.

⟨*Description*⟩ ::= ⟨*RootStat*⟩ ⟨*Definition*⟩*
⟨*Definition*⟩ ::= ⟨*DefId*⟩ ⟨*Dependency*⟩*
⟨*RootStat*⟩ ::= ⟨*UseId*⟩
⟨*Dependency*⟩ ::= ⟨*UseId*⟩
⟨*DefId*⟩ ::= id
⟨*UseId*⟩ ::= id

Abbildung 3 – Abstrakte Syntax zur Umsetzung der Beispiele ⟨nt⟩ sind Nichtterminale, id bezeichnet Terminale, | die Alternative und ∗ den endlichen Abschluss

Der endliche Abschluss entspricht einer Listenproduktion. Beispiel 2 zeigt die komplette Listenproduktion für $\langle \textit{Definition} \rangle^*$.

$\langle \textit{Definitions} \rangle \quad ::= \langle \textit{Defininitions} \rangle \ \langle \textit{Definition} \rangle \mid \varepsilon$

Beispiel 2 – Ersetzung des endlichen Abschluss durch Listenproduktionen für $\langle \textit{Definition} \rangle^*$

In Produktionen steht die Liste $\texttt{X}^*$ demnach für das Nichtterminal `Xs` und die Produktionen `Xs ::= Xs X` und `X ::=` ε. In der ersten Produktion schreiben wir $\texttt{Xs}_0$ für die linke Seite der Produktion, $\texttt{Xs}_1$ für die rechte Seite um die Nichtterminale `Xs` auseinander zu halten.

Einen Ausschnitt aus der konkreten Syntax zur Anforderungsmodellierung im Pumpenumfeld stellt Abbildung 4 dar.

$\langle \textit{Requirement} \rangle$	$\rightarrow$	'rq' label $\langle \textit{Optuid} \rangle$
	$\mid$	'rq' $\langle \textit{Optlabel} \rangle$ $\langle \textit{Optuid} \rangle$ '{' $\langle \textit{ReqStat} \rangle$* '}'
$\langle \textit{Optlabel} \rangle$	$\rightarrow$	label $\mid$ $\langle \textit{empty} \rangle$
$\langle \textit{Optuid} \rangle$	$\rightarrow$	uid $\mid$ $\langle \textit{empty} \rangle$
$\langle \textit{ReqStat} \rangle$	$\rightarrow$	'desc' $\langle \textit{Optdot} \rangle$ text
	$\mid$	'label' $\langle \textit{Optdot} \rangle$ label
	$\mid$	'uid' $\langle \textit{Optdot} \rangle$ guid
	$\mid$	'bug' $\langle \textit{Optdot} \rangle$ integer
	$\mid$	'requires' $\langle \textit{Optdot} \rangle$ ($\langle \textit{Reference} \rangle \mid \langle \textit{References} \rangle$)
$\langle \textit{Reference} \rangle$	$\rightarrow$	guid $\mid$ label
$\langle \textit{References} \rangle$	$\rightarrow$	'[' $\langle \textit{Reference} \rangle^+$ ']'

Abbildung 4 – Konkrete Syntax zur Anforderungsmodellierung (Ausschnitt), $^+$ entspricht dem nicht-leeren endlichen Abschluss

Wie bereits erwähnt, wollen wir die Abhängigkeiten von Anforderungen analysieren, der dafür notwendige Graph sei auf Basis der abstrakten Syntax wie folgt definiert:

Definition 4. *Ein Quadrupel* $Dep \triangleq (G, DefIds, UseIds, R)$ *mit*

- $G \triangleq (V, E)$ *ein gerichteter Graph mit* $V \triangleq DefIds \cup UseIds$ *und* $E \subseteq DefIds \times UseIds$,
- $DefIds$ *der Menge der* definierten Bezeichner,
- $UseIds$ *der Menge der* benutzten Bezeichner *und*
- R *der Menge der Wurzeln*

heißt ***Abhängigkeitsgraph***.

Der Abhängigkeitsgraph kann unter Verwendung der abstrakten Syntax konstruiert werden.

In Definition 4 enthält V auch Elemente, die undefiniert sind oder nicht verwendet werden. Diese Definition entspricht der Möglichkeit den Graph anhand der abstrakten Syntax aus Abbildung 3 (Abschnitt 4) zu beschreiben. Die abstrakte Syntax selbst hat nur begrenzt Aussagekraft bezüglich der Korrektheit des Graphen. Diese Korrektheit gilt es in einer Implementierung zu prüfen.

Auf der Basis des Abhängigkeitsgraph können die Eigenschaften definiert werden, die durch die Implementierung einer DSL für die Anforderungsmodellierung bei der Pumpenentwicklung zu prüfen sind.

Die folgende Eigenschaft beschreibt die klassische Namensanalyse mit der zusätzlichen Eigenschaft, dass die Wurzelelemente definiert sind. In der Pumpendomäne gibt es nur eine ausgezeichnete Wurzel (siehe Abschnitt 1).

Definition 5. *Ein Abhängigkeitsgraph* $Dep \triangleq (G, DefIds, UseIds, R)$ *heißt* ***vollständig*** *genau dann, wenn*

- $UseIds \subseteq DefIds$ *und*
- $R \subseteq DefIds$.

Notation: *Ist ein Abhängigkeitsgraph nach dieser Definition vollständig und* $|R| = 1$*, nutzen wir statt der Menge* R *das Element* $r \in DefIds$ *als ausgezeichnetes Wurzelelement.*

Ohne die grundlegende Eigenschaft der *Vollständigkeit*, insbesondere dem Teil, der der Namensanalyse entspricht, können die anderen Eigenschaften nicht geprüft werden. Die Anzahl der Wurzeln ist bereits durch die abstrakte Syntax und das Metamodell gesichert. Eine weitere Eigenschaft, die bereits in der Einleitung, wie auch in Abschnitt 2 motiviert wurde, ist dass, ausgehend von der Wurzel, jede Anforderung mindestens einmal verwendet werden muss.

Definition 6. *Ein vollständiger Abhängigkeitsgraph* $Dep \triangleq (G, DefIds, UseIds, r)$ *heißt* ***gebunden*** *genau dann, wenn für alle* $v \in V$ *ein Pfad von* r *nach* v *in* G *existiert.*

Definition 7. *Ein vollständiger Abhängigkeitsgraph* $Dep \triangleq (G, DefIds, UseIds, r)$ *heißt* ***azyklisch*** *genau dann wenn* G *azyklisch ist.*

Definitionen 5, 6 und 7 müssen für die Anforderungsmodellierung in der Pumpendomäne erfüllt sein, dabei darf es nur eine ausgezeichnete Wurzel geben, da alle Anforderungen sich aus der Anforderung für Druck und Durchfluss der Pumpe ergeben. Die Azyklizität dient dem Sicherstellen, dass es eine Anforderungshierarchie gibt. Ist die „Hierarchie“ zyklisch kann keine Aufgabenfolge zur Implementierung von Abhängigkeiten definiert werden, die gleichzeitig auch parallelisierbar ist. Bei Werkzeugen zur Erstellungsautomatisierung wie GNU make[SMS13] ist die Bindung von Elementen so streng wie von uns gefordert nicht notwendig, jedoch Azyklizität und Vollständigkeit[SMS13] – aus Sicht der Anforderungsmodellierung werden in einem Makefile mehrere Produkte beschrieben, sodass es mehrere Ausgangsanforderungen geben kann. Auch bei Werkzeugen zur Erstellungsautomatisierung gibt für eine Ausführung des Werkzeugs eine ausgezeichnete Wurzel: den vom Nutzer angegebenen Parameter oder das Element `all`.

Wir verzichten darauf weitere Eigenschaften der Namensanalyse wie Mehrfachdeklaration und Blockschachtelung zu definieren und zu prüfen, da wir keine Erkenntnisse erwarten, die nicht bereits bekannt sind: mit [KW91] findet sich bereits ein Modul für die Namensanalyse im Übersetzerbau, ebenso existiert mit [CGQ+06] eine Bibliothek von Ausdrücken zur Verwendung im Modell-basierten Umfeld.

Im folgenden beschreiben wir die Umsetzung zur Prüfung dieser Kriterien auf Basis der gewählten Abstraktion – ohne Analyse oder Darstellung weiterer Informationen bei der Anforderungsmodellierung – anhand zweier möglicher Implementierungen. Unsere Implementierungen sind zueinander kompatibel, sodass eine Umwandelung der Modelle in das entsprechende Format schnell durchführbar ist.

4.1 Umsetzung OCLinEcore

Mittels OCLinEcore wurde ein Meta-Modell des Eclipse Modelling Framework (EMF) um Konsistenzprüfungen erweitert, analog wie es mit Kermeta in [JBF11] durchgeführt wird.

Der Vorteil von OCLinEcore besteht darin, dass einerseits mit [CGQ+06] eine Sammlung von nützlichen OCL-Ausdrücken besteht und somit der Modellierungsaufwand verringert wird und andererseits, dass OCL-Ausdrücke kompakt sind und damit eine Anpassung an die konkrete Struktur des Meta-Modells leichter fällt.

Die Prüfungen der Bindung und Azyklizität sind in Zeile 2 bzw. 8 von Quelltext 2 aufgeführt. Die Bedingungen in den Kontexten `DefId` und `UseId` drücken die Namensanalyse aus. Die Modellierung über das Metamodell aus Abbildung 2 erlaubt mit wenigen Anpassungen auch die Eingabe der Daten im XML-Austauschformat (XMI) ohne einen generierten Editor. Einfachere Metamodelle zur Anforderungsmodellierung, wie z.B. in [MS09] genutzt, erfordern das Wissen um die Indizes der Anforderungen um die Benutztung einer solchen eingeben zu können. Die Prüfung auf eine Wurzel ist nicht notwendig, da dies durch das Metamodell (2) bereits spezifiziert ist.

Die genutzte OCL-Operation `closure` ermittelt die transitive Hülle, bspw. über Rollen. Damit löst diese Operation ein weiter gefasstes Problem, als dies für das Finden von Zyklen notwendig ist, allerdings ist es gleichzeitig die einzige Möglichkeit dies in OCL auszudrücken. Der OCL-Ausdruck zum Finden von Zyklen entspricht einer Standardlösung[CGQ+06].

```
1    context Description:
2      inv bound:
3          let rootdef  = root.ref.defnId.parent,
4              rootdeps = rootdef->closure(getdefs())
5
6          in rootdeps->union(rootdef->asSet())->includesAll(defs)
7
8      inv acyclic: defs->forAll(c |
9                          c->closure(getdefs())
10                            ->excludes(c.def));
11
12   context Definition:
13     def getdefs() =  deps.use.defnId.parent
14
15
16   context UseId:
17     inv defuse: UseId.allInstances()->forAll(x | def-use(x,x))
18
19   context DefId:
20     inv names: DefId.allInstances()->forAll(x |
21                  DefId.allInstances()->forAll(y |
22                    def-use(x,y) implies x.id = y.id))
```

Quelltext 2 – Umsetzung mittels OCLinEcore für Metamodell aus Abbildung 2
`closure` bestimmt die transitive Hülle, **`let`** erzeugt Namen

Diese Umsetzung basiert auf OCLinEcore unter Verwendung der modernen Pivot-Implementierung von OCL, da hier zuerst die `closure`-Operation zur Verfügung stand. Ohne diese Operation muss die zu prüfende Menge manuell, über Mengenvereinigung und Navigation aufgebaut werden. Andere Implementierungen wurden von uns im Rahmen dieser Arbeit nicht betrachtet.

Die Anzahl der Zeilen der OCLinEcore-Variante beträgt 45 Zeilen, wobei circa 20 Zeilen auf die Definition des Metamodells und der darin verwendeten Rollen und Multiplizitäten entfallen.

4.2 Umsetzung Attributgrammatik

Quelltext 3 zeigt die detaillierte Attributierung zum Aufbau des Abhängigkeitsgraph, die Namensanalyse und den Aufruf der semantischen Funktion zur Feststellung der Azyklizität und Gebundenheit (letzten 4 Zeilen von Quelltext 3). Durch die Verwendung der Attribute `defs` und `env` ist es uns möglich Benutzung vor Definition zu erlauben, sodass ein Bezeichner vor der Definition in der Beschreibung verwendet werden kann.

Wir verwenden zur Implementierung das Werkzeug eli (siehe u.a. [GLH+92]), welches geordnete Attributgrammatiken[Kas80] bereit stellt. Wir können in der Implementierung angeben, ob bei einem Fehler die Attributauswertung abbrechen soll. Da in Quelltext 3 viele Details nicht notwendig sind und viele Werkzeuge zur Generierung der Attributauswertung Erweiterungen bieten, nutzen wir folgende Definition aus, um die Attributierung zu vereinfachen.

Definition 8. *Für eine attributierte Grammatik* $AG \triangleq (\mathsf{G}, A, R, B)$ *mit abstrakter Syntax* $\mathsf{G} \triangleq (T, N, P, Z)$ *und einer Liste* X^* *existieren folgende* ***typische Muster*** *:*

Durchreichen von Attributen *Sei* $Y ::= u\ X^*\ v$ *eine Produktion aus* P, $u, v \in (N \cup T)^*$. *Die Attributierungsregel* $X.\mathsf{a} \leftarrow f(\ldots)$ *eines Attributes* $\mathsf{a} \in A$ *steht dann für*

```
1  rule Y ::= u Xs v
2  attr Xs.a ← f(...)
```

und

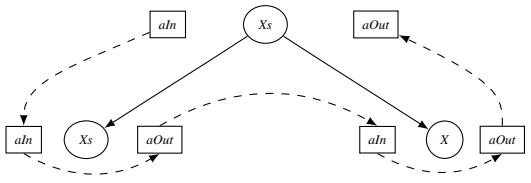
```
1  rule Xs ::= Xs X
2  attr Xs1.a ← Xs.0a
3       X.a ← Xs.a
```

```
1  rule RootStat ::= UseId
2  attr RootStat.root ← UseId.sym
3  cond UseId.sym ∈ RootStat.env → {error "UNKNOWN ROOT" UseId.sym }
4
5  rule Definition ::= DefId Dependencies
6  attr Definition.defsOut ← Definition.defsIn ∪ DefId.sym
7
8       Definition.edgesOut ← Definition.edgesIn ∪ Dependencies.edgesOut
9       Dependencies.edgesIn ← ∅
10
11      Dependencies.source ← DefId.sym
12
13      Dependencies.env ← Definition.env
14 cond DefId.sym ∉ Definition.defsIn → {error "ALREADY DEFINED" DefId.sym }
15
16 rule Dependency ::= UseId
17 attr Dependency.sym ← UseId.sym
18 cond UseId.sym ∈ Dependency.env → {error "UNKNOWN SYMBOL" UseId.sym }
19
20 rule Definitions ::= Definitions Definition
21 attr Definitions1.defsIn ← Definitions0.defsIn
22      Definition.defsIn ← Definitions1.defsOut
23      Definitions0.defsOut ← Definition.defsOut
24
25      Definitions1.edgesIn ← Definition0.edgesIn
26      Definition.edgesIn ← Definitions1.edgesOut
27      Definitions0.edgesOut ← Definition.edgesOut
28
29      Definitions1.env ← Definitions0.env
30      Definition.env ← Definitions0.env
31
32 rule Definitions ::= ε
33 attr Defininitions.defsOut ← Definitions.defsIn
34
35 rule Dependencies ::= Dependencies Dependency
36 attr Dependencies1.edgesIn ← Dependencies0.edgesIn
37      Dependencies0.edgesOut
38              ← Dependencies1.edgesOut∪{( Dependencies0.source, Dependency.sym )}
39
40      Dependencies1.env ← Dependencies0.env
41      Dependency.env ← Dependencies0.env
42
43 rule Dependencies ::=ε
44 attr Dependencies.edgesOut ← Dependencies.edgesIn
45
46 rule Description ::= RootStat Definitions
47 attr Definitions.defsIn ← ∅
48      RootStat.env ← Definitions.defsOut
49      Description.graph ← G(Definitions.defsOut, Defininitions.edgesOut)
50 cond acyclic (Description.graph, RootStat.root)
51         → {error "CYCLIC DEFINITIONS" cyclesOf (Description.graph, RootStat.root) }
52      bound (Description.graph, RootStat.root)
53         → {error "UNREACHABLE DEFINITIONS" unboundOf (Description.graph, RootStat.root) }
```

Quelltext 3 – Attributierung zum Aufbau des Abhängigkeitsgraphen mit Namensanalyse und Prüfung des Graphen über die Bedingungen in Zeilen 50 bis 53

Kettenberechnungen ([GLH+92]) ***chain*** a ***for*** *X* steht für die Attributierungsregeln des Attributs* a *mit*

```
1  rule Xs ::= Xs X
2  attr Xs1.aIn ← Xs0.aIn
3       X.aIn ← Xs1.aOut
4       Xs0.aOut ← X.aOut
5  rule Xs ::= ε
6  attr Xs.aOut ← XsaIn
```



Contributions (u.a. [Hed11]) *Contributions*[5] *erweitern Kettenberechnungen um die Initialisierung von* aIn *und die Berechnung von* `X.`aOut*, wenn mengenwertige Attribute (bzw. Listen) berechnet werden; dabei steht* ***`contribution`*** a ***`for`*** `X.*`b *für*

```
1  chain b for X*
2
3  rule Y ::= u X* v
4  attr Xs.bIn ← ∅
5
6  rule X ::= u
7  attr X.bOut ← X.bIn ∪ { X.a }
```

wobei die erste Regel für alle Produktionen mit X^* *gilt, und die zweite Regel für Produktionen mit linker Seite* `X` *steht.*

Die Definition für typische Muster nutzt rein syntaktische Ersetzungsregeln, eine Erweiterung der Semantik von Attributgrammtiken ist nicht notwendig.

Unter Verwendung der typischen Muster (Definition 8) ergibt sich Quelltext 4 für die Attributberechnungen aus Quelltext 3. Weitere Reduzierungen lassen sich in der Implementierung unter Ausnutzung anderer Erweiterungen attributierter Grammatiken, wie z.B. Vererbungsmechanismen, erreichen.

```
 1  rule Description ::= RootStat Definition*
 2  attr Definition.env ← Definition.defsOut
 3
 4  rule RootStat ::= UseId
 5  attr RootStat.root ← UseId.sym
 6  cond UseId.sym ∈ RootStat.env → {error "UNKNOWN ROOT" UseId.sym }
 7
 8  contribution sym for Definitions*.defs
 9  rule Definition ::= DefId Dependency*
10  attr Dependency.env ← Definition.env
11       Dependency.src ← DefId.sym
12       Definition.sym ← DefId.sym
13  cond DefId.sym ∉ Definition.defsIn → {error "ALREADY DEFINED" DefId.sym }
14
15  rule Dependency ::= UseId
16  attr Dependency.edge ← (Dependency.src, UseId.sym )
17  cond UseId.sym ∈ Dependency.env → {error "UNKNOWN SYMBOL" UseId.sym }
18
19  contribution edge for Dependency*.edges
20  chain edges for Definition*
21  rule Definition ::= DefId Dependency*
22  attr Definition.edgesOut ← Definition.edgesIn ∪ Dependency.edgesOut
23
24  rule Description ::= RootStat Definition*
25  attr Definition.graph ← G(Definition.defsOut, Definition.edgesOut)
26  cond acyclic (Description.graph, RootStat.root)
27          → {error "CYCLIC DEFINITIONS" cyclesOf (Description.graph, RootStat.root) }
28       bound (Description.graph, RootStat.root)
29          → {error "UNREACHABLE DEFINITIONS" unboundOf (Description.graph, RootStat.root) }
```

Quelltext 4 – Attributierung zum Graphaufbau, Namensanalyse und Graphanalyse unter Verwendung von Definition 8; Trennung der Namensanalyse und Graphaufbau durch Zeilen 1 und 2

Quelltext 4 zeigt, wie mit Attributgrammatiken der vollständige Abhängigkeitsgraph aus Definitionen 4 und 5 aufgebaut wird. Dabei prüfen wieder die letzten 4 Zeilen auf Gebundenheit und Azyklizität. Die Menge $UseIds$ (Definition 5) kann aus der Kantenmenge bestimmt werden.

Wird zur Prüfung der Bindung eine klassische Implementierung über Referenzzählung genutzt, dann kann es zu ungenauer Fehlerausgabe und falsch-positiven Ergebnissen kommen. Zur Problemveranschaulichung sei folgende Variante als Alternative angenommen: Statt in Quelltext 4 Kanten zu erzeugen werden in der Menge der definierten Bezeichner zusätzlich gespeichert ob ein Bezeichner verwendet wurde. Nicht-gebundene Bezeichner sind dann jene, die nicht in dieser Menge als „benutzt“ markiert sind. Die Markierung und Abfrage der Markierung erfolgt über `SetUsed` und `IsUnused`. Der entsprechende Ausschnitt könnte damit wie in Quelltext 5 aussehen.

[5] engl. Beitrag

```
1  chain defs for Dependency*
2  rule Definition ::= DefId Dependency*
3  attr Dependency.defsIn ← Definition.defsIn
4       Definition.defsOut ← Dependency.defsOut
5
6  rule UseId ::= id
7  attr UseId.defsOut ← SetUsed(UseId.defsIn, UseId.sym)
8
9  rule Description ::= Statements
10 attr Description.unused ← { x ∈ Statements.defs : IsUnused(x)}
11 cond Description.unused = ∅ →
12        {error "UNUSED SYMBOLS" Description.unused }
```

Quelltext 5 – Umsetzung über Referenzzählung mit Seiteneffekt auf der Definitionstabelle „defs“

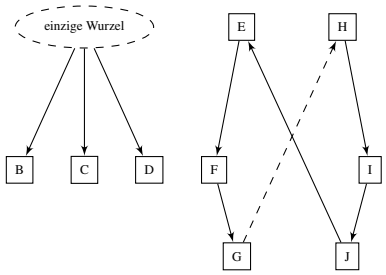


Abbildung 5 – Nicht-gebundene Elemente bei Verwendung von Referenzzählung
unerkannt durch gestrichelte Kante $G \to H$; *Wurzel* = oval, gestrichelt

Bei dieser Implementierung muss dann jedoch das Kettenattribut `defs` auch ein Kettenattribut für `Dependency*` sein. Die Folge solch einer Implementierung lässt sich gut durch Abbildung 5 veranschaulichen: Aufgrund der Implementierung aus Quelltext 5 sind alle Knoten gebunden, jedoch nicht an die Wurzel. Die Implementierung würde hier also ein falsch-positives Ergebnis im Sinne von Definition 6 bestimmen. Wird die Kante $G \to H$ entfernt, so wäre nur das Element H als ungebunden markiert, jedoch sind auch die Elemente E, F, G, H, I, J nicht an die Wurzel gebunden. Letztlich bleibt noch die Alternative direkt in der Attributgrammatik die Tiefensuche anzugeben und auf Basis der aktuell gebundenen Bezeichner die benutzten Bezeichner zu markieren und zu den noch zu betrachtenden hinzuzufügen. Dieser Ansatz ist mit geordneten Attributgrammatiken nicht möglich, auch wird beim Erlauben von Zyklen in vielen Werkzeugen gefordert, dass es einen Fixpunkt gibt, der mittels Fixpunktiteration gefunden werden kann.

Da Graphen nicht nur im Übersetzerbau (siehe dazu u.a. Abschnitt 1) eine häufig einsetzbare Abstraktion bilden, existieren bereits sehr viele Bibliotheken zur Darstellung und Analyse von Graphen. Wir arbeiten momentan daran, auf Basis der Boost Graphbibliothek[SLL02], die Funktionen `acyclic`, `bound`, `cyclesOf` und `unboundOf` mit Einstellungen zu Abbruchkriterien und Fehlerausgabe für eli zur Verwendung mit Attributgrammatiken zur öffentlichen Verfügung zu stellen.

Das Werkzeug um die Attributberechnungen durchzuführen und für das wir die genannten Funktionen zur Verfügung stellen wollen ist eli[GLH+92]. Die Spezifikation zur Umsetzung in eli ohne Hilfsfunktionen zum Bibliotheksaufruf besteht aus 124 Zeilen, wovon 18 Zeilen auf die Weitergabe von Kommandozeilenoptionen entfallen. 40 Zeilen entfallen auf die reine Namensanalyse, sowie weitere 22 Zeilen, welche Quelltext 4 entsprechen.

5 Vergleich der Implementierungen

Anhand von Beispiel 1 zeigen wir in Abschnitt 5.1 die Fehlerausgabe durch die beiden Umsetzungen, zuvor betrachten wir jedoch die Laufzeiten der Implementierungen.

Das Testsystem ist ein Sabayon Linux mit Kernel Version 3.11. Das System hat eine Intel Core i7-3770 CPU und verfügt über 16 GiB Arbeitsspeicher. Die Reallaufzeiten wurden unter Verwendung von GNU Time 1.17 ermittelt. Java ist in Version 1.7.0u45 vorhanden.

Durch einen Testgenerator wurden für beide Implementierungen der abstrakten Syntax aus Abschnitt 4 entsprechende Eingaben erstellt. Da Graphprobleme behandelt werden, stellt dieser Testgenerator einige Optionen zur Verfügung, die sich in Tabelle 1 wiederfinden. Die Spalte „*zyklisch*“ bezeichnet, dass zyklische Graphen generiert werden sollen. Mittels der Option „*Tot*“ wird die Prozentzahl der nicht-erreichbaren Knoten des Graphen festgelegt – die eigentliche Anzahl an Knoten mittels „*Knoten*“. Mit der Option „*Dichte*“ wird, nach Abzug der toten Knoten, festgelegt, wieviele Prozent der maximal noch möglichen Kanten (einem vollständigen Graphen entsprechend) zu erzeugen sind. Die Eigenschaft „Knoten“ bestimmt die definierten Bezeichner, die anderen Eigenschaften deren Benutzung. Der Testgenerator erzeugt keine Dokumente in denen undefinierte Bezeichner verwendet werden.

Eigenschaften				Def 6		Gesamt		Parse	
Knoten	**Dichte**	**Tot**	**zyklisch**	**OEcore**	**eli**	**OEcore**	**eli**	**eli**	**Ecore**
100	3	10	nein	3.37 s	< 0.01 s	3.42 s	< 0.01 s	< 0.01 s	2.70 s
1000	4	2	nein	5.00 s	0.01 s	5.08 s	0.01 s	< 0.01 s	2.82 s
1000	4	2	ja	3.54 s	0.06 s	3.61 s	0.07 s	< 0.01 s	2.87 s
2000	5	0	ja	3.69 s	0.04s	3.80 s	0.04 s	< 0.01s	2.95s
2000	6	10	nein	17.47 s	0.08 s	17.60 s	0.08 s	0.02 s	2.99 s
2000	10	0	nein	34.61 s	0.17 s	35.10 s	0.17 s	0.03 s	3.04 s
5000	10	10	nein	6 m 30 s	1.10 s	6 m 43 s[6]	1.10 s	0.41 s	3.70 s
5000	10	0	ja	**∅**	13.76 s	**∅**	13.95 s	0.41 s	3.76 s
10000	2	4	ja	**∅**	17.50 s	**∅**	17.58 s	0.57 s	3.68 s
10000	2	4	nein	**>10 m**	1.48 s	**>10 m**	1.48 s	0.73 s	3.77 s

Tabelle 1 – Laufzeiten der Werkzeuge
OEcore ≙ Ecore Eingabe, Prüfung mit OCL; Def 6 ≙ Laufzeit zum Einlesen, ggf. Baumaufbau und Prüfung von Definition 6; Gesamt ≙ Gesamtlaufzeit; Parse ≙ Laufzeit zum Einlesen und ggf. Baumaufbau

Zur besseren Vergleichbarkeit wurde die Laufzeit zur Prüfung der Definition 6 getrennt aufgeschlüsselt, dabei sind Ein- und Ausgabe, Programmstart sowie ggf. Graphaufbau mit einbezogen. Ebenfalls getrennt aufgeschlüsselt (Spalte „*Parse*“) wurden die Zeiten, die benötigt werden um ein Dokument einzulesen, ohne Ausgabe zu erzeugen oder die Semantik zu analysieren. Die Beschreibungen für Verwendung mit der OCL-Variante sind, aufgrund des zugrunde-liegenden XML-Formats, höchstens 40% größer als die entsprechenden Dokumente für die eli-Variante. Ecore bezieht sich in Spalte „*Parse*“ auf die der OCL-Variante zugrunde liegenden Technologie. „OEcore“ bezeichnet die Laufzeiten für die Eingabe und das Analysieren mittels OCL. Die Spalten *Gesamt* beziehen sich hingegen auf die Gesamtlaufzeit der Programme und enthalten auch die Analyse aller Definitionen aus Abschnitt 4. Zur Ermittlung der aufgeführten Zeitwerte wurden je Knotenzahl und Zeitwert 1000 Versuche durchgeführt und das arithmetische Mittel der Laufzeit bestimmt. Dauerte ein einzelner Versuch länger als 10 Minuten wurde dieser abgebrochen und neu gestartet. Gab es zehn aufeinanderfolgende Versuche mit jeweils mehr als 10 Minuten Laufzeit wurde der Versuchslauf abgebrochen – in Tabelle 1 sind diese Versuchsläufe fett markiert. Ebenfalls markiert mit $\emptyset$ sind die Werte, in denen die OCL-Variante kein Ergebnis liefern kann, da ein Stapelüberlauf auftritt.

Bemerkung 1. *Die Ursache für den Speicherüberlauf sind die Standardeinstellungen der Java VM. Erst eine Vervierfachung des zulässigen Speicher sorgt dafür, dass für die markierten Beispiele Ergebnisse durch OCLinEcore geliefert werden können. Jedoch sollte es der Java VM möglich sein bei ca. 14 GiB freiem Speicher automatisch mehr zu nutzen. Weiterhin ist die* `closure` *Operation unter Verwendung von Introspektion und Rekursion so implementiert, dass selbst ein aggressiver Übersetzer keine Optimierung durchführen kann.*

Wird der erlaubte Speicher vergrößert, dann kann OCL, aufgrund des fehlerhaften Modells, frühzeitig abbrechen. Dieses Verhalten ergibt sich aus der Kurzauswertung logischer Ausdrücke. Kann OCL jedoch nicht frühzeitig abbrechen, d.h. das Modell ist korrekt, ist die Laufzeit von OCL verglichen mit der eli-Variante um Größenordnungen schlechter. Da wir die Laufzeit und nicht den Speicherverbrauch betrachten, wurden keine ausführlichen Tests mit anderen Speichereinstellungen der Java VM durchgeführt. Die Sprünge bei den Laufzeiten (Zeilen 8 bis 10 der

[6] Aus Zeitgründen konnten zur Bestimmung diesen Eintrags bisher nur zehn Testläufe durchgeführt werden.

Tabelle) für die eli-Variante ergeben sich aus der umfangreicheren Ausgabeerzeugung für zyklische Komponenten. Bei den reinen Zeiten für die reine Syntaxanalyse ergibt sich, dass die Initialisierung von Java zusätzlich Zeit benötigt. Die Sprünge der Parsezeiten bei nahezu gleichen an den Testgenerator übergebenen Eigenschaften ergibt sich aufgrund der unterschiedlichen Hierarchie, die durch den Testgenerator (zufallsbasiert) erstellt wird: längere Laufzeiten bedeuten eine flachere Hierarchie, aber einen breiteren abstrakten Syntaxbaum, d.h. mehr Kanten pro Knoten.

Die Anzahl der Anforderungen für die Beispiele wurde der täglichen Praxis entnommen. Bei ersten Umwandelungen in unser Format hat die Anforderungshierarchie eine Dichte im einstelligen Prozentbereich, sodass die von uns gewählten Beispiel durchaus realistisch sind.

Bemerkung 2. *Eine Laufzeitanalyse für eine Variante mit XText haben wir nicht durchgeführt. Nach unserem Eindruck hat XText Probleme mit der Verwendung von Zeichenketten, sodass Laufzeit und Speicherverbrauch ungeeignet für große Beispiele sind. Zeichenketten kommen jedoch gerade bei der Anforderungsmodellierung, in Form von Anforderungsbeschreibungen, Verantwortlichkeiten oder bei Angaben zur Dokumentenerzeugung, häufig vor.*

5.1 Evaluierung der Fehlerausgabe

Quelltext 6 zeigt die Ausgabe der eli-generierten Anwendung für Beispiel 1. Es ist klar zu erkennen, welche Elemente und Kanten einen Zyklus ausmachen und wo diese im Quelltext vorkommen, sowie welche Elemente nicht erreichbar (Zeilen 16, 17) sind.

```
1  "PUMP.RQ", 1:51 ERROR: Unknown Symbol: "NUMMER8"
2   "PUMP.RQ", 1:1 ERROR: Found a cyclic component, Path:
3   "PUMP.RQ", 1:1 ERROR:   "DRUCKERZEUGUNG" -> "ANLAGENANBINDUNG"
4    "PUMP.RQ", 3:5 INFO:       Definition of: "DRUCKERZEUGUNG"
5    "PUMP.RQ", 4:5 INFO:       Definition of: "ANLAGENANBINDUNG"
6   "PUMP.RQ", 1:1 ERROR:   "ANLAGENANBINDUNG" -> "DRUCKERZEUGUNG"
7    "PUMP.RQ", 4:5 INFO:       Definition of: "ANLAGENANBINDUNG"
8    "PUMP.RQ", 3:5 INFO:       Definition of: "DRUCKERZEUGUNG"
9   "PUMP.RQ", 1:1 ERROR: Found a cyclic component, Path:
10  "PUMP.RQ", 1:1 ERROR:   "ANLAGENANBINDUNG" -> "DRUCKERMITTELUNG"
11   "PUMP.RQ", 4:5 INFO:       Definition of: "ANLAGENANBINDUNG"
12   "PUMP.RQ", 2:5 INFO:       Definition of: "DRUCKERMITTELUNG"
13  "PUMP.RQ", 1:1 ERROR:   "DRUCKERMITTELUNG" -> "ANLAGENANBINDUNG"
14   "PUMP.RQ", 2:5 INFO:       Definition of: "DRUCKERMITTELUNG"
15   "PUMP.RQ", 4:5 INFO:       Definition of: "ANLAGENANBINDUNG"
16  "PUMP.RQ", 7:5 ERROR: Unreachable : "TEMPERATURMESSUNG"
17  "PUMP.RQ", 6:5 ERROR: Unreachable : "GPSNAVI"
```

Quelltext 6 – Fehlerausgabe (eli-Variante) für Pumpenbeispiel aus Abschnitt 2

```
1  ERROR: Diagnosis of DescriptionImpl@32b9bd47{file:pump.xml#/}
2    ERROR: The 'bound' constraint is violated on
3           'DescriptionImpl@32b9bd47{file:pump.xml#/}'
4    ERROR: The 'acyclic' constraint is violated on
5           'DescriptionImpl@32b9bd47{file:pump.xml#/}'
```

Quelltext 7 – Fehlerausgabe (OCL-Variante) für Pumpenbeispiel aus Abschnitt 2

Demgegenüber steht die Ausgabe durch die Variante, die OCL zur Prüfung nutzt. Die Ausgabe der OCL-Variante ist in Quelltext 7 zu sehen. Quelltext 8 zeigt die Fehlerausgabe einer alternativen Umsetzung der OCL-Invariante zur Prüfung auf Zyklen. Die Änderung besteht darin diese Invariante (ohne `forAll`) in den Kontext des Elements zu schieben. Der Nachteil an dieser Invariante ist, dass die Laufzeit der OCL-Variante dadurch weiter erhöht wird: die Laufzeit zur Prüfung auf Azyklizität in Tabellenzeile 3 braucht statt der angegebenen die doppelte Zeit, die vorletzte Zeile circa 30 Minuten zum Finden der zyklischen Komponente wenn erhöhter Speicherverbrauch eingestellt wurde.

```
1  ERROR: Diagnosis of DescriptionImpl@4d3b6cf3{file:pump.xml#/}
2    ERROR: The 'bound' constraint is violated on
3           'DescriptionImpl@4d3b6cf3{file:pump.xml#/}'
4    ERROR: The 'bound' constraint is violated on
5           'DefinitionImpl@7c7686f8{file:pump.xml#//@defs.1}'
6    ERROR: The 'acyclic' constraint is violated on
7           'DefinitionImpl@4997439e{file:pump.xml#//@defs.2}'
8    ERROR: The 'acyclic' constraint is violated on
9           'DefinitionImpl@69d85fd0{file:pump.xml#//@defs.3}
```

Quelltext 8 – Erweiterte Fehlerausgabe (OCL-Variante) für Pumpenbeispiel aus Abschnitt 2

Wie anhand der Quelltexte 6,7 und 8 erkannt werden kann, ist es in der eli-generierten Variante selbst bei großen Zyklen möglich die Elemente eines Zyklus zu finden, wohingegen die OCL-Varianten nicht einmal eine Kante angeben kann, die bei Entfernung den Zyklus aufheben würde; bei Nutzung eines generierten Editors würden die Fehlermeldungen statt z.B. `@defs.1` den Bezeichner enthalten.

Wie am Ende von Abschnitt 4.2 beschrieben, lassen sich die zur Prüfung und Fehlerausgabe genutzten Funktionen in der eli-Variante parametrisieren, sodass detaillierte Fehlerausgabe, umfangreichere Fehlerausgabe oder schneller Abbruch ermöglicht werden. Die reine Bestimmung ob Zyklen vorhanden sind, ist schnell erledigt – selbst für sehr große Beispiele (> 5000 definierte Elemente mit einer Dichte von 10%) in weniger als anderthalb Sekunden. Sollen dagegen alle Zyklen mit allen dazugehörigen Pfaden ausgegeben werden, so entspricht dies der Ausgabe aller Zusammenhangskomponenten. Die möglichen Parameter kann der Endanwender dem Übersetzer übergeben.

6 Verwandte Arbeiten

Zur Analyse von Anforderungen und deren Umsetzung mittels Domänen-spezifischer Sprachen gibt es eine Reihe von Arbeiten und Werkzeugen, einen etwas älteren Überblick über die offenen Probleme der Anforderungsanalyse gibt [NE00]. Viele Arbeiten im Bereich der Anforderungsanalyse mittels Domänen-spezifischer Sprachen, wie [DCS+13, GKvdB10], beschreiben nur dass und ggf. wie eine Sprache umgesetzt wurde, selten wird auf die Prüfung der Semantik der Sprache eingegangen, wie wir es hier getan haben. Keine der uns bekannten Arbeiten der Anforderungsanalyse bietet in diesem Rahmen einen Laufzeitvergleich an. Vielmehr sind Umfang der Beschreibungen (im Sinne von Modellinstanz) und die umgesetzten Analysen von Interesse, so auch bei [TÅJ12], einer Arbeit, die zeigt wie (ersetzbare) Referenzattributgrammatiken genutzt wurden um ein bestehendes System auf Basis von Attributgrammatiken zu ersetzen.

Referenzattributgrammatiken (RAGs) wurden unter diesem Namen erstmals in [Hed00] beschrieben, jedoch reicht unserer Ansicht nach die Motivation in [Hed00] nicht aus, da für die gewählten Beispiele an Problemen mit reinen Attributgrammatiken eine Definitionstabelle ausreichend ist. In [Hed00] kommt zur Prüfung der Typhierarchie von PicoJava, einer Teilmenge der Java-Sprache, unter Verwendung von Attributgrammatiken dieselbe Prüfung zum Einsatz, wie wir sie in OCL genutzt haben (siehe Quelltext 2, Zeile 2). Eine Erweiterung von RAGs, ersetzbare Referenzattributgrammatiken, wurden alternativ zu der von uns genutzten Variante, die auf[Gro09] basiert, in [TÅJ12] benutzt um die Semantik von Modellen spezifizieren zu können. Das Hauptaugenmerk von [TÅJ12] ist nicht Laufzeit sondern Umfang der Implementierung – der Umfang des Quelltext wurde von der per Hand geschriebenen Variante um 75% reduziert. Die wenigen Aussagen, die [TÅJ12] zur Laufzeit trifft deuten an, dass, trotz verlängerter Übersetzungsdauer und langsamerer Namensanalyse, der erzeugte Interpreter inetwa gleich schnell wie die ursprüngliche (handgeschriebene) Variante ist. Ersetzbare Referenzattributgrammatiken (ReRag) wurden u.a. in [EH04] vorgestellt und dort in einem Vergleich der Laufzeiten eines Werkzeugs zur Prüfung der statischen Semantik von Java 1.4 mit der Laufzeit des Java Übersetzers verglichen: die Laufzeit unter Verwendung von ReRAG war annähernd 4 mal höher als die des Java Übersetzers. Die eben genannten Arbeiten ordnen wir, trotz Einbettung in die Meta-Modellierung, dem Übersetzerbau zu.

JastAdd (siehe u.a. [Hed11]) ist ein weiteres Werkzeug zur Generierung von Programmen aus Attributgrammtiken. Die in [Hed11] als Contributions bezeichnete Technik lässt sich mit Hilfe weniger Bibliotheksfunktionen auch in eli ausdrücken.

In [JBF11] wird gezeigt, dass die Semantik von Meta-Modellen, bzw. statische Analysen, unter Verwendung von OCL umgesetzt werden können. Die Arbeit selbst beschreibt das Werkzeug „Kermeta“, das, analog zu eli[GLH+92], eine Verbindung mehrerer Technologien bereitstellt und diese zur vereinfachten Sprachentwicklung verknüpft. Auch [JBF11] bietet keine Laufzeiten zum Vergleich an. [HJK+11] zeigt, wie OCL auch in der beschriebenen DSL genutzt werden kann, jedoch gehen diese Arbeiten nicht auf die maschinelle Prüfung der so beschriebenen Semantik ein.

Einen Überblick über OCL enthält [Pan11], in dem auch beschrieben wird, dass OCL zur Prüfung der Semantik von Sprachen sowie Modelltransformationen genutzt wird.

7 Zusammenfassung

Die Möglichkeit Beschreibungen und Programme in Form von Graphen darzustellen und diese zu transformieren ist im Rahmen des Übersetzerbaus bekannt. Zur Beschreibung der statischen Semantik von DSLs kann OCL zwar genutzt werden, doch lassen sich mit den Methoden des Übersetzerbaus Programme generieren, die nicht nur um 3-4 Größenordnungen performanter sind, sondern ebenso eine bessere Fehlerdiagnose bereitstellen.

Zwar zeigen [JBF11, KER99] die Möglichkeit OCL zur Definition der statischen Semantik von Sprachen zu nutzen. Dennoch ist im Modell-basierten Umfeld die Spezifikation der Semantik einer Sprache noch eine offene Fragestellung [FR07, SGBvB12]. Unsere Erkenntnisse zeigen, dass Attributgrammatiken ein geeigneter Formalismus im Modell-basierten Umfeld sein kann.

Wie bereits [SSF+07], bei der exemplarisch eine Domänen-spezifische Sprache unter Verwendung von OCL und manueller Transformation in Attributgrammatiken erstellt wurde, motiviert auch unsere Arbeit die Implementierung der automatischen Transformation von OCL-Ausdrücken in schnelle Attributgrammatiken.

Wenngleich sich mit wenig Einarbeitungszeit erste Domänen-spezifische Sprachen im Modell-basierten Umfeld erstellen lassen, so ist die Performance dieser nach unseren Erkenntnissen ungeeignet um die modell-basierte Organisation[CFK+13] zu unterstützen. Trotz des höheren Abstraktionsniveau von OCL kann unter Verwendung von Hilfsfunktionen und Mustern auch bei Attributgrammatiken ein nahezu vergleichbares Abstraktiosniveau erreicht werden. Auch bei OCL hängt das Abstraktionsniveau stark von Bibliotheken und vordefinierten Funkionen ab. Auch mit Attributgrammatiken lassen sich kurze Spezifikationen erstellen, durch die Erweiterungen kann das Verständnis sogar noch gesteigert werden. UUAG[VSM12] stellt Attributgrammatiken in Haskell zur Verfügung, sodass auch bei den semantischen Funktionen selbst ein höherer Abstraktionsgrad gewonnen werden kann. Eine ausführliche Betrachtung der Erweiterungen und Implementierungsformen von Attributgrammatiken war jedoch nicht Gegenstand dieser Arbeit.

Die von uns gewählten Beispiele sind zwar durch einen Generator erstellt, doch sind die Einstellungen dieses Generators so gewählt, dass dieser realitätsnahe Dokumente erzeugt. Aus Sicht eines Endanwenders ist es irrelevant, ob OCL oder Attributgrammatiken genutzt wurden um die Sprache zu implementieren, Geschwindigkeit und Fehlerausgabe sind von Relevanz. Einer Sprachentwicklung durch Domänen-Experten stehen wir aus Sicht des Übersetzerbaus kritisch gegenüber. Unserer Ansicht nach müssen Sprach-Experte und Domänen-Experte gemeinsam an der Entwicklung einer DSL beteiligt sein.

Bemerkung 3. *Beide Autoren haben an der Implementierung einer industriellen DSL, Hart-DD, mitgewirkt. Die Sprache wurde nahezu ausschließlich durch Domänen-Experten spezifiziert. Hierbei zeigte sich einerseits, dass Sprachdefinitionen bei steigendem Umfang mehrdeutig wurden. Andererseits wurden Konstrukte eingeführt, die Definitionen (von Bezeichnern) in anderen Namensräumen so beeinflussen konnten, dass die Definition selbst ersetzt wurde (*`LIKE` *und* `IMPORT`*).*

Nicht nur in den Eingangs erwähnten Sprachen werden Bezeichner definiert und benutzt, auch in [DR07], einer Sprache zur Beschreibung eingebetteter Steuerungen in Echtzeit-Systemen, ist dies der Fall. Gleiches gilt auch für SysML, einer in der Automobilindustrie verwendeten Modellierungssprache (siehe u.a. [AP10]). Unsere Betrachtung hat also in sehr vielen Sprachen Relevanz, wenngleich die Wichtigkeit der Informationen unterschiedlichen Grad haben. Es liegt in der Natur der unterschiedlichen Domänen, dass unterschiedliche Eigenschaften bei der Modellierung wichtig sind – Zyklen sind nicht immer ein Fehler, Gebundenheit ist je nach Sprache nur zur Optimierung notwendig.

Wenngleich es mehr als eine OCL-Implementierung gibt, so war es nicht unser Anliegen diese unterschiedlichen Implementierungen untereinander zu vergleichen, sondern für ausgewählte Probleme OCL und Attributgrammatiken gegenüber zu stellen. Wie sich die Implementierungen für eine Variante zur Semantik-Spezifikation – OCL oder Attributgrammatiken – untereinander verhalten, ist eine noch offene Fragestellung.

Im Umfeld von Eclipse existieren bereits eine Reihe von Technologien zur Generierung von Editoren und Visualisierungen, dies ist erst in Ansätzen für bewährte Werkzeuge des Übersetzerbaus vorhanden[SK03]. An dieser Stelle existieren allerdings auch noch weitere Strategien oder andere Arten von Editoren, die nur zum Teil eine Umsetzung in aktuellen Technologien haben.

Im Modell-basierten Umfeld werden nicht selten im generierten Code Anpassungen vorgenommen. Die Nutzung von Bibliotheken funktioniert entweder über direkte Anpassungen des generierten Quelltext oder über direkte Nutzung generierter Klassen (z.B. Vererbung). Diese Vorgehensweise ist problematisch, weil diese manuellen Anpassungen verloren gehen, wenn der Code neu generiert wird. Auch das direkte Nutzen oder Erben generierter Klassen löst dieses Problem nicht, da durch Änderungen im Modell sich die Klassenstruktur verändern kann. Übersetzerbauwerkzeuge zeigen hier eine andere Möglichkeit, um Bibliotheksaufrufe oder Quellcode in den generierten Code einzubinden: die DSL erlaubt auch die Einbeziehung von Codefragmenten (inkl. Funktionsaufrufen) in die Modelle.

Die in dieser Arbeit gewonnenen Erkenntnisse bieten Ansatzpunkte, um umfangreiche Domänen-spezifsche Sprachen, im Gegensatz zu Mini-Sprachen, im Modell-basierten Umfeld entwickeln zu können. Außerdem eröffnet die Nutzung von Übersetzerbauwerkzeugen für DSLs auch neue Möglichkeiten zur Definition der statischen Semantik dieser. Damit können mehr Konsistenzbedingungen effizient überprüft werden als dies mit OCL möglich wäre. Desweitern können Methoden der Programmanalyse auf DSLs übertragen werden, um beispielsweise Rückverfolgbarkeit auch über verschiedene Entwicklungsphasen von der Anforderungsanalyse bis hin zur Implementierung, zu realisieren. Die durch uns gewonnenen Ergebnisse motivieren die automatische Generierung von effizienten Attributgrammatiken aus OCL-Bedingungen zu implementieren.

Danksagung
Wir danken unseren Projektpartnern für die gute Zusammenarbeit und die Bereitstellung umfangreicher Beispiele aus dem aktuellen Arbeitsalltag. Weiterhin danken wir dem Projektträger VDI/VDE-IT und dem BMBF, die diese Arbeit im Rahmen des Projekts ELSY (Nr. 16M3202D) betreuen und fördern. Den anonymen Reviewern danken wir für Anmerkungen, Kritik und nützliche Hinweise.

Literatur

[ALSU06] Alfred V. Aho, Monica S. Lam, Ravi Sethi und Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison Wesley, 2. ausgabe. Auflage, September 2006.

[AP10] Eric Andrianarison und Jean-Denis Piques. SysML for embedded automotive Systems: a practical approach. In *Conf. on Embedded Real Time Software and Systems*, 2010.

[BKWA11] Christoff Bürger, Sven Karol, Christian Wende und Uwe Aßmann. Reference Attribute Grammars for Metamodel Semantics. In Brian Malloy, Steffen Staab und Mark Brand, Hrsg., *Software Language Engineering*, Band 6563 der Reihe *Lecture Notes in Computer Science*, Seiten 22–41. Springer, 2011.

[CFK+13] Tony Clark, Ulrich Frank, Vinay Kulkarni, Balbir Barn und Dan Turk. Domain specific languages for the model driven organization. In *Proceedings of the First Workshop on the Globalization of Domain Specific Languages*, GlobalDSL '13, Seiten 22–27. ACM, 2013.

[CGQ+06] Dolors Costal, Cristina Gómez, Anna Queralt, Ruth Raventós und Ernest Teniente. Facilitating the definition of general constraints in UML. In *Model Driven Engineering Languages and Systems*, Seiten 260–274. Springer, 2006.

[CSW08] Tony Clark, Paul Sammut und James Willans. Applied metamodelling: a foundation for language driven development., 2008.

[DCS+13] Diego Dermeval, Jaelson Castro, Carla Silva, João Pimentel, Ig Ibert Bittencourt, Patrick Brito, Endhe Elias, Thyago Tenório und Alan Pedro. On the Use of Metamodeling for Relating Requirements and Architectural Design Decisions. In *Proceedings of the 28th Annual ACM Symposium on Applied Computing*, SAC '13, Seiten 1278–1283. ACM, 2013.

[DL85] Tom DeMarco und Tim Lister. Programmer Performance and the Effects of the Workplace. In *Proceedings of the 8th International Conference on Software Engineering*, ICSE '85, Seiten 268–272. IEEE, 1985.

[DR07] Gwenaël Delaval und Éric Rutten. A domain-specific language for multitask systems, applying discrete controller synthesis. *EURASIP journal on embedded systems*, 2007, 2007.

[EGK+02] John Ellson, Emden Gansner, Lefteris Koutsofios, StephenC. North und Gordon Woodhull. Graphviz— Open Source Graph Drawing Tools. In Petra Mutzel, Michael Jünger und Sebastian Leipert, Hrsg., *Graph Drawing*, Band 2265 der Reihe *Lecture Notes in Computer Science*, Seiten 483–484. Springer, 2002.

[EH04] Torbjörn Ekman und Görel Hedin. Rewritable Reference Attributed Grammars. In Martin Odersky, Hrsg., *ECOOP 2004 – Object-Oriented Programming*, Band 3086 der Reihe *Lecture Notes in Computer Science*, Seiten 147–171. Springer, 2004.

[FR07] Robert France und Bernhard Rumpe. Model-driven Development of Complex Software: A Research Roadmap. In *2007 Future of Software Engineering*, FOSE '07, Seiten 37–54. IEEE, 2007.

[GKvdB10] Arda Goknil, Ivan Kurtev und Klaas van den Berg. Tool Support for Generation and Validation of Traces Between Requirements and Architecture. In *Proceedings of the 6th ECMFA Traceability Workshop*, ECMFA-TW '10, Seiten 39–46. ACM, 2010.

[GLH+92] Robert W. Gray, Steven P. Levi, Vincent P. Heuring, Anthony M. Sloane und William M. Waite. Eli: A complete, flexible compiler construction system. *Communications of the ACM*, 35(2):121–130, 1992.

[Gro09] Richard C. Gronback. Eclipse modeling project: a domain-specific language (DSL) toolkit, 2009.

[Hed00] Görel Hedin. Reference attributed grammars. *Informatica (Slovenia)*, 24(3):301–317, 2000.

[Hed11] Görel Hedin. An introductory tutorial on JastAdd attribute grammars. In *Generative and Transformational Techniques in Software Engineering III*, Seiten 166–200. Springer, 2011.

[HJK+11] Florian Heidenreich, Jendrik Johannes, Sven Karol, Mirko Seifert, Michael Thiele, Christian Wende und Claas Wilke. Integrating OCL and Textual Modelling Languages. In Juergen Dingel und Arnor Solberg, Hrsg., *Models in Software Engineering*, Band 6627 der Reihe *Lecture Notes in Computer Science*, Seiten 349–363. Springer, 2011.

[JBF11] Jean-Marc Jézéquel, Olivier Barais und Franck Fleurey. Model driven language engineering with kermeta. In *Generative and Transformational Techniques in Software Engineering III*, Seiten 201–221. Springer, 2011.

[Kas80] Uwe Kastens. Ordered attributed grammars. *Acta Informatica*, 13(3):229–256, 1980.

[KER99] Stuart Kent, Andy Evans und Bernhard Rumpe. UML Semantics FAQ. In Ana Moreira, Hrsg., *Object-Oriented Technology ECOOP'99 Workshop Reader*, Band 1743 der Reihe *Lecture Notes in Computer Science*, Seiten 33–56. Springer, 1999.

[Knu68] Donald E. Knuth. Semantics of context-free languages. *Mathematical systems theory*, 2(2):127–145, 1968.

[KS00] Juha Kuusela und Juha Savolainen. Requirements Engineering for Product Families. In *Proceedings of the 22Nd International Conference on Software Engineering*, ICSE '00, Seiten 61–69. ACM, 2000.

[KW91] Uwe. Kastens und William M. Waite. An abstract data type for name analysis. *Acta Informatica*, 28(6):539–558, 1991.

[KWB03] Anneke G. Kleppe, Jos Warmer und Wim Bast. The model driven architecture: practice and promise, 2003.

[MHS05] Marjan Mernik, Jan Heering und Anthony M. Sloane. When and how to develop domain-specific languages. *ACM Comput. Surv.*, 37(4):316–344, Dezember 2005.

[MS09] Niklas Mellegård und Miroslaw Staron. A domain specific modelling language for specifying and visualizing requirements. In *The First International Workshop on Domain Engineering, DE@ CAiSE, Amsterdam*, 2009.

[NE00] Bashar Nuseibeh und Steve Easterbrook. Requirements Engineering: A Roadmap. In *Proceedings of the Conference on The Future of Software Engineering*, ICSE '00, Seiten 35–46. ACM, 2000.

[OL07] Manfred Oesterle und Fred Leidig, Hrsg. *Methodisch sichere, schnelle Produktionsanläufe in der Mechatronik (MESSPRO) - Band 2 der Reihe „Schneller Produktionsanlauf in der Wertschöpfungskette"*. VDMA-Verlag, 2007.

[Pan11] R. K. Pandey. Object constraint language (OCL): past, present and future. *SIGSOFT Softw. Eng. Notes*, 36(1):1–4, Januar 2011.

[Par10] Helmut A. Partsch. *Requirements Engineering systematisch*. Springer, 2010.

[Reg13] Björn Regnell. reqT.org - Towards a Semi-Formal, Open and Scalable Requirements Modelling Tool. In Joerg Doerr und Andreas L. Opdahl, Hrsg., *Requirements Engineering: Foundation for Software Quality*, Band 7830 der Reihe *Lecture Notes in Computer Science*, Seiten 112 – 118. Springer, 2013.

[SGBvB12] Yu Sun, Jeff Gray, Karlheinz Bulheller und Nicolaus von Baillou. A model-driven approach to support engineering changes in industrial robotics software. In *Proceedings of the 15th international conference on Model Driven Engineering Languages and Systems*, MODELS'12, Seiten 368–382. Springer, 2012.

[SK03] Carsten Schmidt und Uwe Kastens. Implementation of visual languages using pattern-based specifications. *Software: Practice and Experience*, 33(15):1471–1505, 2003.

[SLL02] Siek Siek, Lee-Quan Lee und Andrew Lumsdaine. *The Boost Graph Library: User Guide and Reference Manual*. Addison-Wesley, 2002.

[SMS13] Richard M. Stallman, Roland McGrath und Paul D. Smith. *GNU Make – A Program for Directing Recompilation*. Free Software Foundation, 2013.

[SSF+07] Jörg Schmittwilken, Jens Saatkamp, W. Forstner, Thomas H. Kolbe und L. Plumer. A semantic model of stairs in building collars. *Photogrammetrie, Fernerkundung, Geoinformation*, 2007(6):415, 2007.

[SVEH07] Thomas Stahl, Markus Völter, Sven Efftinge und Arno Hasse. *Modellgetriebene Softwareentwicklung: Techniken, Engineering, Management*. dpunkt.verlag, 2007.

[TÅJ12] Alfred Theorin, Karl-Erik Årzén und Charlotta Johnsson. Rewriting JGrafchart with Rewritable Reference Attribute Grammars. In *Industrial Track of Software Language Engineering 2012*, September 2012.

[VSM12] Marcos Viera, Doaitse Swierstra und Arie Middelkoop. UUAG Meets AspectAG: How to Make Attribute Grammars First-class. In *Proceedings of the Twelfth Workshop on Language Descriptions, Tools, and Applications (LDTA)*, Seiten 6:1–6:8. ACM, 2012.

[WG84] William M. Waite und Gerhard Goos. *Compiler Construction*. Springer, 1984.

[wsE] Electronic Device Description Language. http://www.eddl.org. letzter Zugriff 11. Dezember 2013.

[wsY] Yakindu Requirements. http://www.yakindu.de/requirements. letzter Zugriff 11. Dezember 2013.

[Zim13] Wolf Zimmermann. Modell-basierte Programmgenerierung und Methoden des Übersetzerbaus - Zwei Seiten derselben Medaille? In Stefan Wagner und Horst Lichter, Hrsg., *Software Engineering (Workshops)*, Band P-215, Seiten 23–25. GI, 2013.