Erweiterung eines Open Source Language Workbenchs um Funktionen zur Migration von Modellen einer domänenspezifischen Sprache

Hauke Wittern

Department Informatik
HAW-Hamburg
Berliner Tor 7
20099 Hamburg
hauke.wittern@haw-hamburg.de

Abstract: Es wurde ein Konzept zur Generierung von Migrationsskripten bei der Evolution einer domänenspezifischen Sprache entwickelt. Das Konzept ermöglicht es zu erkennen, welche Transformationen unresolvable sind und damit das Eingreifen einer Person erfordern. Die Implementierung des Konzepts basiert auf dem Spoofax Language Workbench, welcher Open Source ist. Dieses Paper präsentiert die Erfahrungen des Autors bei der Implementierung.

1 Einleitung

Eine domänenspezifische Sprache (Domain Specific Language, DSL) ist eine auf eine bestimmte Domäne zugeschnittene Programmiersprache [FP11, Hu98]. Im Gegensatz zu universell einsetzbaren General Purpose Languages (GPL) hat eine DSL eine eingeschränkte Ausdruckskraft [FP11]. Wegen dieser Eigenschaften erleichtern DSLs, bestimmte Teile von Software-Systemen zu verstehen. Ebenso ist die Implementierung bestimmter System-Artefakte mit einer DSL üblicherweise einfacher, schneller und weniger fehleranfällig als mit einer GPL [FP11, Hu98].

Die Vorteile einer DSL beruhen darauf, dass sie besonders geeignete Sprachkonstrukte zur Formulierung von Problemen der Domäne hat. Ein wesentlicher Faktor ist dabei, dass eine DSL das Vokabular von Domänenexperten verwendet. Außerdem erleichtert die konkrete Syntax einer DSL deren Verwendung. Wenn eine DSL unabhängig von anderen Sprachen als externe DSL implementiert wird, kann die Syntax vollständig maßgeschneidert werden [FP11]. Bei internen DSLs kann die Syntax jedoch nicht ohne Einschränkungen gestaltet werden. Denn eine interne DSL baut immer auf einer vorhandenen GPL auf und ist an deren Syntax gebunden [FP11, BH12].

Der Erfolg einer DSL hängt sehr von einer angemessenen Werkzeugunterstützung ab [KV10]. Der Bedarf von sprachspezifischen Werkzeugen ist deshalb ein Nachteil von externen DSLs. Interne DSLs können mit all den Werkzeugen entwickelt werden, die für die

verwendete Sprache verfügbar sind. Hingegen wird für externe DSLs, wegen der maßgeschneiderten Syntax, zumindest ein eigener Parser benötigt. Weitere Werkzeuge, welche die Entwicklung von DSLs und die Arbeit mit DSLs erleichtern, müssen speziell für die Syntax einer externen DSL entwickelt oder angepasst werden. Solche Werkzeuge können beispielsweise Debugger und Editoren für DSL-Programme sein. Sogenannte Language Workbenches decken einen Großteil dieser Funktionalität ab. Ein Language Workbench ist eine integrierte Entwicklungsumgebung (Integrated Development Environment, IDE) sowohl zur Entwicklung von externen DSLs als auch zur Verwendung der DSLs [FP11].

So wie auch bei IDEs für General Purpose Languages kann unter Umständen Bedarf für Anpassungen oder Erweiterungen von Language Workbenches bestehen. Zum einen ist der Funktionsumfang von Language Workbenches sehr unterschiedlich und deckt nicht immer alle Anforderungen ab. Zum anderen kann es nötig sein, einen Language Workbench an den eigenen Entwicklungsprozess und die vorhandenen Werkzeuge anzupassen. Solche Erweiterungen und Anpassungen können es eventuell erfordern, Änderungen am Quelltext des Workbenchs vorzunehmen. Das ist natürlich nur möglich, wenn der Quelltext Open Source ist.

Dieses Paper präsentiert die Erfahrungen des Autors bei der Erweiterung des Open Source Language Workbenchs Spoofax. Spoofax soll mit Funktionen zur gekoppelten Evolution von DSLs und Modellen der DSLs erweitert werden. Durch Migration wird die Konformität von vorhandenen Modellen mit einer weiterentwickelten DSL-Version wiederhergestellt. Das Konzept dazu wurde in [Wi13] erläutert. Das Konzept wurde mit der Zielsetzung entwickelt, automatisiert erkennen zu können, wann ein Entwickler bei der Migration von Modellen manuell eingreifen muss. Die hier behandelte Erweiterung von Spoofax setzt jenes Konzept bei der Generierung von Skripten zur Migration von Modellen um. Die Implementierung soll später bei der Evaluierung des Konzepts verwendet werden.

Das Paper ist wie folgt gegliedert: Abschnitt 2 gibt zunächst einen Überblick über die zugrunde liegenden verwandten Arbeiten bezüglich der Evolution von DSLs. Abschnitt 3 skizziert das Konzept zur Migration von Modellen. Anschließend wird in Abschnitt 4 die grundsätzliche Entscheidung für Open Source und die Auswahl des Spoofax Language Workbenchs begründet. In Abschnitt 5 werden Implementierungsaspekte erläutert und Probleme genannt, die bei der Implementierung auftraten. Abschließend folgt in Abschnitt 6 ein Fazit.

2 Evolution domänenspezifischer Sprachen

Solange eine domänenspezifische Sprache (DSL) verwendet wird, findet üblicherweise eine Evolution der Sprache statt [Fa05, FP11]. Zum einen gewinnen die Entwickler der DSL mit zunehmender Zeit an Domänenerfahrung [Gr07, DRIP11]. Zum anderen kann in der Domäne selbst eine Weiterentwicklung stattfinden [PJ07, Ge08, CP10]. Ebenso können sich die Anforderungen an das System ändern [MJ82, KVW10, Ge08] und es kann eine technologische Weiterentwicklung stattfinden [HVW11]. Diese Veränderungen können es

erfordern, eine DSL daran anzupassen [BD07, HVW11, KVW10, Ge08, FP11, PJ07].

Die Evolution einer DSL kann dazu führen, dass mit der DSL verfasste Modelle ungültig werden [Kr11, Ro12, Ge08, Ci08, HVW11, HBJ09, HBJ08, Be07, DRIP11, Wa07, FP11]. Die vorhandenen Modelle müssen dann an die neue DSL Version angepasst werden, was als Migration bezeichnet wird [HBJ09, Ci08, DRIP11, FP11, Ge08, HVW11, HBJ08, Be07].

Die Veränderungen an einer DSL können entweder durch Vergleich zweier DSL-Versionen ermittelt werden [Ci08, Be07, DRIP11], explizit vom Benutzer als Abweichungsmodell angegeben werden [Na09] oder als Sequenz von Transformationsoperatoren angegeben werden [Wa07, HBJ09, HVW11]. In diesem Paper wird der Operator-Ansatz verfolgt.

Im Operator-basierten Ansatz findet üblicherweise eine gekoppelte Evolution von DSL-Metamodell und Modellen statt (Co-Evolution, Coupled Evolution) [VV08, Ci08, HBJ09, HRW10, HVW11]. Eine Metamodell-Transformation wird mit der zugehörigen Migration über den Operator gekoppelt (Coupled Operator) [HVW11]. In [Be07, Ci08, HBJ09, HRW10, HVW11] wurden zahlreiche solcher Operatoren genannt. Die Operatoren sind jeweils spezifisch für das Metametamodell mit dem die DSL entwickelt wird.

Manche Metamodell-Transformationen haben nur Auswirkungen auf das Metamodell, sodass keine Migration von Modellen notwendig ist. Solche Transformationen werden als *non-breaking* [Ci08, Be07] oder *Model-preserving* [HVW11] bezeichnet. Umgekehrt werden Metamodell-Transformationen die eine Migration von Modellen erfordern als *breaking* [Ci08, Be07] oder *Model-migrating* [HVW11] bezeichnet. Von vielen Metamodell-Transformationen kann die notwendige Migration automatisch abgeleitet werden. Solche Transformationen werden als *resolvable* bezeichnet [Be07, Ci08]. In einigen Fällen kann von einer Metamodell-Transformation jedoch nicht automatisch abgeleitet werden, wie die Migration ablaufen muss. Die Transformation ist dann *unresolvable* [Be07, Ci08]. Und es muss dann eine Person festlegen, wie die Migration ablaufen muss [Ci08, Be07].

Wie eine Migration ablaufen muss, kann auf verschiedene Weise festgelegt werden. Die nötigen Migrationsaktionen können anhand einer manuellen Zuordnung von Elementen der alten DSL-Version auf Elemente der neuen Version ermittelt werden [Na09]. Eine andere Möglichkeit ist, für jede Art von Änderung eine Regel zu formulieren [DRIP11, HBJ09]. In diesem Paper wird der zuletzt genannte Ansatz verwendet.

3 Konzept zur gekoppelten Evolution von DSLs und Modellen

COPE [HBJ09] beziehungsweise Edapt[1] und EMFMigrate [DRIP11] sind zwei Ansätze zur gekoppelten Evolution von DSLs und Modellen. Bei beiden beschränkt sich die Behandlung von Änderungen, die unresolvable sind, auf die Verwendung von Standardwerten. Das Standardverhalten kann zwar überschrieben werden, es gibt aber keine Möglichkeit zu erkennen, wann dies nötig ist. Deshalb wurde in [Wi13] zunächst ein entsprechendes Konzept zur Generierung von Migrationsskripten entwickelt, um es später auf Basis ei-

[1] COPE wird mittlerweile unter dem Namen Edapt weiterentwickelt, siehe http://www.eclipse.org/edapt/

nes Language Workbenchs zu implementieren. Ein Migrationsskript ist ein Computerprogramm. Es besteht aus einer Sequenz von Anweisungen, welche die vorhandenen Modelle einer DSL so umwandeln, dass sie gültige Modelle der neuen DSL Version werden.

Das Konzept aus [Wi13] ermöglicht es automatisiert zu erkennen, welche Modellelemente mit vordefinierten Aktionen automatisch migriert werden können und, wann ein Entwickler eingreifen muss, indem er spezielle Aktionen zur Migration bestimmter Elemente implementiert. Weiterhin ermöglicht das Konzept automatisiert zu erkennen, welche Modell-Elemente überhaupt migriert werden müssen und welche nicht. Dadurch kann die Anzahl der zu implementierenden speziellen Migrations-Aktionen für Transformationen die unresolvable sind minimiert werden. Unnötiger Aufwand bleibt den Entwicklern damit erspart. Nachdem die Entwickler alle Migrationsaktionen implementiert haben, kann jedes Modell vollautomatisch migriert werden.

3.1 Grundsätzliche Funktionsweise

Der Ausgangspunkt für die Generierung eines Migrationsskriptes ist eine Sequenz von Transformationen des DSL-Metamodells. Eine einzelne Transformation wird durch Anwendung eines Transformationsoperators auf das Metamodell der DSL durchgeführt. Die Transformationssequenz repräsentiert die Evolution einer DSL von einer alten zu einer neuen Version.

Nach dem Prinzip der gekoppelten Evolution sind Metamodell-Transformationsoperatoren mit der zugehörigen Migration gekoppelt [VV08, Ci08, HBJ09, HRW10, HVW11]. Für jeden Transformationsoperator gibt es eine Migrations-Definition. Diese spezifiziert, unter welchen Bedingungen welche Modellelemente mit welchen Aktionen migriert werden müssen. Die Bedingungen beziehen sich auf Eigenschaften des ursprünglichen Metamodells oder des weiterentwickelten Metamodells. Zum Beispiel kann eine Bedingung für die Instanziierung eines Modellelements sein, dass das Modellelement obligatorisch ist. Listing 1 zeigt die Migrationsdefinition des *CreateAttribute*-Operators. Bei diesem Operator ist eine Migration nur notwendig, wenn das Attribut obligatorisch ist. Dies wird mit der Bedingung `LowerBound > 0` in Zeile 7 spezifiziert.

Listing 1: Migrationsdefinition des CreateAttribute-Operators

```
1 class CreateAttribute : Transformation {
2   FeatureQualifier AttributeQualifier;
3   Cardinality LowerBound;
4   Cardinality UpperBound;
5
6   override void EvaluateMigrationDefinition(MigrationContext context) {
7     if (LowerBound > 0) {
8        var cvp = context.NewCustomFeatureValueProvider();
9       // hole das Attribut aus dem evolvierten Metamodel:
10      var attribute = (Attribute)context.New[AttributeQualifier];
11      context.MigrateEachInstance(attribute,
            MigrationAction.SetValue(cvp));
12    }
13  }
```

Während der Generierung eines Migrationsskripts wird der Reihe nach für jede einzelne Transformation die Migrations-Definition des zugehörigen Operators ausgewertet. Es werden diejenigen Aktionen aus der Migrations-Definition in das Skript eingetragen, für die alle Bedingungen erfüllt sind.

3.2 Entfernung unnötiger Migrations-Aktionen

Nicht alle aus den Migrations-Definitionen gewonnenen Aktionen sind immer notwendig, um die Gültigkeit eines Modells mit der neuen DSL-Version herzustellen. Unter Umständen kann sich aus der Sequenz von Transformationen ergeben, dass für ein Metamodellelement keine Instanz in einem Modell vorhanden sein kann. Beispielsweise kann ein Attribut nicht instanziiert worden sein, wenn die besitzende Klasse durch eine Transformation zuvor erzeugt wurde und es keine weitere Transformation gibt, durch die eine Instanz der Klasse obligatorisch wurde. Aktionen zur Migration von nicht existenten Modellelementen sind überflüssig und werden nicht in das Migrationsskript eingetragen. Umgekehrt muss ein Migrationsskript aber Anweisungen zur Migration von allen betroffenen Elementen enthalten, die potenziell in einem Modell existieren können.

Auf die potenzielle Existenz von Modellelementen beziehungsweise deren Nichtexistenz wird anhand von Vorbedingungen der Transformationssequenz geschlossen. Für jeden Transformationsoperator sind Vor- und Nachbedingungen spezifiziert. Abbildung 3.2 zeigt als Beispiel die Vor- und Nachbedingungen des *CreateAttribute*-Operators. Die Vor- und Nachbedingungen der Operatoren werden verwendet, um die Vorbedingungen der Transformationssequenz bis zu der Transformation zu ermitteln, für welche die nötigen Migrationsaktionen gesucht werden. Die Berechnung erfolgt iterativ auf Basis der Vor- und Nachbedingungen des aktuellen Operators und des vorherigen Schritts.

$$
\begin{aligned}
PRE_{CreateAttribute} &= \{classExists(c), \neg featureExists(c, a), \\
&\qquad isExistingPrimitiveType(t)\} \\
POST_{CreateAttribute} &= \{ownedAttributeExists(c, a)\}
\end{aligned}
$$

Abbildung 1: Die Vor- und Nachbedingungen des CreateAttribute-Operators

3.3 Migration von unresolvable Transformationen

Das Entfernen unnötiger Migrations-Aktionen vermeidet unnötigen Entwicklungsaufwand bei Metamodell-Transformationen, die unresolvable sind. Sollte eine Transformation unresolvable sein, spezifiziert die Migrations-Definition, dass eine benutzerdefinierte Aktion

notwendig ist. Eine Referenz auf die benutzerdefinierte Aktion wird in das Migrationsskript eingetragen. Die Aktion selbst muss später von einem Entwickler implementiert werden. Um dies zu erleichtern, kann ein Stub generiert werden. Der zuvor bereits genannte *CreateAttribute*-Operator ist unresolvable, sofern das erzeugte Attribut obligatorisch ist. Die in diesem Fall notwendige benutzerdefinierte Aktion ist die Berechnung des Attributwertes. Die Migrationsdefinition in Listing 1 spezifiziert deshalb in Zeile 8, dass ein neuer Value-Provider erzeugt wird. Dieser wird anschließend in Zeile 11 zum setzten der Attributwerte verwendet.

4 Entscheidung für Open Source und Auswahl des Language Workbenchs

Kein verfügbares Werkzeug unterstützt die gekoppelte Evolution von DSLs und Modellen so, wie sie im vorherigen Abschnitt beschrieben wurde. Es wurde deshalb ein geeigneter Language Workbench gesucht, welcher mit dem Konzept zur Migration erweitert werden kann. Dieser Language Workbench muss alle wichtigen funktionalen Anforderungen bei der Entwicklung und Verwendung von DSLs erfüllen.

Infrage kamen nur Open Source Language Workbenches. Denn es war absehbar, dass das Migrationskonzept nur mit Anpassungen des Quelltextes umgesetzt werden kann. Ein eventuell vorhandener Erweiterungsmechanismus wäre wohl nicht ausreichend. Ein Erweiterungsmechanismus ist nur dann geeignet, wenn dessen Entwickler die Art der nötigen Erweiterungen vorhergesehen haben und deren Unterstützung implementiert haben. Die Implementierung des Konzepts zur Migration erfordert einen invasiven Eingriff in den Language Workbench. Es muss nicht nur Funktionalität hinzugefügt werden. Das Konzept hat auch einen Einfluss auf den Modellierungsprozess. Dies erfordert Anpassungen des Workbenchs, die von den Entwicklern des Workbenchs vermutlich nicht vorhersehbar waren. Unabhängig vom Vorhandensein eines Erweiterungsmechanismus wurde Open Source als Anforderung festgelegt, weil nicht alle nötigen Erweiterungen im Voraus planbar sind. Die Entscheidung für Open Source fiel also, um die Erweiterbarkeit sicherzustellen.

Mit der Anforderung, dass der Language Workbench Open Source sein muss, geht ein weitere wichtige nichtfunktionale Anforderung einher. Der Language Workbench muss von seinen Entwicklern weiterhin gewartet und weiterentwickelt werden.

Bei der Auswahl des Language Workbenchs spielten darüber hinaus einige funktionale Anforderungen eine Rolle. Das Konzept aus Abschnitt 3 ist prinzipiell für textuelle und grafische DSLs geeignet. Die vom Autor verwendeten DSLs sind jedoch alle textuell. Der Workbench muss deshalb textuelle DSLs unterstützen. Für die textuellen DSLs muss es einen Editor geben, der Syntaxhighlighting unterstützt. Das Syntaxhighlighting muss anpassbar sein. Syntaktische und semantische Fehler müssen im Text-Editor hervorgehoben werden können. Außerdem müssen DSL-Programme debuggt werden können. Wünschenswert ist, DSL-Definitionen automatisch testen zu können und Modelle refaktorieren zu können.

Language Workbenches sind noch relativ neu und häufig noch unausgereift. Deshalb kamen letztendlich nur zwei infrage, welche die Anforderungen erfüllen: Spoofax[1] und Xtext[2]. Es wurde Spoofax gewählt, weil dieser Workbench benutzerdefinierte Refaktorierungen unterstützt [JV13]. Die benutzerdefinierten Refaktorierungen spielen bei der Implementierung im nächsten Abschnitt eine wichtige Rolle.

5 Entwurf und Implementierung

Es wurde entschieden, das Konzept zur Migration von Modellen aus Abschnitt 3 auf Basis des Spoofax Language Workbenchs zu implementieren. Die Wiederverwendung von einigen Komponenten von Spoofax verspricht, bei der Implementierung des Konzepts erheblichen Entwicklungsaufwand zu ersparen. Um die Funktionen von Spoofax zur Migration von Modellen zu verwenden, sind jedoch einige Anpassungen des Quellcodes notwendig. Im Folgenden wird erläutert, welche Funktionalität von Spoofax wiederverwendet werden kann und welche Faktoren die Anpassung des Spoofax Language Workbenchs erschweren.

5.1 Migration von Modellen als Refaktorierung

Mit Spoofax können die Benutzer Modelle automatisiert refaktorieren [JV13]. Die Benutzer können eigene Refaktorierungen für ihre DSLs definieren [JV13]. Bei einer Refaktorierung mit Spoofax wird eine strukturelle Transformation auf dem abstrakten Syntaxbaum (abstract Syntax Tree, AST) eines Modells ausgeführt [JV13].

Die Migration eines Modells hat viele Gemeinsamkeiten mit einer Refaktorierung eines Modells. Auch bei einer Migration wird ein Modell, um es verarbeiten zu können, zunächst geparst. Das Ergebnis des Parsens ist ein abstrakter Syntaxbaum. Dies ist eine Datenstruktur, die konform zur abstrakten Syntax der alten DSL Version ist. Der abstrakte Syntaxbaum wird anschließend transformiert, sodass er konform zu der abstrakten Syntax der neuen DSL-Version ist. Abschließend wird aus der Datenstruktur DSL-Code erzeugt, der konform zur konkreten Syntax der neuen DSL-Version ist. Eine Modell-Refaktorierung und eine Modell-Migration unterscheiden sich im Wesentlichen also in der DSL Version, zu der das Ergebnis konform ist. Bei einer Refaktorierung sind Eingabe und Ausgabe konform zur selben DSL. Bei einer Migration ist die Eingabe konform zu einer anderen DSL-Version als die Ausgabe. Abbildung 2 veranschaulicht den Ablauf bei der Migration eines Modells. Die Implementierung führt eine zusätzliche Transformation der ASTs in generische Modelle durch. Die generischen Modelle sind konform zu dem Metametamodell, welches die Migrationskomponente verwendet. Nur so ist es möglich, dass die Migrationskomponente Modelle beliebiger domänenspezifischer Sprachen migrieren kann.

Wegen der Gemeinsamkeiten des Refaktorierens und Migrierens von Modellen sieht der Entwurf vor, bei der Implementierung der Migrationsfuntionalität auf den Refaktorie-

[1] http://www.spoofax.org/
[2] http://www.eclipse.org/Xtext/

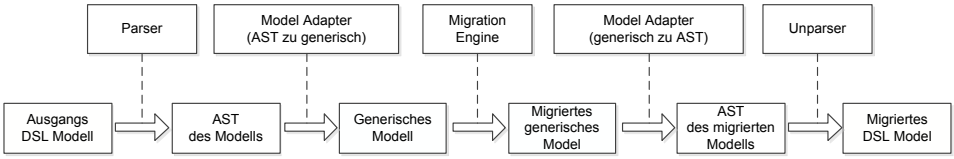

Abbildung 2: Der Ablauf bei der Migration eines Modells. Die Pfeile repräsentieren Modelltransformationen von einer Repräsentation zu einer anderen. Die Komponenten, welche die jeweiligen Transformationen implementieren, sind mit einer gestrichelten Linie verbunden.

rungsfunktionen von Spoofax aufzubauen. Ein Migrationsskript soll im Prinzip wie eine Refaktorierung ausgeführt werden. Die Verwendung von der vorhandenen Funktionalität erspart es, einige Komponenten selbst zu entwickeln. Von besonderem Nutzen sind die Unparsing-Funktionen in Spoofax, mit denen aus ASTs DSL-Code erzeugt wird. Diese Funktion wird nach der Migration eines Modells benötigt. Die Entwicklung eines Unparsers [Ro97] beziehungsweise eines Unparser-Generators ist komplex und wäre mit erheblichem Aufwand verbunden.

Die Implementierung von Migrationsskripten als Refaktorierungen erfordert jedoch eine wesentliche Änderung am Spoofax Language Workbench. Spoofax unterstützt es nicht, zwei verschiedene Versionen einer DSL gleichzeitig zu Nutzen. Dies ist jedoch zur Migration von Modellen notwendig. Denn es wird die alte Version beim Einlesen des Modells und die neue Version bei der Ausgabe des migrierten Modells benötigt. Die Implementierung erfordert Änderungen am Quelltext von Spoofax.

5.2 Schwierigkeiten bei der Implementierung

Die Implementierung der Migrations-Funtionalität und die damit verbundenen Änderungen am Quelltext von Spoofax sind mit einigen Problemen verbunden. Anders als ursprünglich vorgesehen wurde von Windows zu Linux als Betriebssystem der Entwicklungsumgebung gewechselt. Spoofax ist in Eclipse integriert und in Java implementiert. Dennoch ist die Kompilierung des Quelltextes an Unix-basierte Systeme gebunden. Denn Spoofax verwendet den Package-Manager Nix, welcher nur für Unix verfügbar ist [NIX].

Ein weiteres Problem ist die unzureichende Dokumentation von Spoofax. Der Quelltext ist kaum dokumentiert. Und es ist gar keine Dokumentation der Architektur und der Komponenten verfügbar. Die Komponenten und Unterprojekte sind außerdem sehr zahlreich. Es ist deshalb äußerst schwer, den Zweck der Komponenten nachzuvollziehen und deren Zusammenspiel zu verstehen. Neben Java sind einige Komponenten von Spoofax in speziellen DSLs programmiert worden. Diese DSLs sind ebenfalls undokumentiert und ohne Wissen über Interna von Spoofax kaum zu verstehen. Letztendlich erschweren diese Faktoren, Änderungen am Quelltext vorzunehmen.

Gleichfalls erschwert die fehlende Dokumentation, die Komponenten von Spoofax zu verwenden. Die Implementierung der Migrations-Funtionalität kann im Wesentlichen ohne

Änderungen am Quelltext durchgeführt werden, indem die vorhandenen Komponenten verwendet werden und neue hinzugefügt werden. Doch wie schon von Niklaus Wirth kritisiert behindert die fehlende Spezifikation auf den Komponenten aufzubauen:

> "[...] the open source movement ignores and actually hinders the perception of one of the most important ideas in designing complex systems, namely their partitioning in modules, and their formation as an orderly hierarchy of modules. The key idea is that the designer of a module using (importing) other modules, need not know any of the source code of them. He must rely solely on a clear specification of the interface of these modules. Most of the time, modules lack such clear, complete, and unambiguous interface specifications." [Mo09]

Bei Spoofax trifft ein zentraler Punkt von Niklaus Wirths Kritik zu: Die Entwickler müssen den Quelltext kennen, um Spoofax zu erweitern. Das größte Problem ist aber, dass man das Zusammenspiel der Komponenten bloß anhand des Quelltextes nur schwer überblicken kann. Anhand des Quelltextes kann man die Funktionsweise einer einzelnen Komponente noch nachvollziehen. Das Zusammenspiel der Komponenten geht daraus aber nicht hervor. Um das komplexe System der Komponenten von Spoofax zu verstehen, muss man umfassende Kenntnis über den Quelltext der Komponenten haben.

5.3 Stand der Implementierung

Ein erster Prototyp, der das Konzept aus Abschnitt 3 implementiert, ist fertiggestellt. Migrationsskripte können mit dem Prototyp generiert und Modelle damit migriert werden. Allerdings können Modelle momentan nur dann migriert werden, wenn die konkrete Syntax der neuen DSL-Version abwärtskompatibel zu der vorherigen Version ist. Denn die nötigen Anpassungen, damit Spoofax die alte und die neue DSL Version in der selben Instanz laden kann, ist wegen der zuvor in 5.2 genannten Probleme noch nicht abgeschlossen. Momentan wird deshalb die selbe DSL-Version zum Parsen von Modellen und zum Unparsen der migrierten Modelle verwendet. Ebenfalls ist die Integration in die Benutzungsschnittstelle von Spoofax noch nicht abgeschlossen.

6 Fazit

Es wurde ein neuartiges Konzept zur Migration von Modellen bei der Evolution einer DSL entwickelt. Der Spoofax Language Workbench wurde als Basis für die Implementierung des Konzepts ausgewählt. Der Spoofax Language Workbench wurde unter anderem deshalb ausgewählt, weil er Open Source ist. Denn die Implementierung erfordert Anpassungen am Language Workbench, die ohne Änderungen am Quelltext nicht möglich sind.

Migrationsskripte wurden als Modell-Refaktorierungen implementiert. Damit wurde die

Entwicklung eines Unparsers gespart. Jedoch wurde die weitere Implementierung durch die unzureichende Dokumentation erschwert. Der Zweck der einzelnen Komponenten und wie sie zusammenwirken, ist ohne umfassendes Studium des Quelltextes nicht zu erkennen. Dies ist jedoch nicht im Sinne einer modularen Programmierung. Das Prinzip des Information Hiding wird verletzt. Wären die Schnittstellen ausreichend dokumentiert, müsste man sich den Quelltext nur dort ansehen, wo Änderungen vorgenommen werden sollen. Der gesamte Quelltext wird hier eigentlich nur zum Kompilieren benötigt. Angesichts der aufgetretenen Probleme wird zurzeit erwogen, die Implementierung auf Basis von Xtext statt Spoofax fortzusetzen. Xtext hat eine größere Entwicklergemeinde und eine Dokumentation ist verfügbar.

Mit der Implementierung wird das in Abschnitt 3 genannte Konzept evaluiert werden. Die Evaluierung findet im Rahmen des Projekts Smart Power Hamburg[2] mit DSLs in der Energie und Smart Grid Domäne statt. Es wird evaluiert werden, wie das Migrationskonzept im Vergleich zu anderen Ansätzen und der manuellen Erzeugung von Migrationsskripten abschneidet. Darüber hinaus ist eine Weiterentwicklung des Ansatzes denkbar, indem auch Nachbedingungen direkt zur Entfernung unnötiger Migrationsaktionen verwendet werden. Momentan werden diese nur zur Berechnung von Vorbedingungen von Transformationssequenzen verwendet.

Literatur

[BD07] Bosch, J.; Dittrich, Y.: Domain-Specific Languages for a Changing World. 2007.

[Be07] Becker, S. et al.: A Process Model and Classification Scheme for Semi-Automatic Meta-Model Evolution. In *Proc. 1st Workshop MDD, SOA und IT-Management (MSI'07)*, Seiten 35–46. GiTO-Verlag, 2007.

[BH12] Barringer, H.; Havelund, K.: Internal versus External DSLs for Trace Analysis. In Khurshid, S.; Sen, K., Hrsg., *Runtime Verification*, Jgg. 7186 of *Lecture Notes in Computer Science*, Seiten 1–3. Springer Berlin Heidelberg, 2012.

[Ci08] Cicchetti, A. et al.: Automating Co-evolution in Model-Driven Engineering. In *Proceedings of the 12th International IEEE Enterprise Distributed Object Computing Conference*, EDOC '08, Seiten 222–231, Washington, DC, USA, 2008. IEEE Computer Society.

[CP10] Cazzola, W.; Poletti, D.: DSL evolution through composition. In *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, RAM-SE '10, Seiten 6:1–6:6, New York, NY, USA, 2010. ACM.

[DRIP11] Di Ruscio, D.; Iovino, L.; Pierantonio, A.: What is needed for managing co-evolution in MDE?. In *Proceedings of the 2nd International Workshop on Model Comparison in Practice*, IWMCP '11, Seiten 30–38, New York, NY, USA, 2011. ACM.

[Fa05] Favre, J.-M.: Languages evolve too! Changing the Software Time Scale. In *Proceedings of the Eighth International Workshop on Principles of Software Evolution*, IWPSE '05, Seiten 33–44, Washington DC, USA, 2005. IEEE Computer Society.

[2] http://www.smartpowerhamburg.de, gefördert durch BMWi via EnEff:Wärme

[FP11] Fowler, M.; Parsons, R.: *Domain-specific languages*. Addison-Wesley, Upper Saddle River, N.J, 2011.

[Ge08] Geest, G. d.: Building a framework to support Domain-Specific Language evolution. Diplomarbeit, Delft University of Technology, Delft, the Netherlands, 2008.

[Gr07] Gruschko, B.: Towards synchronizing models with evolving metamodels. In *Proc. Int. Workshop on Model-Driven Software Evolution held with the ECSMR*, 2007.

[HBJ08] Herrmannsdoerfer, M.; Benz, S.; Juergens, E.: Automatability of Coupled Evolution of Metamodels and Models in Practice. In Czarnecki, K. et al., Hrsg., *Lecture Notes in Computer Science*, Seiten 645–659. Springer, Berlin, Heidelberg, 2008.

[HBJ09] Herrmannsdoerfer, M.; Benz, S.; Jürgens, E.: COPE - Automating Coupled Evolution of Metamodels and Models. In *ECOOP 2009 – Object-Oriented Programming*, Jgg. 5653 of *Lecture Notes in Computer Science*, Seiten 52–76. 2009.

[HRW10] Herrmannsdoerfer, M.; Ratiu, D.; Wachsmuth, G.: Language Evolution in Practice: The History of GMF. In Hutchison, D. et al., Hrsg., *Software Language Engineering*, Jgg. 5969 of *Lecture Notes in Computer Science*, Seiten 3–22. Springer, Berlin, Heidelberg, 2010.

[Hu98] Hudak, P.: Modular Domain Specific Languages and Tools. In *Proceedings of the 5th International Conference on Software Reuse*, ICSR '98, Seiten 134–143, Washington DC, USA, 1998. IEEE Computer Society.

[HVW11] Herrmannsdoerfer, M.; Vermolen, S.; Wachsmuth, G.: An extensive catalog of operators for the coupled evolution of metamodels and models. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, Seiten 163–182, Berlin, Heidelberg, 2011. Springer-Verlag.

[JV13] Jonge, M. d.; Visser, E.: Implementing Refactorings in the Spoofax Language Workbench. Technical report, 2013. http://swerl.tudelft.nl/twiki/pub/Main/TechnicalReports/TUD-SERG-2013-008.pdf.

[Kr11] Kruse, S.: On the Use of Operators for the Co-Evolution of Metamodels and Transformations. In *ME 2011 - Models and Evolution, Workshop at ACM/IEEE 14th International Conference on Model Driven Engineering Languages and Systems*, Seiten 54–63. 2011.

[KV10] Kats, L. C.; Visser, E.: The spoofax language workbench: rules for declarative specification of languages and IDEs. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications (OOPSLA '10)*, Seiten 444–463. ACM, New York, NY, USA, 2010.

[KVW10] Kats, L. C.; Visser, E.; Wachsmuth, G.: Pure and declarative syntax definition: paradise lost and regained. In *Proceedings of the ACM international conference on Object oriented programming systems languages and applications - OOPSLA '10*, Seiten 918–932. ACM Press, 2010.

[MJ82] McCracken, D. D.; Jackson, M. A.: Life cycle concept considered harmful. *ACM SIGSOFT Software Engineering Notes*, 7(2):29–32, 1982.

[Mo09] Morris, R.. Niklaus Wirth: Geek of the Week. Online, 07 2009. https://www.simple-talk.com/opinion/geek-of-the-week/niklaus-wirth-geek-of-the-week/.

[Na09] Narayanan, A. et al.: Automatic Domain Model Migration to Manage Metamodel Evolution. In *Proceedings of the 12th International Conference on Model Driven Engineering Languages and Systems*, MODELS '09, Seiten 706–711, Berlin, Heidelberg, 2009. Springer-Verlag.

[NIX] The Nix project: *About Nix*. http://nixos.org/nix/.

[PJ07] Pizka, M.; Jürgens, E.: Tool-Supported Multi-Level Language Evolution. In *Proceedings of SVM'07: Software and Services Variability Management Workshop Concepts, Models and Tools*, 2007.

[Ro97] Rozenberg, G.: *Handbook of Graph Grammars and Computing by Graph Transformation, Vol 1: Foundations*. World Scientific, 1997.

[Ro12] Rose, L. M. et al.: Graph and model transformation tools for model migration. *Software & Systems Modeling*, Seiten 1–37, 2012.

[VV08] Vermolen, S.; Visser, E.: Heterogeneous Coupled Evolution of Software Languages. In *Proceedings of the 11th international conference on Model Driven Engineering Languages and Systems*, MoDELS '08, Seiten 630–644, Berlin, Heidelberg, 2008. Springer.

[Wa07] Wachsmuth, G.: Metamodel Adaptation and Model Co-adaptation. In Ernst, E.; Wachsmuth, G., Hrsg., *Proceedings of the 21st European Conference on Object-Oriented Programming (ECOOP'07)*, Jgg. 4609, Seiten 600–624. Springer-Verlag, 2007.

[Wi13] Wittern, H.: Determining the necessity of human intervention when migrating models of an evolved DSL. In *Proceedings of the 17th IEEE International EDOC Conference Workshops (EDOCW 2013)*, 2013. Im Druck.