Model Based Development of Data Integration in Graph Databases using Triple Graph Grammars

Seminar zu Graphtechnologien: Graphdatenbanken, Graphalgorithmen, Graphtransformation

Lei Xu Christian Konrad

Sommersemester 2020

Zusammenfassung

Graphdatenbanken sind hochskalierbare, schemalose Datenbanken, welche ein einfaches Interface bereitstellen, um große Mengen von Daten aus verschiedenen Quellen zu integrieren. Wenn ähnliche Daten aus heterogenen Quellen - zum Beispiel relationale Datenbanken - integriert werden sollen, wird die Integrität derer Schemata aufgrund der schemalosen Natur der Graphdatenbanken in der Regel innerhalb der Anwendungsschicht sichergestellt. Unter der Verwendung von Triple-Graph-Grammatiken kann die Integration der Schemata bereits auf Modelebene beschrieben werden. Daten aus unterschiedlichen Quellen können in ein einzelnes Zielschema integriert werden, indem `Gremlin`-Code aus Regeln, die mittels einer Triple-Graph-Grammatik beschrieben wurden, generiert wird. Der beschriebene Ansatz erlaubt die Verwendung von Forward-Regeln einer Triple-Graph-Grammatik zur Generierung von `Gremlin`-Anfragen und -Kommandos, um einen Quellgraphen in *neo4j* in ein Zielschema initial zu überführen. Das endgültige Ziel der Forschung soll es sein, eine Datenintegration vollständig auf Modelebene beschreiben zu können.

1 Einleitung

Unter *Datenintegration* versteht man die Kombination von Daten aus unterschiedlichen Quellen in eine vereinheitlichte Darstellung (Schema) [1]. Die Schemata dieser Daten sind in der Regel strukturell heterogen, während die Besitzer der Datenquellen ihre Autonomie bewahren möchten. Dies können Organisationen sein wie Unternehmen, die ihre Daten teilen oder eigene Dateninseln auflösen möchten, Forschungsgruppen, die gemeinsam an wissenschaftlichen Großprojekten arbeiten, deren Daten aber unabhängig voneinander generiert werden; aber auch Informationsbeschaffungssysteme wie Suchmaschinen profitieren von der

Integration der unzähligen frei verfügbaren Datenquellen des Internets [2]. Eine Datenintegration löst diese Heterogenität auf, während sie aber die Autonomie der Daten-Besitzer bewahrt. Organisationen können Ihre Daten teilen und in ihre eigenen Schemata integrieren.

Graphdatenbanken sind schemalose, skalierbare Datenbanken, die für große Mengen semi-strukturierter Daten geeignet sind und einfache Benutzerschnittstellen bereitstellen [3]. Sie basieren auf einem Graph-Datenmodell und können daher häufig fachliche Modelle im Originalschema in das logische Modell abbilden [4]. Damit sind sie für Modellierungsanforderungen vorgebender Domänen oft besser geeignet als relationale Datenbanken - eine hervorragende Grundlage für Datenintegrationen. Bisherige Ansätze zur Datenintegration müssen aber überarbeitet werden, um den Ansprüchen an die Flexibilität von Graphdatenbanken und der repräsentierten Schemata gerecht zu werden.

Unter *Triple-Graph-Grammatiken (TGG)* versteht man eine visuelle Modellierungssprache, die zur Beschreibung bidirektionaler Modell-zu-Modell-Transformationen verwendet wird [5]. Eine TGG beschreibt mittels Produktionsregeln den Aufbau eines sogenannten Korrespondenzgraphen, der Teile eines Quell- und Zielgraphen miteinander verknüpft. Aus diesen Regeln können dann Transformations- und inkrementelle Synchronisationschritte abgeleitet werden, die zur Verknüpfung, Synchronisierung und Abbildung von Daten zwischen verschiedenen Modellen verwendet werden [6], [7].

Die zugrundeliegenden Forschungsarbeiten beschreiben einen Ansatz zur Integration von Daten in Graphdatenbanken aus heterogenen Quellen auf Modelebene mittels einer Triple-Graph-Grammatik (TGG) [8]. Dazu wird `Gremlin`-Code aus TGG-Regeln generiert, um Abfragen und Aktualisierungsoperationen auf importierten Quelldaten in *neo4j* durchzuführen. Die Datenintegration selbst wird auf der Modellebene beschrieben, wobei UML zur Beschreibung der Schemata und TGGs zur Beschreibung der Zuordnung zwischen diesen im Eclipse Modeling Framework verwendet werden. Dieser Ansatz bietet folgende Vorteile:

- **Höheres Abstraktionslevel.** Durch die Beschreibung der Integration mit modelbasierten Technologien erhöht sich das Abstraktionslevel.
- **Skalierbarkeit.** Die Verwendung von Graphdatenbanken zur Persistenz der Korrespondenzen erlaubt ein hohes Maß an Skalierbarkeit des Verfahrens im Vergleich zu relationalen Datenbanken.
- **Strukturiert und schema-sicher.** Trotz der schemalosen Natur von Graphdatenbanken wird durch diesen Ansatz die Schemasicherheit durch die Verwendung typisierter Transformationsregeln einer TGG auf Modellebene - bei sorgfältigem Entwurf der Regeln - und ihrer korrekten Übersetzung in die Abfragesprache der Datenbank gewährleistet.
- **Einfache Vorabvalidierung der Datenintegration.** Die Integration kann auf Modelebene mit Werkzeugen des Eclipse Modeling Frameworks vor der eigentlichen Ausführung auf den produktiven Daten in der Graphdatenbank validiert werden kann.

- **Integration weiterer Stakeholder.** Durch die Abstraktion auf eine visuell erfass- und gestaltbare Modelebene können weitere Beteiligte einer Organisation an der Datenintegration partizipieren, ohne dass die darunterliegende Technologie (in diesem Fall Gremlin und neo4j) von diesen tiefergehender verstanden oder Code geschrieben werden muss.

In dieser Ausarbeitung wird Nachfolgendes beschrieben: In Abschnitt 2 werden die Schwierigkeiten der Datenintegration aus heterogenen Quellen in Graphdatenbanken anhand eines Beispiels beschrieben. Dieses Beispiel wird in Abschnitt 3 wieder aufgegriffen, um den vorliegenden Lösungsansatz praktisch zu demonstrieren. Dieser Abschnitt ist in weitere Abschnitte unterteilt, die sich an den einzelnen Schritten des Lösungsansatzes orientieren: Abschnitt 3.1 erklärt, wie Meta-Modelle mittels *UML* im *Eclipse Modeling Framework (EMF)* beschrieben werden. Diese Modelle sind notwendig, um die in Abschnitt 3.2 beschriebene Abbildung der Schemata und Testinstanzen in *Neo4j*-Graphdatenbanken mittels *NeoEMF* durchzuführen, als auch um die Triple-Graph-Grammatik und ihre Regeln in *eMoflon* zu erstellen. Dies wird in Abschnitt 3.3 erläutert. Abschnitt 3.4 beschreibt schließlich, wie aus diesen Regeln ausführbarer `Gremlin`-Code via *Acceleo* generiert wird, mit dem die Datenintegration in der Graphdatenbank durchgeführt werden kann. Abschnitt 4 widmet sich der Evaluierung des Ansatzes, während Abschnitt 5 weitere Arbeiten zu diesem Thema auflistet und vergleicht und in Abschnitt 6 ein Fazit gezogen und zukünftige Entwicklungen diskutiert werden.

2 Probleme der Datenintegration in Graphdatenbanken aus heterogenen Quellen

Die *Datenintegration* in relationale Datenbanken ist ein wohlbekanntes Anwendungsgebiet [9], während sie in Bezug auf Graphdatenbanken einen noch relativ jungen Forschungsbereich darstellt [10]. Aufgrund der Schemalosigkeit und Semistrukturiertheit von Graphdatenbanken wird eine Datenintegration beziehungsweise die Definition von Transformationsschritten in der Regel händisch in der Anwendungsschicht durchgeführt. Eine so definierte Datenintegration ist vor der Integration von produktiven Daten schwer zu validieren; häufig sind Replikate beziehungsweise ein Staging erforderlich, um manuell definierte Assertions durchführen zu können. Ein weiteres Problem stellt die Flexibilität von Graphdatenbanken dar: Die Struktur der Daten kann sich spontan durch Hinzufügen weiterer Knoten und Kanten ändern.

Für Organisationen ist es von Interesse, eine Teilmenge ihrer Daten kontinuierlich untereinander zu teilen und dabei ihre Domänenhoheit zu bewahren, also ihre Schemata und darauf aufbauende Prozesse unverändert zu belassen. Durch eine Integration in Graphdatenbanken können die zu integrierenden Daten in großen Mengen lose miteinander verknüpft werden, während durch die Flexibilität der Datenbanken auch häufige Änderungen an Daten und Datenschemata

unterstützt werden. Ursprüngliche Integrationsansätze erfüllen diese Anforderung im Kontext von Graphdatenbanken bisher nicht [10].

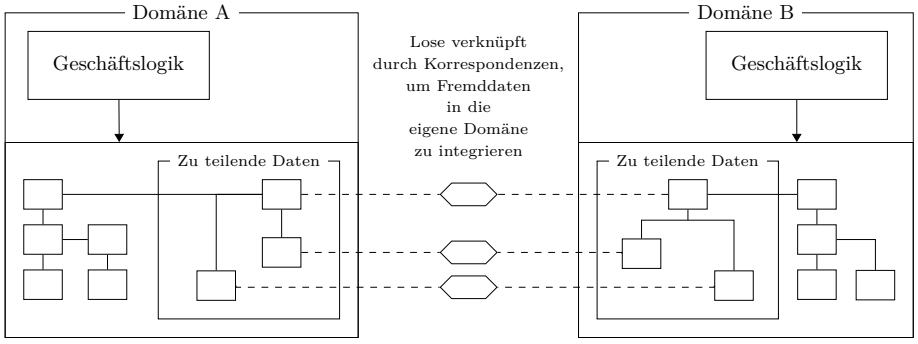


Abbildung 1: Typisches Integrationsszenario

Der Ansatz zur Datenintegration wird in der zugrundeliegenden Forschungsarbeit mittels eines fortlaufenden Beispiels beschrieben. Das Beispiel umfasst die beiden in Großbritannien ansässigen Unternehmensdaten-Anbieter *CompaniesHouse (CH)* und *CompanyCheck (CK)*. Beide Anbieter stellen umfangreiche Daten zu Unternehmen der "Limited"-Rechtsform in Großbritannien bereit [11]. Eine Zielgruppe dieser Daten sind B2B-Unternehmen, die diese Daten in ihre eigene Datenbasis aufnehmen möchten, um beispielsweise die Lead-Generierung zu unterstützen. Die beiden Datenquellen sind heterogen, unterschiedlich konsistent und weisen teilweise eine unterschiedliche Semantik in den zu teilenden Datenattributen auf. Die beiden Domänen sind also entsprechend gut geeignet, um die Herausforderungen der Datenintegration abzubilden. Grundsätzlich sollen die Datensätze aus beiden Quellen verknüpft werden, um fortlaufende bidirektionale Aktualisierungen durchführen zu können. Abbildung 4 zeigt vereinfachte Schemata der Daten beider Domänen als Abbildung in einer Graphdatenbanken, anhand derer das Integrationsszenario veranschaulicht wird.

Grundsätzlich soll der vorgestellte Integrationsansatz folgende Anforderungen erfüllen [8]:

- **Bidirektionale Integration.** Die Integration soll in beide Richtungen funktionieren, ohne die Datenhoheit der Domänen zu verletzen.
- **Skalierbarkeit.** Die Integration soll skalierbar sein, also auf sehr große Mengen von Daten performant anwendbar sein.
- **Visuelle Integration.** Die Integration soll die Teilhabe von Fachgebietsexperten ohne technisches Wissen über die darunterliegende Daten- und Anwendungsschicht durch eine visuelle Darstellung der Integrationsregeln unterstützen.
- **Agile und inkrementelle Integration.** Die Integration soll eine inkrementelle Transformation und Synchronisation der Modelle erlauben und auf

einfachen, atomaren, leicht veränder- und erweiterbaren Regeln basieren.

- **Heterogenität.** Die Integration soll heterogene Datenstrukturen in einer gemeinsamen vereinheitlichten Struktur abbilden.
- **Validierbar und schemasicher.** Die Integration soll Daten innerhalb einer Graphdatenbank schemasicher aus einem Quell- in ein Zielschema überführen. Die Schemasicherheit der Integration soll vorab validiert werden können.

3 Lösungsansatz unter Verwendung von Triple-Graph-Grammatiken

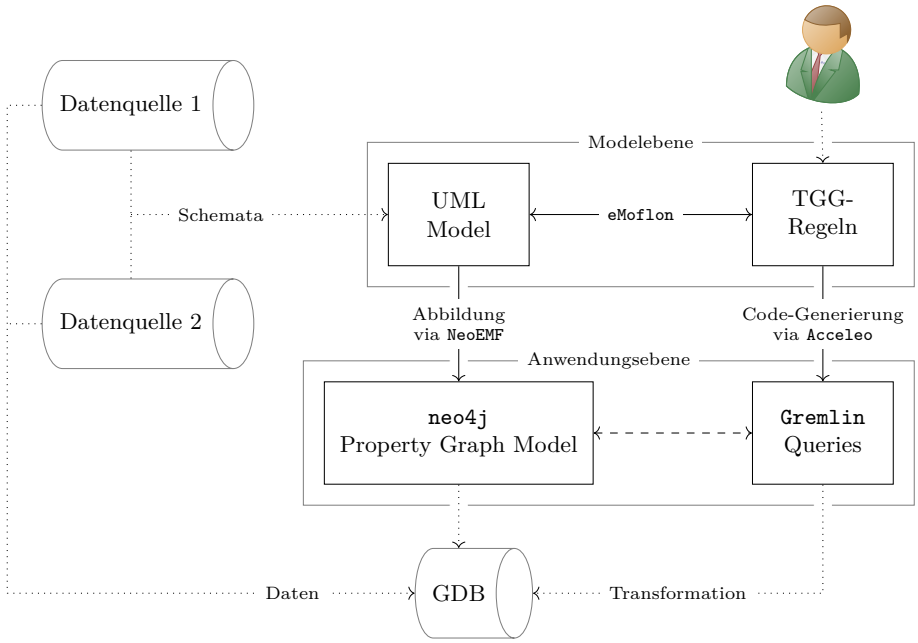


Abbildung 2: Architektur der Datenintegration mit Triple-Graph-Grammatiken

Abbildung 2 beschreibt die Architektur des Integrationsansatzes schematisch. Zuerst werden die Schemata der Daten und die Transformation zwischen diesen auf der **Modelebene** definiert. Dies erlaubt einen hohen Grad an Abstraktion. Die Datenmodelle der beiden Domänen werden in der Modellierungssprache *UML* beschrieben. Abbildungen beziehungsweise Relationen zwischen diesen Datenmodellen werden mittels Regeln einer *Triple-Graph-Grammatik (TGG)* [5] beschrieben. Die Beschreibung und Validierung der Transformation durch

diese Regeln geschieht in *eMoflon* [12], einer Erweiterung des *Eclipse Modelling Framework (EMF)* [13].

Die Daten der beiden Domänen werden innerhalb einer Graphdatenbank integriert. Die Abbildung und tatsächliche Durchführung der Regeln innerhalb der **Anwendungsebene** erfolgt, indem die Schemata über jeweils eine repräsentative Instanz in der Graphdatenbank abgebildet werden. Anhand dieser Instanzen kann ein Importskript erstellt werden, um die tatsächlichen Daten aus den Quellen zu importieren, zum Beispiel über CSV-Dateien [14], [15]. Die Abbildung der Schemata erfolgt in *NeoEMF* [16]. Die TGG-Regeln werden in die Anwendungsebene abgebildet, indem über den Code-Generator *Acceleo* [17] äquivalenter `Gremlin`-Code für die gewünschte Transformationsrichtung aus den Regeln erzeugt wird. Dieser wird auf den zuvor schemasicher importierten Daten angewendet, um den Zielgraphen zu erhalten. Die Transformationen können inkrementell und fortlaufend angewendet werden, um die beiden Datenquellen synchron zu halten. Damit ist die Integration abgeschlossen.

Anschließend können die integrierten Daten wieder in die relationale Datenbank der Domäne importiert oder in der Graph-Datenbank belassen werden, um darauf Anwendungslogik aufzubauen.

3.1 Abbildung der zu integrierenden Datenmodelle in UML

Die Datenintegration wird zuerst unabhängig von der darunterliegenden Plattform auf der Modellebene beschrieben. Dazu müssen die zu integrierenden Datenmodelle (Schemata) durch *UML-Klassendiagramme* abgebildet werden.

Modellierung mit dem Eclipse Modeling Framework (EMF)

Das *Eclipse Modeling Framework (EMF)* [13] ist ein Modellierungsframework basierend auf Eclipse und bietet Codegenerierungsfunktionen zum Erstellen von Tools und anderen Anwendungen auf der Grundlage strukturierter Modelle (Schemata) an.

In EMF werden sogenannte `Ecore`-Domänenmodelle erzeugt, die in Form von UML-Klassendiagrammen dargestellt werden. Ein solches `Ecore`-Modell soll den Problembereich einer Domäne abbilden. Der Ansatz zur Datenintegration verwendet diese Modelle, um die zu integrierenden Datenmodelle der Quell- und Zieldomäne exakt abzubilden. Diese `Ecore`-Modelle werden benötigt, und die nachfolgende Modelltransformation zu beschreiben.

Abbildung 3 zeigt die `Ecore`-Modelle für die beiden zu integrierenden Beispieldomänen entsprechend der vorgegebenen Schemata der Domänen (siehe auch die Darstellung als Property-Graphen in Abbildung 4). Die Objekte beider Domänen sind sich semantisch sehr ähnlich, jedoch unterschiedlich strukturiert und benannt. Diese Nichtübereinstimmung muss konzeptionell behoben werden, indem die deklarativen TGG-Regeln verwendet werden, um entsprechende Konzepte in Beziehung zu setzen, sowie operativ, indem die relevanten TGG-Datentransformationsregeln abgeleitet werden.

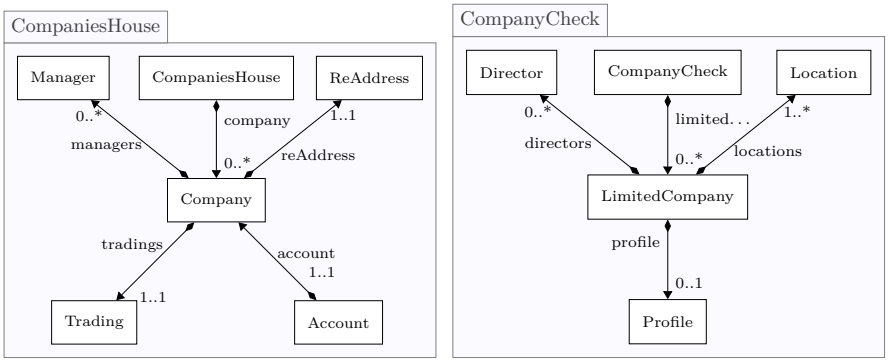


Abbildung 3: Abbildung der Domänenmodelle/Datenstrukturen in EMF

3.2 Abbildung in eine Neo4j-Graphdatenbank über NeoEMF

Um die Daten der beiden Domänen innerhalb einer Graphdatenbank zu integrieren, müssen die Daten der Quelldomäne (oder beider) zuerst in die Graphdatenbank importiert werden. Aus dem importierten Graphen wird der jeweilige Zielgraph durch Transformationsregeln erzeugt. Dazu ist *neo4j* eine geeignete Graphdatenbank, da sie einfache Werkzeuge zum paketweisen Import großer Datenmengen aus unterschiedlichen Quellen anbietet.

Der Neo4j-Property-Graph

Neo4j ist eine populäre Graphdatenbank, die auf dem Property-Graph-Model basiert, also einem gerichteten Multigraphen bestehend aus Knoten und Kanten, die jeweils Attribute (Properties) in Form von Schlüssel-Wert-Paaren besitzen können [18]. In Abbildung 4 ist dies für die beiden Domänen des fortlaufenden Beispiels exemplarisch dargestellt. Die Knoten eines solchen Graphen entsprechen den persistierten Objekten, während Kanten die Relationen zwischen diesen Objekten beschreiben. Ein solcher Graph kann mit speziell dafür gestalteten Sprachen wie `Gremlin` [19] und Frameworks wie *Apache TinkerPop* effizient traversiert und manipuliert werden. Durch diesen Aufbau eignet sich eine Graphdatenbanken wie *neo4j* insbesondere dort, wo in relationale Datenbanken das Datenmodell zu komplex oder die Abfragen zu inperformant werden (in der Regel durch exzessive Joins). Der Bedarf an Graphdatenbanken steigt stetig durch die Entwicklung und Verwendung von Anwendungen und Systemen wie zum Beispiel sozialen Netzwerken, medizinischen/biologischen Daten, Wissensdatenbanken, Seiten des semantischen Webs und Suchmaschinen, Empfehlungsnetzwerken und insbesondere Domänen des maschinellen Lernens [3], [20]–[22]. Der Vorteil der Flexibilität - die Instanzschemata der Daten können sich zur Laufzeit durch das Hinzufügen neuer Knoten und Kanten ändern - kann abhängig von den Anforderungen zum Nachteil werden [23]. Wenn eine Schemasicherheit benötigt wird, so wird diese auf einer anderen Ebene als der Datenebene gewährleistet, in

der Regel innerhalb der Anwendungsschicht.

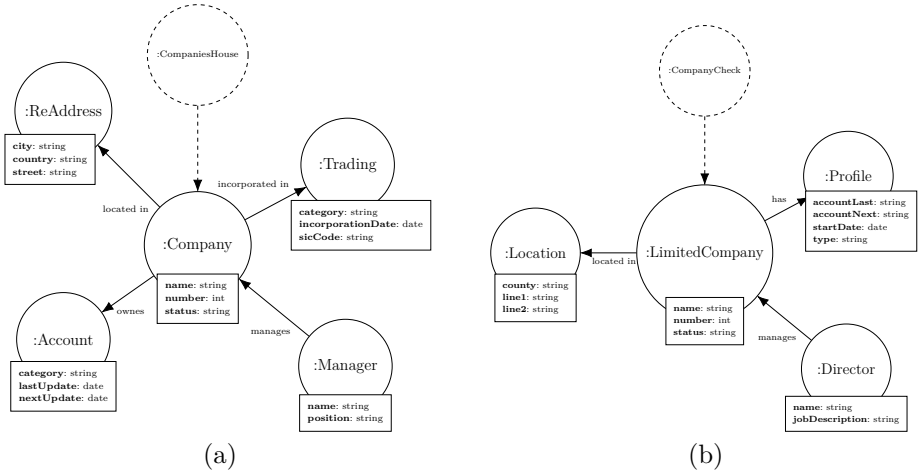


Abbildung 4: Beispiel gültiger Property-Graphen-Modelle für die Schemata *CompaniesHouse (a)* und *CompanyCheck (b)*

Schemasicherer Datenimport mit NeoEMF

NeoEMF ist ein Persistenz-Framework für das Eclipse Modeling Framework [16]. Mit NeoEMF können (Java-)Klassen und tatsächliche Instanzen dieser Klassen aus EMF-Schemen und -Instanzen (`.xmi`-Dateien) generiert werden, die dann auf einfachem Wege in verschiedene Datenbanken oder andere Persistenzebenen gespeichert werden können. Dabei werden alle wichtigen Graphdatenbanken unterstützt, die das *Apache TinkerPop*-Framework implementieren, darunter auch *neo4j*, so dass die Instanzen via `Gremlin` persistiert werden können.

In diesem Ansatz wird NeoEMF verwendet, um eine repräsentative Modelinstanz als Graph in *neo4j* zu persistieren. Dazu wird das in EMF abgebildete UML-Schema der zu importierenden Datenquelle in NeoEMF importiert. NeoEMF stellt für die importierten Schemata Klassen und dazugehörige Factories bereit, damit der Anwender mit geringem Aufwand Instanzen der Modelle erstellen kann (ab Zeile 11 in Listing 1). Über diese Factories wird eine repräsentative Instanz erzeugt, die das Schema abbildet, die dann über das *Blueprint-Interface* des *Apache TinkerPop*-Frameworks (Zeile 3) in der Graphdatenbank persistiert wird (Zeile 20). Diese Instanz verknüpft das Wurzelelement (im Anwendungsbeispiel `Company` respektive `LimitedCompany`) mit einem domänenbeschreibenden Masterknoten, der dauerhaft persistiert und mit allen Wurzelelementen der jeweiligen Domäne verknüpft wird, damit Knoten der jeweiligen Domäne schnell gefunden und dieser zugeordnet werden können.

```
1 public static void main(String[] args) throws IOException {
```

```
    // create local database using tinkerpop blueprint
    URI uri = new BlueprintsUriFactory()
        .createLocalUri("models/evaluation.graphdb");
    ImmutableConfig config = new BlueprintsTinkerConfig();

    ResourceSet resourceSet = new ResourceSetImpl();
    Resource resource = resourceSet.createResource(uri);

    // create instance(s) on the fly
    NeoEMF_CompaniesHouseFactory factory =
        NeoEMF_CompaniesHouseFactory.eINSTANCE;

    Company company = factory.createCompany();
    company.setName("ACME Inc.");

    // ...

    // save instance(s) to database
    resource.getContents().add(company);
    resource.save(config.toMap());
    resource.unload();
}
```

Listing 1: Beispielcode zum Import einer repräsentativen Modelinstanz in Neo4j via NeoEMF

Die so persistierte Instanz wird sodann als `Cypher`-Erstell-Skript exportiert. Dieses Skript dient als Vorlage für weitere Datenimportierte und wird dementsprechend generalisiert, d.h. Instanz-Label und -Attribute werden durch Platzhalter ersetzt. Über das Command-Line-Interface (CLI) von *neo4j* und passenden Import-Erweiterungen wie `neo4j-shell-tools` [14] oder `LOAD CSV` [15] wird das generalisierte Skript verwendet, um im `CSV`-Format exportierte Daten aus der Datenquelle effizient und paketweise in die Graphdatenbank zu importieren. Zuvor sollten in *neo4j* Indizes und Constraints dem Schema entsprechend festgelegt werden [24]. Die über diesen Weg importierten Daten erfüllen nun die Anforderungen an die Schemasicherheit (siehe Abschnitt 4).

```
import-cypher -i data.csv -b 10000
CREATE (c:Company {
    name: {name},
    number: {number},
    status: {status} })
CREATE (reAd:ReAddress {
    city: {city},
    country: {country},
    street: {street} })
CREATE (c)-[:LOCATED_IN]->(reAd)
```

```
11 ...
```

Listing 2: Beispiel eines generalisierten Cypher-Skripts zum Import von Daten über eine CSV-Datei

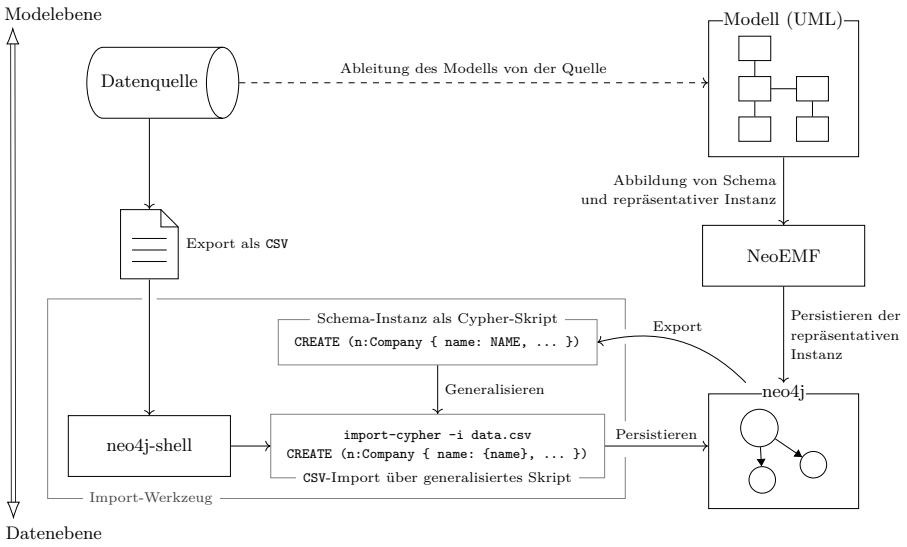


Abbildung 5: Schemasicherer Import-Workflow mit NeoEMF

3.3 Transformation mit Triple-Graph-Grammatiken in eMoflon

Triple-Graph-Grammatiken (TGG) wurden als Technik zur bidirektionalen Modelltransformation eingeführt [5]. TGGs stammen ursprünglich aus dem Umfeld der Metamodellierung, lassen sich aber auch zur Transformation konkreter Datenmodelle verwenden. Da sie graphenbasiert sind, ist die Übertragung auf Graphen in Graphdatenbanken sehr einfach und benötigt keiner weiteren Abstraktion.

Eine Triple-Graph-Grammatik wird verwendet, um eine Schema-Abbildung mittels sogenannter *Korrespondenzen* zu beschreiben. Dabei werden Produktionsregeln auf dem Quell- und Ziel- sowie einem Korrespondenzgraph definiert, aus denen wiederum Transformationsschritte abgeleitet werden, um einen Graphen (ein Schema) aus einem anderen zu erstellen und zwischen diesen beiden Graphen konsistente Verknüpfungen (im Korrespondenzgraph) aufrecht zu erhalten, um auf Veränderungen reagieren zu können [25, p. @giese2009model]. Ein Vorteil von TGGs ist es, dass Transformationsschritte deklarativ definiert werden und dennoch in beide Richtungen ausgeführt werden können. Die Transformationen können schrittweise (inkrementell) angewendet werden. Dadurch kann fortlau-

fend auf Änderungen der beteiligten Modelle reagiert werden, um die Modelle zur Laufzeit zu synchronisieren.

Um den Aufbau und die Anwendung einer TGG zu erklären, werden zuerst *Graph-Grammatiken* im Allgemeinen erörtert. Anhand dieser Grundlagen werden TGGs und daraus ableitbare Transformationen erklärt. Anschließend wird die Beschreibung von TGGs durch die Ermittlung einfacher Produktionsregeln für die beiden Domänen des Anwendungsbeispiels konkretisiert und durch die Verwendung eines geeigneten Frameworks angewendet.

Graph-Grammatiken

Eine *Graph-Grammatik* ist eine Grammatik, die mit ihren Produktionsregeln den Aufbau gültiger Graphen beschreibt. Soll mit einer Graph-Grammatik die Ersetzung eines (Teil-)Graphen durch bzw. die Transformation in einen anderen beschrieben werden, spricht man auch von einem Graphersetzungssystem.

Eine Graph-Grammatik für einen Graphen $G = (E, V)$, bestehend aus einer Menge von Knoten V und Kanten E, ist also ein Tupel (G, P, S), wobei P die Menge aller Produktionsregeln und S ein (ggf. leerer) Start-Graph ist. Eine Produktionsregel beschreibt einem Morphismus $m : L \to R$ von einem *Kontextgraphen* L in einen *Ersetzungsgraphen* R. Der Kontextgraph wird auch als *Linke Seite (LHS, Left Hand Side)* und der Ersetzungsgraph analog als *Rechte Seite (RHS, Right Hand Side)* bezeichnet. Diese Bezeichnungen orientieren sich an der Leserichtung gewöhnlicher Grammatiken. Der Kontextgraph ist ein Graphmuster, für das eine Übereinstimmung mit einem Teilgraphen G' von G gefunden werden soll, das heißt, es existiert ein Isomorphismus $L \to G'$ oder mit anderen Worten, es wird ein Muster in G gefunden, dass der linken Regelseite entspricht. Diesen Schritt bezeichnet man auch als *Graph Matching*. Werden ein oder mehrere entsprechende Teilgraphen G' in G gefunden, so werden diese durch das Muster des Ersetzungsgraphen R ersetzt. Dabei können Matching/Erhaltungs-, Erstell- und Löschoperationen innerhalb einer Produktionsregel definiert werden.

TGGs verwenden eine eingeschränkte Form von Graph-Grammatiken als Grundsprache. Dabei wird auf Löschregeln verzichtet, da diese in TGGs nicht möglich sind. TGGs erlauben nur monotone, konstruktive Produktionen. Durch Löschen von Knoten können Korrespondenzen nicht mehr aufgebaut werden und bidirektionale Transformationen wären nicht mehr möglich.

Die visuelle Repräsentation von Graph-Grammatiken ist aufgrund graphbasierten Natur einfach und verständlich. Dabei werden häufig unterschiedliche Farben verwendet, um entsprechende Operationen darzustellen. Mit **Schwarz** werden die Teile des Graphen dargestellt, die bereits vorhanden sind (gegen die gematched werden kann), mit **Grün** werden die zu erstellenden Teile gekennzeichnet und **Rot** beschreibt Löschoperationen.

Abb. 6 veranschaulicht dies anhand einer sehr einfachen Produktionsregel einer Graph-Grammatik auf der Basis des CompaniesHouse-Schemas des Anwendungsbeispiels. Diese Regel fügt jedem Knoten des Typs `CompaniesHouse` einen weiteren Knoten `Company` hinzu.

Im *Pattern-Matching*-Schritt (Linke Seite) wird eine vorhandene Instanz eines `CompaniesHouse`-Objekts als Voraussetzungen für die Ausführung der Regel gesucht. Im Graphersetzungs-Schritt (Rechte Seite) wird dann das gefundene Objekt erhalten und ein neuer Knoten sowie die dazugehörige Kante hinzugefügt.

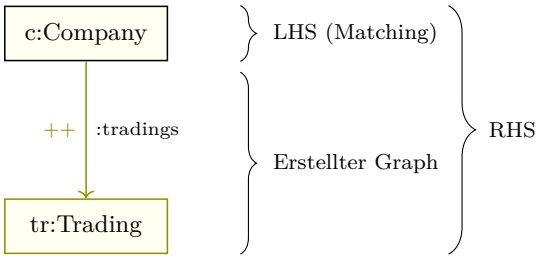


Abbildung 6: Beispielregel einer Graph-Grammatik

Da Löschvorgange in TGGs nicht möglich sind, wird hier auf eine entsprechende Einführung verzichtet.

Triple-Graph-Grammatiken

Eine Triple-Graph-Grammatik beschreibt Produktionsregeln, die in der Summe eine Schema-Abbildung mittels *Korrespondenzen* beschreiben. Die Produktionsregeln werden durch einen typisierten und attribuierten Graphen dargestellt und bestehen aus einfachen Unterregeln zu drei Teilgraphen; zwei davon repräsentieren den Quell- und den Zielgraph, die aufeinander abzubildenden sind, während die Gesamtheit der Korrespondenzen den dritten Graphen bildet [27]. Eine Produktionsregel beschreibt also eine Abbildung von einem Tripel-Graph (vor der Regelanwendung) auf einen produzierten Tripel-Graph, daher auch der Name. Die Kombination der drei Regeln beschreibt eine Möglichkeit, wie zwei Graphen und der zugehörige Korrespondenzgraph simultan verändert oder erstellt werden können. Diese Grammatiken ermöglichen es also, die Beziehung zwischen verschiedenen Arten von Modellen zu definieren und ein Modell in ein anderes zu überführen, als auch Instanzgraphen zu transformieren.

Ausgehend von der vorherigen Beschreibung der Graph-Grammatiken können wir TGGs nun folgendermaßen formal beschreiben:

Eine Triple-Graph-Grammatik ist eine Graph-Grammatik auf einem Tripel-Graph $G = (G^S \leftarrow G^C \rightarrow G^T)$ bestehend aus dem Quellgraphen G^S, Zielgraphen G^T und dem Korrespondenzgraphen G^C [28]. Zwischen den Graphen gibt es zwei Morphismen $s_G : G^C \rightarrow G^S$ und $t_G : G^C \rightarrow G^T$. Ein Tripel-Graph-Morphismus ist ein Morphismus $m : G \rightarrow H$ zwischen zwei Tripel-Graphen G und H, bestehend aus drei Graph-Morphismen m^S, m^T und m^C. Ein vollständiger Triple-Graph-Morphismus kann also folgendermaßen beschrieben werden:

Eine TGG-Produktionsregel r ist ein injektiver Tripel-Graph-Morphismus $r = (r^S; r^C; r^T) : L \rightarrow R$ auf dem Kontextgraph L (LHS) und dem Ersetzungsgraph R (RHS) mit Erhalt von L in R. Analog zu einfachen Graph-Grammatik-

$$\begin{array}{ccccc} G = (G^S & \xleftarrow{s_G} & G^C & \xrightarrow{t_G} & G^T) \\ \downarrow m \quad \downarrow m^S & & \downarrow m^C & & \downarrow m^T \\ H = (H^S & \xleftarrow[s_H]{} & H^C & \xrightarrow{t_H} & H^T) \end{array}$$

Abbildung 7: Tripel-Graph-Morphismus

Produktionsregeln wird eine TGG-Produktionsregel angewandt, indem der Kontextgraph L auf einen Teilgraph aus G gematched wird. Anschließend wird L in G mit R ersetzt. Einen solchen Transformationsschritt kann man als $G \overset{r,m}{\Longrightarrow} H$ notieren. Eine vollständige TGG für einen Graphen G ist also ein Tupel $TGG = (G, P, S)$, wobei P die Menge aller TGG-Produktionsregeln und S ein Graph-Tripel als Startaxiom (siehe Regel a) aus Abbildung 11) ist. Die Produktionsregeln beschreiben, wie alle drei Graphen simultan aufgebaut werden können. Aus diesen Regeln können einfach gerichtete Transformationsschritte abgeleitet werden, um den Ziel- (Vorwärtstransformation) oder den Quellgraphen (Rückwärtstransformation) aus dem entsprechenden bereits existierenden Graphen zu erzeugen oder beide Graphen fortlaufend zu synchronisieren [28].

Um eine Datenintegration mit einer TGG zu beschreiben, ist also ein Tripel-Graph G, insbesondere ein Korrespondenzgraph G^C, zu den zu integrierenden Daten-Graphen G^S und G^T und eine Folge von Produktionsregeln $P = r_1, \ldots, r_n$ zu finden, so dass aus den Produktionsregeln entsprechende Vorwärtstransformationsschritte $t_1, \ldots, t_n$ abgeleitet werden können für die gilt $G^S \overset{r_1}{\Rightarrow} \ldots \overset{t_n}{\Rightarrow} G$ mit $G = (G^S, G^C, G^T)$. Die Vorraussetzung ist also, dass eine solche Folge gefunden werden kann, ansonsten sind die Strukturen nicht vollständig integrierbar.

Vorwärts- und Rückwärts-Transformationen

In erster Linie beschreiben TGG-Produktionsregeln den simultanen Aufbau zweier Graphen (Modelle) und ihrer Korrespondenzen. In der Praxis lassen sich aber Transformationsregeln ableiten, um einen neuen Graphen aus einem existierenden Graph zu erstellen. Der vorliegende Ansatz bedient sich dabei Vorwärts- und Rückwärts-Transformationen, um initial oder paketweise Transformationen durchzuführen (*Batch-Transformationen*). Neben diesen Transformationstypen können auch Synchronisationsregeln definiert werden, um Modelle und Instanzgraphen zur Laufzeit synchron zu halten, doch die Anwendbarkeit solcher Regeln für die Datenintegration sind noch Gegenstand jüngster Forschung und nicht dieser Ausarbeitung [28].

Im Falle der Datenintegration soll ein existierender Quellgraph in einen Zielgraphen transformiert werden. Dazu ist also zunächst eine Graph-Grammatik zu finden, welche das Modell der Quelldomäne beschreibt. Die Morphismen, welche die Regeln dieser Grammatik beschreiben, werden nun zu zur Zieldomäne passenden Tripel-Graph-Morphismen erweitert. Dazu ist ein Schema-Mapping vorzunehmen, das heißt semantisch ähnliche Objekte von Quell- und Zielmodell zu finden, um die Korrespondenzen dieser Objekte zu beschreiben. Für

das fortlaufende Anwendungsbeispiel sind diese Korrespondenzen und Morphismen schnell gefunden, siehe Abbildung 11. Für die erstellten Regeln ist eine Ableitungsreihenfolge zu finden, die einen korrekten Zielgraphen erzeugt.

Für eine Vorwärtstransformation werden Übersetzungsregeln aus den Produktionsregeln abgeleitet. Das Quellmodell ist bereits existent und die Elemente der Quelldomäne sollen nicht mehr erneut erzeugt werden. Entsprechend sind die Objekte der Quelldomäne nicht Teil des produzierten Graphs der Produktionsregeln, sondern werden im Kontextgraph beziehungsweise der linken Seite der Regel (LHS) mit einer Matching-(Erhaltungs-)Operation gekennzeichnet. Bei Ausführung einer solchen Vorwärtstransformation werden diese Teile im Graph per *Pattern-Matching* wie zuvor für Graph-Grammatiken beschrieben gefunden. Dabei wird aber nicht nur der Quellgraph betrachtet. Die Knoten und Kanten im Kontextgraph werden auf bereits existierende Elemente sowohl im Quell-, Ziel- als auch im Korrespondenzgraphen abgebildet. Hierbei können unterschiedliche Bedingungen ergänzt werden, wie negative Matching-Bedingungen (eine Regel wird nur angewendet, wenn ein verknüpftes Objekt im Quellgraph nicht existiert) oder Assertions auf Attribute. Dabei werden Knoten und Kanten, die in einer Produktionsregel im Quellgraph erstellt worden wären, in einem Vorwärtstransformationsschritt in das Pattern-Matching aufgenommen und bei erfolgreichem Matching als *gebunden* markiert. Dabei muss sichergestellt werden, dass jedes Element des Quellmodels nur genau einmal gebunden wird. Eine Transformation ist dann vollständig abgeschlossen, wenn alle Elemente des Quellgraphen erfolgreich als gebunden markiert sind. Gibt es noch ungebundene Elemente, so wurden Teile des Quellgraphen nicht bearbeitet. In diesem Fall sollten die Regelausführungen werden nach bestimmten Backtracking-Verfahren zurückgesetzt und eine andere Ausführungsreihenfolge versucht werden.

Durch die Birektionalität von TGGs kann nun auch das Quellmodell aus dem Zielmodell über die Grammatik erstellt werden. Man spricht dann von einer Rückwärtstransformation.

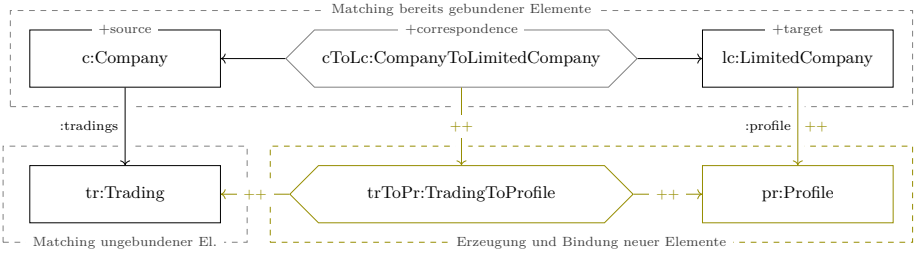
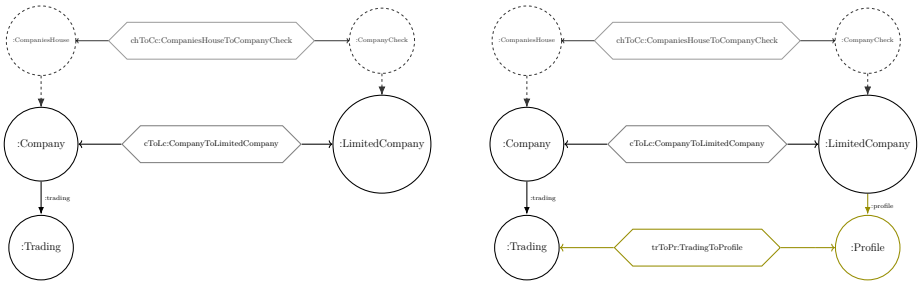


Abbildung 8: Beispiel einer Vorwärtstransformationsregel einer Triple-Graph-Grammatik

Auch bei TGGs eignet sich eine visuelle Repräsentation besser für ein konkretes Verständnis. In Abbildung 8 ist eine solche Vorwärtstransformations-Regel dargestellt. Die Darstellung ist der von einfachen Graph-Grammatik-Regeln aus Abbildung 6 ähnlich, doch besteht die Darstellung der TGG-Regel darüber



Pattern Matching (LHS)

Erstellung in Korrespondenz- und Zielgraph (RHS)

Abbildung 9: Zustand des Graphenmodells vor und nach Anwendung der Vorwärtstransformation zur Regel aus Abbildung 8)

hinaus aus Teilgraphen beider Domänen sowie den Korrespondenzen. Die linken Seite der Darstellung (nicht zu verwechseln mit der Bezeichnung der linken Seite (LHS) in Hinsicht auf den Kontextgraph) zeigt die Knoten und Kanten der Quelldomäne, während die Objekte der Zieldomäne auf der rechten Seite dargestellt werden. Der mittlere Teil beschreibt die Korrespondenz zwischen den Elementen beider Domänen. In der Regel wird auf eine gesonderte Darstellung der LHS (Kontextgraph) einer Produktionsregel verzichtet und diese in Form einer kompakten Darstellung (schwarz gezeichnete Elemente) innerhalb der RHS angezeigt. Die grünen Elemente beschreiben auch hier die Elemente, die durch Anwendung der Regel erstellt werden. In Abbildung 11 sind sämtliche Regeln für da Anwendungsbeispiel gelistet. Die Regeln zeigen, dass die Transformation für unser Beispiel vollständig auf Modellebene mittels einer TGG beschrieben werden kann.

Die Relevanz der Reihenfolge der Regelausführung ist besonders hervorzuheben. In Abbildung 10 wird ein Zustand nach einer Vorwärtstransformation ausführlich dargestellt. Die schwarzen Bereiche des Graphmodels visualisieren bereits existierende Knoten und Kanten von Quell-, Ziel- und Instanzgraphen. Für eine bessere Übersicht wurde auf die Darstellung weiterer Knoten des Quellgraphen verzichtet und ein kleiner Ausschnitt gewählt, der bereits Korrespondenzen besitzt und als gebunden markiert wurde. Die grünen Elemente stellen die Knoten und Kanten dar, die nach Anwendung der Vorwärtstransformation im Ziel- und im Korrespondenzgraphen erstellt wurden. Die hervorgehobenen Teile des Graphen sind dabei für das Pattern Matching relevant. Nach Anwendung der Regel *e)* wird also eine neue Korrespondenz zwischen `Account` des Quell- und `Profile` des Zielgraphen erzeugt, wobei der Knoten `Profile` bereits zuvor in Regel *c)* erstellt wurde. An dieser Vorbedingung erkennt man leicht die Relevanz der Reihenfolge der Regelausführung, da nicht nur der axiomatische Teil der TGG-Regel für diese relevant ist. Im Prinzip könnten Regel c) und e) auch zusammengefasst werden, da eine Regel nicht nur einzelne Knotenpaare miteinander verknüpfen kann. Für die Datenintegration ist es aber sinnvoll, atomare Regeln zu definieren, die möglichst wenige Knoten in einer Korrespondenz be-

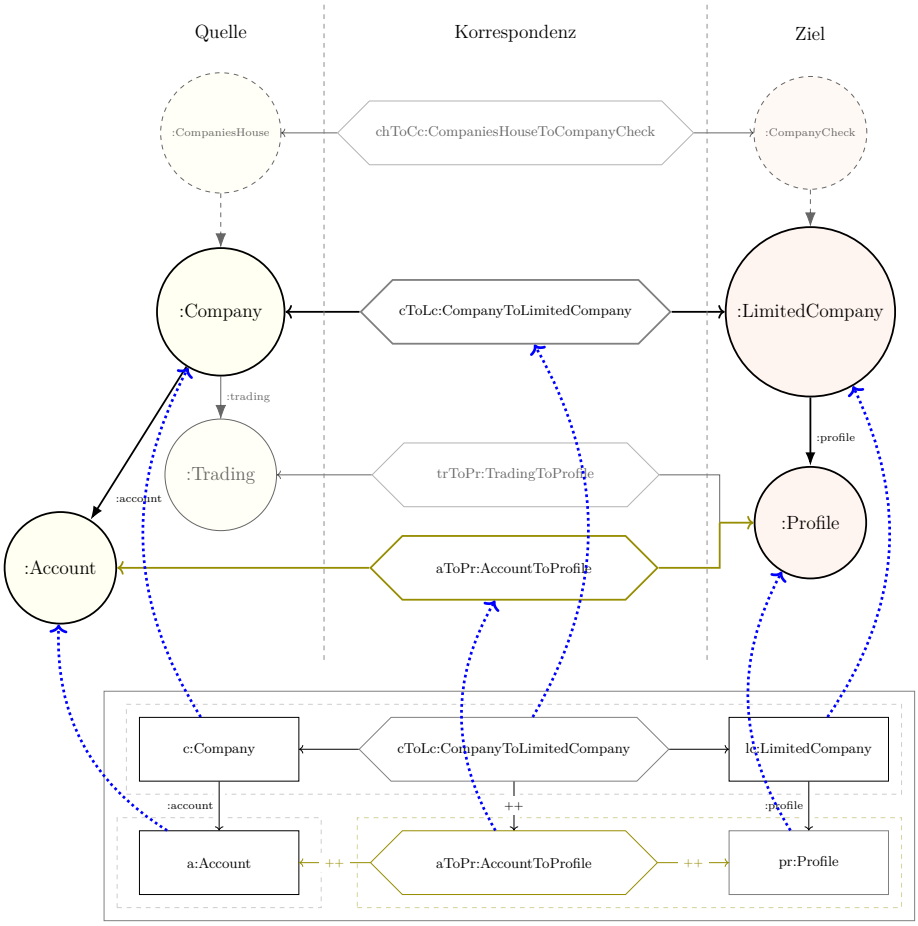


Abbildung 10: Visualisierung eines Zustands nach Anwendung der Regel e)

schreiben, damit inkrementelle Transformationen einfach durchführbar sind. Des Weiteren können Relationen unterschiedlicher Kardinalitäten nur in atomaren Regeln korrekt abgebildet werden. Ist ein Graphelement optional, so kann eine Regel, die ein solches Element enthält, ebenso nicht mit einer anderen Regel zusammengefasst werden.

Definition und Ausführung von TGG-Regeln in eMoflon

eMoflon ist eine Tool-Suite für Model-Driven-Engineering (MDE) in Form einer Erweiterung für EMF, die eine Reihe visueller und formaler Sprachen für die (Meta-) Modellierung und das Modellmanagement bietet [29]. Insbesondere können TGG-Regeln mit *eMoflon* assistentengestützt erstellt und auf Instanzdateien in EMF angewendet und validiert werden.

Um TGG-Regeln zu definieren, müssen die in EMF erstellten Modelle in Form

von `.ecore`-Dateien in ein *eMoflon*-Projekt importiert werden. Basierend auf den Modellen für die zu integrierenden Datenquellen der beiden Domänen wird ein weiteres Schema ergänzt, um die Struktur der Korrespondenzen zu definieren. Im Anschluss können die TGG-Regeln auf der Basis dieser drei Teile erstellt werden. An dieser Stelle müssen Quell- und Zieldomäne einmalig festgelegt werden; dies hat aber keinen Einfluss auf die mögliche Richtung der Integration, da aus den Regeln sowohl Vorwärts- als auch Rückwärtstransformationen abgeleitet werden können.

EMoflon benötigt zur Definition einer TGG zuerst eine Schema-Datei (siehe Listing 3). In dieser werden die Schemata der Quell- und Zieldomäne importiert und die Korrespondenzklassen festgelegt. Diese müssen vor der eigentlichen Regeldefinition festgelegt werden. An dieser Stelle werden noch keine Transformationsoperatoren erwartet. Listing 4 zeigt die spezifische Implementierung einer Regeln in *eMoflon*. Zuerst sind der Name der Regel festzulegen sowie das zuvor definierte TGG-Schema zu referenzieren. Die weitere Syntax ist denkbar einfach: Für die bereits im Schema festgelegten Domänen (`#source` und `#target`) sowie für die Korrespondenz (`#correspondence`) gibt es einen vordefinierten Bereich, in dem die Matching- und Erstelloperationen festgelegt werden. Die Operationen werden durch durch bestimmte Operatoren definiert. Der für das Anwendungsbeispiel relevante Operator ist der Erstell-Operator `++`, durch den festgelegt wird, welche Knoten und Kanten durch Anwendung der Regel zu erstellen sind. Wird auf den Operator verzichtet, werden die entsprechenden Knoten und Kanten als Erhaltungsoperation interpretiert und sind für das Pattern-Matching relevant. Durch simples Anpassen der Operatoren können Vorwärts- und Rückwärtstransformationen schnell definiert werden. Ein weiterer Bereich steht für sogenannte Attributbedingungen zur Verfügung, um Assertions aufzustellen, wie Attributwerte zueinander in Beziehung stehen. Die Elemente aus den Domänen und der Korrespondenz können innerhalb der Attributbedingungen verwendet werden.

```
1  #import ".../CompaniesHouseEMF.ecore"
2  #import ".../CompanyCheckEMF.ecore"
3
4  #schema CompanyTGG
5
6  #source {
7      CompaniesHouseEMF
8  }
9
10 #target {
11     CompanyCheckEMF
12 }
13
14 #correspondence {
15     CompaniesHouseToCompanyCheck {
16         #src->CompaniesHouse
17         #trg->CompanyCheck
```

```
    }
    // ...
}

#attributeConditions { }
```

Listing 3: TGG-Schema zum beschriebenen Regelsatz

```
#rule RuleC #with CompanyTGG

#source {
    c:Company {
        -tradings->tr
    }
    tr:Trading
}

#target {
    lc:LimitedCompany {
        ++ -profile->pr
    }
    ++ pr:Profile
}

#correspondence {
    cToLoc:CompanyToLimitedCompany {
        #src->c
        #trg->lc
    }

    ++trToPr:TradingToProfile {
        #src->tr
        #trg->pr
    }
}

#attributeConditions {
    eq_string(c.name, lc.name)
    // ...
}
```

Listing 4: TGG-Datei zu Regel c) aus Abb. 8

Um die Daten des Anwendungsbeispiels der Domänen *CompaniesHouse* und *CompanyCheck* zu integrieren, werden sechs Regeln implementiert. Diese sind in Abbildung 11 aufgelistet. Diese Darstellung ist noch einmal kompakter

als die bisherige Darstellung und verzichtet auf eine explizite Zeichnung der Korrespondenzknoten. Diese werden über ein Label an der Korrespondenzkante angedeutet.

Regel a) versucht zuerst, den domänenbeschreibenden Masterknoten in der Quelldomäne zu finden. Dieser ist sinnvoll, um die in einer gleichen Graphdatenbank nebeneinander und mit Korrespondenzen verknüpften Knoten eindeutig der richtigen Domäne zuordnen und schnell finden zu können. Bei erfolgreichem Matching wird ein entsprechender gegenüberliegender Masterknoten für die Zieldomäne erzeugt und über einen Korrespondenzknoten miteinander verknüpft. Das Ergebnis der Ausführung dieser Regel ist nun Teil der Voraussetzungen für die Ausführung der nächsten Regeln. Diese Regel bezeichnet man auch als Axiom, da sie keine Vorbedingungen besitzt und die Anwendung der Grammatik mit dieser beginnt.

Die Folgeregel b) wendet nun das Axiom als Vorbedingung an und prüft auf das Vorhandensein eines `Company`-Knotens in der Quelldomäne, der mit dem Masterknoten über eine entsprechende Kante verknüpft ist. Die weitere Regelausführung entspricht dem, was zuvor in der Beschreibung zu Abbildung 10 erläutert wurde.

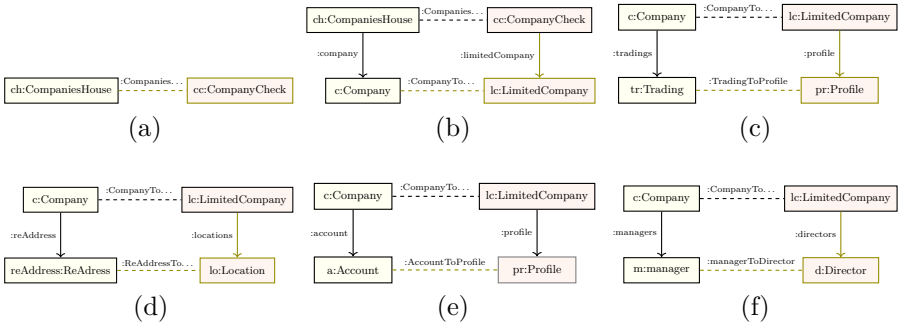


Abbildung 11: Kompakte Darstellung von TGG-Regeln zur Vorwärtstransformation in eMoflon

3.4 Implementierung und Durchführung der Transformation mit Gremlin und Acceleo

Nachdem eine gültige TGG-Ableitungsreihenfolge für Vorwärtstransformationen in *eMoflon* gefunden und beschrieben wurde, müssen die TGG-Regeln in eine Abfragesprache für *neo4j* übersetzt werden, damit die zuvor schemasicher über den *NeoEMF*-Import-Workflow in die Graphdatenbank importierten Quelldaten transformiert werden können. Als Zielsprache dieser Übersetzung eignet sich die Graphdatenbank-Abfragesprache `Gremlin` aufgrund ihrer einfachen Syntax und Plattformunabhängigkeit [19]. `Gremlin` nutzt das *Apache TinkerPop*-Framework als Adapter für unterschiedliche Graphdatenbanken als auch für unterschiedliche implementierende Programmiersprachen und Frameworks. `Gremlin` selbst ist

eine Graphtraversierungssprache, die im Vergleich zu Sprachen wie `SQL` oder `Cyper` keine besondere, eigene Syntax aufweist, sondern in Host-Sprachen implementiert beziehungsweise eingebettet ist, so dass sich Anwender keine neuen Sprachkonstrukte aneignen müssen. Zu diesen Host-Sprachen zählen populäre Sprachen wie `Java`, `JavaScript`, `Python` oder auch `Groovy`.

Die Übersetzung nach `Gremlin` wird durch den Code-Generator *Acceleo* unterstützt [17]. *Acceleo* ist ein Textgenerator für *EMF*, der mittels einer eigenen Template-Sprache beliebigen Code aus importierten Modelldateien, darunter auch TGG-Dateien, erzeugen kann. In Listing 6 ist ein Auszug für das Generator-Template abgebildet. Der Generator importiert zuvor in *eMoflon* definierte `.tgg`-Dateien und erstellt für die gefundenen Regeln einzelne Dateien, die den Transformationsschritt abbilden. Dazu kann `Acceleo` auf die TGG-Objekte zugreifen, über diese iterieren und entsprechende Ausgaben (den Code der Zielsprache) in eine Datei schreiben. Die Syntax erlaubt die Definition von Hilfsmethoden in Form von Untertemplates, um seine Templates zu strukturieren (Zeile 37-55).

Bei der Abbildung von TGG-Regeln auf Gremlin-Code werden dabei Schritte für die gewünschte Transformationsrichtung aus den Regeln abgeleitet. Die linke Seite (LHS) einer TGG-Regel wird über die Pattern-Matching-Funktionen von `Gremlin` abgebildet, während der Produktionsgraph der rechten Seite über graphmanipulierende `Gremlin`-Funktionen erstellt wird. Dabei werden sowohl die Knoten und Kanten des Ziel- als auch des Korrespondenz-Graphs in der Datenbank erzeugt.

Pattern-Matching (LHS). Beim Pattern-Matching wird versucht, geeignete, mit dem Muster der linken Seite einer Regel übereinstimmende Teilgraphen zu finden. In Graphdatenbanken ist das Pattern-Matching der teuerste Schritt, während Erstellung und Traversierung von Knoten und Kanten eine konstante Komplexität aufweisen. In Listing 5, Zeile 3-4, ist das Pattern-Matching für Regel c) aus Abbildung 8 dargestellt. Das Skript wurde mit *Acceleo* erzeugt. Das Matching erfolgt über die Befehle `.v()`(Knoten-Matching anhand von Attributen) und `.out()` (Traversierung zu Knoten über ausgehende Kanten). Über die Befehle `.as()` und `.table()` können die gefundenen Knoten in einer Tabelle zwischengespeichert werden, während `.loop()` das Matching für weitere passende Treffer wiederholt.

Graph-Produktion (RHS). Ein `TransactionalGraph`-Objekt von Gremlin behält den aktuellen Traversierungszustand bei (analog einem Cursor bei relationalen Datenbanken), so dass nach dem Pattern-Matching auf den gefundenen Knoten gearbeitet werden kann. In Zeile 7-11 von Listing 5 wird die Generierung der neuen Knoten und Kanten in Ziel- und Korrespondenzgraph beschrieben.

```
1 { TransactionalGraph g ->
2     // LHS of ruleA
3     g.v(name, cToLc).as(x).out(source).as(y).table(t).loop(x)
```

```
    g.v(name, c)
    ...
    // RHS of ruleA
    g.addVertex(name, trToPr)
    g.addVertex(name, pr)
    g.addEdge(trToPr, tr, source)
    g.addEdge(trToPr, pr, target)
    g.addEdge(lc, pr, profile)
}
```

Listing 5: Generierte Gremlin-Abfrage und -Kommando zur TGG-Regel c) aus Abb. 8

```
...

[for (i :TGGRule | aClass.tggRule)]
  [file (i.name.concat('.groovy'), false)]
  { TransactionalGraph g ->
    // LHS of [i.name/]
    t = new Table()
    [for (j :TGGObjectVariable | i.objectVariable)]

      [if (j.domain.type.toString().equalsIgnoreCase('CORRESPONDENCE'))]
       [if (not j.bindingOperator.toString().equalsIgnoreCase('CREATE'))]
          [for (l :TGGLinkVariable | aClass.tggRule ->
            select(name.equalsIgnoreCase(rname)).linkVariable)]
            [if (l.source.name.toString()
              .equalsIgnoreCase(j.name.toString()))]
             g.v(name,[j.name/]).as(x)
               .out([l.name/]).as(y)
               .table(t).loop(x)
            [/if]
          [/for]
       [/if]
      [/if]
      ...
      [if (j.domain.type.toString().equalsIgnoreCase('SOURCE'))]
        (g.v(name,[j.name/]))
      [/if]
      ...
    [/for]
    ...

    // RHS of [i.name/]
    [AddVertices(i)/]
    [AddEdges(i)/]
    ...
  }
```

```
36    [/file]
37  [/for]
38
39  ...
40
41  [template public AddVertices(rule :TGGRule)]
42    [for (k :TGGObjectVariable | rule.objectVariable)]
43      [if (k.bindingOperator.toString().equalsIgnoreCase('CREATE'))]
44        [if (k.domain.type.toString().equalsIgnoreCase('CORRESPONDENCE'))]
45          g.addVertex(name,[k.name/])
46        [/if]
47      [/if]
48    [/for]
49    ...
50  [/template]
51
52  [template public AddEdges(rule :TGGRule)]
53    [for (k :TGGLinkVariable | rule.linkVariable)]
54      ...
55      g.addEdge([m.source.name/]], [m.target.name/], [m.name/])
56      ...
57    [/for]
58    ...
59  [/template]
60  ...
```

Listing 6: Auszug des Templates zur Generierung von Gremlin-Queries aus TGGs in Acceleo

Die Ausführung der generierten Regel-Skripte erfolgt in einem kompakten, einbettbaren `Groovy`-Skript. `Groovy` ist eine von *Apache* veröffentlichte Skriptsprache, die auf der *Java Virtual Machine (JVM)* ausgeführt wird [30], [31]. Sie vereinfacht und erweitert die Sprachkonzepte von Java, ist aber nicht weniger mächtig. Die Sprache hat eine hohe Ähnlichkeit zu Python und kann in verschiedene Umgebungen wie eine Datenintegrationsumgebung eingebettet werden.

Ein großer Vorteil von `Groovy` ist das Konzept der *Closures* [32]. Closures sind anonyme Funktionen, die wie Lambda-Funktionen für `Java` funktionale Programmierparadigmen in die Sprache `Groovy` bringen. Sie sind aber darüber hinaus eigene Objekte einer dedizierten `Closure`-Klasse. Closures werden in diesem Ansatz verwendet, um via `Acceleo` generierte Skripte (je ein Skript pro Regel) als Datei in das eingebettete Groovy-Skript zu laden, um sie dann inkrementell auszuführen. Listing 7 zeigt eine einfache Klasse `TransformationRule`, die ein solches Regel-Closure entgegennimmt und innerhalb einer sicheren `Gremlin`-Transaktion ausführen kann. Listing 8 beschreibt das einbettbare Skript, um diese Closure-Skripte aus Dateien zu laden und über die `TransformationRule`-Klasse auszuführen. Dabei ist die Reihenfolge des Ladens der Skripte und damit

der Regelausführung maßgeblich relevant. Da dieses Verfahren kein Backtracking implementiert, müssen die Regeln und ihre Reihenfolge vorab auf der Modelebene auf eine Vollständigkeit der Transformation hin validiert werden. Es ist vorstellbar, die Liste der verfügbaren Closure-Skripte automatisch zu ermitteln oder sogar ein CLI-Skript aus dieser Prozedur zu erzeugen.

```
class TransformationRule {
    String generatedRuleFile // file containing generated rule code

    void apply(TransactionalGraph graph) {
        Closure rule = new GroovyShell().evaluate(
            (new File(generatedRuleFile)).getText('UTF-8')
        )
        graph.tx().open() // open transaction
        rule.call(graph) // run generated transformation rule
        graph.tx().commit() // persist graph after applying rule
    }
}
```

Listing 7: Groovy-Klasse zur Abbildung generierter Gremlin-Queries über Closures

```
// ...
TransactionalGraph graph = Neo4jGraph.open("/integrationGraph")

// order matters
List<String> ruleFiles = [
    './ruleA.groovy',
    './ruleB.groovy',
    // ...
]

List<TransformationRule> rules = ruleFiles.collect {
    file -> new TransformationRule ( generatedRuleFile: file )
}

rules.each { rule -> rule.apply(graph) }
// ...
```

Listing 8: Groovy-Skript zur Ausführung der generierten Gremlin-Queries in Neo4j

4 Auswertung

4.1 Korrektheit

Um die Korrektheit des vorliegenden Ansatzes zu beweisen, also ob die über `Acceleo` generierten `Gremlin`-Kommandos und Abfragen äquivalent zu den formulierten TGG-Regeln sind, werden die Regeln innerhalb des EMF auf ein Instanzmodell und die generierten Transformationsbefehle in `Gremlin` auf einen zu diesem Instanzmodell äquivalenten Graph in `neo4j` angewendet. Bei Korrektheit der Methode muss ein Isomorphismus zwischen dem resultierenden Instanzmodell und dem Zielgraphen bestehen. Dieser Isomorphismus wird nachfolgend bewiesen [8]. Durch die Erzeugung der Testfälle durch `NeoEMF` kann die Äquivalenz des initialen Source-Instanzmodells und des entsprechenden Graphen angenommen werden:

$$G_{emf} \cong G_{neo}$$

Eine Transformation $G_{emf} \overset{TGG}{\rightarrow} G_{emf'}$ eines Modells mittels einer TGG im EMF kann wie folgt beschrieben werden, wobei jede Regel $r_1, ...r_n$ der TGG einen einzelnen Transformationsschritt beschreibt:

$$G_{emf} = G_{emf_0} \overset{r_1}{\rightarrow} ... \overset{r_n}{\rightarrow} G_{emf_n} = G_{emf'}$$

Das gleiche Abbildungsschema gilt auch für die Graphtransformation $G_{neo} \overset{\texttt{Gremlin}}{\rightarrow} G_{neo'}$ in `neo4j`. Dabei sind $q_1, ..., q_n$ die aus den Regeln $r_1, ..., r_n$ generierten `Gremlin`-Kommandos/-Abfragen:

$$G_{neo} = G_{neo_0} \overset{q_1}{\rightarrow} ... \overset{q_n}{\rightarrow} G_{neo_n} = G_{neo'}$$

Ist die Korrespondenz der TGG-Regeln r_{i+1} zum generierten `Gremlin`-Code q_{i+1} sichergestellt, so gilt $G_{emf_i} \cong G_{neo_i} \implies G_{emf_{i+1}} \cong G_{neo_{i+1}}$. Per Induktion ist somit der Isomorphismus des resultierenden Zielgraphen zum transformierten Instanzmodell und somit die Korrektheit der Transformation bewiesen:

$$G_{emf'} \cong G_{neo'}$$

5 Fazit und zukünftige Arbeiten

Der dargestellte Ansatz erlaubt die Verwendung von Triple-Graph-Grammatiken (TGG), um die Datenintegration in Graphdatenbanken auf Modelebene beschreiben zu können. Diese Methode ermöglicht somit eine höhere Abstraktionsebene zur Definition von Datenintegrationen. Durch eine solche Abstraktion können an der Datenintegration Beteiligte zukünftig in der Lage sein, ihre Integrationen zu beschreiben, ohne Kenntnisse der darunter liegenden technischen Ebene zu haben, was die Zusammenarbeit für Unternehmen und Forschende vereinfacht. Ferner kann durch dieses Verfahren die Qualität von Datenintegrationen sichergestellt werden, da die Integration auf Modelebene mit Werkzeugen des Eclipse Modeling Frameworks vor der Ausführung auf den produktiven Daten in der Graphdatenbank validiert werden kann. Die Integration skaliert zudem gut mit der Menge

der zu integrierenden Daten durch die Verwendung von Graphdatenbanken und damit dem Verzicht auf die Notwendigkeit exzessiver Joins.

Das Verfahren verwendet `neo4j` zusammen mit der Abfragesprache `Gremlin` als zugrundeliegende Datenbank, doch es sei an dieser Stelle zu erwähnen, dass auch andere Graphdatenbanken `Gremlin` (zum Teil nativ) unterstützen. Das Verfahren sollte dank der Unterstützung des *Tinkerpop-Blueprints*-Interfaces in `NeoEMF` [33] auch auf diesen Datenbanken anwendbar sein.

Die Forschungsarbeiten beschreiben nicht, wie die Transformation auf Attributebene durchzuführen ist. TGGs ermöglichen aber auch die Beschreibung von Transformationen auf Attributebene, so dass die Generierung des entsprechenden `Gremlin`-Codes unproblematisch sein sollte. Im Prinzip lassen sich die Attribute eines Property-Graphen auch beschreiben, indem der Graph als erweiterter Graph (*E-Graph*) notiert wird, also Attribute als weitere Knoten (= Attributwerte) und verbindende Kanten (= Attributbezeichner) dargestellt werden [34]. Dies ist insbesondere dann ein wichtiger Betrachtungsgegenstand, wenn Attribute aggregiert [28] oder Knoten zusammengefasst werden, da ansonsten das Binding und Backtracking scheitern können. Des Weiteren ist nicht beschrieben, wie Kanten des Zielgraphen einer Transformation dem Schema eines Property-Graphen entsprechend, also eine tatsächliche Beziehung beschreibend, benannt werden können und wie mit umgekehrten Kantenrichtungen umzugehen ist. Es ist außerdem notwendig, eigene Regeln für Kanten zu entwerfen, wenn Attribute von Kanten Bestandteil der Datenintegration sein sollen, da diese auch Korrespondenzen benötigen. Dies ist auch dann notwendig, wenn es unterschiedliche Kantentypen gibt, welche aber die gleichen Knotentypen verknüpfen (zum Beispiel zur Darstellung unterschiedlicher Personenrollen). Außerdem werden Graph-Parsing-Strategien zur Bestimmung der Ausführungsreihenfolge der Regeln und möglicherweise notwendiges Backtracking nicht diskutiert. Um eine valide Transformation respektive Datenintegration zu erhalten, müssen die Regeln zuvor (zum Beispiel auf entsprechenden Instanzmodellen) dahingehend getestet werden, ob nach Ausführung des Regelsets auch alle Graphelemente gebunden sind.

Aktuelle und zukünftige Arbeiten beschäftigen sich mit der Integration von mehr als zwei Datenquellen, der Implementierung von Synchronisationsregeln und weiterer Bedingungen für TGGs (zum Beispiel negatives Matching) sowie der Optimierung der Werkzeugkette. Da das Verfahren bisher nur Regeln für Vorwärts- und Rückwärtstransformationen implementiert, können nur initiale Transformationen durchgeführt werden. Auf laufzeitrelevante Änderungen am Schema eines Graphen kann zurzeit nicht reagiert werden. Zu einer solchen inkrementellen Transformation notwendige Synchronisationsregeln, die automatisch auf Änderungen reagieren, sind Gegenstand der aktuellen Forschung. Dazu sollen Änderungen (Deltas) des Quell- und Zielgraphen mittels `ChangeFeed` [35] von *GraphAware* automatisiert verfolgt werden.

Die verwendeten Werkzeuge und Frameworks sind zurzeit noch schwierig zu bedienen, benötigen viele manuelle zeitintensive Zwischenschritte und sind teilweise unpraktisch, was für größere Integrationsprojekte unvorteilhaft ist. Außerdem ist noch ein tiefergehendes technisches Verständnis erforderlich, um

das Verfahren anzuwenden. Um diesem Problem zu begegnen, wird ein weiteres Werkzeug entwickelt, welches die Werzeugkette orchestriert und bis zu einem gewissen Grad automatisiert sowie die Abhängigkeiten dieser untereinander verwaltet. Des Weiteren soll die Erstellung von TGG-Regeln durch die Auswahl entsprechender Editoren für Anwender graphischer und verständlicher gemacht werden [36], [37].

Referenzen

[1] M. Lenzerini, "Data integration: A theoretical perspective," in *Proceedings of the twenty-first acm sigmod-sigact-sigart symposium on principles of database systems*, 2002, pp. 233–246, doi: 10.1145/543613.543644.

[2] A. Halevy, A. Rajaraman, and J. Ordille, "Data integration: The teenage years," in *Proceedings of the 32nd international conference on very large data bases*, 2006, pp. 9–16.

[3] C. Vicknair, M. Macias, Z. Zhao, X. Nan, Y. Chen, and D. Wilkins, "A comparison of a graph database and a relational database: A data provenance perspective," in *Proceedings of the 48th annual southeast regional conference*, 2010, doi: 10.1145/1900008.1900067.

[4] V. M. de Sousa and L. M. del V. Cura, "Logical design of graph databases from an entity-relationship conceptual model," in *Proceedings of the 20th international conference on information integration and web-based applications & services*, 2018, pp. 183–189, doi: 10.1145/3282373.3282375.

[5] A. Schürr, "Specification of graph translators with triple graph grammars," in *International workshop on graph-theoretic concepts in computer science*, 1994, pp. 151–163.

[6] F. Hermann, H. Ehrig, U. Golas, and F. Orejas, "Formal analysis of model transformations based on triple graph grammars," 2014.

[7] A. Königs and A. Schürr, "Tool integration with triple graph grammars-a survey," *Electronic Notes in Theoretical Computer Science*, vol. 148, no. 1, pp. 113–150, 2006.

[8] A. Alqahtani and R. Heckel, "Model based development of data integration in graph databases using triple graph grammars," in *Federation of international conferences on software technologies: Applications and foundations*, 2018, pp. 399–414.

[9] A. Doan, A. Halevy, and Z. Ives, *Principles of data integration*. Elsevier, 2012.

[10] P. Barceló, J. Pérez, and J. Reutter, "Schema mappings and data exchange for graph databases," in *Proceedings of the 16th international conference on*

database theory, 2013, pp. 189–200, doi: 10.1145/2448496.2448520.

[11] "Free company data product." CompaniesHouse, http://download.companieshouse.gov.uk/en_output.html, 2020.

[12] *Neo4EMF, a scalable persistence layer for emf models*. https://emoflon.org/eclipse-plugin/emoflon_2.15.0/handbook/handbook_eMoflon_2.15.0.pdf: eMoflon team, 2016.

[13] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: Eclipse modeling framework*. Pearson Education, 2008.

[14] M. Hunger, *Import data into your neo4j database from the neo4j-shell command*. https://github.com/jexp/neo4j-shell-tools: neo4j, 2017.

[15] D. R. T. Michael Hunger Ryan Boyd & William Lyon, *Graph database deployment strategies*. https://neo4j.com/blog/rdbms-graph-database-deployment-strategies/: neo4j, 2016.

[16] A. Benelallam, A. Gómez, G. Sunyé, M. Tisi, and D. Launay, "Neo4EMF, a scalable persistence layer for emf models," in *European conference on modelling foundations and applications*, 2014, pp. 230–241.

[17] J. Musset *et al.*, "Acceleo user guide," vol. 2, p. 157, 2006.

[18] R. Angles, "The property graph database model," in *AMW*, 2018.

[19] M. A. Rodriguez, "The gremlin graph traversal machine and language (invited talk)," in *Proceedings of the 15th symposium on database programming languages*, 2015, pp. 1–10, doi: 10.1145/2815072.2815073.

[20] C. T. Have and L. J. Jensen, "Are graph databases ready for bioinformatics?" *Bioinformatics*, vol. 29, no. 24, p. 3107, 2013.

[21] T. Schindler, "Anomaly detection in log data using graph databases and machine learning to defend advanced persistent threats." 2018, [Online]. Available: http://arxiv.org/abs/1802.00259.

[22] P. Cudré-Mauroux and S. Elnikety, "Graph data management systems for new application domains," *Proceedings of the VLDB Endowment*, vol. 4, no. 12, pp. 1510–1511, 2011.

[23] R. Angles, "A comparison of current graph database models," in *2012 ieee 28th international conference on data engineering workshops*, 2012, pp. 171–177.

[24] *Neo4j constraints*. https://neo4j.com/docs/cypher-manual/current/administration/constraints/: neo4j, 2020.

[25] H. Giese, S. Glesner, J. Leitner, W. Schäfer, and R. Wagner, "Towards verified model transformations," in *Proc. Of the 3rd international workshop*

on model development, validation and verification (modev 2a), genova, italy, 2006, pp. 78–93.

[26] H. Giese and R. Wagner, "From model transformation to incremental bidirectional model synchronization," *Software & Systems Modeling*, vol. 8, no. 1, pp. 21–43, 2009.

[27] W. Schäfer and G. Engels, "Graphtransformationen für komponentenbasierte softwarearchitekturen," p. 15, 2008.

[28] F. Hermann *et al.*, "Model synchronization based on triple graph grammars: Correctness, completeness and invertibility," *Software & Systems Modeling*, vol. 14, no. 1, pp. 241–269, 2015.

[29] N. Weidmann, A. Anjorin, L. Fritsche, G. Varró, A. Schürr, and E. Leblebici, "Incremental bidirectional model transformation with eMoflon:: IBeX. In 2019 (ceur workshop proceedings), james cheney and hsiang-shang ko (eds.), vol. 2355. CEUR-ws. Org, 45–55." 2019, doi: 10.1007/978-3-030-23611-3_8.

[30] "Groovy - a multi-faceted language for the java platform." Apache, https://groovy-lang.org, 2020.

[31] D. Koenig, A. Glover, P. King, G. Laforge, and J. Skeet, *Groovy in action*. Manning Publications Co., 2007.

[32] *Groovy - closures*. https://groovy-lang.org/closures.html: Apache, 2020.

[33] *Supported backends - neoemf wiki*. https://github.com/atlanmod/NeoEMF/wiki/Supported-Backends: Atlanmod, 2018.

[34] N. Khan, "Concept and implementation of an extension of a visualisation tool for the flowchart-based visual satellite control language spell-flow," 2015.

[35] GraphAware, "GraphAware neo4j changefeed." https://github.com/graphaware/neo4j-changefeed, 2016.

[36] E. Kindler and R. Wagner, "Triple graph grammars: Concepts, extensions, implementations, and application scenarios," Technical Report tr-ri-07-284, University of Paderborn, 2007.

[37] D. Binanzer and C. Ermel, "Konzeption und implementierung eines werkzeugs zur konfliktanalyse von emf-modelltransformationen mit tripelgraphgrammatiken," 2012.