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

Abdullah Alqahtani and Reiko Heckel

Department of Informatics,
University of Leicester, UK
`{aqa2,rh122}@le.ac.uk`
`http://www2.le.ac.uk`

Abstract. Graph databases such as `neo4j` are designed to handle and integrate big data from heterogeneous sources. For flexibility and performance they do not ensure data quality through schemata but leave it to the application level. In this paper, we present a model-driven approach for data integration through graph databases with data sources in relational databases. We model query and update operations in `neo4j` by triple graph grammars and map these to `Gremlin` code for execution. In this way we provide a model-based approach to data integration that is both visual and formal while providing the data quality assurances of a schema-based solution.

Keywords: Data Integration; Graph Databases; Model-based Development; Triple Graph Grammars

1 Introduction

Data integration is the process of combining data from heterogeneous sources in a unified and consistent way [11]. In changing market conditions businesses have to be flexible, able to merge, cooperate with or acquire other businesses [9]. To work together effectively, such newly related businesses will have to integrate at least some of their data. As with application integration in general, approaches to data integration should support flexibility of future evolution and allow to share data while retaining ownership. For example, two businesses may agree to share their customer, product and supplier data but keep their internal processes separate. In such scenarios, data integration should be loose and partial to guarantee sustainability in the face of changing business goals [22].

Business data is often high in volume and velocity of change [11]. Data sources may be too large to replicate or merge fully and both data and data models may undergo changes at different rates. Keeping data loosely coupled or linked, it is easier to maintain integration [31]. Unfortunately, legacy data integration concepts do not address these requirements. Graph databases (GDB) such as `neo4j` [27] provide a scalable semi-structured data store based on a simple and flexible graph data model. They are able to store large data sets [21] and query them efficiently using navigational query languages such as `Cypher` and `Gremlin` [14]. Triple graph grammars (TGG) are a declarative language to relate heterogeneous data through a relational structure. They support uni- and bidirectional transformations as well as linking existing data [25]. TGGs have originally been developed for model transformation, integration and synchronisation [17, 10].

In this paper, we present an approach using TGG as data integration language on top of a GDBs. This is supported by the generation of code from TGG rules for GDB query and update operations. The overall architecture is shown in Figure 1. Data integration is specified at a model layer, describing sources using UML models and the links to be created between them using TGG rules. This also allows to maintain the consistency of links and to update them when the source data changes. To implement the data integration, TGG rules are translated to the agnostic GDB query language `Gremlin` [24]. Then, the relevant data is imported to the GDB through source adapters, such that the `Gremlin` code can be executed.

Advantages of this approach are (i) the level of abstraction is increased due to the use of model-driven technology, (ii) the use of a GDB to maintain the link structure supports scalability, (iii) schema safety is maintained due to the use of typed TGG rules at model level and their correct translation to `Gremlin`, (iv) the visual nature of the model will allow business experts to understand and support the development of the integration.

In particular, correctness and scalability will be evaluated experimentally.

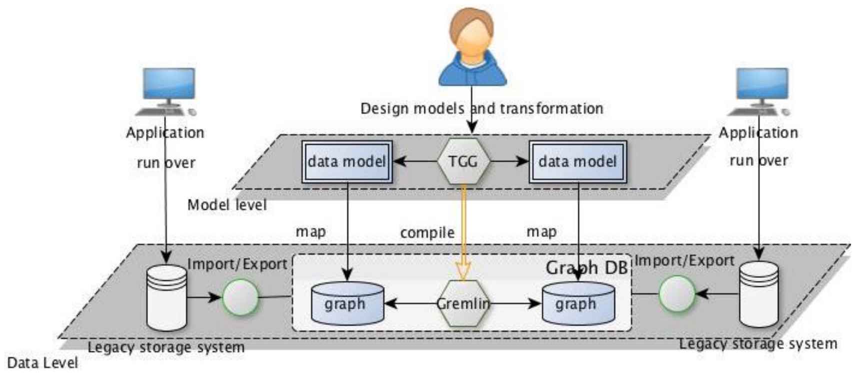

Fig. 1. Presenting TGG and UML Models for Data Integration in Graph Databases.

The remainder of the paper is organised as follows: Section 2 motivates the problem by an integration scenario. Section 3 introduces the overall approach. Model-level data integration using TGGs is discussed in Section 4. Section 5 defines the mapping from TGGs to `Gremlin` while Sections 6 and 7 present the evaluation of the approach and related work respectively. Section 8 concludes the paper and discusses future work.

2 Application Scenario

The domain of business data provision is used as a running example throughout paper. CompaniesHouse (CH) and CompanyCheck (CC) are two well-known UK business data providers. Both are responsible for filing data of all limited companies in the UK providing data on four million companies to be accessed online or downloaded [1]. However, the data provided by both sources is generally not consistent, differing in what data is provided per company but also in the semantics of apparently shared fields. In general, data can be incomplete or missing.

To mitigate these problems and benefit from the full extent of the data provided, we would like to link corresponding records in both databases and support mutual updates in both directions. More precisely, the requirements for integration are as follows:

① Bidirectional Integration: In this scenario data may be moved from CH to CC or vice versa. In case either side are missing data, they should be updated accordingly while retaining independent ownership.
② Scalability: Data is provided for millions of companies, so scalability of the integration process is important.
③ Visual Integration: Business data integration requires input by domain experts. This should be supported by visualising the integration components and rules.
④ Agile Integration: To be able to evolve our understanding of the integration incrementally, we have to support partial integration and existing rules and models should be easy to modify [15].
⑤ Heterogeneity: Both domains structure their data differently. Data representing the same objects are described using different names. Thus integration should be able to map heterogeneous representations into a shared consolidated structure.

3 Model Driven Approach to Data Integration

Our approach is based on data modelling and model transformation rules to be compiled into GDB code for execution. The two levels are briefly discuss below.

3.1 Model Level

This layer describes the integration at a high level of abstraction by representing source data models using UML class diagrams. This provides a platform-independent in terms of common modelling features such as classes, attributes, associations, etc. Relations between source data models are described using TGG rules, which can be used to link, synchronise and map data between the different models [12, 18]. TGG rules are created by Eclipse Modelling Framework (EMF) and `eMoflon` [26, 2]. In particular, `eMoflon` is a metamodelling tool for creating and executing TGG rules.

TGGs are used to create and maintain relations between source and target elements [12, 19]. Such relations can be used to update their constituents incrementally thus supporting data evolution. Therefore, despite their origins in meta modelling, TGGs can be used as a data integration and mapping language at the application level. Model transformation using TGGs to support data integration in GDBs has been suggested for these reasons. First, TGGs provide a solution to link heterogeneous components, establishing correspondences between elements that describe or share similar information, hence supporting requirement ⑤. They can copy and update such data if needed, e.g., to react incrementally to changes on either side in the integration, supporting requirement ④. Second, TGGs are a visual query language for GDBs, meeting requirement ③. Third, TGGs support bidirectional transformations, meeting requirement ①.

3.2 Implementation Level

UML models and TGG rules are mapped into `noe4j` property graphs and `Gremlin` queries. This allows us to leave the execution to the GDB. UML models are mapped to `neo4j` using the `NeoEMF` framework [4]. The mapping from TGGs to `Gremlin` is implemented using the `Acceleo` tool [23]. Data sources are imported via CSV files, loaded to `neo4j` using the `neo4j-shell` commands [16]. Such commands are modified based on the model mapping using `NeoEMF`.

Due to the scalability of `neo4j`, its use together with `Gremlin` to execute the integration helps us meet requirement ②. In addition to the incremental development of TGG rules, the flexibility of schemaless data in the GDB also supports requirement ④.

4 Model-Level Data Integration using TGGs

Declarative TGG rules describe how two models are related, however, these relations can be translated to perform batch transformation. The derived rules are needed to copy data back and forth between sources. In addition, translation of TGG can also be used to relate elements at different sources which describe the same phenomena without moving data across. Such different translations depend on the same TGG specification between the sources [12]. Therefore, we derive three types of which are forward, backward, and consistency checking rules (see [12]). However, due to lack of space we will only show an example of forward transformation in Section 5.

Figure 2 shows the class diagrams for both domains, CompaniesHouse (CH) in 2(a) and CompanyCheck (CC) in 2(b). CH contains the main Company class as well as information such as the registered address (RegAddress) of the company, Account, Managers, and Trading classes. In the CC domain most information is similar in content but differently structured and named. This mismatch needs to be resolved conceptually, using the declarative TGG rules to relate corresponding concepts, as well as operationally by deriving the relevant TGG data transformation rules.

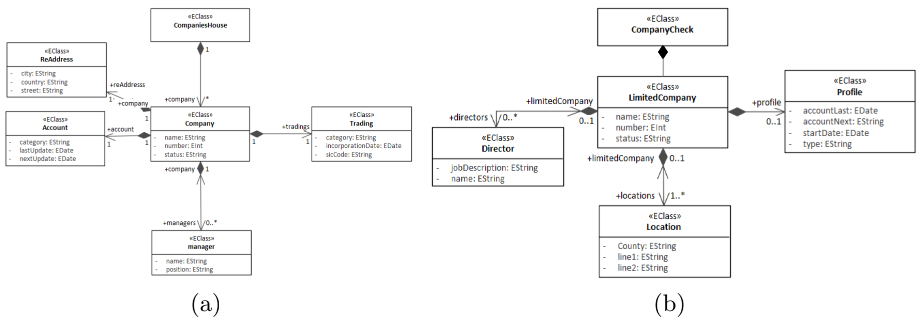

Fig. 2. 2(a) CompaniesHouse model and 2(b) CompanyCheck model

Figure 3 presents six TGG rules. Rules are created using the TGG tool in `eMoflon`. Rule 3(a) creates a pair of classes, CH and CC, without any preconditions (i.e., an empty left-hand side). This type of rule refers to as an axiom, and it is applicable before and independently of any other rule. Rule 3(b) relates the Company and LimitedCompany classes. It requires the pattern created by 3(a) as precondition (left-hand side). This is denoted by black elements while the elements newly generated are shown in green. Rules 3(c), 3(d) and 3(f) are used to create consistent pairs of Trading with Profile, RegAddress with Location, and Manager with Director classes. They can all be applied independently since they do not depend on each other but only on Rule 3(b).

In rule 3(e), the Account class is created and linked to the Profile class from rule 3(d). The left-hand side includes the precondition pattern of 3(d) and the class Profile. The right-hand side only adds the Account class from the source domain and the AccountToProfile class from the correspondence domain. This rule presents a good example of how backward transformation can be derived from a TGG specification, E.g., a new element is created in one domain that relates to an existing elements of the other domain.

5 Mapping TGGs to Graph Databases

GDBs are based on a simple graph model known as *property graph*. It is defined by sets of nodes N and relationships R, attributed by key-value pairs known as properties. The `NeoEMF` framework [4] has been used to map EMF objects manipulated by TGG rules to corresponding `neo4j` nodes and edges.

We only map a subset of TGGs, which does not include all features such as negative application conditions. Mapping derives directed (operational) TGG rules for forward, backward and consistency transformation from the same declarative TGG rules. The left-hand side of a TGG operational rule is implemented using the pattern matching capability of `Gremlin`. If the `Gremlin` query is successful, the graph will be updated according to mutation statements derived from the right-hand side. Nodes and edges of a specific domain are created based on the type of the operational rule from given context elements.

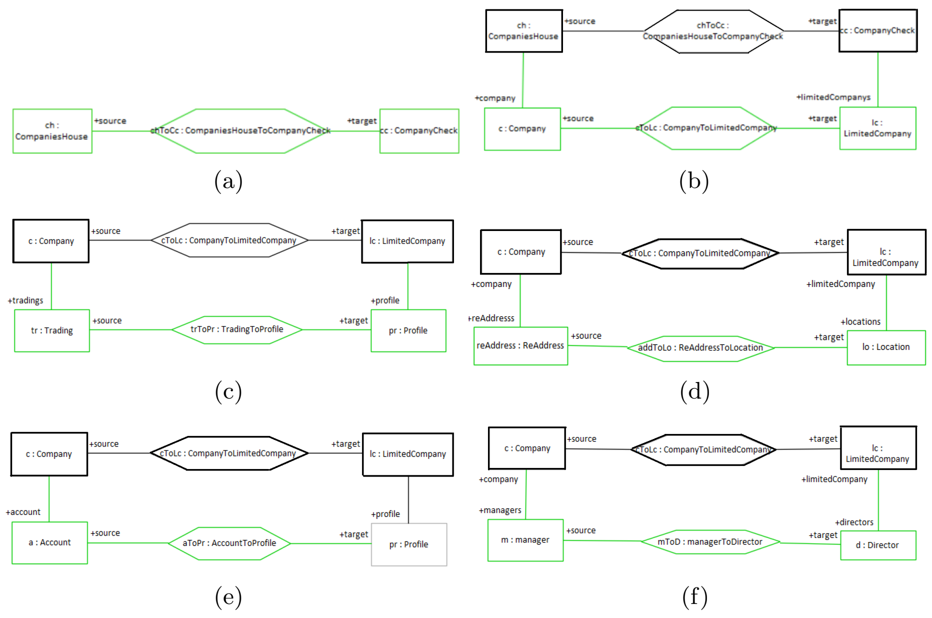

Fig. 3. TGG rules of running example

The concepts of Gremlin and TGGs can be aligned to demonstrate the mapping of rules into queries. TGG concepts are categorised based on the types of elements used in the transformation and their binding type, for instance objects and links. There are three types of bindings of TGG elements within rules: (1) check, (2) create, and (3) destroy [28]. The check type states a precondition of the TGG rule, requiring objects and/or links to be present to apply the rule. The create type is instructing objects or links to be created. In our mapping, we only consider check and create bindings. According to each binding type, elements and nodes are translated into the logic of Gremlin statements. Table 1 presents the mapping of concepts between TGGs and Gremlin based on the binding types of links and objects.

For illustration of the mapping, we focus on a forward transformation rule derived from TGG rule depicted in 3(b) of Figure 3. In such a rule, correspondence and target elements are created from given source and context elements. Figure 4 shows the forward rule, whose left-hand side is denoted by black elements and the new elements in the right-hand side are denoted by green elements. Our mapping example only uses directed rules for forward and backward translation. However, as discussed previously, from the same symmetric TGG productions we can generate consistency checking translation rules.

We implemented the mapping using Acceleo, a general EMF-based tool to generate text from models [23]. In our translation, we rely on the TGG metamodel defined by eMoflon [28]. The metamodel is fed to the Acceleo genera-

Table 1. Mapping of TGG rules to Gremlin

TGG Elements	GDB Elements	Semantics	
		TGG	Gremlin Expression
TGG Object	Neo4j Vertex	Check Only Binding (LHS)	- graph.v(VertexName) for single vertex matching - graph .V(VertexName).as('x').out (EdgeLabel).as('y') - graph .V(VertexName).outE (EdgeLabel). inV(VertexName) For Multi-vertices matching
		Create Binding (RHS)	graph.AddVertex(Properties)
TGG Link	Neo4j Edge	Check Only Binding (LHS)	- graph.e(EdgeId) For single Edge matching - graph.v(VertexName).bothE For multi-edges matching of a specific vertex
		Create Binding (RHS)	- graph.AddEdge(Properties)

tor which uses metamodel elements based on designated templates to generate `Gremlin` code.

Pattern Matching (LHS) This phase involves matching entities of the left-hand side pattern graph with corresponding elements in the property graph. That means, the pattern is interpreted as a query. Pattern matching is the most expensive part of executing graph transformations [5]. One of the features of `Gremlin` is to provide efficient techniques for complex pattern matching regardless of the size of the graph. Therefore, we use `Gremlin` to implement pattern matching. For example, the left-hand side of the rule in Figure 4 is encoded by the `Gremlin` query in Listing 1.1, showing how context elements are retrieved and matched using *out* and *as* steps, and how the result is stored and returned in the form of a *table*.

Listing 1.1. Gremlin query for LHS of forward rule in Fig. 4

```
t= new Table ()
g.v('name', chToCc).as('x').
    out(target).as('y').table(t).loop('x')
=> [x : v(5) , y: v(8)]
```

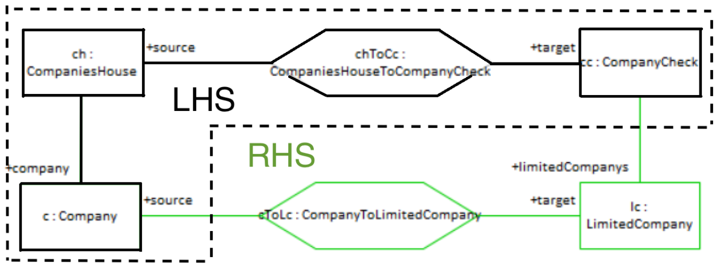

Fig. 4. Compact representation of forward transformation rule derived from 3(b) in Figure 3

```
g.v('name', chToCc).as('x').
    out(source).as('y').table(t).loop('x')
=> [x : v(5) , y: v(6)]
g.v(name, c)
=> v(7)
```

In the above `Gremlin` code we first find all target nodes of the outgoing *target* and *source* edges from the *chToCc* node. The table step selects the 1st and 2nd edge of the all paths that reach the nodes and inserts them into the table t as rows. The loop step is used for recursive matching, i.e., if there are any multiply connected nodes with the same link label. Finally, the query returns the node of the source model *c* of type *Company*.

Graph Manipulation (RHS) This phase executes the creation and deletion of nodes and edges of the target graph. We implement these operations using `Gremlin` methods and statements. Data methods to update the graph are implemented within a transactional block to ensure that each transformation step is atomic [30]. Methods include `graph.addVertex()` and `graph.addEdge()` for node and edge creation and `graph.removeVertex, graph.removeEdge()` for node and edge deletion. However, for operational rules derived from TGGs, deletion operations are not needed. Referring to our forward rule in Figure 4, its righ-hand side is implemented using the following `Gremlin` code using the query defined in Listing 1.1.

Listing 1.2. Gremlin code for RHS of forward rule in Fig 4

```
graph.tx().onReadWrite(Transaction)
g.addVertex('name',cToLc )
g.addVertex('name', lc)
g.addEdge( cToLc , c, source)
g.addEdge( cToLc,  lc, target)
g.addEdge(cc, lc, limitedCompanys)
graph.tx ().commit()
```

The above code creates nodes *cToLc* in Line 2 and *lc* in line 3. Edges of type *source, target* from correspondence to source and target nodes are created in Lines 4 and 5. Line 6 shows the creation of a *limitedCompanys* edge required for conformance to the CC class diagram.

6 Experimental Evaluation

The queries generated by `Acceleo` from TGG rules in `eMoflon` are evaluated for correctness and performance.

6.1 Correctness

To evaluate correctness of our approach, we need to ensure that the generated `Gremlin` queries implement the same behaviour of TGGs. We generate test cases based on the TGG specification, and map them to `neo4j` using the `NeoEMF` framework [4]. Then, `Gremlin` queries are applied using the blueprint interface. TGG transformations result in consistent pairs of source and target models. Based on the direction of transformation, source or target model serves as test inputs and transformation implemented by `Gramlin` is System Under Test (SUT). The target model of the TGG transformation represents the expected output [33]. The *test generator* [33] is implemented in `eMoflon` and uses TGG specification to generate valid and adequate test suits (pairs of test models and valid outputs based on TGG rules). The tool automatically generates test cases either for all TGG rules (large test cases) or individual TGG rules (small test cases). It is based on a grammar-based generation approach and uses auxiliary functions to support *traversal strategy* and *stop criterion*. It also passes the generated tests to a component that evaluates the quality of test cases based on gathered coverage data from the TGG metamodel and applicable rules and produces a quality report accordingly.

The generated test suit of each rule consists of five test cases. The stop criterion is based on the size of the sample model. The minimum size of tests is 20 elements, and the maximum size is 7,000 elements. They cover the structural features of the source model, as well as its classes, attributes and associations *w.r.t* applicable TGG rules. The number of objects in each test case and the selection of TGG rules are manually encoded before the generation. Then, we generate the test cases for the complete transformation. The traversal strategy of the applicable rules is parameterised based on the interdependency of the rules to meet a certain application sequence, i.e. in our running example, we have six rules (r(a), r(b), * r(c) * r(d) * r(e) * r(f)), such that r(a) is the axiom or initial rule, and r(b) depends on the application of r(a). The * means that the remaining rules can be randomly applied. We limited the number of applications of the axiom rule to one to ensure that we have one root element for every test graph. Each generated EMF object graph (test case) is mapped into a `neo4j` graph [4].

Table 2 presents a summary of the generated test cases and the applications of the rules (the elements of each rule). It is presented to demonstrate the quality of generated test cases. Each test case is bounded in size, as shown in the first row (numbers within brackets). The number of generated elements for each rule out of the defined upper bound is presented for each test case. Note that the test cases also cover the edges required by the TGG rules.

Table 2. Summary of applications for each rule of generated test cases

Rules	$\#Test_1(20)$	$\#Test_2(200)$	$\#Test_3(1000)$	$\#Test_4(3000)$	$\#Test_5(7000)$
r(a)	1/20	1/200	1/1000	1/3000	1/7000
r(b)	5/20	43/200	205/1000	602/3000	1358/7000
r(c)	4/20	42/200	189/1000	580/3000	1426/7000
r(d)	2/20	31/200	201/1000	611/3000	1398/7000
r(e)	4/20	42/200	197/1000	539/3000	1382/7000
r(f)	4/20	41/200	207/1000	613/3000	1435/7000

To compare the result of the execution of the queries with the expected output based on the TGG specification we have to establish a graph isomorphism between output graphs G_{neo} generated by `Gremlin` in `neo4j` with EMF graphs G_{emf} generated by the original TGG rules in `eMoflon`.

In the initial mapping to the GDB, we maintain consistency of identifiers of input model elements. This is ensured by running `EMFCompare` [29], a tool for comparing two EMF models, to match corresponding elements, creating a partial isomorphism that covers all elements retained from the given object graph. However, the newly created elements by `Gremlin` are not known to `NeoEMF` resource which makes it impossible to use `EMFCompare` to complete the test.

In addition to manual testing for small test cases via visualising the graphs using `Gephi` [3],a graph visualisation tool, we implemented the isomorphism test after each transformation step using the `igraph` package, employing a standard graph isomorphism algorithm [6]. `Igraph` implements the `VF2` isomorphism algorithm in Python. The `VF2` algorithm is a simple isomorphism check based on tree search and backtracking [20].

This is done incrementally after each application of a query. A TGG model transformation $G_{emf} \xrightarrow{*} G_{emf'}$ breaks down into individual steps $G_{emf} = G_{emf_0} \xrightarrow{r_1} \dots \xrightarrow{r_n} G_{emf_n} = G_{emf'}$ with rules $r_1, \dots, r_n$.

The same structure can be identified in the `neo4j` transformation $G_{neo} \xrightarrow{*} G_{neo'}$ as $G_{neo} = G_{neo_0} \xrightarrow{q_1} \dots \xrightarrow{q_n} G_{neo_n} = G_{neo'}$ with queries $q_1, \dots, q_n$ derived from the rules above.

In order to be correct, the execution should result in isomorphic graphs $G_{neo'} \cong G_{emf'}$. Based on the `NeoEMF` mapping we can assume $G_{neo} \cong G_{emf}$. Then, the isomorphism of $G_{emf_{i+1}}$ and $G_{neo_{i+1}}$ is obtained by the correspondence of r_{i+1} and q_{i+1} from isomorphic graphs G_{emf_i} and G_{neo_i}. By induction this ensures $G_{neo'} \cong G_{emf'}$.

We follow an iterative process to update `Acceleo` templates for preserving the behaviour of the TGG rules. We manually mutate the mapped inputs for some of the test cases (e.g. by inverting the directions of some edges or by changing the names of nodes) before the application of `Gremlin`, in which the application fails during the matching pattern phase for respective queries posed against the modified structure; therefore, no updates were made as we instructed our implemented GDB application. The reason for the mutation step is to ensure that the modified graphs cannot be executed by the generated `Gremlin`. Therefore, we produce invalid graphs as a part of the testing because the tool only generates valid pairs of source and target models.

Both $G_{neo'}$ and $G_{emf'}$ are incrementally sent to the `igraph` function, which returns the isomorphism of the valid generated graphs (after attempts of corrections) and the *non-isomorphism* for mutated graphs. To correct the errors in the `Acceleo`, we relied on small test cases to visualise the graphs and compare them manually because the `igraph` does not manifest the differences for the given graphs. Most of the errors were `Gremlin` based, hence scripts were updated without changing the main mapping rules. We run the function after each correction attempt until we ensured that the given (valid) graphs are isomorphic. We complete our test by successfully covering all generated test cases.

6.2 Performance

We conducted an experiment to compare execution times of TGGs using `eMoflon` and of the translated `Gremlin` queries using `neo4j`. Results show that the GDB engine provides a highly scalable platform, executing our queries on large graphs. Using the six rules discussed in Section 4 we execute a complete forward transformation from CH to CC. Source models of various size have been automatically generated using model generators based on the TGG rules. These provide the input EMF graphs G_{emf} which are mapped to `neo4j` using `NeoEMF` into property graphs G_{neo}. Then, TGG rules are applied using `eMoflon` and `Gremlin` queries are applied using `neo4j` and the times for both executions are measured. The experiment has been conducted on Macintosh machine with a 2.5 GHz Intel Core i7 processor and 16 GB 1600 MHz DDR3 RAM memory. The results show that `neo4j` outperforms the execution using `eMoflon`, especially with larger graphs of more than 100,000 elements. Figure 5 plots execution times in seconds.

In our application scenario discussed in Section 2, information of 200,000 companies requires one million data objects which have been executed in 180 seconds which was the largest information that can be transformed using `eMoflon` using our example.

CompaniesHouse provides data of approximately four million companies. For this purpose, we execute the `Gremlin` queries alone to evaluate the scalability of the approach w.r.t real-world examples. Memory configurations of `neo4j` were modified to set the parameters of the heap size that is responsible for query execution and caching transactions as per `neo4j` recommendations. Based on the same hardware specification, the result shown in Figure 6 indicates that the approach scales up to transform information of 1.5 million companies, and can be executed in 1388 seconds. Due to memory overhead on the virtual machine, an error was thrown when transforming graphs consist of more than 8 million nodes.

To recap, the evaluation shows unsurprisingly that our approach achieves better performance than current implementation of `eMoflon` due to the use of GDB as the underlying storage and engine. Also, the approach can scale to large data, beyond the ability of the current EMF-based transformation tools.

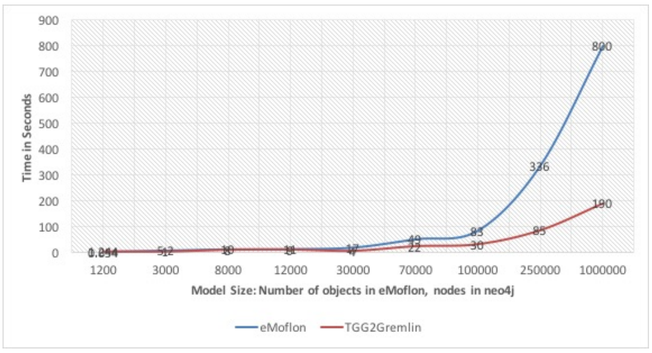

Fig. 5. Execution times of eMoflon and Neo4j

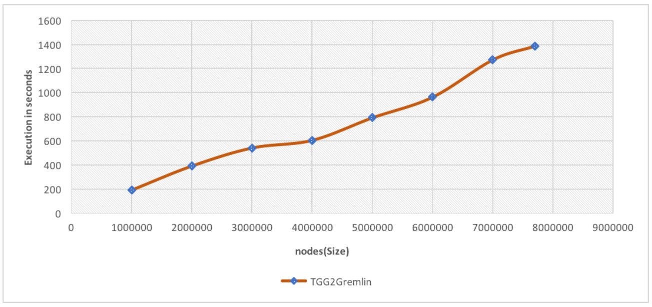

Fig. 6. Stand alone execution of Neo4j

6.3 Threats to Validity

There are some features that might threaten the validity of experimental evaluation. We discuss them briefly as follows: The limited number of TGG rules makes it difficult to generalise the conclusion. Only six TGG rules of a single transformation are tested. There is no firm conclusion on the effect of TGG complexity on execution time. E.g. there are other scenarios which require transformations where the individual rules may be large and complex. Although we only demonstrated an example of forward transformation, this threat is mitigated via maintaining non-functionality through code generation of backward transformation from the same specification. Thus, we rely on bidirectional behaviour of TGG in general rather than reducing complexity of the TGG design. We also rely on the fact of existence of similar integration examples especially in small and medium size businesses.

7 Related Work

We distinguish between the use of non-relational databases as scalable solutions for the implementation of MDE tools and as a scalable persistence layer for data-driven applications.

In [8], the authors propose mappings of UML models to GDBs in two stages. First, a model transformation is defined between the UML metamodel and the graph metamodel. Second, a framework is designed to generate Java code that can access the GDB. Our approach shares with this framework the concept mapping from class diagrams to the property graph model. However, we define a new mapping of TGG rules to native GDB code rather than access it by Java.

Mogwai [7] presents a lightweight model query language using the Object Constraint Language (OCL), extending the `NeoEMF` [4] mapping. They translates OCL into `Gremlin` expressions and compute queries on GDB representations of models. The approach uses model-to-model transformation from OCL to `Gremlin`. Instead, our approach uses model-to-text transformations based on `Acceleo` and supports side effects arising from update operations of translated TGG rules. Such implementation of `Acceleo` produces valid `Gremlin` queries *w.r.t* TGG rule without the need to establish a complex metamodel for `Gremlin` language.

In the second category, BXE2E [13] is a bidirectional approach to support import and export of electronic medical records. The study focusses on defining a mapping between the OSAR and E2E medical record systems based on embedding TGG rules that relate both data models. The implementation of BXE2E is based on Java code to support native operations of data transformation. To reduce execution costs, the design of the TGG rules does not use pattern matching, but encapsulates complex queries with lenses operations. Although this approach exploits TGGs for data-driven applications, it works with a restricted form of TGG rules which does not permit to use the full power of TGGs as a declarative language. Moreover, TGG rules do not describe bidirectional data integration for data-driven, schema-less applications.

GRAPE [32] presents as scalable graph transformation engine based on `neo4j`. It employs `Coljure` as domain-specific language for the textual syntax of rewriting rules and `GraphViz` for rule visualisation. Unidirectional rule operations such as addition, deletion and matching operations are compiled into the `Cypher` query language. This compilation utilises the pattern matching capability of

`Cypher` on `neo4j` graphs. The engine provides persistence backtracking facilities based on the property graph model and transaction features of `neo4j`. Despite the fact that the transformation engine is built on top of `neo4j`, imperative and unidirectional transformations are compiled into `Cypher` and do not avoid vendor lock-in since `Cypher` runs only on `neo4j`.

In our approach, TGGs are formally translated to an agnostic GDB query language to execute data operations for scalable performance. We compare our approach with aforementioned related work in terms of the requirements in Section 2. Agility is compared based on incremental development of TGGs to support evolving requirements. We also refer to the use of GDBs as a persistence and computation engine. Table 3 shows a summary of the comparison.

Table 3. General comparison of relevant approaches based on discussed requirement in Section 2

Approach	Bidirectional	GDB Support	Agility	Visual Query	Heterogeneity Support
Mogwai	✗	✓	✗	✗	✗
UMLtoGraphDB	✗	✓	✗	✓	✗
Grape	✗	✓	✗	✓	✗
BXE2E	✓	✗	✓	✓	✗
TGG2Gremlin	✓	✓	✓	✓	✓

8 Conclusion and Future Work

In this paper, we show a how graph databases and model-driven development tools can work together to build data integration solutions. Concepts and techniques of both technologies can lead to advantages at both design and execution level. As future work, we plan to enhance our mapping to cover most features and useful extensions of TGG such as negative application conditions in order to provide more options for data integration designers. Moreover, we plan to cope with a round-trip engineering of `Gremlin` queries to support the evolution of EMF models and changing requirements of the integration.

References

1. Companieshouse (2011), available at `https://assets.publishing.service.gov.uk/government/uploads/system/uploads/attachment_data/file/247006/0284.pdf`
2. Anjorin, A., Lauder, M., Patzina, S., Schürr, A.: eMoflon: Leveraging EMF and Professional CASE Tools. In: 3. Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe2011) (2011)
3. Bastian, M., Heymann, S., Jacomy, M., et al.: Gephi: an open source software for exploring and manipulating networks. Icwsm 8, 361–362 (2009)

4. Benelallam, A., Gómez, A., Sunyé, G., Tisi, M., Launay, D.: Neo4EMF, a scalable persistence layer for EMF models, pp. 230–241. Springer (2014)
5. Bergmann, G., Horváth, Á., Ráth, I., Varró, D.: A benchmark evaluation of incremental pattern matching in graph transformation. Graph Transformations pp. 396–410 (2008)
6. Csardi, M.G.: Package igraph 3(09), 214–217 (2013)
7. Daniel, G., Sunyé, G., Cabot, J.: Mogwaï: a Framework to Handle Complex Queries on Large Models. In: International Conference on Research Challenges in Information Science (RCIS 2016). Grenoble, France (Jun 2016), `https://hal.archives-ouvertes.fr/hal-01344019`
8. Daniel, G., Sunyé, G., Cabot, J.: UMLtoGraphDB: Mapping conceptual schemas to graph databases. In: Conceptual Modeling: 35th International Conference, ER 2016, Gifu, Japan, November 14-17, 2016, Proceedings 35. pp. 430–444. Springer (2016)
9. Fensel, D., Ding, Y., Omelayenko, B., Schulten, E., Botquin, G., Brown, M., Flett, A.: Product data integration in B2B e-commerce. IEEE Intelligent Systems 16(4), 54–59 (2001)
10. Giese, H., Hildebrandt, S.: Efficient model synchronization of large-scale models. No. 28, Universitätsverlag Potsdam (2009)
11. Halevy, A., Rajaraman, A., Ordille, J.: Data integration: the teenage years. In: Proceedings of the 32nd international conference on Very large data bases. pp. 9–16. VLDB Endowment (2006)
12. Hermann, F., Ehrig, H., Golas, U., Orejas, F.: Formal analysis of model transformations based on triple graph grammars. Mathematical Structures in Computer Science 24(04), 240408 (2014)
13. Ho, J., Weber, J., Price, M.: Bxe2e: A bidirectional transformation approach for medical record exchange. In: Guerra, E., van den Brand, M. (eds.) Theory and Practice of Model Transformation. pp. 155–170. Springer International Publishing, Cham (2017)
14. Holzschuher, F., Peinl, R.: Performance of graph query languages: comparison of cypher, gremlin and native access in neo4j. In: Proceedings of the Joint EDBT/ICDT 2013 Workshops. pp. 195–204. ACM
15. Hughes, R.: Agile Data Warehousing: Delivering world-class business intelligence systems using Scrum and XP. IUniverse (2008)
16. Hunger, M.: neo4j-shell tools. `https://github.com/jexp/neo4j-shell-tools` (2013), gitHub repository
17. Kindler, E., Rubin, V., Wagner, R.: An adaptable TGG interpreter for in-memory model transformation. In: Proc. of the 2nd International Fujaba Days 2004, Darmstadt, Germany. Technical Report, vol. tr-ri-04-253, pp. 35–38. University of Paderborn (Sep 2004)
18. Knigs, A., Schrr, A.: Tool integration with triple graph grammars - a survey. Electronic Notes in Theoretical Computer Science 148(1), 113 – 150 (2006), `http://www.sciencedirect.com/science/article/pii/S1571066106000454`, proceedings of the School of SegraVis Research Training Network on Foundations of Visual Modelling Techniques (FoVMT 2004)
19. Leblebici, E., Anjorin, A., Schrr, A.: A catalogue of optimization techniques for triple graph grammars. In: Modellierung. vol. 19, p. 21 (2014)
20. Levendovszky, T., Charaf, H.: Pattern matching in metamodel-based model transformation systems. Periodica Polytechnica Electrical Engineering 49(1-2), 87–107 (2006)

21. Miller, J.J.: Graph database applications and concepts with neo4j. In: Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA March 23rd-24th (2013)
22. Münch, T., Buchmann, R., Pfeffer, J., Ortiz, P., Christl, C., Hladik, J., Ziegler, J., Lazaro, O., Karagiannis, D., Urbas, L.: An innovative virtual enterprise approach to agile micro and sme-based collaboration networks. In: Working Conference on Virtual Enterprises. pp. 121–128. Springer (2013)
23. Musset, J., Juliot, É., Lacrampe, S., Piers, W., Brun, C., Goubet, L., Lussaud, Y., Allilaire, F.: Acceleo user guide. See also http://acceleo. org/doc/obeo/en/acceleo-2.6-user-guide. pdf 2 (2006)
24. Rodriguez, M.A., De Wilde, P.: Gremlin. URL: https://github. com/tinkerpop/-gremlin/wiki (2011)
25. Schürr, A.: Specification of graph translators with triple graph grammars. In: Graph-Theoretic Concepts in Computer Science. pp. 151–163. Springer (1995)
26. Steinberg, D., Budinsky, F., Merks, E., Paternostro, M.: EMF: Eclipse Modeling Framework. Pearson Education (2008)
27. The neo4j Team: (2018), `https://neo4j.com`, neo4j Graph Database Platform
28. eMoflon team, T.: An introduction to metamodelling and graph transformations with eMoflon. Technical Report, TU Darmsadt (2014)
29. Toulm, A., Inc, I.: Presentation of emf compare utility. In: Eclipse Modeling Symposium. pp. 1–8
30. Varró, G., Friedl, K., Varró, D.: Graph transformation in relational databases. Electronic Notes in Theoretical Computer Science 127(1), 167–180 (2005), proceedings of the International Workshop on Graph-Based Tools (GraBaTs 2004)
31. Wasserman, A.I.: Tool integration in software engineering environments. In: Long, F. (ed.) Software Engineering Environments. pp. 137–149. Springer Berlin Heidelberg, Berlin, Heidelberg (1990)
32. Weber, J.H.: Grape – A graph rewriting and persistence engine. In: de Lara, J., Plump, D. (eds.) Graph Transformation. vol. 10373, pp. 209–220. Springer International Publishing, Cham (2017)
33. Wieber, M., Anjorin, A., Schürr, A.: On the usage of TGGs for automated model transformation testing. In: Di Ruscio, D., Varró, D. (eds.) Theory and Practice of Model Transformations. vol. 8568, pp. 1–16. Springer International Publishing, Cham (2014)