Proceedings of the 33rd European Modeling & Simulation Symposium (EMSS), 020
18th International Multidisciplinary Modeling & Simulation Multiconference

2724-0029 © 2022 The Authors.
doi: 10.46354/i3m.2022.emss.020

Model Verification in Graph Databases and its Application in Neo4j

Christoph Praschl[1,*], Andreas Pointner[1], Oliver Krauss[1], Emmanuel Helm[1] and Andreas Schuler[1]

[1]University of Applied Sciences Upper Austria, Hagenberg i. M., Austria

[*]Corresponding author. Email address: christoph.praschl@fh-hagenberg.at

Abstract

This work introduces a concept for rule based model verification using a graph database on the example of Neo4j and its query language Cypher. An approach is provided that allows to define verification rules using a graph query language to detect transformation errors within a given domain model. The approach is presented based on a running example, showing its capability of detecting randomly generated errors in a transformation process. Additionally, the method's performance is evaluated using multiple subsets of the IMDb movie data with a maximum of 17,000,000 nodes and 41,000,000 relationships. This performance evaluation is carried out in comparison to the Object Constraint Language, showing advantages in the context of highly connected datasets with a high number of nodes. Another benefit is the utilization of a well established graph database as verification tool without any need for re-implementing graph and pattern matching logic.

Keywords: Model Verification; Graph-based Modeling; Neo4j; Cypher

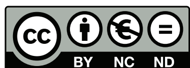
1. Introduction

In this work, an approach is presented that allows the verification of model transformations using graph databases, enabling efficient verifications. If an application already uses a graph database for persisting the data, the presented approach also has the advantage that no additional tooling is required for the verification, since the required verification mechanisms are already available in the database.

Model transformations are widely used in the context of software development to adapt a given model to a particular use case. One example for such transformations is the automatic code generation from UML diagrams as discussed by Sunitha and Samuel (2019), where a chart model is transformed to an abstract syntax tree model. To ensure the correctness of transformations, the resulting models have to be verified. For this task, this work introduces a verification approach based on graph databases.

Models abstract parts of the real world using only relevant properties. They are used in different aspects and stages in the process of software development – beginning with class diagrams using UML, to domain models or database models. In model driven development (c.f. Atkinson and Kuhne (2003)), models play a central role in creating software, by using models as a basis for creating code. For example, UML class models can be used for automatically creating the domain classes of a system for different or multiple programming languages.

In different stages of the development process, as well as in dissimilar parts of the actual system, it is common to have various models describing the same aspect from multiple perspectives. To transform from one model to another, several model transformation techniques are used. The correctness of the target model in reference to the source model has to be ensured. The challenge is to ver-



© 2021 The Authors. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY-NC-ND) license (https://creativecommons.org/licenses/by-nc-nd/4.0/).

ify if the information of the source model is transformed correctly into the target model and no information is lost. This is crucial since developers must be able to rely on the correct semantic information to transfer from one model to another as described by Lano et al. (2012a). Examples of verifications in the context of a model to model transformation like UML to relational database transformations are given by Lano et al. (2012b).

This work introduces an approach for model verification using a database and a query language for the definition of verification rules. The utilization of a graph database (GBD) is proposed instead of a relational database to overcome the problem of the object–relational impedance mismatch as described by Ireland et al. (2009). The presented approach is used to ensure the correctness of a transformation by analyzing the target model, or by comparing the source and the associated target instances. The general usage of the verification approach is shown based on a running example. The performance is evaluated based on a case study using the IMDb dataset (c.f. IMDb.com, Inc (2022)) and is compared to the state-of-the-art model verification framework the Object Constraint Language (OCL) introduced by OMG (2006).

This work aims to answer the following research questions:

- RQ_1: *How can model transformations be verified in graph databases?*
  It is examined if errors in a result model stemming from a faulty model transformation process can be detected using a GBD in combination with a query language.
- RQ_2: *How does a graph database based verification perform and scale?*
  The performance of a verification process can be a crucial point depending on the use case. In certain circumstances, a verification is needed in a short amount of time. Since a model instance can consist of millions of objects depending on the use case, the scaling of the approach can be another crucial factor besides the performance of the verification approach.

In context of the stated questions, this work makes the following contributions:

- A novel model verification approach using a GBD
- A case study evaluating the approach's performance
- A comparison between the presented approach and OCL

2. Approach

The verification approach is described based on the graph database Neo4j in combination with the graph query language Cypher. The query language is used to define verification rules, which are applied to the model. If all rules are fulfilled, the verification process is rated as successful. The architecture for a verification service as well as user defined functions in the context of model verification are presented.

2.1. Verification framework

The core concept of the presented graph database verification approach is to persist the specific models in a first step and to define verification rules using a query language which are evaluated on the models in a second step.

To be able to create a verification framework providing this functionality, it is necessary to encapsulate the Cypher queries, that should be executed. The presented architecture distinguishes between two types of queries. On the one hand queries which return a single value result, e.g. for the result of counting specific nodes or boolean evaluations, and on the other hand queries which return a list of results. For this reason, the architecture differentiates between `TypedQuery` and `MultiResultTypedQuery` – both used as wrapper for the actual Cypher query. The `MultiResultTypedQuery` class is derived from `TypedQuery`, since the only difference is the result type as list of entries. This architecture is shown in Figure 1.

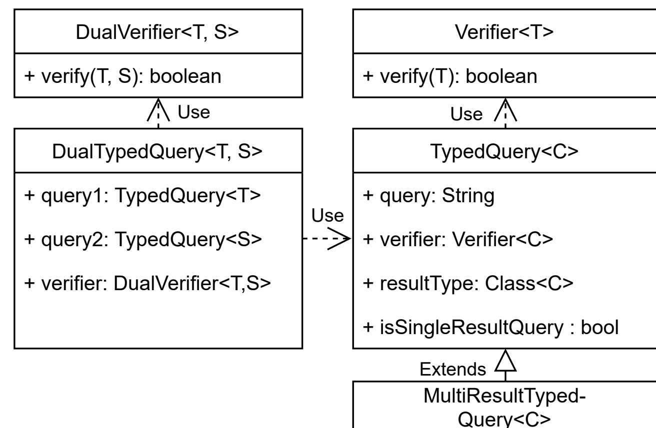


Figure 1. Architecture for a query based verification framework. `TypedQuery` represents one verification query, which wraps the native Cypher query, its expected result type and an associated `Verifier`. The verifier is used to evaluate the query's result and for this represents the actual verification process. The `MultiResultTypedQuery` is an extension of the `TypedQuery` and represents a query with a collection of result values. The `DualTypedQuery` wraps two `TypedQuery` objects and compares their result for the verification.

Neo4j provides the possibility of returning results with dynamic data types. This is left out for the verification, since it requires an additional parsing process with statically typed languages such as Java. In addition to the actual Cypher query, both query wrappers (`TypedQuery`, `MultiResultTypedQuery`) contain a verifier, which evaluates the query result. A verifier takes the result and returns a boolean value, which signals if the result is correct. An appropriate verifier must be used depending on what should be verified with the specific query. In the simplest form, the query itself already returns a boolean value, representing if the verification holds or not. In such cases, the verifier only forwards the query result. Additionally, a verifier can be implemented for equality checks of the result with a value comparison.

Based on the concept of single result and multi result queries, another concept called `DualTypedQuery` is introduced. While `TypedQuery` and `MultiResultTypedQuery` can have any result which is evaluated independently, the idea of the `DualTypedQuery` wrapper is to execute two queries and to compare their results. For this reason, a `DualTypedQuery` consists of two `TypedQuery` instances and a `DualVerifier` which is responsible for the result comparison (shown in Figure 1). Such a `DualVerifier` can be used if the verification should be done by comparing the source and the target model, instead of only analyzing the target.

2.2. User-defined APOC procedures

One crucial functionality for verifications using graphs is the possibility of comparing the equality of two nodes. Neo4j only provides the possibility for id-based equality checking or alternatively manually comparing a node via property based match clauses in a Cypher query. To enhance the equality mechanism of Neo4j, additional user-defined procedures can be created using Neo4j's APOC framework as described by Neo4j, Inc. (2021). For this reason, this work introduces a user defined procedure called `equals()`. This method expects two nodes A and B, as well as two lists L_A and L_B, each containing n property names for the associated node. These property lists are used for the actual equality checking. With this purpose, the position of each element in the first list must be equal to the position of the corresponding property in the second list. This is defined by a function $f : A(L_A(i)) \equiv B(L_B(i)) \mid i = [0, n)$. The implementation of this equality check is shown in Algorithm 1 in the form of a user defined procedure for the Neo4j graph database. Since APOC is a framework based on the turing-complete programming language Java, it can be used to extend the verification mechanism in any way.

Algorithm 1: User defined function for comparing two nodes based on given property lists.

Data: n1, n2: Nodes to be compared;
p1, p2: Property names of associated nodes;
Result: Boolean value that represents given nodes are equal based on their properties.

```
1  @UserFunction("nodeEquals")
2  public boolean equals(
3      @Name("n1") Node n1, @Name("n2") Node n2,
4      @Name("p1") List<String> p1, @Name("p2")
   List<String> p2) {
5          if(p1.size() != p2.size()) return false;
6          return IntStream.range(0, p1.size())
7                  .mapToObj(i -> Pair.of(p1.get(i),
   p2.get(i)))
8                  .allMatch(p -> Objects.equals(
9                          n1.getProperty(p.first(), null),
10                         n2.getProperty(p.second(), null)
11                 )
12         );
13 }
```

3. Evaluation

The evaluation is done in two ways. On the one hand, a functional evaluation and on the other hand, a performance comparison is carried out. The functional evaluation is done using a running example, while the performance measurement is done in comparison to OCL using the public IMDb dataset. Both analyses are based on the Neo4j Community Edition 4.3.6 with the APOC extension 4.3.0.4. In addition to that, Eclipse's OCL implementation in the version 3.10 is used. All tests are executed using an AMD Ryzen 9 3900 X 12-Core processor with 64 GB RAM.

3.1. Functional Evaluation

The functional evaluation is based on mutation testing as described by DeMillo et al. (1978), which can be applied in model transformation as stated in Mottu et al. (2006) and Guerra et al. (2019). In this example, n three-dimensional polygons are randomly created, where each polygon consists of m three-dimensional points. Every point is defined by the three coordinates x, y and z. Those polygons are transformed to two-dimensional counterparts, by removing the z-coordinate. The transformation is randomly executed incorrectly, leading to different types of transformation errors. As shown in Figure 2 one or multiple of the following errors can occur during the transformation:

(a) Removing the transformed `Polygon2D` from the result
(b) Deleting the resulting `Point2D`
(c) Mixing the values of the points' coordinates, so $x' = y$ and $y' = x$ applies for a transformation $t(P(x, y, z)) = P'(x', y')$
(d) Keeping the z-coordinate, so a `Point2D` still consists of 3D information
(e) Adding additional, artificial properties to a `Point2D`

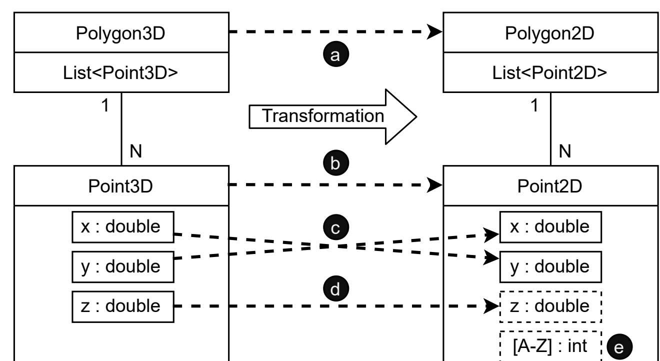


Figure 2. The running example with three- and two-dimensional polygons and associated points, which are randomly transformed incorrectly. The errors are labelled from a to e.

Algorithm 2: Query to compare the number of Polygon3D and Polygon2D nodes.

Result: Boolean that signals if the number of Polygon3D and Polygon2D nodes is equal.

```
1 MATCH(n:Polygon3D), (x:Polygon2D)
2 WHERE (n)--(x)
3 RETURN count(n) = count(x);
```

Algorithm 3: Query to compare the number of created Point2D nodes with the number of the Point3D in the original polygon.

Data: $polyId: Id of the Polygon3D to check
Result: Boolean that signals if the number of points is equal in the polygons.

```
1 MATCH(n:Polygon3D)-->(p3:Point3D)
2 WHERE id(n)=$polyId
3 OPTIONAL MATCH
    (n)--(:Polygon2D)--(p2:Point2D)
4 RETURN COUNT(DISTINCT(p2)) =
    COUNT(DISTINCT(p3));
```

Algorithm 4: Query to compare all points of the created Polygon2D with the source points in its 3D-counterpart.

Data: $pointId: Id of the Point3D
Result: Boolean that signals if the number of points is equal in the polygons.

```
1 MATCH (poly3d:Polygon3D),
2    (poly2d:Polygon2D),
3    (p3:Point3D), (p2:Point2D)
4 WHERE (p3)--(poly3d)--(poly2d)--(p2)
5    AND id(p3) = $pointId
6    AND nodeEquals(p2, p3, [“x”, “y”])
7    AND size(keys(p2)) = size(keys(p3)) - 1
8 RETURN p2 IS NOT NULL;
```

To validate the functionality of the proposed framework, the randomly created errors should be detected with suitable verification rules. For this task, the mentioned transformation process logs which transformed elements contain an error, and the type of error that was introduced. This information is not used by the verification and is also not stored in the database. It is only used as ground truth to evaluate the correctness of the verification process.

Based on the presented running example, the following Cypher queries are used as verification rules to detect the mentioned errors: The first statement shown in Algorithm 2 checks if a `Polygon2D` was created for every three-dimensional entity. The second statement shown in Algorithm 3 is executed once for every `Polygon3D`. It selects the polygon with the given id and tries to match the corresponding `Polygon2D`. Then it compares the number of points of both associated polygons. The third and most sophisticated statement shown in Algorithm 4 iterates over every `Point3D` for a given `Polygon3D` from the source model and checks if there is a matching `Point2D` in the corresponding `Polygon2D` regarding the point's *x* and *y* properties. This query also checks if a `Point2D` has exactly one property less than its corresponding `Point3D` (two instead of three), due to the missing z-coordinate.

The presented verification framework executes all those queries once per model (first query), per polygon (second query) and respectively per point (third query) in a polygon. If all queries result in a positive result (*true*) the model is valid and the transformation is correct, otherwise a faulty transformation is detected.

3.2. Performance Evaluation

In terms of evaluating the performance of the proposed approach, a case study is carried out. This study is based on parts of IMDb's dataset namely the person data (name.basics), the movie data (title.basics) and the associations between movies and persons (title.principals). The structure of the used data is shown in Figure 3. This data set is imported into an empty Neo4j graph database using an open-source Java-based importer published by Pointner and Praschl (2020). For the OCL based comparison, the same data is also imported into an Ecore model using another open-source implementation published by Praschl and Pointner (2021). After the import process both models consist of 10,647,967 entries, which are composed of 4,152,840 persons and 6,495,127 movies, and 41,108,868 relationships between the two entities for the complete data set. In addition to this complete version, six smaller subsets are created, that enable to evaluate the performance of the approach. This is done by selecting and persisting a limited number of *n* relationships with $n \in \{2500; 40000; 160000; 625000; 2500000; 10000000\}$ and the associated nodes. The evaluation is executed using 50 warm up runs, as well as 50 execution runs with a time limit of one hour per run based on the Neo4j and the OCL approach.

Based on these subsets, multiple queries, respectively constraints, are executed to ensure the model's integrity with different complex requirements for the model. These queries can be separated into two groups of verification rules:

1. Queries that only require knowledge of individual entities.
2. Queries that also require knowledge of associated entities based on existing relationships.

Regarding the first group, an initial query is used to check if every person has an attribute called `name` (cf. Algorithm 5 and Algorithm 6). The second query of this group is used to ensure the uniqueness of an actor's `id` property(cf. Algorithm 11 and Algorithm 12). As mentioned,

Algorithm 5: Neo4j query 1 used as verification rule for checking if every person has an attribute called name.

Result: True if all person nodes have the attribute name, else False.

```
1 MATCH (n:Person) WHERE n.name IS NULL
  RETURN COUNT(n) = 0;
```

Algorithm 6: OCL query 1 used as verification rule for checking if every person has an attribute called name.

Result: True if all person nodes have the attribute name, else False.

```
1 context Root inv: self.persons→forAll(a | a.name <>
  null)
```

these queries differ from the remaining two, as they only require knowledge of the individual entities, without including any relationships to other entities. The second group in turn consists of two queries. One query that is used to check if every person is related to at least one movie (cf. Algorithm 7 and Algorithm 8) and another query that is used to verify, that every actor (person with "actor" in the `primaryProfession` list property) is also associated as actor (relationship "part_of" with the value "actor" in the `category` property) with at least one movie (cf. Algorithm 9 and Algorithm 10). Every verification rule is executed using the Neo4j based approach as well as OCL and the specific results are juxtaposed.

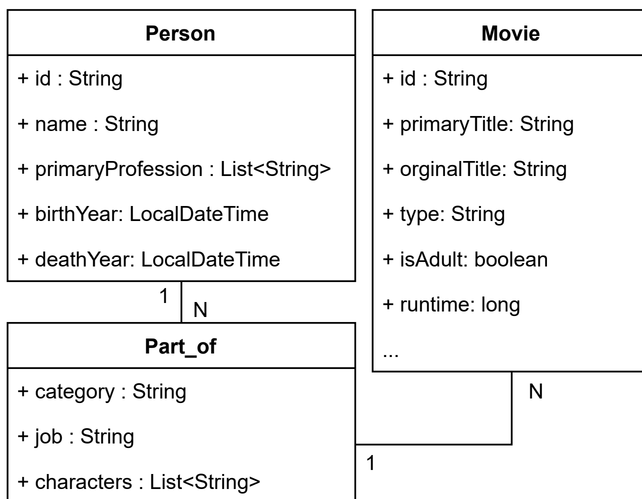


Figure 3. Class diagram of the used IMDb dataset showing the information for a Person, a Movie and the association "part_of" between those entities.

Algorithm 7: Neo4j query 2 used as verification rule for checking if every person is related with at least one movie.

Result: True if all person nodes that are of a relationship, else False.

```
1 MATCH (p:Person) WHERE NOT
  (p)-[:part_of]-(:Movie)
2 RETURN COUNT(p) = 0;
```

Algorithm 8: OCL query 2 used as verification rule for checking if every person is related with at least one movie.

Result: True if all person nodes that are of a relationship, else False.

```
1 context Root inv: self.persons→
2    forAll(a | self.partOf→any(aIm | a = aIm.id) <>
  null)
```

Algorithm 9: Neo4j query 3 used as verification rule for checking if every actor is related with at least one movie as actor.

Result: True if all person nodes with the primaryProfession actor are also part of a relationship with the category actor, else False.

```
1 MATCH (p: Person) WHERE "actor" IN
  p.primaryProfession
2 AND NOT (p)-[:part_of {category:
  "actor"}]-(:Movie)
3 RETURN COUNT(p) = 0;
```

Algorithm 10: OCL query 3 used as verification rule for checking if every actor is related with at least one movie as actor.

Result: True if all person nodes with the primaryProfession actor are also part of a relationship with the category actor, else False.

```
1 context Root inv: self.persons→
2    select(a |
  a.primaryProfession→includes('actor'))→
3    forAll(a | self.partOf→
4       any(aIm | a = aIm.id and aIm.category =
  'actor') <> null
5    )
```

Algorithm 11: Neo4j query 4 used as verification rule for checking if every actor's id is unique.

Result: True if all values of the property id of the person nodes are unique, else False.

```
1 MATCH(x:Person) RETURN
   COUNT(DISTINCT(x.id)) = COUNT(x);
```

Algorithm 12: OCL query 4 used as verification rule for checking if every actor's id is unique.

Result: True if all values of the property id of the person nodes are unique, else False.

```
1 context Root inv: self.persons→isUnique(a | a.id)
```

4. Results

This section lists the results of the functional as well as the performance comparison. The functional comparison is carried using the presented verification approach on a running example with a transformation from three-dimensional to two-dimensional polygons on the one hand. On the other hand, the performance evaluation is done based on verifications for the IMDb data set, which are compared to the state-of-the-art framework OCL.

4.1. Functional Results

Multiple experiments are tested for the polygon-based running example using a different number of nodes that are transformed during the process and evaluated using the presented Cypher queries. Table 1 shows the number of correctly and incorrectly created polygons/points (ground truth) and the detected invalid nodes in the specific run. The table shows that all incorrectly transformed nodes are detected using the presented approach. The running example shows, that the presented verification approach is capable of detecting errors due to a model transformation. Based on the results of this experiment the RQ. 1 "How can model transformations be verified in graph databases?" is answered. The experiment shows successfully the utilization of the presented approach in the context of a model verification process using Neo4j, Cypher and APOC for the given example.

4.2. Performance Results

In context of the RQ.2 "How does a graph database based verification perform and scale?", the results of the performance evaluation are listed in Table 2 and show that the present approach outperforms the Eclipse's OCL implementation in the most cases, based on the mean run time and the standard deviation per query and subset. The mentioned time limit is clearly exceeded by the second OCL query, starting with the data set containing 625,000 relationships. For this reason, this run is incomplete, with

Table 1. Verification results of multiple experiments for the presented running example, showing that all incorrectly transformed polygons and points are detected by the presented framework

	Number of transformed polygons		Number of transformed points		Number of detected faults	
#	Correct	Incorrect	Correct	Incorrect	Faulty Polygons	Faulty Points
1	2	0	58	2	0	2
2	4	1	242	8	1	8
3	7	3	346	154	3	154
4	3	0	750	0	0	0
5	16	4	907	93	4	93
6	3	2	119	51	2	51
7	36	4	1895	105	4	105
8	39	11	2289	211	11	211

only two execution runs at all. Furthermore, the remaining subsets are not executed. Next to Table 2, the Figures 4 to 7 compare the two approaches per query within a logarithmic scale. The first and fourth query in combination with the first two subsets with 2,500 and 40,000 relationships are the only examples in the evaluation, where OCL outperforms Neo4j. In all other situations, the Neo4j based approach is faster. Especially, queries not only considering individual entities but also relationships show the performance advantages of the graph based approach. This is also highlighted in Figure 5 with a linear runtime complexity $O(n)$ using Neo4j and a quadratic complexity $O(n^2)$ using OCL for the second query.

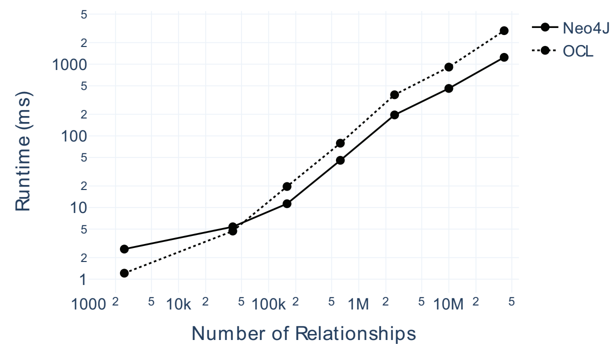


Figure 4. Run time comparison between Neo4j and OCL for the first query, that checks if every `person` in the model has an attribute called `name`.

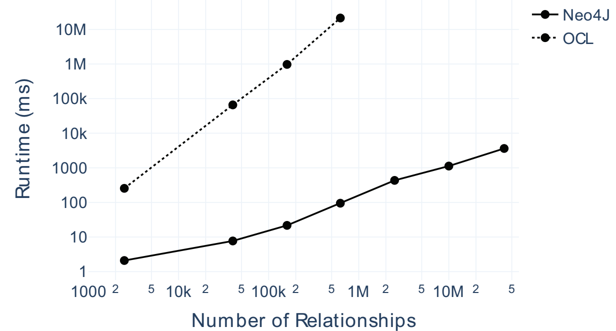


Figure 5. Run time comparison between Neo4j and OCL for the second query, that checks if every `person` is part of at least one relationship. Since the execution of the OCL version exceeds the time limit of one hour, it is not applied onto all data sets.

Table 2. Run time comparison (in ms) of the queries using Neo4j and OCL for various number of relationship in the used IMDb data subset. The speedup section shows the scaling factor between the mean values for every query with OCL compared to Neo4j ($speedup = OCL_{Mean}/Neo4j_{Mean}$). Due to the runtime of 5.93 hours in average for the second OCL query with 625,000 relationships (*), the setup is canceled after two runs and because of this situation it is not executed for the remaining subsets.

		2,500		40,000		160,000		625,000		2,500,000		10,000,000		41,108,868	
		Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev	Mean	StDev
Query1	Neo4j	2.63	0.45	5.48	0.51	11.49	1.11	45.73	1.12	196.95	1.92	459.40	1.44	1,247.89	1.96
	OCL	1.26	0.26	5.01	0.57	19.66	0.10	79.07	0.35	380.56	29.52	940.11	72.55	2,894.14	134.34
	Speedup	0.48		0.91		1.71		1.73		1.93		2.05		2.32	
Query2	Neo4j	2.15	0.25	7.89	1.13	22.09	1.84	95.18	1.32	433.80	3.66	1,124.59	4.18	3,626.21	11.74
	OCL	261.30	9.56	65,751.80	446.24	977,367.68	17,115.71	21,373,379.76*	5,770.55*	-	-	-	-	-	-
	Speedup	121.53		8,333.56		44,244.8		224,557.47*		-		-		-	
Query3	Neo4j	2.26	0.15	13.31	1.12	39.71	1.70	176.84	1.40	760.55	5.78	2,099.20	4.77	6,460.75	11.68
	OCL	25.92	0.77	98.26	5.84	2,072.02	50.70	1,487.39	84.96	7,892.92	121.95	27,224.91	200.91	102,376.40	525.31
	Speedup	11.47		7.38		52.18		8.41		10.38		12.97		15.85	
Query4	Neo4j	2.82	0.42	6.61	0.42	16.81	1.17	63.36	2.50	303.06	47.69	734.96	72.05	2,331.53	79.70
	OCL	1.05	0.19	5.16	0.30	18.79	1.33	73.44	16.56	357.78	51.85	811.50	69.00	2,722.12	198.22
	Speedup	0.37		0.75		1.12		1.16		1.18		1.10		1.17	

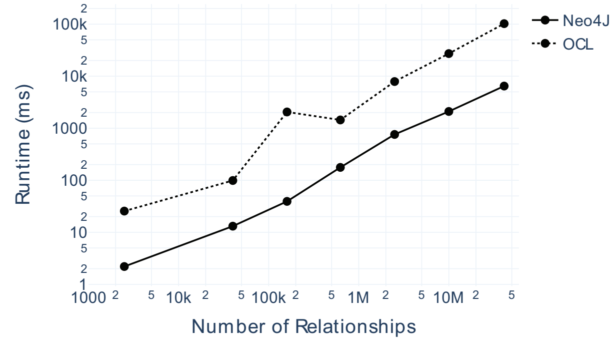


Figure 6. Run time comparison between Neo4j and OCL for the third query, that checks if every `person` that has the value `actor` within the `primaryProfession` list, has also a relationship to at least one movie with the relationship attribute `category` containing this value.

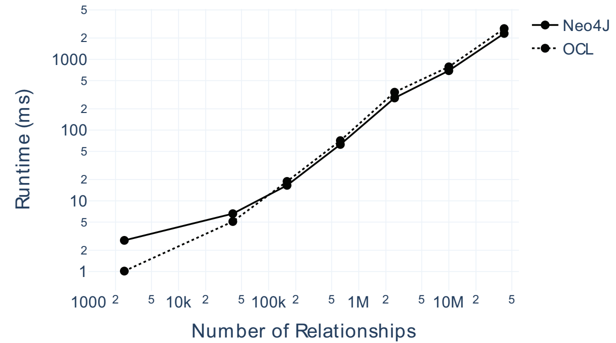


Figure 7. Run time comparison between Neo4j and OCL for the fourth query, that checks if every `person` has a unique id.

5. Related Work

An approach for model verification is presented by Guerra et al. (2012). This verification is performed by specifying visual contracts and compiling them into QVT, which is introduced by OMG (2007), to detect disconformities of transformation results. In contrast to their work, the presented approach is working directly on the database level, using the corresponding query language to create the verification.

Ko et al. (2013) present a verification concept using property matching based transformation and graph comparison algorithms. To achieve this, the authors rely on meta-information and compare associated properties in the source as well as target model using similarity measurements. Compared to the presented work, this verification approach is based on property matching. In contrast, the approach presented in this work does not use similarity measurements, but compares the model properties.

Selim et al. (2013) describe the concept of verifying graph-based model transformations based on properties. The introduced property provers are used in the context of the DSLTrans (c.f. Barroca et al. (2011)) transformation language to check the source and the target model based on constraints. Hence, a constraint results in *true* or *false*, depending on whether it is fulfilled by the respective model or not. The idea of these constraints are comparable

to the verification rules described in this work. The major difference is the field of application, since the approach presented in this work uses a graph database instead of DSLTrans and for this reason benefits from query optimizations of a database.

OCL from OMG (2006) is widely used for the verification of models, for example defined using the Meta-Object Facility (MOF) language as defined by OMG (2016). In this context, Cariou et al. (2010) characterize a method using OCL transformation contracts to verify the result's correctness of a transformation process. Burgueno et al. Burgueno et al. (2013) present a tool for the specification and verification of models. The second of the two mentioned publications is in turn based on OCL constraints. In addition to that also Gogolla and Vallecillo (2011) describe a method for transformation contracts using the constraint language OCL. In contrast to the methods using OCL, the presented approach in this work does not require an explicitly defined model for the verification, because it only relies implicitly on the database's graph meta model.

6. Conclusion

This work contributes, a novel approach that enables verification on a graph-based structure. The approach uses the graph database Neo4j and allows creating verification rules using the Cypher query language. Based on a running example, where 3D polygons are transformed into 2D polygons, the validity of the verification approach is examined and allows answering RQ_1: "*How can model transformations be verified in graph databases?*". Furthermore, to also address RQ_2: "*How does a graph database based verification perform and scale?*", the performance of the presented approach is evaluated based on the IMDb dataset in comparison to OCL. Both evaluations show that the approach is capable of verifying models using Neo4j and Cypher. The performance evaluation shows the advantages of a graph based verification approach, especially for highly-connected data sets with a high amount of entries, compared to the classic approach using OCL. Additionally, the functional as well as the performance evaluation provide examples on how to define verification rules within the presented approach.

In the future, it is planned to evaluate different rule patterns to show best practices for using the presented approach. Furthermore, a query API could be developed, allowing to express the verification rules in a fluent and type-safe way like the JPA Criteria API – specified in JSR 338 chapter 6 by DeMichiel and Jungmann (2017). This would lead to an enhanced and even more powerful model verification framework.

References

Atkinson, C. and Kuhne, T. (2003). Model-driven development: a metamodeling foundation. *IEEE Software*, 20(5):36–41.

Barroca, B., Lúcio, L., Amaral, V., Félix, R., and Sousa, V. (2011). Dsltrans: A turing incomplete transformation language. In Malloy, B., Staab, S., and van den Brand, M., editors, *Software Language Engineering*, pages 296–305, Berlin, Heidelberg. Springer Berlin Heidelberg.

Burgueno, L., Wimmer, M., Troya Castilla, J., and Vallecillo Moreno, A. (2013). Tractstool: Testing model transformations based on contracts. In *MODELS-JP 2013: Invited Talks, Demonstration Session, Poster Session, and ACM Student Research Competition co-located with the 16th Int. Conf. on Model Driven Engineering Languages and Systems, p 76-80*. CEUR-WS.

Cariou, E., Belloir, N., Barbier, F., and Djemam, N. (2010). Ocl contracts for the verification of model transformations. *Electronic Communications of the EASST*, 24.

DeMichiel, L. and Jungmann, L. (2017). Sr 338: Java™persistence api, version 2.2. https://jcp.org/en/jsr/detail?id=338; accessed 10. May 2022.

DeMillo, R. A., Lipton, R. J., and Sayward, F. G. (1978). Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4):34–41.

Gogolla, M. and Vallecillo, A. (2011). Tractable model transformation testing.

Guerra, E., de Lara, J., Wimmer, M., Kappel, G., Kusel, A., Retschitzegger, W., Schönböck, J., and Schwinger, W. (2012). Automated verification of model transformations based on visual contracts. *Automated Software Engineering*, 20(1):5–46.

Guerra, E., Sánchez Cuadrado, J., and de Lara, J. (2019). Towards effective mutation testing for atl. In *2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems (MODELS)*, pages 78–88.

IMDb.com, Inc (2022). Imdb datasets. https://www.imdb.com/interfaces/. (Accessed on 12/07/2022).

Ireland, C., Bowers, D., Newton, M., and Waugh, K. (2009). A classification of object-relational impedance mismatch. In *2009 First International Confernce on Advances in Databases, Knowledge, and Data Applications*, pages 36–43. IEEE.

Ko, J.-W., Chung, K.-Y., and Han, J.-S. (2013). Model transformation verification using similarity and graph comparison algorithm. *Multimedia Tools and Applications*, 74(20):8907–8920.

Lano, K., Kolahdouz-Rahimi, S., and Clark, T. (2012a). Comparing verification techniques for model transformations. In *Proceedings of the Workshop on Model-Driven Engineering, Verification and Validation*, page 23–28. Association for Computing Machinery.

Lano, K., Kolahdouz-Rahimi, S., and Clark, T. (2012b). Verification of model transformations. *Dept. of Informatics, King's College London*.

Mottu, J.-M., Baudry, B., and Le Traon, Y. (2006). Mutation analysis testing for model transformations. In *European Conference on Model Driven Architecture (ECMDA 06)*, pages 376–390.

Neo4j, Inc. (2021). Neo4j apoc library - developer guides.

https://neo4j.com/developer/neo4j-apoc/. (Accessed on 07/12/2022).

OMG (2006). Object constraint language (ocl) specification, version 2.0. https://www.omg.org/spec/OCL/2.0/; accessed 10. May 2022.

OMG (2007). Meta object facility (mof) 2.0 query / view / transformation specification. https://www.omg.org/cgi-bin/doc?ptc/2007-07-07; accessed 10. May 2022.

OMG (2016). Mof meta object facility. https://www.omg.org/spec/MOF/; accessed 10. May 2022.

Pointner, A. and Praschl, C. (2020). Fhooeaist/neo4j-imdb: v1.0. https://zenodo.org/record/4030726.

Praschl, C. and Pointner, A. (2021). Fhooeaist/imdb-ocl-verification: v1.0.0. https://zenodo.org/record/5705169.

Selim, G., Lúcio, L., Cordy, J. R., and Dingel, J. (2013). Symbolic model transformation property prover for dsltrans. Technical report, Technical Report 2013-616, Queen's University.

Sunitha, E. and Samuel, P. (2019). Automatic code generation from uml state chart diagrams. *IEEE Access*, 7.