Managing 3D Simulation Models with the Graph Database Neo4j

Martin Hoppen, Juergen Rossmann, Sebastian Hiester

Institute for Man-Machine Interaction
RWTH Aachen University
Aachen, Germany
Email: {hoppen,rossmann}@mmi.rwth-aachen.de, sebastian.hiester@rwth-aachen.de

***Abstract*—Every simulation is based on an appropriate model. Particularly in 3D simulation, models are often large and complex recommending the usage of database technology for an efficient data management. However, the predominant and well-known relational databases are less suitable for the hierarchical structure of 3D models. In contrast, graph databases from the NoSQL field store their contents in the nodes and edges of a mathematical graph. The open source Neo4j is such a graph database. In this paper, we introduce an approach to use Neo4j as persistent storage for 3D simulation models. For that purpose, a runtime in-memory simulation database is synchronized with the graph database back end.**

Keywords–3D Simulation; 3D Models; Simulation Database; Graph Database; Database Synchronization

I. INTRODUCTION

Before a technical system can get into mass production and is ready for execution it has to pass certain stages of development. Besides the new component's design, an intense test phase is compulsory in order to confirm its operability, increase its prospects and if necessary to initiate some steps of optimization. For such tests, engineers utilize simulation tools like block-oriented simulation or 3D simulation to analyze their target system in a virtual environment. In particular, 3D simulation systems allow to analyze the spatial properties and behavior of the intended system and its interaction with its surroundings in an expressive visual way.

Every simulation is based on an appropriate model describing properties and behavior of the system under development. This model data needs to be managed conveniently. The usage of database technology has established for such requirements. In contrast to flat files, they offer high performance data evaluation and simplify data management with regard to security, reliability, recovery, replication and concurrency control. Currently, relational databases are dominating the market. Due to different problems with scalability and effective processing of big data with relational databases the field of NoSQL ("Not only SQL") databases has emerged [1]. In this context, the approach of graph databases (GDBs) has become popular. GDBs save their data in the nodes and edges of a mathematical graph, in particular, to manage highly linked information. As such, they are ideally suited for 3D simulation models. Like in Computer-aided Design (CAD), such 3D data usually comprises a huge number of parts of many different types (mostly, each with only few instances), structured hierarchically with interdependencies. This recommends a graph-like data structure. For the same reason, the scene graph is a common approach to manage 3D data at runtime.

In this paper, we present a concept for a synchronization interface between a GDB and a 3D simulation database, i.e., the runtime database of a 3D simulation system. The applied data mapping strategy is bidirectional and in part incremental. The approach was developed in the context of a student project and is based on our previous work [2]. Its feasibility is shown with a prototypical implementation using the GDB Neo4j and the 3D simulation system VEROSIM and its Versatile Simulation Database (VSD) (Figure 1).

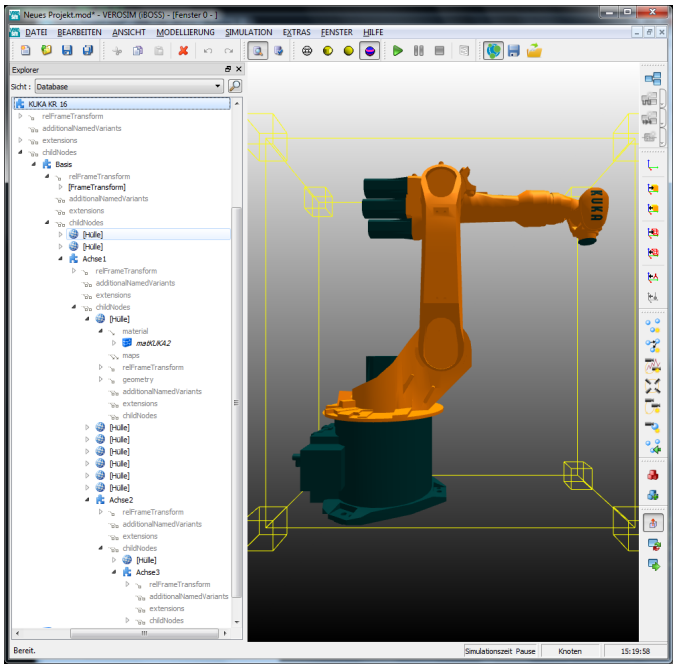

Figure 1. Robot model (from Figure 9) loaded from Neo4j into the in-memory simulation database VSD.

In Section II, we start with the basics of GDBs and a short introduction to Neo4j and VEROSIM including VSD. Section III gives an overview of different GDBs and motivates the decision for Neo4j. In Section IV, the prototype's general requirements are itemized and Section V describes the specific implementation of the interface with Neo4j and the VSD. Subsequently, Section VI presents an evaluation of the

 ISBN: 978-1-61208-486-2

interface and Section VII recaps our work with a concluding statement.

II. State of the Art

In this section, the necessary basics for our work are presented.

A. Graph Databases

The idea of a GDB relies on the mathematical graph theory. Information is saved in the nodes (or vertices) and edges (or relationships) of a graph as shown in Figure 9. A graph is a tuple $G = (V, E)$, where V describes the set of nodes and E the set of edges, i.e., $v_i \in V$ and $e_{i,j} = (v_i, v_j) \in E$ [3]. To specify records, properties of nodes and (depending on the GDB) even relationships can be described by key-value pairs [4]. An important aspect of GDBs is the fact that all relationships are directly stored with the nodes so that there is no need to infer them as in relational databases using foreign keys and joins. Hence, read operations on highly connected data can be performed very fast. During a read access, the graph is traversed along paths so that the individual data records (nodes and edges) can be read in situ and do not have to be searched globally. Therefore, the execution time depends only on the traversal's depth [1].

GDBs also provide standard database features like security, recovery from hard- or software failures, concurrency control for parallel access, or methods for data integrity and reliability.

In contrast to flat files, using a (graph) database, data can be modified with a query language. Such languages are a powerful tool to manipulate the database content so that the data is not only stored persistently and securely but can also be handled simply.

B. Neo4j

Neo4j is a GDB implemented in Java. It can be run in server or embedded mode. Figure 2 shows its data model. Central elements are nodes and relationships containing the stored records. These records are described by an arbitrary number of properties (key-value pairs). Neo4j offers the concept of *labels* and *types* to divide the graph in logical substructures. A node is extendible with several labels characterizing the node's classification. Similarly, a relationship is identified by a type (exactly one). Besides the classification of the data, this also improves reading performance as just a part of the graph must be traversed to find the desired record [4][5]. Apart from that, Neo4j is schemaless, i.e., it does not require any metadata definition before inserting actual user data.

All Neo4j accesses are processed in ACID (Atomicity, Consistency, Isolation, Durability) compliant transactions guaranteeing the reliability, consistency and durability of the database content [1]. Accesses are either performed with Neo4j's own query language called Cypher or using its Java API.

C. VEROSIM and VSD

VEROSIM is a 3D simulation system rooted in the field of robotics [6]. In recent years, it evolved to a versatile framework for simulation in various fields of application (in particular: environment, space, industry). During runtime, simulation models are managed by its VSD. This in-memory

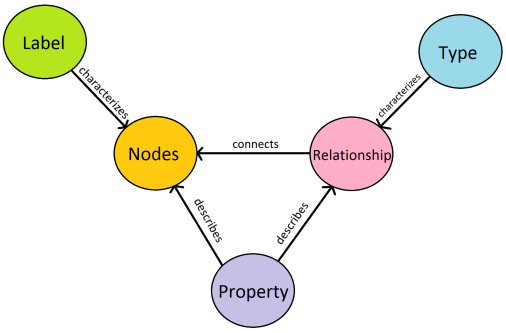

Figure 2. Neo4j data model.

database not only stores the passive structure and properties of a 3D simulation model but also its active behavior, i.e., the simulation algorithms themselves. VSD is an object-orientated GDB.

Objects are called instances and are characterized by properties. Such properties can either be value properties (Val-Properties) with basic or complex data types or reference properties. The latter model $1:1$ (Ref-Properties) or $1:n$ (RefList-Properties) directed relationships between instances. Furthermore, these relationships can be marked to *contain* target instances using an *autodelete* flag allowing to model UML composite aggregations.

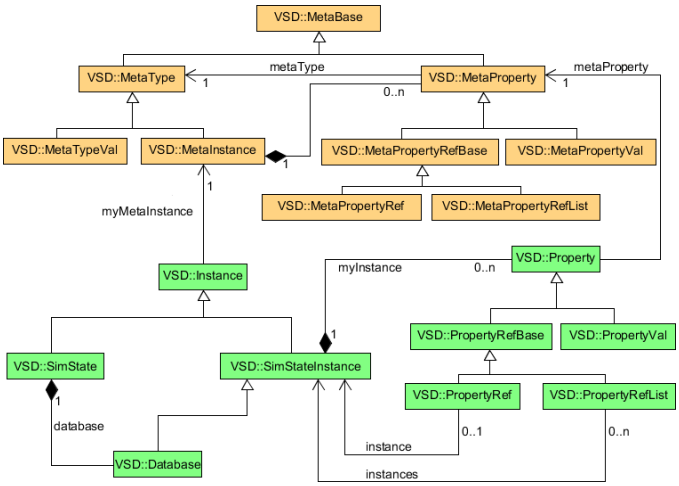

Figure 3. VSD data model (top: metadata, bottom: instance data).

VSD comprises a meta information system providing access to its schema and also to specify its schema. So-called meta instances describe an instance's class (name, inheritance, etc.) and so-called meta properties its properties (name, type, etc.). Figure 3 shows VSD's data model.

III. Related Work

Besides Neo4j, there are many other GDBs in the market. They differ in their conceptual structure and application area.

DEX is a GDB based on the labeled and directed attributed multigraph model. All nodes and edges are classified (labeled), edges are directed, nodes can be extended with properties (attributes), and edges can be connected with more than two

 ISBN: 978-1-61208-486-2

nodes (multigraph) [7]. The graph is represented by bitmaps and other secondary structures. DEX has been designed for high performance and scalable graph scenarios. The good performance is achieved by the bitmap-based structure and the indexing of all attributes, which are efficiently processed by the C++ kernel [8].

Trinity [9][10] is a memory-based graph store with many database features like concurrency or ACID-conform transactions. The graph storage is distributed among multiple well connected machines in a globally addressable memory address space yielding big data support. A unified declarative language provides data manipulation and message passing between the different machines. The great advantage of Trinity is the fast access to large data records. It is based on a multigraph model, which can exceed one billion nodes. Since there is no strict database schema, Trinity can flexibly be adapted to many data sets.

HypergraphDB stores its data in a directed multigraph, whose implementation is based on BerkeleyDB. All graph elements are called atoms. Every atom is characterized by its atom arity indicating the number of linked atoms. The arity determines an atom's type: An arity larger than zero yields an edge atom, or else, a node atom. Each atom has a typed value containing the user data [11].

InfoGrid is a framework specialized in the development of REpresentational State Transfer (REST)-full web applications. One part of this framework is a proprietary GDB used for data management. The graph's nodes are called MeshObjects, which are classified by one or more so-called EntityTypes, properties, and their linked relationships. MeshObjects not only contain the user data but also manage events relevant to the node [12].

Infinite Graph is a GDB based on an object-oriented concept. All nodes and edges are derived from two basic Java classes. Thus, the database schema is represented by user-defined classes. Besides data management, Infinite Graph provides a visualization tool [13]. Since the database can be distributed on multiple machines working in parallel, Infinite Graph can achieve a high data throughput. To manage concurrency, a lock server handles the different lock requests [8].

AllegroGraph [14] provides a REST protocol architecture. With this interface, the user has full control of the database including indexing, query and session management. All transactions satisfy ACID conditions.

Despite this wide range of GDBs, for the following reasons, we decide to use Neo4j in our approach:

- In many tests it proves to process data fast and efficiently,
- it can handle more than one billion nodes – even enough for extremely large 3D simulation models – which could be useful in coming stages of extension,
- Neo4j is a full native GDB so that traversal and other graph operations can be performed efficiently,
- Neo4j provides a comprehensive and powerful query language (e.g., for efficient partial loading strategies in future versions of the presented prototype),
- directed edges allow to model object interdependencies more accurately, however, without disadvantages in traversal performance,
- properties on relationships allow for a more flexible modeling (e.g., to distinguish between shared and composite aggregation relationships),
- finally, Neo4j is currently the most prevalent GDB in the market indicating it to be especially well explored and developed. Hence, it provides the best prospects of success.

Note that while we choose Neo4j for the reasons given above, the presented concepts are mostly independent of the choice of the particular GDB.

IV. Concept

In this section, we describe the fundamental concept and the required features of our synchronization component's prototype. Its implementation using Neo4j and VSD is described in Section V.

A. Structure Mapping

An essential question when synchronizing two databases is: How do we map the different data structures? Depending on the database paradigm, entities with attributes and relationships (connecting two or more entities) are represented differently. For example, a relational database uses relations, attributes and foreign keys while a GDB uses nodes, relationships and properties.

1) **Schema Mapping:** Before synchronizing user data, a generic schema mapping is performed mapping the metadata of one database to the other as described in [15]. This is performed once on system startup. For example, when performed between a relational and an object-oriented database, each table of the former might be mapped to a corresponding class of the latter (columns and class attributes accordingly).
2) **Schemaless Approach:** When a schemaless database is involved, a different approach has to be applied. Here, metadata from a non-schemaless database must be mapped onto the user data of the schemaless one. For example, class names from an object-oriented database are mapped onto node labels of a schemaless GDB.

For the schemaless Neo4j, in our prototype, we chose the second approach.

B. Object Mapper

Another key aspect of the concept is the object mapper. It maps objects from one database to an equivalent counterpart in the other database. For example, an object from an object-oriented database is mapped to a corresponding node in the GDB. The mapping is based on the counterparts' identities and includes a transfer of all property (or attribute) values in between. Based on these mappings, individual object or property changes can be tracked and resynchronized. Summarized the mapping is bidirectional and in part incremental.

C. Transactions

Any changes (insert, update, delete) to the data are tracked and stored in transactions, which can be processed independently. By executing these transactions, data is (re)synchronized on object level. During the accumulation (and before the execution) of such transactions, the operations

stored within can be filtered for redundancies. For example, a transaction for creating a new object followed by a transaction for deleting the very same object can both be discarded.

V. PROTOTYPE

This section gives an insight into the prototypical implementation of the interface between VSD and Neo4j. The prototype should have the ability to save simulation data from VSD in Neo4j and to load it back into VSD. Initially, when storing a simulation model in Neo4j, VSD's contents are archived once. Subsequently, changes in VSD are tracked and updated to Neo4j individually. That is, when a VSD instance has been changed just the changes are transferred as mentioned above. In the current version of the interface, only changes within VSD are tracked for resynchronization. Thus, Neo4j serves as database back end, which can store simulation models persistently.

The prototype is realized in a C++ based VEROSIM plugin, which uses Neo4j's Java API in embedded mode in order to communicate with Neo4j.

A. Data Mapping

In the context of this work, synchronization represents data transfer from one database to another. However, the structure of one database's data elements often differs from the other's. Thus, it becomes necessary to map these different structures on each other. Figure 4 shows our intuitive approach.

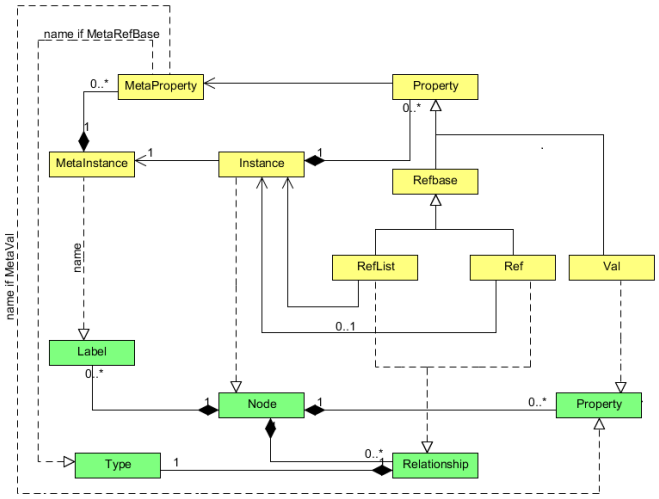

Figure 4. Data mapping of the synchronization component (top: VSD data model, bottom: Neo4j data model).

Single VSD instances are mapped to single Neo4j nodes and references (Ref/List-Properties) from one instance to another are represented by relationships between the corresponding nodes. The relationship is orientated to the referenced node's direction. Furthermore, we transfer the Val-Properties of a VSD instance to Neo4j node properties.

As mentioned above, a basic difference between VSD and Neo4j is that the former comprises metadata describing (and prescribing) a schema while the latter is schemaless. VSD metadata contains important information for the simulation and is indispensable for a correct data mapping. Thus, it is essential to transfer this informations as well. We store VSD metadata on Neo4j's object level:

1) A VSD instance's class name is mapped to it's Neo4j node's label,
2) a VSD Ref/List-Property's name is mapped to it's Neo4j relationship's type, and
3) a VSD Val-Property's name is mapped to it's Neo4j property's key.

Val-Property values are handled depending on their data type. If the type corresponds to one of Neo4j's supported basic types (e.g., integer, float, string, boolean, etc.) the value will be transfered directly. More complex data structures (e.g., mathematical vectors, etc.) are serialized to a binary representation and transfered as such.

To store additional meta information about VSD Ref/List-Properties, we take advantage of Neo4j's feature to add properties to relationships. Currently, every relationship gets a boolean property with the key *autodelete* as introduced above. Additionally, a RefList-Property entry's order is stored as an index in a relationship property.

B. Synchronization Component

Figure 5 depicts the structure of the synchronization component (based on [2]). Its core is the (object) mapper managing mappings between pairs of VSD instances and Neo4j nodes. Each mapping is stored in form of a so-called ObjectState (OS) holding all relevant information. The OS contains both objects' ids, all collected (but not executed) transactions and the state of the relation between the two. This state indicates whether the pair is synchronous, i.e., equal, or whether one of them has been changed and differs from its counterpart. An exemplary list of object states of the mapper is given in Figure 6.

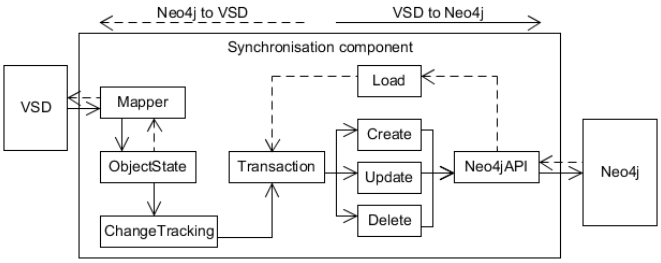

Figure 5. Synchronization component.

VSD-Id	Neo4j-Id	State	Pending-Transactions
...			
6049	0	SYNCED	-
6050	1	PENDING_UPDATE	Update-Transaction1, Update-Transaction2
6051	2	PENDING_DELETE	Delete-Transaction
...			

Figure 6. Exemplary list of object states of the mapper.

Each change to a VSD instance is encapsulated in a transaction stored in the appropriate OS. Subsequently, they can be executed. Depending on the change's type, a create, update, or delete transaction is generated. Furthermore, a separate load transaction is used to load Neo4j contents into VSD. Each transaction comprises all type specific information

 ISBN: 978-1-61208-486-2

necessary for its execution. For example, a create transaction contains the names and values of all Val-Properties, the class name, and information on Ref/List-Properties like target ids, reference names and *autodelete* values.

The last part of the synchronization component is the Neo4jAPI, which interacts with Neo4j's Java API.

1) VSD to Neo4j: VSD is an active database. One aspect of this activity is that changes to its instances are notified to registered components like the synchronization component presented in this work. Notifications include all relevant information about the modification like the instance's id or the changed property. The synchronization component encapsulates this information in an appropriate transaction. Using the instance id, the mapper is able to identify the corresponding OS and retrieve the mapping's state. A change tracking mechanism is used to filter redundant transactions as mentioned above (more details are given in Section V-C).

When the user or some automatic mechanism (e.g., a timer) triggers a resynchronization, all collected transactions are executed modifying Neo4j's contents accordingly.

2) Neo4j to VSD: When loading a Neo4j database's contents to VSD, the Neo4jAPI traverses the graph and generates a load transaction for each visited node. All data is read from Neo4j before entries are stored in the mapper. Load transactions contain the respective node's id, all its property keys and values and the ids of adjacent nodes of outgoing relationships and their respective properties (*autodelete* and index for RefList-Properties). Subsequently, the synchronization component executes all load transactions. For each, a new VSD instance with appropriate properties is created and its id is stored in an OS with the corresponding node's id.

C. Change Tracking

As mentioned above, when collecting transactions, newly created ones may cancel out older ones. A change tracking mechanism performs the necessary filtering of such redundant transactions.

Change tracking is based on the current state of the considered OS. Depending on the incoming transaction's type, the state changes and the list of collected transactions is updated.

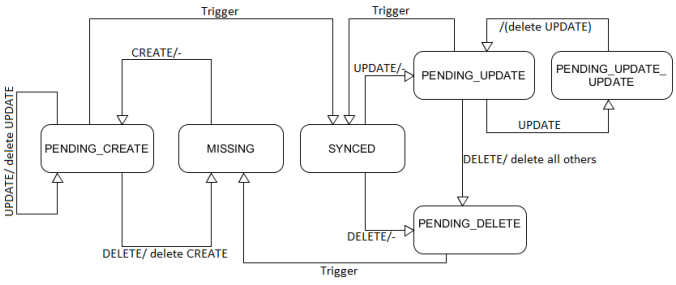

Figure 7. State machine of the change tracking mechanism.

Change tracking is modeled as a state machine as depicted in Figure 7. Here, the input (triggering state transitions) is represented by the incoming transaction type and the output (emitted during state transitions) describes the transaction list's modification. The initial state of any OS for a newly created VSD instance is the *MISSING* state as there is no corresponding Neo4j node. This intermediate state is left as soon as the corresponding create transaction is generated and the state changes to *PENDING_CREATE*. If this VSD instance is deleted before the transaction of type create has been executed, both transactions (create and delete) are removed and the whole OS is deleted. Else, upon a resynchronization trigger, a corresponding Neo4j node is generated, the state changes to *SYNCED*, and all executed transactions are removed from the list. The *SYNCED* state means that a Neo4j node and its VSD instance counterpart are in sync. It is reached every time a resynchronization was performed and is left when the VSD instance is modified (*PENDING_UPDATE*) or deleted (*PENDING_DELETE*).

In *PENDING_UPDATE* state, the changed property of an additional update transaction is compared to existing update transactions to avoid multiple updates of the same property. If two transactions modify the same property only one of them needs to be stored. This is represented by the intermediate *PENDING_UPDATE_UPDATE* state.

VI. Evaluation

Finally, the interface's effectiveness and performance have been evaluated using two simulation models of an industrial robot and a satellite shown in Figure 8. Given the current functional range of the presented prototype, further tests do not appear to provide more insights. Initially, both models are stored in a Neo4j database and, subsequently, loaded back into an (empty) VSD. The robot model yields 170 Neo4j nodes and 209 relationships. The more complex satellite about 20,000 nodes and 25,000 relationships. The highly connected nature of the 3D simulation data is apparent making a GDB ideally suited for its storage.

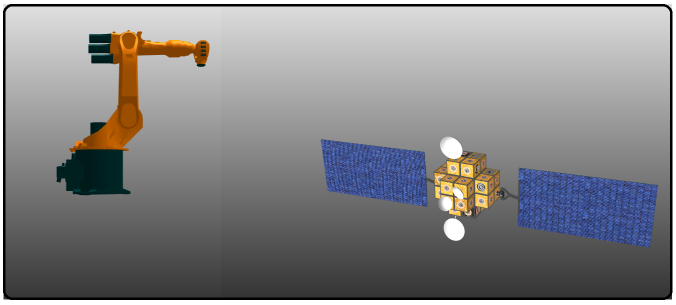

Figure 8. Robot and satellite (data: [16]) simulation model.

Figures 9 and 1 give an impression of the interface's effectiveness. Figure 9 shows an excerpt of the robot model data within Neo4j. Figure 1 shows the same data loaded into the VSD in-memory simulation database. The data mapping operates generically, i.e., independent from the actual data, making the whole synchronization component very flexible. The interface can synchronize arbitrary VSD contents to a Neo4j database.

In Section V, we present the interface's functionality to selectively resynchronize changes to VSD instances. This feature has been tested by changing some VSD instance's properties (e.g., name or postion of a component). In the Neo4j browser, we verified that these modifications were transferred correctly. Inversely, changes to node properties from the Neo4j browser show up in VEROSIM when the model is reloaded.

 ISBN: 978-1-61208-486-2

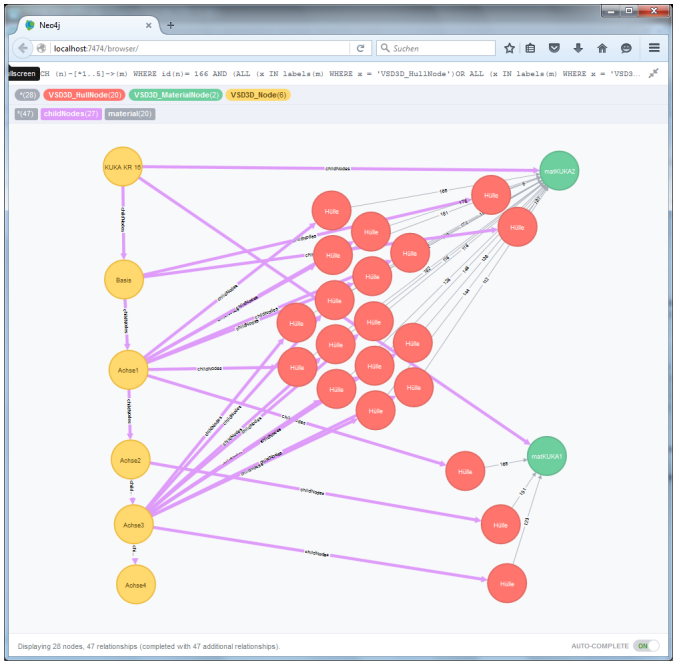

Figure 9. 3D simulation model data (excerpt) of an industrial robot stored within Neo4j.

TABLE I. LOADING AND SAVING TIMES OF THE PROTOTYPE.

	Neo4j		File	
	Robot	Satellite	Robot	Satellite
Loading	0.14	3.99	0.1	2.8
Saving	2.63	10.53	1.8	9.9

This also shows the advantage of selectively modifying data within a database in contrast to a file-based approach.

Another important aspect of the evaluation is the interface's performance. Here, the initial storage of a simulation model into Neo4j and the loading of a whole simulation model from Neo4j were examined and compared to saving and loading models to and from the native VEROSIM file format. Results are given in Table I. The access operations to the GDB are only somewhat slower than the native file operations. For a prototypical implementation from a student project, these results are very promising. First of all, compared to the highly optimized code for reading and writing the native file format, the current prototype is only optimized to a certain degree. Furthermore, the more high-level database access operations will always remain a little more complex than simple, sequential file reading or writing. Yet, the additional benefit from a full-fledged database (providing security, multi-user support, etc.) more than compensates for this small drawback.

Altogether, this shows that a GDB like Neo4j is well suited for highly connected 3D simulation model data and can be handled fast.

VII. CONCLUSION AND FUTURE WORK

Our approach to connect the GDB Neo4j with VEROSIM's simulation database VSD is motivated by the hierarchical structure of 3D simulation models that matches well with a graph structure. The presented interface encapsulates all VSD modification in independent transactions. A mapper maps individual VSD instances to single VSD nodes so that modifications can be processed individually and there is no need to save the complete VSD contents to Neo4j every time a single VSD instance changes. The stored data (as shown in Figure 9) indicates the highly linked structure of the simulation data so that GDBs are an ideal storage back end.

As future work, further performance optimizations and evaluations beyond the results from the student project could be performed. For instance, better traversal algorithms might improve loading speed. Another idea is to use Neo4j's batch inserter in contrast to the transactional structure to reduce resynchronization time. Furthermore, Neo4j might be used as a central database in a distributed simulation scenario with several VEROSIMs and VSDs. Here, an equivalent notification mechanism is needed for Neo4j to be able to track modifications in the central database.

REFERENCES

[1] I. Robinson, J. Webber, and E. Eifrem, Graph Databases-New Opportunities For Connected Data, 2nd ed. O'Reilly, 2015.

[2] M. Hoppen and J. Rossmann, "A Database Synchronization Approach for 3D Simulation Systems," in DBKDA 2014,The 6th International Conference on Advances in Databases, Knowledge, and Data Applications, A. Schmidt, K. Nitta, and J. S. Iztok Savnik, Eds., Chamonix, France, 2014, pp. 84–91.

[3] R. Diestel, Graph Theory, 2nd ed. Springer, 2000.

[4] M. Hunger, Neo4j 2.0 A graph database for everyone (orig.: Neo4j 2.0 Eine Graphdatenbank für alle), 1st ed. entwickler.press, 2014.

[5] Neo4j Team, "The Neo4j Manual v2.2.5," 2015, URL: http://neo4j.com/docs/stable/ [retrieved: May, 2016].

[6] J. Rossmann, M. Schluse, C. Schlette, and R. Waspe, "A New Approach to 3D Simulation Technology as Enabling Technology for eRobotics," in 1st International Simulation Tools Conference & EXPO 2013, SIMEX'2013, J. F. M. Van Impe and F. Logist, Eds., Brussels, Belgium, 2013, pp. 39–46.

[7] N. Martinez-Bazan, S. Gomez-Villamor, and F. Escale-Claveras, "Dex: A high-performance graph database management system," in Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference on, April 2011, pp. 124–127.

[8] R. Kumar Kaliyar, "Graph databases: A survey," in Computing, Communication Automation (ICCCA), 2015 International Conference on, May 2015, pp. 785–790.

[9] Microsoft, "Graph engine 1.0 preview released," 2016, URL: http://research.microsoft.com/en-us/projects/trinity/ [retrieved: May, 2016].

[10] B. Shao, H. Wang, and Y. Li, "Trinity: A distributed graph engine on a memory cloud," in Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data. ACM, 2013, pp. 505–516.

[11] B. Iordanov, "HyperGraphDB: A Generalized Graph Database," in Web-Age information management. Springer, 2010, pp. 25–36.

[12] InfoGrid Team, "Infogrid: The web graph database," 2016, URL: http://infogrid.org/trac/ [retrieved: May, 2016].

[13] J. Peilee, "A survey on graph databases," 2011, URL: https://jasperpeilee.wordpress.com/2011/11/25/a-survey-on-graph-databases/ [retrieved: May, 2016].

[14] "Allegrograph," 2016, URL: http://franz.com/agraph/allegrograph/ [retrieved: May, 2016].

[15] M. Hoppen, M. Schluse, J. Rossmann, and B. Weitzig, "Database-Driven Distributed 3D Simulation," in Proceedings of the 2012 Winter Simulation Conference, 2012, pp. 1–12.

[16] J. Weise et al., "An Intelligent Building Blocks Concept for On-Orbit-Satellite Servcing," in Proceedings of International Symposium on Artificial Intelligence, Robotics and Automation in Space (i-SAIRAS), 2012, pp. 1–8.

 ISBN: 978-1-61208-486-2