Implementing Graph Transformation Languages using RDF Storage and SPARQL Queries

[†]Māris Jukšs, [†,‡]Hans Vangheluwe, [†]Clark Verbrugge
[†]School of Computer Science, McGill University, [‡]Department of Mathematics and Computer Science, University of Antwerp, Belgium

August 31, 2012

Abstract

The Resource Description Framework (RDF) format has seen signficant interest from industry as a flexible format for graph representation. However, RDF has so far not been widely used in the context of graph rewriting tools. In this paper, we take AToMPM, our research-oriented meta-modelling and graph rewriting tool and extend it by adding RDF-based graph transformation functionality. We address the graph matching problem, generating custom SPARQL queries for each rule pattern, and use that to efficiently perform matches on a source RDF graph. In this way we create a research system for RDF graph transformations that also gives us the ability to investigate different possible approaches to using RDF and evaluate the effects on the system. Our experience and performance data serves as a guideline for evaluating the trade-off between the inherent efficiency of a specialized representation and the flexibility of using a standardized representation.

1 Introduction

The size of host graphs is a significant concern in the application of graph rewriting systems. Scalability concerns in terms of graph matching magnify greatly with input graph size, and best performance is achieved through the use of highly optimized, system-specific designs for graph storage and management. The complexity and performance of these subsystems, however, represents a trade-off with flexibility and increased scalability—a large graph model may be the target of multiple, concurrently executing applications, with transactional and distributed support implying a shared, system-agnostic representation is equally important.

The Resource Description Framework (RDF) format is a highly flexible format that easily accommodates a wide variety of representation requirements [11]. The use of generic triples to form associations between elements, combined with namespaces and many other features make RDF an appropriate format for representing models that may be accessed by different applications. Industrial strength database back-ends exist for RDF storage, allowing RDF content to extend to truly enormous graphs, with commercial systems available that accommodate trillions of elements [9]. Despite these advantages, however, graph transformation systems have so far not been widely integrated into RDF contexts.

In this work we explore the use of RDF in the context of a graph transformation system. We extend our, research-oriented, meta-modeling and graph rewriting tool, AToMPM [12], by adding support for different RDF back-ends. Our work allows access to RDF graph representations, and enables graph matching through either a standard interface built on basic CRUD operations, or by using specialized SPARQL queries [14]. With this we are able to do experimental studies on non-trivial systems, evaluating how graph transformation performance of RDF-based representations compares with performance of our original `igraph` [7] (in-memory) back-end. As a more generic representation, our prototype RDF-based systems show significant performance degradation. The effect, however, is not uniform, with different graph operations exhibiting different performance changes. Our experience shows that RDF support is straightforward, and gives experimental evidence for where optimization needs to be concentrated in order to improve performance and so allow graph transformation to feasibly exploit the inherent flexibility and scalability of modern RDF back-ends.

Specific contributions of our work include:

- We augment a state-of-the-art, research-oriented graph transformation system with support for RDF graphs. This includes instrumentation to obtain performance data. This technical contribution is important in the sense that it enables non-trivial experimentation, and comparison with other back-end designs.
- The SPARQL interface to RDF allows for non-trivial graph matching to be expressed, simplifying the matching step, and enables exploiting any optimizations built into the SPARQL system. We show how to

construct customized SPARQL queries that represent the structural requirements of the Left-Hand Side (LHS) of a graph rewriting rule.
- Use of a generic graph representation back-end implies a trade-off between performance and flexibility/scalability. Using our prototype implementation we provide initial experimental data comparing the performance between two different RDF back-ends and a customized `igraph` back-end.

2 Related Work

[2] present a framework for programmable RDF model transformations. Here the transformation is not a problem of graph matching and rewriting but rather RDF semantic data manipulation based on programmable transformation chunks called Atoms. The implementation was applied in the context of middleware message servers. RDF graphs have been used to represent abstract syntax of models and algebraic graph transformations were devised on them in [4]. The approach described aims to manipulate RDF triples directly using rewriting rules. [3] describes algebraic RDF transformations using different methods such as single or double push-out and proves that transformations can be reversible. SPARQL back-end implementations use different approaches to answer queries. [18] suggest using the subgraph matching approach to facilitate the processing of queries with wild cards. The aim was to provide uniform answers to various queries. Graph rewriting applications may run out of physical memory for huge graphs. Rewriting of the graphs located in a relational database could alleviate such problems and is described in [17]. `igraph` library works on graphs that reside entirely in memory and used in AToMPM. A relation database back-end for `igraph` is developed in [1]. [6] describes SPARQL query manipulations to mediate RDF data. The approach used produces new SPARQL queries based on query pattern matching and rewriting. In [10], the authors perform transformations from an object oriented query language to SPARQL queries and vice versa. This simplifies querying semantic data in the RDF stores for object oriented domains.

3 Background

Our work depends on a number of specific technologies and tools. In this section we give essential backround on AToMPM, `igraph`/Himesis, and RDF that is necessary for understanding our work.

3.1 AToMPM

AToMPM, A Tool for Multi-Paradigm Modeling, is a multi-formalism/multi-abstraction meta modeling and model transformations tool, developed as a successor to AToM^3 [8]. One of the distinguishing features of AToMPM is the browser based user interface that eliminates the need for complex installations and setups for domain specific language (DSL) engineers and users. In AToMPM everything is a model, from the tool bars in the browser to the user interface behavior. The tool is in active development and aspires to be the answer to the shortage of simple and usable model-driven engineering tools (MDE).

AToMPM's graph rewriting capability is powered by T-Core [16], a collection transformation primitives which abstracts the graph matching and rewriting aspects of graph transformation rules and provides a universal way of dealing with graph rewriting problems and in particular, of designing transformation languages. Graphs are encapsulated in packets. Packets are passed to Matcher, Rewriter, Iterator, Rollbacker T-Core primitives, and other entities which perform the necessary function based on the packet information. In particular the Matcher performs the matching of model patterns on incoming graph pattern. This separation of duties gives AToMPM significant flexibility, without sacrificing efficiency.

Figure 1 illustrates the basic architecture of this system: at the top, a JavaScript client component provides a GUI interface, which then communicates with a server component. The server itself is split into several components. Client requests are first handled by the Concrete Syntax Server which is responsible for updating the concrete syntax once model transformations take place. It communicates with the Abstract Syntax Server, which maintains the model structure, and forwards requests to the (Python) transformation server, where model transformations are carried out. The latter provides a uniform CRUD (Create, Read, Update, Delete) API interface over different back-ends, and is currently directed toward a Himesis [13] representation based on the `igraph` [7] library. For the work in this paper we modified the Python server to support RDF. Details of which are discussed in the next section.

3.2 `igraph`/Himesis

Graph representation and transformations are actually carried out within the C-based `igraph` library, contained within a Himesis representation in order to support hierarchies. Graph structures in `igraph` have a simple and well-optimized representation. Nodes, for instance, are implicitly represented by an integer index, allowing the system to allocate nodes simply by incrementing a maximum node counter. Once a node is allocated, edges can be constructed as (directed) pairs of node indices, and are themselves referenced by an index. This design allows for straightforward and efficient access to graph structures, using array-like access semantics.

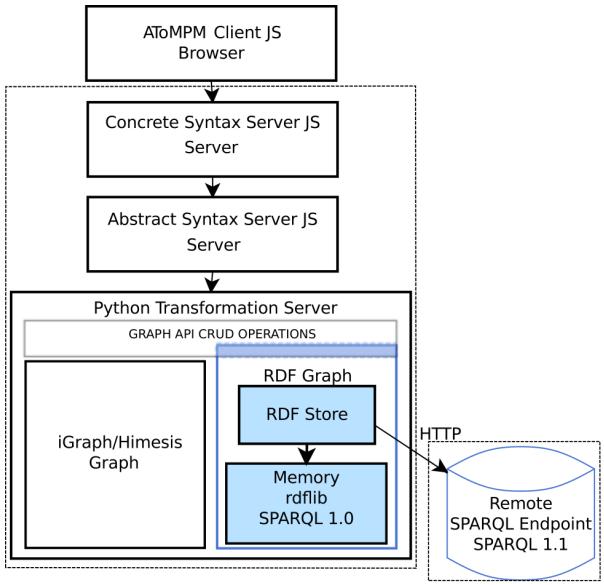

Figure 1: Architecture of our AToMPM and our RDF integration.

3.3 RDF

RDF [11] is a highly flexible data representation, intended to facilitate use within arbitrary semantic domains. It has been widely used in the context of the semantic web, and provides several serialization formats, including XML, Notation-3, N-Triples, Turtle, and N-Quads. Our focus here is on the use of Triples, which allow for straightforward expression of (attributed) graph structures. Generically, each triple consists of a subject, a predicate, and an object, all maintained as strings in the RDF store. Subjects and predicates can be arbitrary RDF URI (Uniform Resource Identifier) references, while objects can be either RDF URI references or simple literals. Use of URIs allows for easy web integration, as well as domain scoping. In our case, triples encode attributed associations between subjects and objects, which may be graph nodes or edges. We discuss this more in the next Section.

RDF can be queried directly, or at a higher level through SPARQL [14], a structured query language that allows for querying of RDF triples with a syntax similar to that of SQL. SPARQL enables construction of complex and compound queries, and is thus suitable for more efficiently matching of multi-node/edge graph patterns, as found in non-trivial graph rewriting systems.

4 RDF Integration

In this section we describe the design of our RDF integration and associated model transformations in more detail. Figure 1 showed the newly added components; these can be summarized as:

- An RDF graph which provides an implementaton of the graph CRUD operation API. The RDF graph form we use can be based on either of two distinct back-ends. We developed support for `rdflib`, an in-memory RDF store that supports SPARQL 1.0, and a remote SPARQL endpoint that supports version 1.1 of the SPARQL draft. For the latter we use AllegroGraph 4.6 SPARQL server [9], although, any SPARQL standard implementation should work.
- An RDF graph matching component. This is completely different from the one used in `igraph` and is one of the most important parts of this work. Matching is performed using SPARQL queries and is thus closely integrated with the RDF graph implementation.

The two RDF back-ends represent different approaches to using RDF. The `rdflib` design uses a local RDF store, with a simplified deployment and setup. The SPARQL back-end incurs additional network overhead, but allows for use of an external, distributed or cloud-based RDF store, improving scalability. To integrate with AToMPM, we mainly reuse the CRUD graph API layer that already wraps Himesis/`igraph`; such reuse facilitates almost effortless switching to new graph implementations that can use different back-ends. The main differences then lie within the RDF graph matching that is done using SPARQL queries, as opposed to VF2 [5] based graph matching. VF2 is an algorithm for efficiently solving sub-graph isomorphism problem as a constraint satisfaction problem. The RDF Graph API has following most notable features:

- Node iteration and node access is provided via the `[]` operator. To achieve this we query the RDF store for nodes located in a particular context and yield node objects to the application. Node object contains all attributes for a given node.
- Support for getting and setting node attributes. Node objects behave like a dictionary, where we can set and get key-value pairs. Internally this is achieved by SPARQL update queries for the SPARQL endpoint, or API calls in case of `rdflib`.
- Iteration over graph edges is allowed, yielding edge objects containing adjacent node identifiers.
- A select function that allows querying nodes based on particular attribute values.

Since AToMPM uses T-Core for controlling and performing graph rewriting operations, we also reuse that in out work. However, we do change the matching part which is not part of a T-Core; more on matching and rewriting is provided in the following sections.

4.1 RDF Graph Representation

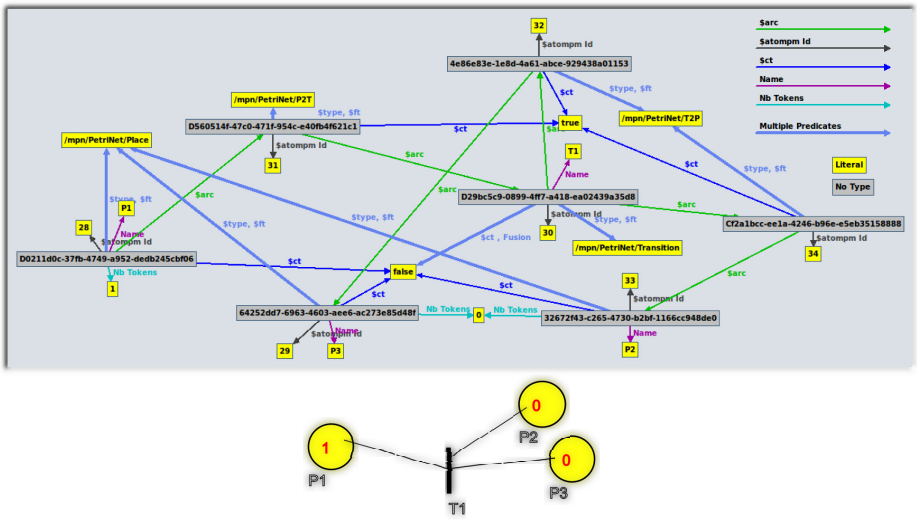

Figure 2: RDF Representation of an AToMPM Petri Net.

Our RDF integration relies on a specific representation of the host graph in RDF form. Figure 2 gives an example of this translation, mapping a small Petri Net model in AToMPM to RDF. In general, every AToMPM node and edge is associated with an unique identifier (UID). Domain specific and implementation related attributes, as well as node/edge associations are then compiled into RDF triples. In the above example, grey nodes represent basic entities in the source Petri Net, with attributed edges from grey nodes representing attached AToMPM node attributes, encoded as RDF predicates or properties. At the end of each edge can be other grey nodes or yellow nodes representing literals. Node P1, for instance, is represented by unique identifier (UID) `D0211d0c-37fb-4749-a952-dedb245cbf06`, which then becomes the subject for all triples pertaining to node P1. P1's name branches from this, described by an edge marked as `name` and connected to literal node `P1`. In the Petri Net model P1 is connected to transition T1, with this edge represented by UID `D560514f-47c0-471f-954c-e40fb4f621c1`, and the connection relation encoded as triple `(D0211d0c-37fb-4749-a952-dedb245cbf06, $arc, D560514f-47c0-471f-954c-e40fb4f621c1)`. Notice that in this picture node P1 has a number of other, implementation dependent attributes, each represented by a corresponding RDF triple.

4.2 RDF Store Operations

The two RDF back-ends used in this paper require two distinct approaches. The nature of each dictates the operations one can perform on them. First, the `rdflib` RDF graph does not support SPARQL 1.1, which means that it is impossible to execute INSERT and DELETE queries. These operations are thus implemented using API calls. In the case of the SPARQL endpoint we use the HTTP protocol to execute various queries and achieve desired results. We chose the network approach to using the AllegroGraph API to have a universal and inherently scalable solution that can be applied on any standard remote SPARQL endpoint, with minimal or no changes. The main drawback of the HTTP approach is of course the network traffic, which we will discuss in the results section.

4.3 Matching and Rewriting

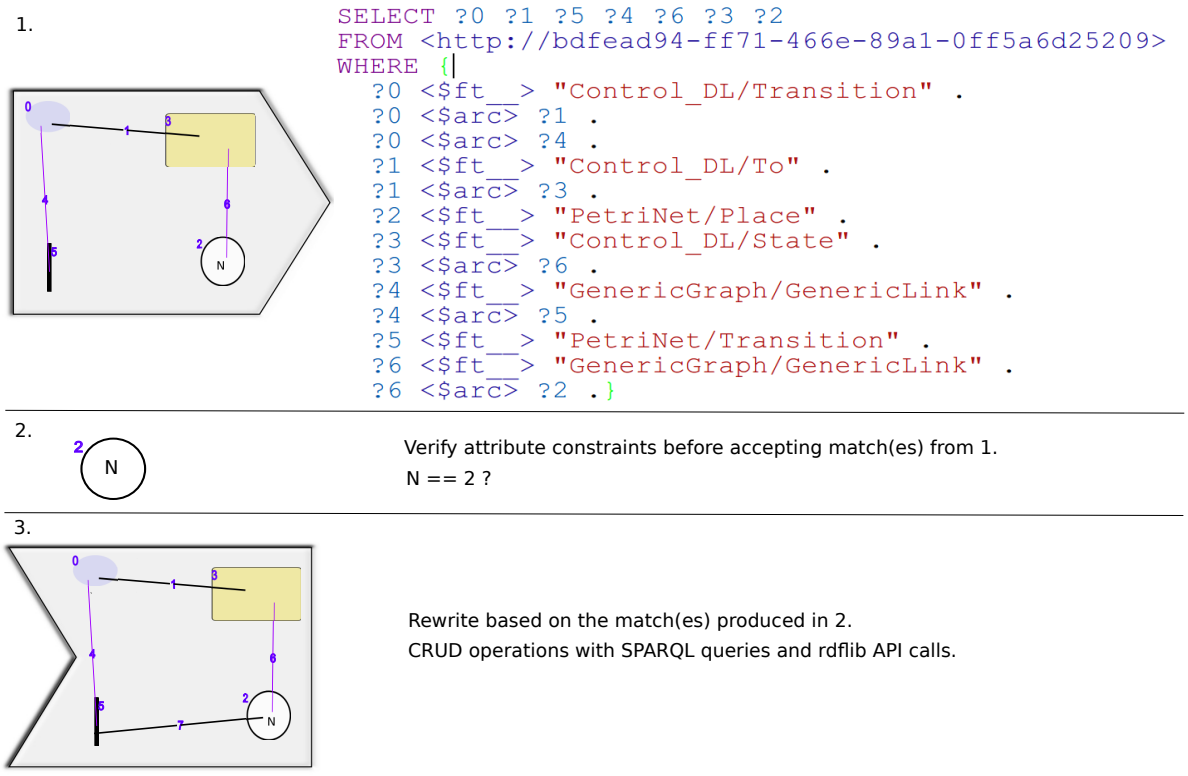

Figure 3: RDF graph rewriting.

An advantage of SPARQL queries is that they are expressive enough to allow matching entire graph patterns. In this way we relay the complexity of matching to the RDF store implementation with the hope that it is as efficient as possible. Figure 3 shows the steps to apply one of the rules in a transformation from automata to Petri Nets. In Step 1 the LHS of the graph rewriting rule is matched to the source graph using the SPARQL query shown on the right. For every LHS or negative application condition (NAC) we generate a query specific to the graph pattern of the rule. Since every entity in the AToMPM rule is uniquely labeled this allows us to use the labels as SPARQL query variables. The query in Step 1 tries to match the blue node with label 0 so that it has a predicate $ft__ (full type) with a literal value of Control_DL/Transition, predicate $arc to a node labeled 1 and $arc to a node labeled 4, and so forth. This query returns the mapping from pattern node labels to unique identifiers within the source graph. In Step 2 the match is validated against any non-structural node attribute constraints. In this case we verify that attribute N on the source graph node is equal to the number 6. Successful matches are used to rewrite the graph in Step 3 according to the RHS of the rewriting rule. In this particular case it is creating a connection from a Petri Net transition to a place with an edge marked by label 7. The rewrite itself is performed using graph CRUD operations with a mix of RDF-specific operations, depending on the back-end used.

5 Experiments

Use of RDF is expected to introduce significant runtime overhead, and so we have done performance experiments to evaluate how well our two RDF back-ends perform, relative to each other and to the baseline `igraph` implementation. Below we present our benchmarks, followed by experimental data and discussion.

5.1 Benchmarks

Benchmarks are aimed at evaluating performance of graph transformations, as well as at determining the cost of basic CRUD operations.

To review the performance of the RDF graph transformations we choose an existing transformation from automata to Petri Nets, and apply it to two distinctly sized automata models. Figure 4 shows the two models, extracted from a full model of a vehicle power-window, and Figure 5 shows the transformations used. The

latter includes non-trivial control flow between rules, witch colored edges between nodes signifying control flow, depending on the outcome of the rule execution. A grey edge means to execute the next rule when the current rule is not applicable, whereas a green edge shows the next rule upon successful application of a rule. Rules inside the transformation have a good mix of matching and rewriting challenges, including extensive use of graph rule negative application conditions (NAC).

Transformations are executed 7 times for each of the `igraph`, `rdflib`, and SPARQL implementations. To reduce cache/startup effects, the first result is discarded, and we report the average of the remaining six. While performing the transformation we record the time for matching and rewriting separately, as well as overall transformation time. It is worth noting that we do not time the Iterator primitive itself, which chooses a single match for rewriting from a set of matches. In these tests the time to choose a match is infinitesimal and not particularly interesting in this context.

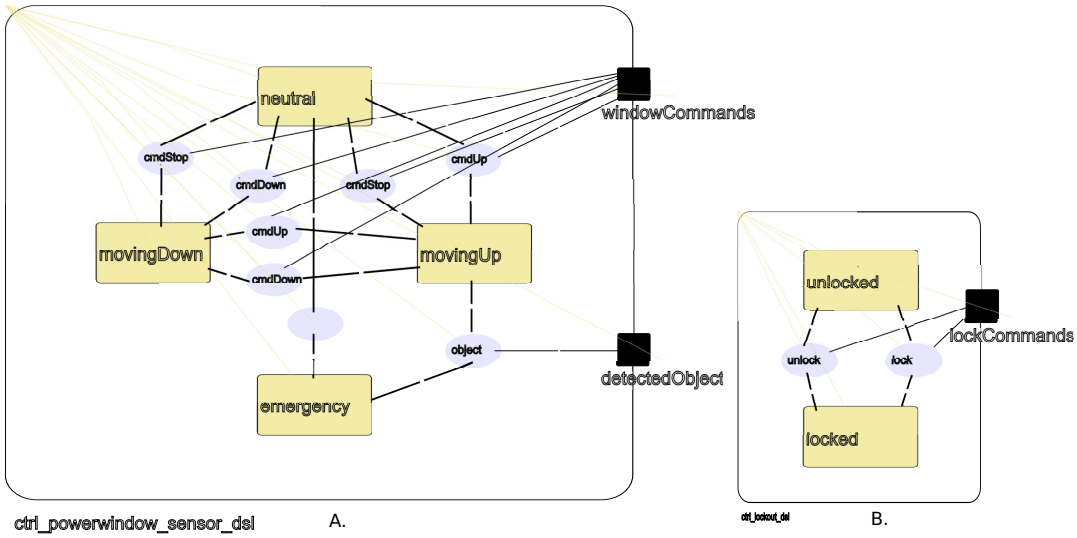

Figure 4: Benchmark models, drawn from a model of power-window behavior.

Figure 5: Automaton to Petri Net transformation.

For CRUD operation evaluation we devised a set of basic tests that are timed and compared against the existing `igraph` based implementation. Extensive performance evaluation of `igraph` was done in [15]. We reevaluate `igraph` CRUD operations in the context of RDF graph rewriting so that comparison was possible. Note that for the CRUD operations test we try to optimize certain things for the RDF graph with SPARQL back end.

In particular, we produce batch queries where possible. These kinds of optimizations are not performed for the `rdflib` RDF graph transformation deployment inside the AToMPM tool.

The initial condition of the CRUD test is a graph with 50 nodes and 25 random edges. `igraph` keeps track of nodes and edges efficiently by simply keeping the counter equal to the number of nodes or edges. In the case of the RDF we store the added node UIDs so that we did not have to query which nodes are in the RDF store. We also store node UIDs that are the source of an edge. These lists would not be part of a normal invocation, and are primarily used to help choose random node UIDs without unnecessary query overhead. Actual tests are drawn from a set of basic operations, {AN,AE,UA,DE,DN}, which are:

- AN - add a node with three attributes. For the RDF graph that means adding three attribute triples to the store. For `igraph`, it creates a node and assigns three attributes.
- AE - add a directed edge. We simply add one triple after randomly choosing source and destination nodes to create an edge for RDF graph. Note that this results in a multi-graph. For `igraph` this requires a simple API call after choosing adjacent nodes in a similar fashion.
- UA - update attributes. This is a straightforward operation for `igraph`. Some SPARQL implementations do not support in-place updates, and so attempting to update an existing triple that is distinct only in the value of an object will result in duplicate triples with separate object values. Thus in RDF we first delete three attributes then add new ones.
- DE - delete edge. The edge is randomly chosen for deletion.
- DN - delete node. We erase a node as well as all of its attributes, and any edges that connect to that node.

Each CRUD operation is performed N times for N = {10,100,1000,10000}, averaged over 10 iterations, disregarding the first iteration result. This timing is then divided by N to give us a time for each CRUD operation individually.

5.2 Results

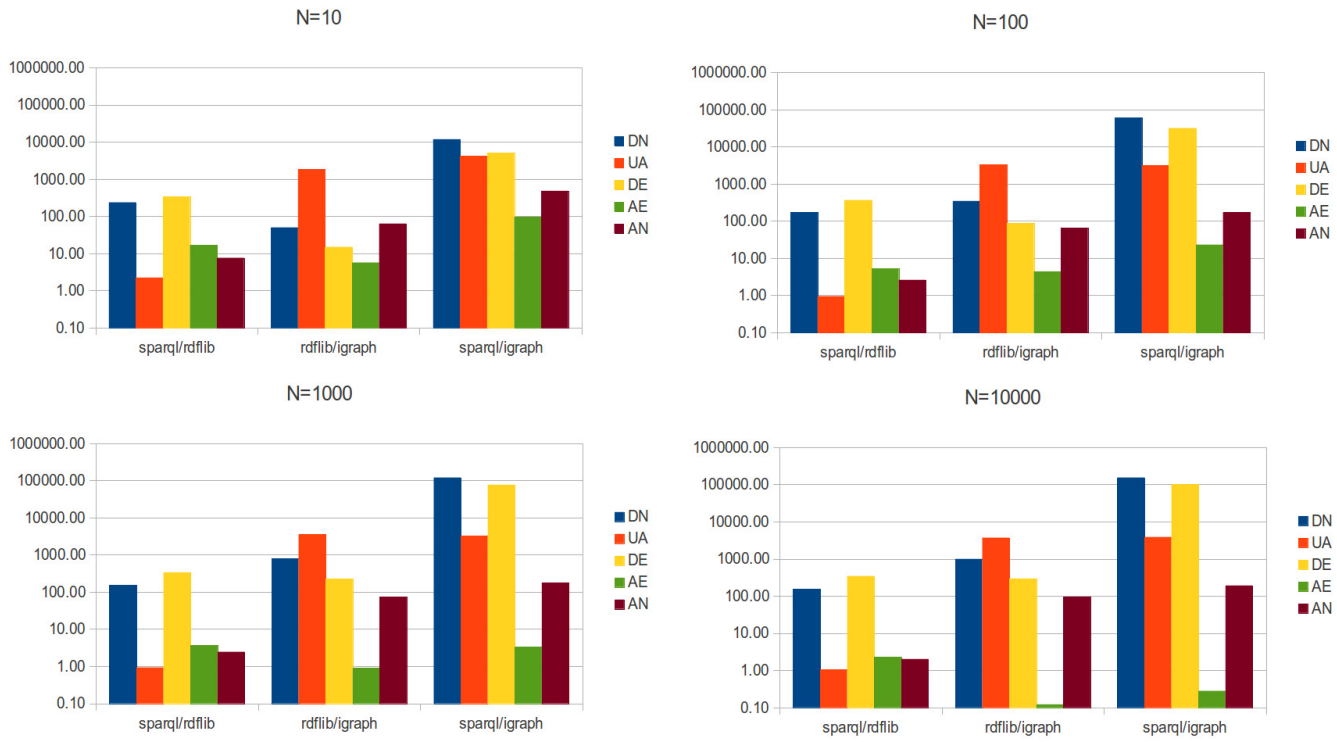

Figure 6: CRUD operation results.

Figure 6 presents the result of our CRUD experiments comparing an existing implementation based on `igraph` to both `rdflib` in-memory and SPARQL endpoint implementations. These results are normalized, to allow easy comparison of the two RDF implementations to each other and to `igraph`—values below 1.0 indicate speedup, whereas values above 1.0 indicate a slowdown by the indicated factor. Unsurprisingly, RDF operations tend to be significantly slower, by several orders of magnitude. This is primarily due to the additional network traffic in the SPARQL implementation and the need for results parsing and manipulation for both RDF back-ends.

Table 1: Average transformation times in second for models A and B as displayed in Figure 4 using the transformations shown in Figure 5.

	Model A			Model B		
	Overall	Match	Rewrite	Overall	Match	Rewrite
`igraph`	0.2071	0.1761	0.0287	1.0491	0.9732	0.0713
`rdflib`	10.1838	7.5235	2.6593	375.6626	361.4132	14.2464
SPARQL	41.8108	23.246	18.563	673.9526	616.0905	57.8582

Interestingly, the AE operation is actually as fast as `igraph` for both `rdflib` and SPARQL starting from N=1000 and faster for N=10000. This is likely due to the simpler way edges are stored in RDF, versus the more complex data structures in `igraph` used to ensure edges are properly associated, and suggests that an RDF design can be competitive, at least for some sets of operations.

Table 1 displays the results of our transformation rule tests. Here it is also obvious that with our naive implementation of RDF, `igraph` still has a dramatic performance edge. Due to nature of our transformation and the frequent use of NACs, the matching part takes the majority of time for the transformation, larger than rewriting itself by approximately a factor of 10 in all implementations. This is to be expected, as rewriting often involves only a few CRUD operations used to perform the structural and non-structural graph changes on a known set of nodes, edges, and attributes. The `rdflib` implementation is faster than the SPARQL implementation in this case, as the CRUD operations evaluation also demonstrated. Again this is due the fact that there is no network traffic, and we are required to do less manipulation of the data the application receives.

6 Conclusions and Future Work

RDF combines advantages of being a well defined, but semantically neutral format with demonstrably good implementation scalability. Use of RDF is thus appealing within a graph transformation system, but since speed is a potential trade-off for this flexibility it is important to investigate practical performance. By integrating support for RDF into a non-trivial graph transformation framework we enable such exploration. Our modifications to AToMPM allow us to gather experimental data, comparing an optimized, but memory-limited internal representation with different RDF back-ends, including an externally hosted SPARQL endpoint. In our current implementation RDF performance is clearly sub-optimal, but gives useful guidance on where further performance optimization needs to be concentrated.

Future work for our system is of course concentrated on optimizing the design based on our initial data. Pattern matching SPARQL queries, for instance, could be expanded to extract extra data to speed up attribute-constraint verification. A caching mechanism, coupled with CRUD operation improvement via SPARQL query optimization would also be beneficial. Other directions include investigating performance for the very large models RDF is best directed at, and looking into the use of hybrid back-ends that delegate to RDF or `igraph` as appropriate.

References

[1] M. Anquetin, P. Baudemont, T. Franville-Lafargue, E. Navarro, and G. Racineux. igraph database project. Technical report, École nationale supérieure d'électronique, d'électrotechnique, d'informatique, d'hydraulique, et de télé- communications, Toulouse (France), 2009.

[2] E. Babkin, A. Sevastianov, A. Shutov, and A. Zhdankin. An approach to programmable RDF-model transformations. In *Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, ICDCS '04, pages 150–157. IEEE Computer Society, 2004.

[3] B. Braatz and C. Brandt. *Graph Transformations for the Resource Description Framework*, volume 2, page 14. EASST, 2008.

[4] B. Braatz and C. Brandt. Domain-specific modelling languages with algebraic graph transformations on rdf. In *Proceedings of the Third international conference on Software language engineering*, SLE'10, pages 82–101. Springer-Verlag, 2011.

[5] L. P. Cordella, P. Foggia, C. Sansone, and M. Vento. A (sub)graph isomorphism algorithm for matching large graphs. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, 26(10):1367–1372, Oct. 2004.

[6] G. Correndo, M. Salvadores, I. Millard, H. Glaser, and N. Shadbolt. SPARQL query rewriting for implementing data integration over linked data. In *Proceedings of the 2010 EDBT/ICDT Workshops*, EDBT '10, pages 4:1–4:11. ACM, 2010.

[7] G. Csrdi and T. Nepusz. igraph reference manual, 2012. `http://igraph.sourceforge.net/`.

[8] J. de Lara and H. L. Vangheluwe. Using AToM^3 as a meta-CASE environment. In *4th International Conference On Enterprise Information Systems*, 2002.

[9] Franz Inc. Allegrograph. `http://www.franz.com/agraph/allegrograph/`.

[10] G. Hillairet, F. Bertrand, and J. Lafaye. Rewriting queries by means of model transformations from SPARQL to OQL and vice-versa. In R. Paige, editor, *Theory and Practice of Model Transformations*, volume 5563 of *Lecture Notes in Computer Science*, pages 116–131. Springer Berlin / Heidelberg, 2009.

[11] G. Klyne and J. J. Carroll. Resource description framework (RDF): Concepts and abstract syntax. Technical Report REC-rdf-concepts-20040210, W3C, 2004.

[12] R. Mannadiar. *A Multi-Paradigm Modelling Approach to the Foundations of Domain-Specific Modelling.* PhD thesis, McGill University, 2012.

[13] M. Provost. Himesis : A hierarchical subgraph matching kernel for model driven development. Master's thesis, McGill University, 2005.

[14] E. Prud'hommeaux and A. Seaborne. SPARQL query language for RDF. Technical Report REC-rdf-sparql-query-20080115, W3C, 2008.

[15] E. Syriani. *A Multi-Paradigm Foundation for Model Transformation Language Engineering.* PhD thesis, McGill University, 2011.

[16] E. Syriani and H. Vangheluwe. De-/re-constructing model transformation languages. *ECEASST*, 29, 2010.

[17] G. Varró, K. Friedl, and D. Varró. Implementing a graph transformation engine in relational databases. *Software and Systems Modeling*, 5:313–341, 2006.

[18] L. Zou, J. Mo, L. Chen, M. T. Özsu, and D. Zhao. gstore: answering sparql queries via subgraph matching. *Proc. VLDB Endow.*, 4(8):482–493, 2011.