Noname manuscript No.
(will be inserted by the editor)

Connecting knowledge to data through transformations in KnowID: system description

Pablo R. Fillottrani · Stephan Jamieson · C. Maria Keet (✉)

Received: date / Accepted: date

Abstract Intelligent information systems deploy applied ontologies or logic-based conceptual data models for effective and efficient data management and to assist with decision-making. A core deliberation in the design of such systems, is how to link the knowledge to the data. We recently designed a novel knowledge-to-data architecture (KnowID) which aims to solve this critical step through a set of transformation rules rather than a mapping layer, which operate between models represented in EER notation and an enhanced relational model called the ARM. This system description zooms in on the novel tool for the core component of the transformation from the Artificial Intelligence-oriented modelling to the relational database-oriented data management. It provides an overview of the requirements, design, and implementation of the modular transformations module that straightforwardly permits extension with other components of the modular KnowID architecture.

Keywords Ontology-mediated Data Access · Data Management · Conceptual Modeling

Pablo R. Fillottrani
Departamento de Ciencias e Ingeniería de la Computación, Universidad Nacional del Sur, Bahía Blanca, Argentina and Comisión de Investigaciones Científicas, Provincia de Buenos Aires, Argentina

Stephan Jamieson
Department of Computer Science, University of Cape Town, South Africa

C. Maria Keet
Department of Computer Science, University of Cape Town, South Africa
E-mail: mkeet@cs.uct.ac.za

1 Introduction

Application ontologies, or logic-based conceptual data models, as well as knowledge graphs, are used to achieve effective data management with faster data analysis thanks to querying at the conceptual layer. This advantage comes at a cost of devising a good and efficient way to connect the knowledge to the data; an introductory overview of several options and considerations are described in [12]. The very recently proposed KnowID architecture [9] (see Fig. 1) is a highly modular architecture for such applications, which takes Extended Entity Relationship (EER) diagrams as application ontologies and connects them to the data layer via the so-called 'Abstract Relational Model' (ARM) of [1] that uses special object identifiers and a strict extension to SQL for path queries (SQLP) that simplify querying [14]. Set within this context of the KnowID architecture, the main contribution of this system description paper is the presentation of the first proof-of-concept implementation of connecting the AI-oriented knowledge layer to the database-oriented data layer through transformations, transforming application ontologies represented in EER to ARM and vice versa. It shows that what ought to work theoretically, indeed does so practically. It has a front-end for model management that is linked but decoupled from the back-end that carries out the transformations and generates a log file that is to be used subsequently for processing knowledge-based queries. The back-end has an API, so that other modules can be 'plugged in' at a later stage. The business logic is achieved with commonly supported data structures, which then are converted into the latest Semantic Web and Knowledge Graph technologies, which is demonstrated here with JSON. The source code, docu-

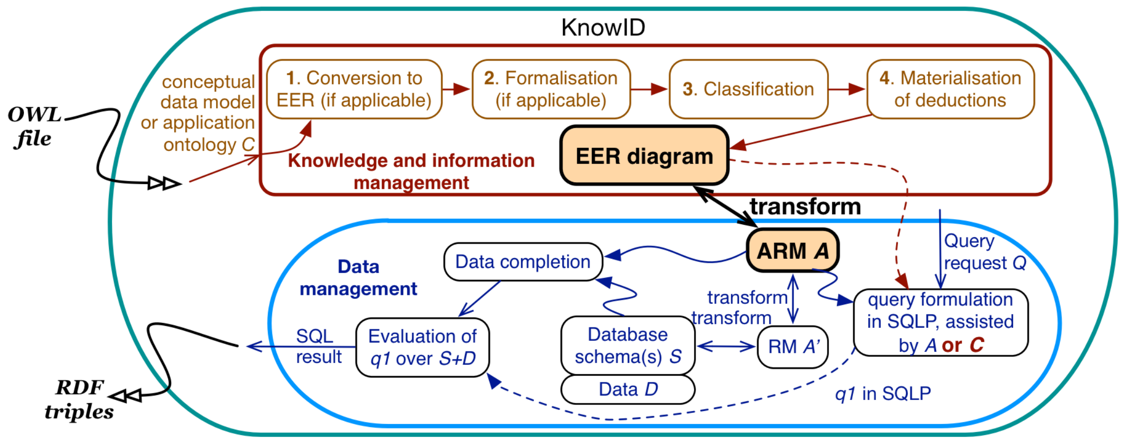

KnowID
OWL file
conceptual data model or application ontology C
1. Conversion to EER (if applicable)
2. Formalisation (if applicable)
3. Classification
4. Materialisation of deductions
Knowledge and information management
EER diagram
transform
ARM A
Query request Q
Data management
Data completion
transform
transform
query formulation in SQLP, assisted by A or C
SQL result
Evaluation of q1 over S+D
Database schema(s) S
RM A'
Data D
q1 in SQLP
RDF triples

Fig. 1 The KnowID architecture, with the focus of the system presented in this paper highlighted in bold black font text in the two dark shaded boxes connected with the thick arrow: providing the knowledge-to-data connection as model transformations.

mentation, and test cases are available at `http://www.meteck.org/KnowID.html`.

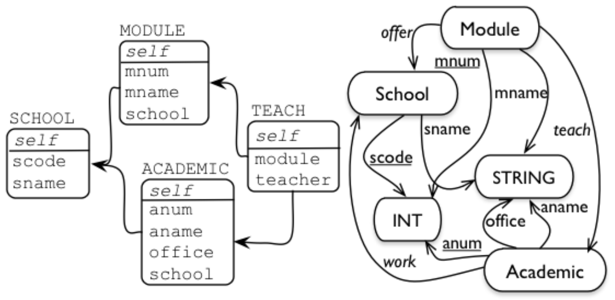
The currently most well-known proposed solution to link knowledge represented in application ontologies to data stored in database is called ontology-based data access (OBDA) [3], which has closely related recent activities with (virtual) knowledge graphs [15,20]. OBDA uses a mapping layer between the ontology and the relational database(s), and avails of query rewriting to answer conjunctive queries. Both features are costly—computationally [10] as well as in design and maintenance [13]. Moreover, data analysts typically want to be able to use the full SQL expressiveness and stay with the closed world assumption that is better known to them than the open world assumption in OBDA. These issues are avoided in KnowID by using transformation rules [9]. That theoretical design was yet to be assessed on practical correctness, feasibility of implementation, and flexible architectures to implement it. The described system realises the knowledge-to-data inter-layer connection in KnowID, and serves as a basis for its further development and application complement the well-known technologies that compose both the knowledge and data layer. Furthermore, although there are approaches to support diverse fragments of the SQL query language in ODBA (see for example [4, 11]), they are incomplete and involve a complex transformation leading to a loss of intuition for the modeller. By using ARM, KnowID ensures that explicit primary and foreign keys are hidden from the user's view so queries appear more naturally to modellers, and also can be expressed more compactly. This is very relevant for example for query explanation [16] as well as query formulation and understanding [14]. See [9] for a more detailed comparison of both architectures.

The remainder of this system description paper first briefly recaps the key points of the KnowID architecture (Section 2), to then expand on the system design and implementation (Section 3). System testing and a use case are touched upon in Section 4. We discuss and close in Section 5.

2 Theoretical background

Our approach to combining *intensional knowledge* ($\mathcal{K}$) with *large amounts of data* ($\mathcal{D}$) is depicted in Fig. 1. It consists in a pipeline starting with a conceptual data model or application ontology $\mathcal{K}$ represented as an EER diagram. After the steps of formalisation and materialisation of deductions, this EER diagram is used to obtain an abstract relational model schema in ARM, which is the basis for i) the relational database schema for $\mathcal{D}$, ii) the data completion of $\mathcal{D}$ with the intensional knowledge, iii) the reformulation of SQLP queries into SQL queries to be evaluated over the completed data. The pipeline has the following three key components:

– Knowledge, at the type/class-level (i.e., with entities that can be instantiated), which is represented formally in a logic. This type-level theory should be independent of systems design aspects (so, e.g., no PK/FK and OO object identifiers and similar in the model);
– Structured data, stored in a relational database or an RDF triple store (not unstructured text documents).
– Automated reasoning support with, as a minimum, querying the data using the vocabulary elements from the knowledge layer. It should avail of the formally represented knowledge in some way (so, not just a graphical query interface for the stored data like in query-by-diagram).



```
relation Academic ( (self OID, anum INT, aname STRING,
        office STRING, school OID),
  primary key (self),
  constraint sch foreign key (school)
        references school,
  pathfd (anum) -> self,
  disjoint with (module, school, teach) )
```

Fig. 2 Top: Graphical renderings of a sample ARM schema, resembling a relational model graphical notation (left) and a knowledge graph notation (right). Bottom: textual representation of the `Academic` relation from the ARM schema

Table 1 Comparison between OBDA and KnowID on the main distinguishing features of computational cost.

Feature	**OBDA**	**KnowID**
World	OWA+CWA	CWA
Language for $\mathcal{K}$	OWL 2 QL	relational, DL
Language for $\mathcal{D}$	relational or RDF	relational
Query language	SPARQL + SQL (fragment)	SQLP
Automated reasoning	yes	yes
Reasoning w.r.t. data	query rewriting	data completion
Mapping layer	yes	no
Transformations	no	yes
Entity recasting	yes	no
Syntactic sugar	available	possible

EER is a conceptual data modelling language [17] which is able to express in a graphical notation advanced requirements for information systems, and has been widely applied in database design as well as in other more abstract modelling tasks. Even though other popular languages exist that could also be used in the knowledge layer of KnowID, EER was chosen because of the well known transformation between EER and the relational model (which is similar enough to the ARM to provide a solid ground), and also because it is widely used in the database community. There are also several logic-based reconstructions of EER, which therewith facilitates automated reasoning over the diagrams.

The Abstract Relational Model (ARM) is a generalisation of the relational model (RM) which includes an abstract domain of entities `OID` used for object identifiers, and an associated new `self` attribute for each relation in an ARM. This `self` with an `OID` has an underlying theory of referring expressions such that the identifiers in ARM remain virtual and thus does not cost one another column in a database table. Its details are described in [1,14,18]. This seemingly simple extension allows for various other constructs beyond primary and foreign keys, such as declaring disjointness constraints, explicit inheritance, and path functional dependencies. In Fig. 2 we show a graphical representation of a conceptual model in ARM, and its textual representation.

ARM generalises RM in two ways: first, there is no explicit choice in the representation of primary keys for each relation, one simply may assume it exists; second, a set of declarative constraints is available to model the domain data. In this sense, ARM is, strictly, conceptual model-*like*. ARM can be formalised in a Description Logic (DL) with n-ary relations ($n \geq 1$), e.g., the PTIME decidable $\mathcal{CFDI}_{nc}^{\forall -}$ [19] In that case, the problem of deciding when for some ARM model Σ, constraint φ (e.g., a disjointness constraint), and relation T_i (e.g., `Academic`, in Fig. 2), $\Sigma \models (\varphi \in T_i)$ holds in that ARM schema can be reduced to reasoning about logical consequence [1]. It has a complementary query language, called SQLP [1], which is an extension of full SQL in order to allow queries over paths, availing over those virtual identifiers in the background, and has been shown to improve querying when used with ARM [14].

Automated reasoning is a fundamental gear for this architecture, in both the knowledge and the data layers. First, it is needed to classify model elements and materialize deductions in the third and forth step at the knowledge and information management layer in Fig. 3. This involves reasoning in the logic chosen by the previous formalization step. The optional first step of transformation of OWL files into EER diagrams may also involve OWL reasoning. Furthermore, automated reasoning is used in the data completion process at the data management level, availing of the formally represented knowledge and data.

The system described in the next section completes this approach by providing a bi-directional connection between the ontology layer and ARM schemata. A comparison with OBDA is included in Table 1. From this follows what would be the best choice when; e.g., for queries beyond UCQs or evolving schemas, KnowID will perform better, whereas with rapidly changing data and static schemas, OBDA may outperform KnowID.

3 System design and implementation

The system design process overall took an approach of: (i) specifying a list of requirements; (ii) exploring the solution space with 21 3rd-year students in seven teams as their BSc 'capstone project'; (iii) devising the final architecture and refining and updating the best components of the capstone implementations. The second step was included, as it would permit a faster exploration of a range of options. While the transformation rules obviously remain the same, 'externalities' to meet the requirements and turn it into a usable tool varied greatly among the proposed and implemented solutions. Aside from different programming languages (Java, Python, React, JavaScript, and combinations thereof), the main differences in the solutions were regarding interaction with front ends for the EER and ARM models (text or diagrams, self-made vs. borrowed from draw.io), more or less restrictive storage of models (a structured plain text file vs. JSON or XML with or without XSD and a tailor-made parser), and the internals of the algorithms how to achieve the implementation of the rules more or less efficiently. The system that is presented here, took the system of one of the teams as starting point[1], which we updated and extended, both regarding some intricate details of the rules and corner cases and by being informed by the other capstone systems and prospects for extensibility to other tasks.

Requirements The hard requirements were to implement the rules as in the KnowID paper [9], both the EER-to-ARM and ARM-to-EER transformations, to be able to open and save those models, report on success or failure of a transformation, and have some user interface for these actions and in/output. Ideally, the prospective tool would also report on those elements or constraints that could not be transformed, and it would be helpful if the tool were to report on what happened with each element (since this will be useful for processing SQLP queries). For our version, the log file and graphical rendering was deemed a hard requirement. The students, meanwhile, were free to choose between a textual or graphical representation of the models, including that it was permissible to extend a current open source EER tool to do this provided it supports the required language features.

Architecture and implementation Since the link between the knowledge and data layer is the first implementation component of KnowID, and considering the various possible extensions, the design was made such that

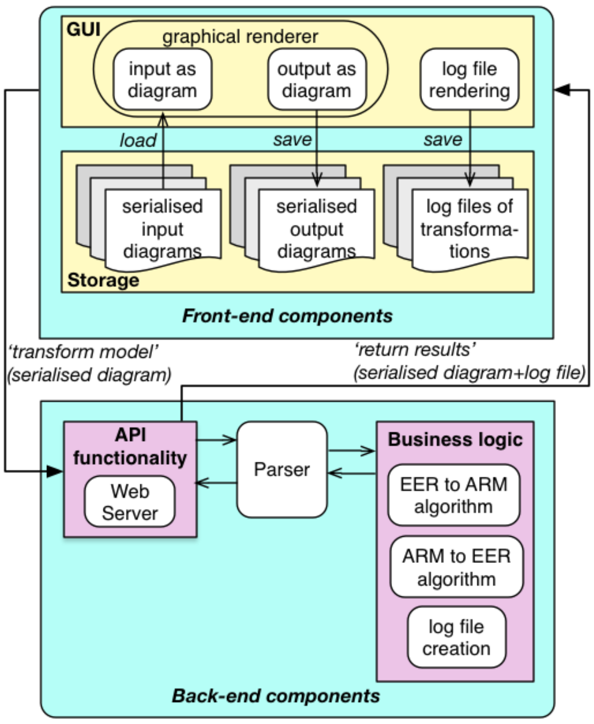


Fig. 3 Architecture to realise the transformations between the knowledge and data layer.

it is highly modular. The architecture of the system is shown in Figure 3. It is divided into a front-end for user interaction and model management and a back-end for the business logic for the transformations.

The front-end renders diagrammatically the EER and ARM models in the GUI and stores them as JSON files. This is realised in JavaScript with React. Three functions are available to the user: load, transform, and save a model. The GUI displays the input model and the generated output side-by-side, with the log file at the lower part of the screen.

The back-end functionality commences with the Web Server, which functions as an API to the business logic. At present, it takes the file received from the front-end component and passes it through the JSON Parser, converting it into an object representation in memory. It then determines whether it is an EER or an ARM model and feeds it into the applicable transformation algorithm in the business logic model. The bird's eye view of the transformation algorithms are shown in Algorithms 1 and 2, including differences for strong entity types that have their own identifier and weak ones that have a partial identifier. The output of the algorithm is a model of the other type along with a log file. These are passed through the JSON Parser and the resultant file returned to the web server, which forwards it to the front-end to render it, and if the user so wishes, to save it. The back-end was coded in Python, availing of, in

[1] The team members kindly permitted us to use and revise their code (by default they are the copyright holders).

particular, the Enum and JSON libraries, and Flask (in the Web Server module).

Algorithm 1 EER to ARM algorithm at a glance

```
 1: $\Sigma$: ARM schema; $R$: set of relations in $\Sigma$; $\Omega$: an ERD;
    $S$: regular or strong entity type; $W$: weak entity type; $P$:
    set of all pathfd constraints.
Precondition: The input model is valid EER.
 2: function GenerateARM( )
 3:     Group ETs from $\Omega$                ▷ strong $S$ and weak $W$
 4:     for each $s \in S$ do
 5:         create a relation $r \in R$
 6:         fire the other relevant transformation rules
 7:             ▷ add self, pathfd, PK, FK, other attributes
 8:     end for
 9:     for each $w \in W$ do
10:         create a relation $r \in R$
11:         fire the other relevant transformation rules
12:         ▷ add self, pathfd, PK with FK, other attributes
13:     end for
14:     Set all attributes' datatype to anyType
15:     $P \leftarrow$ List all pathfd constraints from each $r \in R$
16:     Compute disjointness and covering from $P$
17:     Add disjointness or covering constraints to each $r \in R$
18:     return $\Sigma$
19: end function
```

Algorithm 2 ARM to EER algorithm at a glance

```
 1: $\Sigma$: ARM schema; $R$: set of relations in $\Sigma$; $\Omega$: an ERD;
    $S$: regular or strong entity type; $W$: weak entity type; $M$
    m:n relationship; $P$: set of all pathfd constraints.
Precondition: The input model is valid ARM.
 2: function GenerateEER( )
 3:     Classify each $r \in R$  ▷ strong, regular, weak, m:n, isa
 4:     for each $r \in R$ that is strong, regular, isa do
 5:         create an Entity Type $s \in S$
 6:         fire the other relevant transformation rules
 7:                 ▷ identifiers, attributes, remove datatypes
 8:     end for
 9:     for each $r \in R$ that is weak do
10:         create an Entity Type $w \in W$
11:         fire the relevant transformation rules
12:                 ▷ partial identifier, add relationship, etc.
13:     end for
14:     for each $r \in R$ that is m:n do
15:         create a Relationship $m \in M$
16:         fire the relevant transformation rules
17:     end for
18:     Create relationships and isa between the entity types
19:     Add cardinality constraints
20:     return $\Omega$
21: end function
```

Flexibility, maintainability, and extensibility The business logic, i.e., the transformation algorithms, use generic data structures (arrays) so that it can be independent of transient technologies. For instance, JSON is gaining popularity for representing knowledge graphs over serialisations in RDF or XML, noting that it is possible to convert between JSON and RDF [5,6]. Thanks to the modular design, another parser could be added for the input and the output, which will not affect the actual transformation algorithms. For instance, the array of objects could be written into XML, passed back to the Web Server, and then loaded into a preferred modelling tool, such as draw.io, ERwin, NORMA etc. that serialise in XML.

Further extensions can be added easily to the current architecture, such as a separate back-end component for the automated reasoning over the conceptual model before the transformations, and a module for query formulation with SQLP over either the EER diagram or ARM model.

System testing A formal unit test suite was constructed for the business logic, i.e., the core KnowID transformation algorithms from EER to ARM and ARM to EER. This was used in two stages: the first stage concerned robustness testing to verify all components worked and the second stage included tests with plausible models that had actual names for entities. Each test case comprises a pair of JSON files encoding an EER (or ARM) input model and a expected ARM (or EER) output model. They covered each rule, corner case, and intended failure amounting to 83 tests. This started with the base case of a single entity type, progressing to two entity types and a binary relationship first with the base cases of cardinality constraints (1:n, 1:1, n:1, m:n) and mandatory or optional participation (i.e., universal or existential quantification, when formalised), to subsequently start adding attributes with correct handling of datatypes, testing n-aries, weak entity types with partial identifiers, and subsumption followed by disjointness and completeness on subsumption. This then moved to the more involved aspects, such as multiple or conflicting `pathdf` constraints to `self`, cardinality constraints unsupported in ARM, recursive relations, and composite attributes. The outputted diagrams and the log files were inspected manually; when a transformation was incorrect or a log entry incomplete, revisions were made to the code and the model transformation tested again. All user test cases are available in the online supplementary material, which contains a `notes.txt` in both EER-to-ARM and the ARM-to-EER folders that describe for each test case what it is testing, a corresponding `_Test.py` file for automating test execution, the respective JSON files of the input models, and the output as screenshots, so that it also can be verified without installing the tool.

4 Example

We present the transformation tool's working by demonstrating the generation of an ARM model from an EER Diagram, where the end state of the transformation is shown in Fig. 4.

Let us assume a small application ontology or conceptual data model about academics working for a department that offers modules that are taught by those academics. This may be stored in OWL, serialised in XML, or a similar notation, and that can be converted into JSON. For instance, there may be declarations such as (in Description Logics notation):

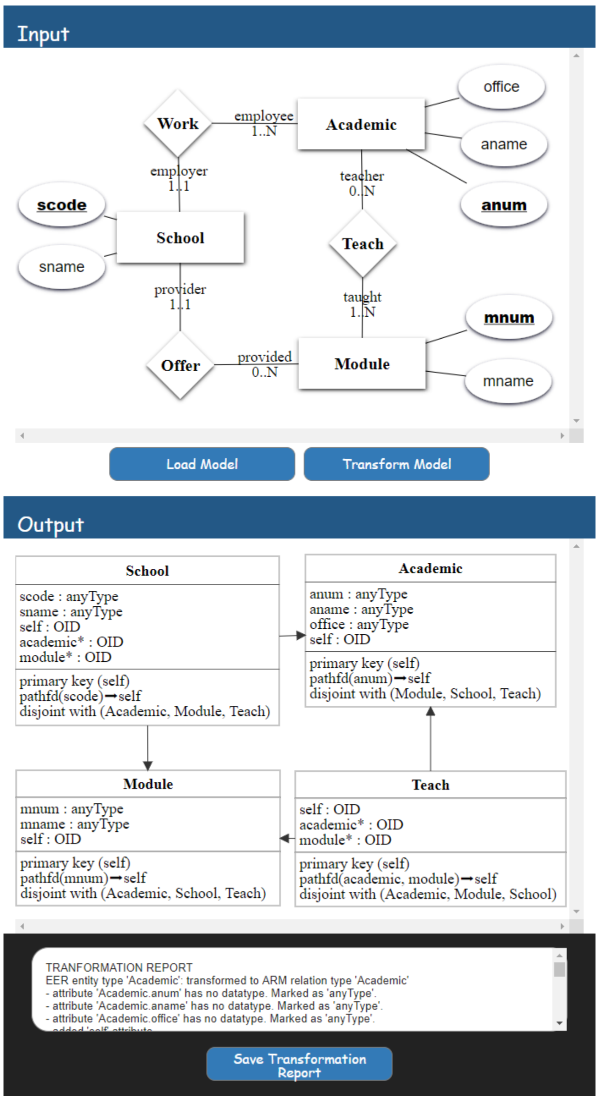
$\mathsf{School} \sqsubseteq \exists \mathsf{employs.Academic}$
$\mathsf{Academic} \sqsubseteq = 1\, \mathsf{worksFor.School}$
$\mathsf{Module} \sqsubseteq = 1\, \mathsf{offeredBy.School}$
...

This is stored eventually in the corresponding JSON file. For instance, the entity type Person is stored as follows (we have removed whitespace to save layout):

```
{ "entityTypes":
    [   {  "name": "Person",
            "attributes": [  {"name": "name"},
                {"name": "ZA_ID"} ],
            "identity": { "attributes": [
                    "ZA_ID" ],
                "type": "complete" } },
        ... ] ... }
```

In the tool's interface, one clicks Load to load that JSON file, which is rendered graphically in a simple ERD notation, shown in the left pane in Fig. 4. Upon clicking Transform, it will transform the model into ARM, which generates a log file that can be saved (bottom pane in Fig. 4) to a JSON file and it generates a text-based ARM model with relation specifications as shown in compact mode in Fig. 2 (saved in JSON format), which is then rendered graphically in the right-hand pane of the interface (see Fig. 4). The choice of ARM graphical elements was chosen here to be relational model-like. As can be seen, default datatypes (anytype) are added, since they are not part of an ERD but are part of the ARM specification, and there are auto-generated names for ARM relations created from $m : n$ relationships, such as joinRelation [Academic-Module], and the arrows point to the source relation of the foreign key as usual.

The ARM file thus generated lies now in the data layer of the KnowID architecture (Fig. 1). With this file it is now possible not only to start the relational database design if necessary, but also to realise the data completion step that materialises the conceptual model's implicit knowledge. Physical data characteristics such as DBMS specific definitions, referential integrity, and performance optimisation may now be integrated in the full process of knowledge management.



TRANFORMATION REPORT
EER entity type 'Academic': transformed to ARM relation type 'Academic'
- attribute 'Academic.anum' has no datatype. Marked as 'anyType'.
- attribute 'Academic.aname' has no datatype. Marked as 'anyType'.
- attribute 'Academic.office' has no datatype. Marked as 'anyType'.

Fig. 4 Screenshots of the tool with the example, in the state after having transformed the ERD input into an ARM model.

5 Discussion and conclusions

This system description paper presented the novel implementation to connect the knowledge layer with the data layer using transformations, as an alternative option to the currently common intermediate mapping layer. Its design is highly modular in order to facilitate easy addition of other modules with additional functionalities and to be compatible with a number of Web technologies for the representation of the models and for reasoning over them.

The next phase in the implementation will be extending the use of SQLP from ARM to EER so as to realise the application ontology-based conceptual queries, which is at least theoretically feasible thanks to the transformations. Also, it may be linked to, e.g., *crowd* [2] for online editing of the conceptual models and automated reasoning over them (roughly: steps 1-3 in Figure 1), which has its UML models stored in JSON format, which would practically facilitate the theoretical sufficient interchangeability [7,8].

Acknowledgements We would like to thank all the capstone students for their efforts, and in particular J. Du Plessis, St J. Grimbly, and G. Stein for their design and code. SJ and CMK acknowledge the partial support by the NRF of South Africa (grant number 115376).

References

1. Borgida, A., Toman, D., Weddell, G.E.: On referring expressions in information systems derived from conceptual modelling. In: Proc. of ER'16, *LNCS*, vol. 9974, pp. 183–197. Springer (2016)
2. Braun, G., Estevez, E., Fillottrani, P.: A reference architecture for ontology engineering web environments. J. Comp. Sci. & Tech. **19**(1), 22–31 (2019)
3. Calvanese, D., Cogrel, B., Komla-Ebri, S., Kontchakov, R., Lanti, D., Rezk, M., Rodriguez-Muro, M., Xiao, G.: Ontop: Answering SPARQL queries over relational databases. Semantic Web Journal **8**(3), 471–487 (2017)
4. Calvanese, D., De Giacomo, G., Lembo, D., Lenzerini, M., Rosati, R.: Eql-lite: Effective first-order query processing in description logics. In: M.M. Veloso (ed.) IJCAI 2007, Proceedings of the 20th International Joint Conference on Artificial Intelligence, Hyderabad, India, January 6-12, 2007, pp. 274–279 (2007)
5. Davis, I., Steiner, T., Hors, A.J.L.: RDF 1.1 JSON Alternate Serialization (RDF/JSON) (2013). URL `http://www.w3.org/TR/rdf-json/`. 07 November 2013
6. Dimou, A., Sande, M.V.: RDF mapping language (RML). Unofficial draft, Ghent University (2014). URL `http://rml.io/spec.html`. 17 September 2014
7. Fillottrani, P.R., Keet, C.M.: Conceptual model interoperability: a metamodel-driven approach. In: A. Bikakis, et al. (eds.) Proc. of RuleML'14, *LNCS*, vol. 8620, pp. 52–66. Springer (2014)
8. Fillottrani, P.R., Keet, C.M.: Evidence-based languages for conceptual data modelling profiles. In: T. Morzy, et al. (eds.) Proc. of ADBIS'15, *LNCS*, vol. 9282, pp. 215–229. Springer (2015)
9. Fillottrani, P.R., Keet, C.M.: KnowID: An architecture for efficient knowledge-driven information and data access. Data Intelligence p. (in print) (2020)
10. Gottlob, G., Kikot, S., Kontchakov, R., Podolskii, V.V., Schwentick, T., Zakharyaschev, M.: The price of query rewriting in ontology-based data access. Artif. Intell. **213**, 42–59 (2014)
11. Kontchakov, R., Lutz, C., Toman, D., Wolter, F., Zakharyaschev, M.: The combined approach to query answering in dl-lite. In: F. Lin, U. Sattler, M. Truszczynski (eds.) Principles of Knowledge Representation and Reasoning: Proceedings of the Twelfth International Conference, KR 2010, Toronto, Ontario, Canada, May 9-13, 2010. AAAI Press (2010)
12. Krötzsch, M., Rudolph, S.: Is your database system a semantic web reasoner? Künstl Intell **30**, 169–176 (2016)
13. Lubyte, L., Tessaris, S.: Automated extraction of ontologies wrapping relational data sources. In: Proc of DEXA'09, pp. 128–142. Springer (2009)
14. Ma, W., Keet, C.M., Oldford, W., Toman, D., Weddell, G.: The utility of the abstract relational model and attribute paths in SQL. In: C. Faron Zucker, et al. (eds.) Proc. of EKAW'18, pp. 195–211. Springer (2018)
15. Noy, N., Gao, Y., Jain, A., Narayanan, A., Patterson, A., Taylor, J.: Industry-scale knowledge graphs: Lessons and challenges. Queue **17**(2), 20:48–20:75 (2019)
16. Roy, S., Suciu, D.: A formal approach to finding explanations for database queries. In: C.E. Dyreson, F. Li, M.T. Özsu (eds.) International Conference on Management of Data, SIGMOD 2014, Snowbird, UT, USA, June 22-27, 2014, pp. 1579–1590. ACM (2014). DOI 10.1145/2588555.2588578. URL `https://doi.org/10.1145/2588555.2588578`
17. Thalheim, B.: Extended entity relationship model. In: L. Liu, M.T. Özsu (eds.) Encyclopedia of Database Systems, vol. 1, pp. 1083–1091. Springer (2009)
18. Toman, D., Weddell, G.E.: Fundamentals of Physical Design and Query Compilation. Synthesis Lectures on Data Management. Morgan & Claypool Publishers (2011)
19. Toman, D., Weddell, G.E.: On adding inverse features to the description logic $\mathit{CFD}^{\forall}{}_{\mathrm{nc}}$. In: Proc. of PRICAI'14, pp. 587–599 (2014)
20. Xiao, G., Ding, L., Cogrel, B., Calvanese, D.: Virtual knowledge graphs: An overview of systems and use cases. Data Intelligence **1**, 201–223 (2019)