MEDINFO 2019: Health and Wellbeing e-Networks for All
L. Ohno-Machado and B. Séroussi (Eds.)

doi:10.3233/SHTI190197

Query Translation Between AQL and CQL

Georg Fette[a,b], Mathias Kaspar[b], Leon Liman[a], Maximilian Ertl[b], Jonathan Krebs[a], Georg Dietrich[a], Stefan Störk[b], Frank Puppe[a]

[a] *Chair of Computer Science 6, University of Würzburg, Würzburg, Germany,*
[b] *Comprehensive Heart Failure Center, University Hospital of Würzburg, Würzburg, Germany*

Abstract

Secondary use of electronic health records using data aggregation systems (DAS) with standardized access interfaces (e.g. openEHR, i2b2, FHIR) have become an attractive approach to support clinical research. In order to increase the volume of underlying patient data, multiple DASs at different institutions can be connected to research networks. Two obstacles to connect a DAS to such a network are the syntactical differences between the involved DAS query interfaces and differences in the data models the DASs operate on. The current work presents an approach to tackle both problems by translating queries from a DAS using openEHR's query language AQL (Archetype Query Language) into queries using the query language CQL (Clinical Quality Language) and vice versa. For the subset of queries which are expressible in both query languages the presented approach is well feasible.

Keywords:

Electronic Health Records, Data Warehousing, Information Systems

Introduction

The secondary use of electronic health records (EHR) has become an important field in medical informatics [1]. Routine clinical data is reused for various scientific purposes, like prospective estimation of study cohort sizes or support of study cohort acquisition. When this routine data is scattered in various data sinks in various heterogeneous data formats, it is difficult to access. In order to improve access to the EHR, solutions have been developed to aggregate the routine data and to make it accessible via a standardized interface. Two paradigms are possible when constructing such a data aggregation system (DAS): The first method is to leave the data in its original place and to provide a standardized access by aggregating the requested data chunks on the fly using appropriate extraction transformation load (ETL) pipelines. SMART on FHIR [2] projects for example realize this paradigm by aggregating requested FHIR resources on the fly. This approach works well for patient centered access (i.e. queries operating on the data of a single patient). However, for population centered access (i.e. queries operating on data of all patients) this approach lacks indices covering the involved data sources. Without such indices, query speed performance does not scale related to the amount of queried patient data. On large volumes of EHR data this would heavily afflict the usability of FHIR related query languages like FHIR-REST-API[1] or CQL (Clinical Quality Language)[2]. The second method to provide access to aggregated routine data is to provide an additional data sink with a generic data model, into which the heterogeneous routine data is persistently transferred via an ETL pipeline. The data sink is supplied with appropriate indices, so it can be efficiently queried via a standardized query language. Examples of this architecture are i2b2 [3], openEHR [4] or OMOP [5]. HAPI[3] and VONK[4] are data sink servers that do not only use FHIR as an access interface in patient centered mode, but they also use it as the data model in which the EHR data is stored, so it can be queried in population centered mode using query languages like FHIR-REST-API or CQL.

Clinical Research Networks

In order to increase the volume of underlying patient data for larger and thus more expressive query results, DASs at different institutions can be connected to clinical research networks. Examples for such networks are PCORnet, the OHDSI research network or EHR4CR. In a DAS network a query is distributed to connected nodes, where it is independently evaluated. Each node's results are returned and combined to an aggregated result. Systems like SHRINE [6] for i2b2 or SNOW [7] for openEHR perform the query distribution and result aggregation automatically. Each network, however, only allows DASs having the same query language to be part of the network. If a DAS with a different query language is intended to be integrated into the network, the data from that DAS has to be transferred (like in [8]) into a new dedicated DAS supporting the networks query language. However, parallel support of multiple DASs containing the same redundant data at one institution creates an overhead in support and hardware.

An alternative approach is to translate queries of an incompatible DAS into the query language required by the DAS network. The current work examines the feasibility of translating queries formulated in the query language CQL into AQL (Archetype Query Language, the query language of openEHR) and vice versa. To accomplish this task, the query is first translated into an intermediate query graph model, then potentially necessary graph transformation are applied on the query graph and, finally, the graph is translated into the desired target query language.

Methods

Query Languages

Before going into detail about the translation process, the two languages shall be briefly introduced:

CQL is a functional query language. A CQL script is defined as a so called *library* in which, besides the actual queries, also

[1] https://www.hl7.org/fhir/search.html
[2] https://cql.hl7.org
[3] http://hapifhir.io
[4] https://fire.ly/vonk/vonk-fhir-server

meta information related to the queries can be defined. CQL is independent of a concrete data model, as the model to be used is explicitly defined in each CQL library. The CQL data model elements can be composed of the following data types: primitives (Booleans, Strings, numbers and timestamps), clinical codes, quantities, intervals, lists and structured types. Structured data objects, which can again contain other structured data objects, can be accessed with a path syntax (e.g. *patient.contact[0].name.family*). Besides the data model, a CQL library can define its search context, which is either *Patient* or *Population* centered. The main functional part in a CQL library are the *statements*. Each statement can be seen as an individual query. Usually a statement defines which data sources the query is operating on (e.g. *[Patient] A* or *[Observation] B*), how the elements of the queried sources are constrained (e.g. *B.valueString = 'x'*), how the various data sources have to be related to each other (e.g. *[Patient] A with [Observation] B such that B.subject = A*) and which elements of potential matches have to be returned as results (e.g. *Return Tuple {id:A.identifier, B.valueString}*). CQL provides a rich repertoire of operators (e.g. comparison, logical, arithmetic, list access, aggregation) to perform calculations on the data model.

OpenEHR's query language AQL is SQL-inspired and as well a functional language. It operates on a data model that is defined by the openEHR data modeling language ADL. OpenEHR's root data model elements are called *archetypes*. An archetype can be composed of the following data element types: primitives (Booleans, Strings, numbers and timestamps), other archetypes and generic container structures (e.g. lists). Structured data objects are accessed via a path syntax. An AQL query mainly consists of three parts: The *FROM* part defines which archetypes are queried and how these archetypes are related to each other concerning meronymity relations (e.g. *EHR A contains Observation B*). The *WHERE* part constrains the archetype elements (e.g. *B/value > 1*). The *SELECT* part defines which elements have to be returned in the results (e.g. *select A/ehr_id, B/value*). Compared to CQL, AQL provides a smaller set of operators (comparison, logical, *matches*, *exists*). For the sake of brevity archetype names in the following chapters are shortened (e.g. openEHR-EHR-OBSERVATION.LabResult.v1 → Observation[LabResult]).

Currently, the CQL translation has the following constraints: 1. The CQL queries may only contain a single statement and 2. The CQL queries are defined in the context *Population*. If a CQL query with context *Patient* has to be translated to AQL, the given patient identifier has to be integrated into the CQL statement as a constrained *Patient.identifier*.

Queries as Graphs

Queries can be seen as graphs. In order to perform the query translation, the CQL/AQL (depending on the translation direction) query to be translated is transformed into an intermediate graph model. The graph is subsequently translated into the desired target query language. The graph model is depicted in Figure 1. A graph contains a set of data model elements that are connected via relations. Although the type of these relations is not specifically defined, they could be interpreted as meronymity (i.e. *contains*) relations. Additionally, a graph contains a set of operators, which contain as parameters either other operators, data model elements or literals. A similar same approach has already been applied in [9] where AQL was translated into the query language of i2b2.

For parsing AQL queries the parser from the AQL-processor of the EtherCIS project[5] was taken and combined with a graph builder written by the authors. The parser for CQL libraries was taken from the cql-2-elm project[6] and combined with a graph builder also written by the authors. The graphs retain 1. the structure of data model elements mentioned in the query, 2. the constraints and operators on data element values and 3. which data elements have to be contained in the returned results. Whenever elements in a query are named with an alias, all references to that alias create *IsRelatedTo/ HasParameter* relations to the graph node identified with that alias.

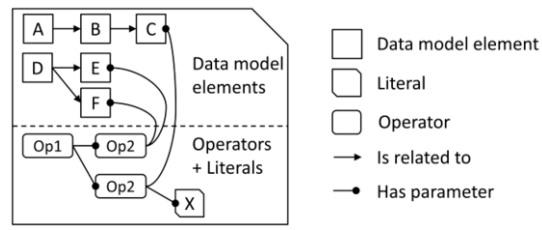

Figure 1 – Model of graphs, which are used as intermediate data model during query translation.

Graph Transformations

Because of properties of CQL/AQL that do not exist in the respective other language or non-matching properties of the incorporated data models, the graph has to be appropriately transformed, in order to fit all required properties of the target language. Currently there exist the following types of transformations: Meronymity-Equality-Transformations and their reverse, Path-Transformations, Concept-Code-Mappings, Operator-Mappings, Resolve-Quantity-Transformations, Resolve-Interval-Transformations (see Figure 2).

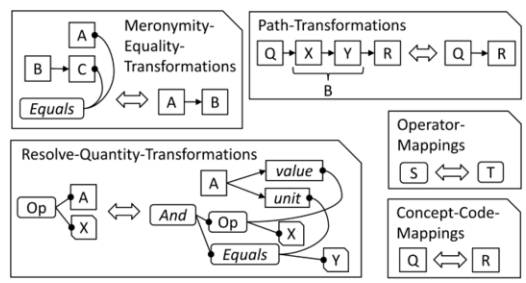

Figure 2 – Query graph transformations

Meronymity-Equality-Transformations

Meronymity-Equality-Transformations are needed because in CQL the connection between two root data model elements is represented by an equality constraint between one root element and a child element of the other root element (e.g. *[Patient] A with [Observation] B such that B.subject = A*). The definition of both involved element types (*[Observation]* and *[Patient]*) determine the type for the element *B.subject*, which would otherwise not be specified. In contrast, in AQL the main mechanism to define structural relationships between data model elements are meronymity relations, which also define the type of the contained elements (e.g. *Ehr A contains Observation[LabResult] B*). Meronymity-Equality-Transformations are parameter free. The transformation looks for any three data model elements *A*, *B* and *C*, with a relation between *B* and *C* and an *Equals* operator containing *A* and *C* as parameters. The element *C* and the *Equals* operator are deleted and the element *A* gets connected to the element *B*. The reverse Equality-Meronymity-Transformation requires a parametrization because only a defined set of related elements *A* and *B* should be transformed and the name of the newly

[5] http://ethercis.org

[6] https://github.com/cqframework/clinical_quality_language

created element *C* has to be given. For all directly connected elements *A* and *B* the connection gets removed and instead a new element *C* gets created and connected to *B*. Furthermore, a new *Equals* operator gets created receiving *A* and *C* as parameters.

Path-Transformations

Path-Transformations provide the possibility to shorten or lengthen data model element paths. They are needed because elements from the source data model, which have to be mapped to their semantically identical counterparts in the target data model, might be encapsulated in additional element wrappers. These wrappers have to be removed if they do not exist in the target data model. E.g. the intermediate element *valueQuantity* in *Observation.valueQuantity.value* (FHIR) has no counterpart in *Observation[LabResult]/value* (openEHR). The transformation can be configured to either shorten or lengthen a path. In order to shorten a path, the transformation searches for nodes *Q* and *R* connected by a list of nodes *B* and removes the intermediate nodes. The reverse transformation introduces new intermediate nodes *B* into the graph between two directly connected nodes *Q* and *R*.

Concept-Code-Mappings

Concept-Code-Mappings map element identifiers of the source data model to element identifiers of the target data model by renaming nodes.

Operator-Mappings

Operator-Mappings map operators of the source query language to operators of the target query language by renaming nodes.

Resolve-Quantity-Transformations

AQL does not contain quantities as build in types. Therefore, operators on quantity types have to be exchanged by *and*-connected clauses that check the requested value as well as the requested quantity type (e.g. *days, cm*) (e.g. *[Encounter] A where A.length > 120 days → [Encounter] A where A.length.value > 120 and A.length.unit = 'days'*). The transformation looks for data model elements *A* contained as a parameter in an operation *Op* (with potential further parameters). The node *A* get connected to two newly created data model elements *value* and *unit* and gets removed from the parameters of *Op*. Additionally, a new *equals* operator is created which gets the new *unit* node and a newly created literal *Y* as parameters. The value of *Y* has to be given as a transformation configuration and defines the unit of the quantity to be computed. The new *equals* node and the *Op* node get connected in a newly created *and.*

Resolve-Interval-Transformations

Similar to quantity types, AQL does not contain intervals as build in types. Operators on interval types have to be exchanged by *and*-connected clauses, that check the requested constraints (e.g. *[Patient] A where A.birthDate in A.contact[0].address.period → [Patient] A where A.birthDate > A. contact[0].address.period.low and A.birthDate < A. contact[0].address.period.high*).

Graphs to Queries

After all necessary graph transformations have been executed, the graph can be translated into the desired target query language via respective Graph2QueryString writers implemented by the authors.

The proposed method was tested on manually designed AQL/CQL queries and on queries contained in the AQL/CQL documentation. A query was (when translatable) translated into its respective counterpart and re-translated into its original language. The original query was compared for semantic equality to its translated counterpart as well as to its re-translation into the original query language.

Results

Translations Constraints

The proposed methodology using the currently available set of graph transformations allows the translation of all CQL/AQL queries having the following properties: A query has to contain the data type *Patient* (*EHR* in AQL) as a query source. The query may contain an arbitrary amount of additional data sources of arbitrary type. A query may return an arbitrary subset of query sources or data model elements that are reached from the data sources via paths. The paths can have arbitrary length. Data model elements can be constrained using the logical comparators listed below, parametrized with literals or with other data model elements. A query may contain the data types *Boolean*, *String*, *Integer*, *Decimal*, *Timestamp*, *Date*, *DateTime*, *Quantity* and *Interval*. Data model elements or literals can be processed with operators from the set of mutual operators listed below.

Translations are only possible for queries containing exclusively operators which exist in both query languages with the same semantics and the same interface. These operators (using their CQL-displaynames) grouped into operator categories are:

- Boolean operators *{ and, or, not }*
- comparator operators on numeric/date/timestamp types *{ =, <, <=, >, >=, != }*
- comparator operators on Strings *{ =, !=, matches }*
- operators on lists/iterators *{ in, exists, with, without }*.

All other operators either have to be substituted by appropriate graph transformations or render a query untranslatable.

CQL does not contain an equivalent to AQL's *contains* operator. This operator, which performs a type matching on a given to be contained data type, returns the matched child data element. The operator is substituted by Meronyminity-Equality-Transformations.

The mutual set of operators is reduced by the following limitations of the two languages, which could not be substituted by graph transformations:

AQL contains no arithmetic operators (e.g. *+, -, *, /*). AQL comprises a limited set of list operators (e.g. *first, last, sort, count* do not exist). It is only possible to access specific list elements via a given index or via the *matches* operator. The *contains* operator can only be used to constrain the data model structure and not to check the containment of a single element in a list (e.g. *[ProcedureRequest] A where A.notes contains 'x'*).

AQL allows no aliased valueset definitions. All valuesets have to be directly given as parameters to the operator using them, instead of having the possibility to reference a previously defined valueset with an alias. CQL, on the other hand, allows only valueset aliases based on valueset ids but no definition of valuesets by listing their contained codes. As the current implementations is restricted to CQL statements instead of CQL libraries, the usage of valuesets or therein contained codes still has to be resolved.

Table 1 shows a selection of example queries that have been automatically translated given proper graph transformation configurations.

Table 1 – Automatically translated example queries

CQL	AQL
[Patient] A where A.active = true and A.gender = 'male'	select e from EHR e where e/active = true and e/gender = 'male'
[Patient] A where exists(A.name B where B.given = 'John')	select e from EHR e contains composition a[HumanName] where a/given = 'John'
[Patient] A with [Encounter] B such that not(exists(B.discharge)) and B.subject = A	select e from EHR e contains composition a[Encounter] where not exists a/content[Discharge]
[Patient] A with [Observation] B such that B.code = 'Calcium' and B.valueQuantity > 10'mg' and B.subject = A	select e from ehr e contains observation a[LabResult] where a/code = 'Calcium' and a/valueQuantity/value > 10 and a/valueQuantity/unit = 'mg'

Figure 3 pictures an example of an CQL query being transformed into AQL. The query represents a request for patients having at least one *Calcium* laboratory measurement with a value of more than 10. The CQL query is parsed into the uppermost graph depicted in Figure 3. Successively, Meronymity-Equality-Transformations, Path-Transformations and Concept-Code-Mappings are applied to the graph (parametrized with the configurations from Table 1, CQL → AQL). The Meronymity-Equality-Transformation links the *Patient* node with the *Observation* node, which is the required representation in AQL. The Path-Transformation removes the *valueQuantity* node, because in the openEHR data model the value element is a direct child element of the *LabResult* archetype. The Concept-Code-Mappings rename the nodes *Patient* and *Observation* as *Ehr* and *Observation[LabResult]*, as those are the semantically equivalent data elements in the openEHR data model. Finally, the transformed graph is translated into an AQL query String.

Table 2 – Manually defined configurations used in the example in Figure 3. The column headings reference the nodes from Figure 2.

CQL → AQL		
Path-Shorten-Transformations		
Q	**B**	**R**
Observation[LabResult]	{valueQuantity}	value
Concept-Code-Mappings		
Q	**R**	
Ehr	Patient	
Observation[LabResult]	Observation	
AQL → CQL		
Equality-Meronymity-Transformations		
A	**B**	**C**
Patient	Observation	subject
Path-Lengthen-Transformations		
Q	**B**	**R**
Observation[LabResult]	{valueQuantity}	value
Concept-Code-Mappings		
Q	**R**	
Ehr Observation[LabResult]	Patient Observation	

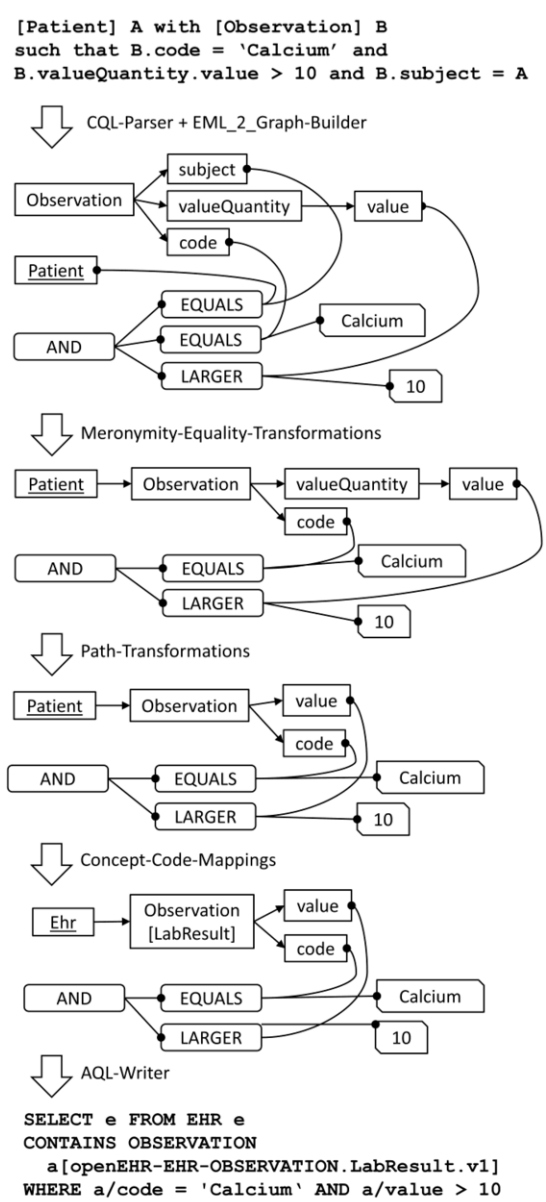

Figure 3 – Translation of an CQL query into AQL. Underlined nodes denote elements that have to be included in the result

The translation from AQL to CQL goes accordingly with the transformation configurations from Table 1, AQL → CQL. It is noted that the order of the execution of the different transformation types remains the same, so the re-translation is not according to the reverse direction in Figure 3, which has to be considered in the transformation configurations. The AQL query is first parsed into a graph. Subsequently the Equality-Meronymity-Transformation creates the child node *subject*, connects it to the *Observation[LabResult]* node, removes the relation between *Ehr* and *Observation[LabResult]* and adds a new *Equals* operator with *subject* and *Ehr* as parameters. The Path-Transformation adds the node *valueQuantity* between *Observation[LabResult]* and *value*. The Concept-Code-Mappings rename the nodes *Ehr* and *Observation[LabResult]* as *Patient* and *Observation*. Finally, the graph is translated into a CQL query String.

Discussion

The approach to translate a query into another query to improve or facilitate its evaluation is known as query rewriting [10]. The early work on query rewriting is related to view-based relational database querying and deals with the translation of queries formulated in the same language for both, source and target. In recent years query rewriting was as well used in conjunction with graph data query system, e.g. [11] resembles the presented approach, whereas there XQuery was translated into SPARQL.

The discussed graph model and the graph transformations have been specified and implemented by the authors. Both elements could be exchanged by a standard from the graph computing community like e.g. Owl to model graphs and graph transformation frameworks like GROOVE to model the transformations.

Although the used graph model is similar to the parser graphs produced by each of the systems, it was more convenient for the authors to use a separate model in order to be independent of possible system specific modeling paradigms.

The concept-mappings and path-transformations could be substituted by a mechanism using FHIR ConceptMaps, which encode the mappings from elements of one data model to equivalent data model elements in a target system.

When applying the presented approach on productive data models, the configuration of the various transformations could become cumbersome when the data models are large. It would be beneficial if some configurations could be automatically deduced by an analysis of the source and the target data model. When both data models are annotated with terminology codes, the configuration of concept-mappings could be deduced automatically by identifying equivalent data model elements annotated with the same codes.

An issue that could appear when applying query translations in combination with concrete query engines system implementations could be that the translations are semantically equivalent, but they can contain differences in their syntactical structure. E.g. the two CQL queries *[Patient] A with [Encounter] B such that B.patient = A* and *[Patient] A where exists [Encounter] B where B.patient = A* are semantically equivalent, but a query execution engine could handle the execution of a *with-such-that* expression differently than a *where-exists* with a nested sub query.

An aspect not yet covered by the presented approach is the translation of *valueset*, *concept*, *code* and *codesystem* definitions. Due to the omnipresence of valueset references in the query examples of both query languages, the presented approach could have difficulties to be applied in productive systems without this issue being solved.

A further aspect not tackled in the presented approach is how query results are returned by the respective systems. Both systems have their own method and syntactical encoding for delivering results. For that topic, proper adapters would have to be developed as well. The example from Figure 3 for example returns complete patient objects, which are serialized differently by the respective systems. In order to prevent differences in serializations, the return types of translatable queries could be restricted to primitive types (e.g. patient ids instead of complete patient objects).

As the presented approach is still work in progress, it can be extended whenever the query language specifications are changed or extended and thus set of mutual operators grows. E.g. the AQL specification already contains announcements of future language elements like arithmetic operators, an extended *matches* operator or alias definitions (i.e. *let*).

Conclusions

An approach was presented to translate Clinical Quality Language (CQL) queries into Archetype Query Language (AQL) queries and vice versa. Several examples were shown to illustrate the capabilities of the presented apprach and for one example the translation process was illustrated in detail. As CQL and AQL do not comprise the same sets of operations, the translation capabilities of the presented approach are restricted to a subgroup of possible queries, which have to be composed of a set of common operators of both query languages. Despite this limitation, queries expressible in both languages can be automatically translated, which would allow both query systems to be transparently included in a distributed research network of the other type.

Acknowledgements

An implementation of the presented approach in Java is available at https://gitlab2.informatik.uni-wuerzburg.de /gef18bg/cdw_querymapper. The authors thank B. Haarbrandt, A. Zautke and G. Vella for helpful comments. This research was funded by grant of German Federal Ministry of Education and Research (Comprehensive Heart Failure Center Würzburg, grants #01EO1004 and #01EO1504).

References

[1] Prokosch H, Ganslandt T, Perspectives for medical informatics. Reusing the electronic medical record for clinical research, *Methods Inf Med* **48(1)** (2009), 38-44.

[2] Mandel JC, Kreda DA, Mandl KD, et al, SMART on FHIR: a standards-based, interoperable apps platform for electronic health records, *J Am Med Inform Assoc* **23(5)** (2016), 899-908.

[3] Murphy S, Weber G, Mendis M et al, Serving the Enterprise and beyond with Informatics for Integrating Biology and the Bedside (i2b2), *J Am Med Inform Assoc* **17(2)** (2010), 124-30.

[4] Kalra D, Beale T, Heard S, The openEHR Foundation. *Stud Health Technol Inform* **115** (2005), 153-73.

[5] Overhage JM, Ryan PB, Reich CG et al, Validation of a common data model for active safety surveillance research, *J Am Med Inform Assoc* **19(1)** (2012), 54-60.

[6] Weber GM, Murphy SN, McMurry AJ, et al, The Shared Health Research Information Network (SHRINE): a prototype federated query tool for clinical data repositories, *J Am Med Inform Assoc* **16(5)** (2009), 624-30.

[7] Hailemichael MA, Marco-Ruiz L, Bellika JG, Privacy-preserving Statistical Query and Processing on Distributed OpenEHR Data, *Stud Health Technol Inform* **210** (2015), 766-70.

[8] Haarbrandt B, Tute E, Marschollek M, Automated population of an i2b2 clinical data warehouse from an openEHR-based data repository, *J Biomed Inform*, **63** (2016), 277-94.

[9] Fette G, Kaspar M, Liman L, et al. Query Translation between openEHR and i2b2. (in press) *Stud Health Technol Inform* (2019)

[10] Calvanese D, De Giacomo G, Lenzerini M, et al. What is query rewriting?, *Proc. of KRDB* (2000), 17–27.

[11] Droop M, Flarer M, Groppe J, et al. Embedding Xpath Queries into SPARQL Queries, *ICEIS* (2008).

Address for Correspondence

Georg Fette, georg.fette@uni-wuerzburg.de