Using a Generic Modular Mapping Framework for Simulation Model Composition

Philipp Helle and Wladimir Schamai
Airbus Group Innovations
Hamburg, Germany
Email: {philipp.helle,wladimir.schamai}@airbus.com

***Abstract*—This paper presents a new framework for solving different kinds of mapping problems, the Generic Modular Mapping Framework (GEMMA). It is geared towards high flexibility for dealing with a large number of different challenges. To this end it has an open architecture that allows the inclusion of application-specific code and provides a generic rule-based mapping engine that allows users without programming knowledge to define their own mapping rules. The paper provides a detailed description of the concepts inherent in the framework and its current architecture. Additionally, the evaluation of the framework in two different application cases, simulation model composition and testbench setup, is described.**

***Keywords*–Mapping; Framework; Simulation Model Composition.**

I. Introduction

Recently, several of our research challenges could be reduced to a common core question: How can we match data from one or more data sources to other data from the same and/or different data sources in a flexible and efficient manner? A search for an existing tool that satisfied our application requirements did not yield any results. This sparked the idea of a new common generic framework for data mapping. The goal in designing this framework was to create an extensible and user-configurable tool that would allow a user to define the rules for mapping data without the necessity for programming knowledge and that yet still has the possibility to include application-specific code to adapt to the needs of a concrete application.

The results of our efforts so far and a first evaluation based on our existing research challenges are presented in this paper.

This paper is structured as follows: Section II provides information regarding related. Section III provides a detailed description of the framework, its core concepts and its architecture. Next, Section IV describes the application cases that have been used to drive and evaluate the framework so far. Finally, Section V concludes the paper.

II. Related Work

The related work divided into two major categories: on the one hand, record linkage and data deduplication tools and frameworks, and on the other hand semantic matching frameworks for ontologies.

Record linkage as established by Dunn in his seminal paper [1] and formalised by Felligi and Sunter [2] deals with the challenge to identify data points that correspond with each other in large data sets. Typically, this involves databases of different origin and the question, which of the data on one side essentially are the same on the other side even if their name does not match precisely. The same approach is also called data deduplication [3] where the goal is to identify and remove redundancies in separate data sets. An overview of existing tools and frameworks can be found in [4]. The research work in that area focuses on efficient algorithms for approximate and fuzzy string matching since the size of the data sets involved often leads to an explosion of the run times. These tools [5] often include phonetic similarity metrics or analysis based on common typing errors, i.e., analysis based on the language of the input data. Furthermore, they concentrate on the matching of the string identifiers whereas our framework is more open and flexible in that regard and also includes the possibility to base the matching on available semantic meta-information. The goal in record linkage is always finding data points in different sets representing the same real-world object. Our framework was developed with the goal to match data from different sources that is somehow related but not necessarily referencing the same object.

Semantic matching is a type of ontology matching technique that relies on semantic information encoded in ontologies to identify nodes that are semantically related [6]. They are mostly developed and used in the context of the semantic web [7], where the challenge is to import data from different heterogeneous sources into a common data model. The biggest restriction to their application is that these tools and frameworks rely on the availability of meta-information in the form of ontologies, i.e., formal representations of concepts within a domain and the relationships between those concepts. While our framework can include semantic information, as shown in Section IV-A, it is not a fixed prerequisite.

In conclusion, we can say that our framework tries to fit into a middle ground between record linkage and semantic matching. We use methods applied in both areas but we leave the user the flexibility to choose, which of the features are actually needed in a mapping project.

III. Generic Modular Mapping framework

The Generic Modular Mapping Framework (GEMMA) is designed to be a flexible multi-purpose tool for any problem that requires matching data points to each other. The following subsections will introduce the artefacts that make up the core idea behind GEMMA, describe the kind of mapping rules that can be implemented, show the generic process for the usage of GEMMA and describe the software architecture and the current GEMMA implementation.

A. Artefacts

GEMMA is centred around a set of core concepts that are depicted by Figure 1. In an effort to increase the flexibility of GEMMA, the core concepts have been defined in an abstract fashion.

The following artefacts are used:

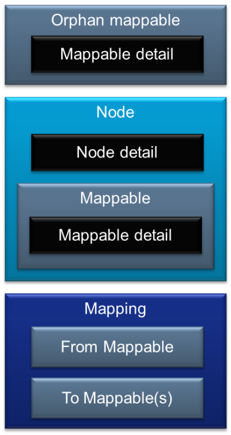

Figure 1. Overview of relevant artefacts

- **Node** - Something that has properties that can be mapped to some other properties.
- **Mappable** - Something that can be mapped to some other thing according to specified mapping rules. Orphan mappables are mappables whose owning node is not known or not relevant to the problem.
- **Mapping** - The result of the application of mapping rules, i.e., a relation between one FROM mappable and one or more TO mappables. Note that the semantic interpretation of a mapping highly depends on the application scenario.
- **Mapping rule** - A function that specifies how mappings are created, i.e., how one mappable can be related to other mappables.
- **Mappable or node detail** - Additional attribute of a mappable or a node in the form of a {detail name:detail value} pair. Details are optional and can be defined in the context of a specific application scenario.

To illustrate these abstract definitions, Figure 2 provides a simple example, where real-world objects depicted on the left hand side are represented on the right hand side in the form of our GEMMA concepts.

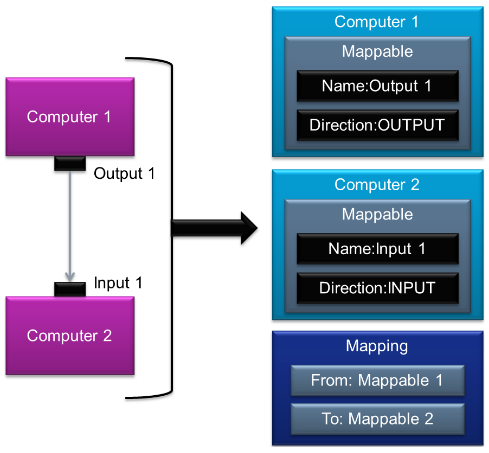

Figure 2. Simple example

In this context, the abstract concept definitions provided above are interpreted as follows:

- **Node** - A computer with input and output ports
- **Mappable** - An input or output port of a computer
- **Mapping** - The connection between ports
- **Mapping rule** - Output ports must be connected to input ports according to some specified criteria such as having the same port name or the same data type.
- **Mappable detail** - Every port has a detail called direction, which defines if the port is an input of output port of the computer

B. Mapping rules

One goal of GEMMA is to allow a large degree of freedom regarding the definition of the mapping rules, so that the framework can be used flexibly for very different kind of application scenarios. So far, the following kinds of mapping rules have been identified and are supported by GEMMA:

- **Name matching**, e.g., map a mappable to other mappables with the exact same name.
- **Fuzzy name matching**, or other forms of **approximate string matching** [8], e.g., map a mappable to other mappables with a similar name (similarity can be based on the Levenshtein distance (LD) [9], i.e., ”map” can be matched to ”mop” if we allow an LD of 1).
- **Details**, e.g., map a mappable with value of detail X=x to other mappables with values of details Y=y and Z=z or more concretely, map a mappable with detail direction=”output” to mappables with detail direction=”input”.
- **Structured rewriting** of search term based on name, details and additional data, e.g., construct a new string based on the properties of a mappable and some given string parts and do a name matching with the new string (e.g., new string = ”ABCD::” + $mappable.detail(DIRECTION) + ”::TBD::” + $mappable.detail(LOCATION) would lead to a search for other mappables with the name ”ABCD::Input::TBD::Front”).
- **Semantic annotations** such as user-predefined potential mappings (bindings) using mediators as described in [10], e.g., map a mappable whose name is listed as a client of a mediator to all mappables whose name is listed as a provider of the same mediator.

And, of course, any combination of the above mentioned kinds of rules can be used. For example, structured rewriting could also be applied on the target mappables, which would in effect mean defining aliases for every mappable in the mappable database in the context of a rule.

Additionally, it is planned that several rules can be selected and prioritized for a mapping configuration with options for defining their application, e.g., only if the rule with the highest priority does not find any matches then rules with a lower priority are evaluated.

C. Process

The process for the usage of GEMMA is generic for all kinds of applications scenarios and consists of five steps:

1) Import
2) Pre-processing
3) Matching
4) Post-processing
5) Export

The mapping process is configured using an XML configuration file that defines which parsers, rules, resolvers and exporters (see III-D for a detailed explanation of the terms) will be used in the mapping project. The open character of GEMMA allows implementing different data parsers for importing data, resolvers for post-processing of the mappings and data exporters for exporting data.

Import loads data into the framework. GEMMA provides the interfaces *DataParser*, *MappableSource* and *NodeSource* to anyone who has the need to define a new data parser for an application-specific configuration of GEMMA. All available parsers are registered in an internal parser registry where the Run Configuration can instantiate, configure and run those parsers, which are required by the configuration file. The data will then be stored in the mappable database.

Pre-processing of data involves selection of mappables that will require matching using whitelists and/or blacklists and structured rewriting of, e.g., mappable names based on mappable details as already discussed in Section III-B. Pre-processing will be user-defined in a set of rules in a file that can be edited with a standard text editor and does not require programming knowledge. The set of rules which should be applied in one mapping project will be defined by the configuration.

Matching involves running queries on the mappable database to find suitable matches for each mappable that is selected for mapping. The queries are derived from the mapping rules. A mapping is a one to (potentially) many relation between one mappable and all the matches that were found.

Post-processing or match resolving is an optional step that is highly driven by the specific application as will be shown in Section IV. It potentially requires the interaction with the user to make a selection, e.g., a mapping rule might say that for a mappable only a one-to-one mapping is acceptable but if more than one match was found then the user must decide, which of the found matches should be selected. Post-processing also allows the user to apply the graphical user interface to review and validate the generated mapping results and thereby to check the completeness and correctness of the defined rules and to manipulate mappings manually, e.g., remove a mappable from a mapping if the match was not correct.

Export is also highly application-specific. Exporting involves transformation of the internal data model into an application-specific output file. Similar to the *DataParser* interface, a generic *MappingExporter* interface allows the definition of custom exporters that are registered in a exporter registry where they can be accessed by the run configuration as dictated by the configuration file.

D. Architecture and implementation

As already stated before, the Generic Mapping Framework is designed as a flexible answer to all sorts of mapping problems. This is represented in the architecture of the framework, which is depicted in Figure 3 in a simplified fashion. GEMMA modules can be categorized either as core or as application-specific. Core components are common for all GEMMA usage scenarios whereas the application specific components have to be developed to implement features that are very specific to achieve a certain goal. For example, data parsers are application-specific as applications might need data from different sources whereas the mappable database and query engine is a core component that is shared. Table I provides a brief description of the most important modules in GEMMA and their categorization.

TABLE I. GENERIC MAPPING FRAMEWORK MODULES

Module	Description	Core	Specific
Data Parser	Reads data (nodes and/or mappables) into the internal data model and feeds the mappable database		x
Generic Mapper	Generates mappings between mappables based on rules	x	
GUI	Interface for loading configuration, displaying mappings as well as allowing user-decisions and displaying of data based on resolver	x	
Issue Logger	Logs mapping related issues	x	
Mappable Database	Stores mappables and allows searches	x	
Resolver	Resolves mappings based on application specific semantics		x
Rule Manager	Reads mapping rules and provides rules information to other components	x	
Run Configuration	Holds the configuration that defines which parsers, exporters and rules are used in the current mapping project	x	
Specific Exporter	Exports the internal data model into a specific file format		x
Standard Exporter	Exports the internal data model into an XML file	x	

The implementation of GEMMA was done in Java. As far as possible, open source libraries and frameworks have been used. The choice for the mappable database, for example, fell on Apache Lucene [11]. Lucene is is a high-performance, full-featured text search engine library. The choice of Lucene might seem odd because we are not using it for its originally intended purpose, indexing and searching of large text files, but it offers a lot of the search capabilities like fuzzy name matching that we need and is already in a very stable state with a strong record of industrial applications.

IV. EVALUATION

The evaluation so far has been done using two application cases, simulation model composition and testbench setup.

A. Simulation model composition

The description of the application case simulation model composition requires the introduction of the bindings concept as presented in [10]. The purpose of bindings is to capture the minimum set of information required to support model composition by an automated binding or connecting mechanism. For example, for the outputs of a given component, we wish to identify the appropriate inputs of another component to establish a connection.

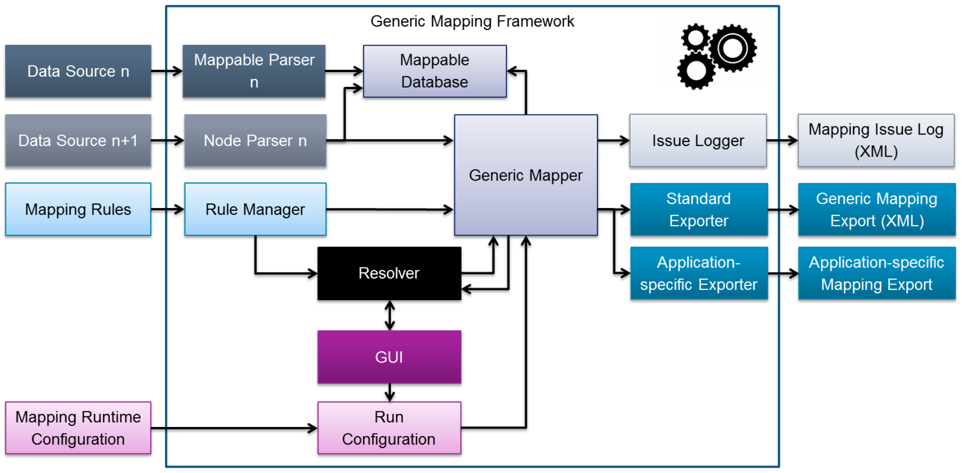

Figure 3. Generic mapping framework architecture

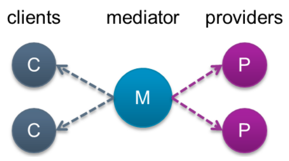

Figure 4. Bindings concept

To this end [10] introduces the notions of clients and providers. Clients require certain data; providers can provide the required data. However, clients and providers do not know each other a priori. Moreover, there may be multiple clients that require the same information. On the other hand, data from several providers may be needed in order to compute data required by one client. This results in a many-to-many relation between clients and providers. In order to associate the clients and the providers to each other the mediator concept is introduced, which is an entity that can relate a number of clients to a number of providers, as illustrated in Figure 4. References to clients and providers are stored in mediators in order to avoid the need for modifying client or provider models.

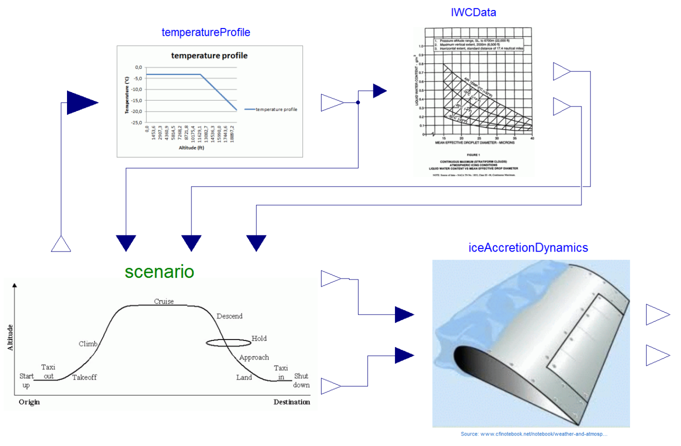

Figure 5. Assembled ice-accretion simulation based on [12]

After the bindings concept was introduced we can now turn to the description of the application case. Generally speaking, the application case is the automatic creation of connections between different model components in a model. Typically in modelling tools, to create a connection between one port of one component to another port of another component requires the user to draw each connection as one line from one port to the other port. If the components' interfaces or the model structure change, then all of the connections have to be checked and some of them have to be redrawn. If we consider a large set of models that have to be changed frequently or if we want to create the models dynamically, then the effort for creating and maintaining the connections between the components in the models becomes a serious issue. The goal of our application case is the formalization of the often implicit rules which the user applies to create the connections to automate this process.

Consider the model depicted by Figure 5, which is a part of the model from the public aerospace use case in the CRYSTAL project [12]. It consists of component models such as flight scenario profile, ice accretion dynamics, and tables for temperature or liquid water content. All of the component models must be interconnected. For example, the temperature profile component requires the current aircraft altitude, which is provided by the flight scenario component; the ice accretions dynamics component requires the current aircraft speed, which is also provided by the scenario component, etc. The individual models were built using the Modelica tool Dymola and exported as Functional Mockup Units [13] (FMUs) in order to be integrated, i.e., instantiated and connected, in a co-simulation environment.

However, assume that the models were created without this specific context in mind. They neither have agreed interfaces, nor do the name and type of the component elements to be connected necessarily match. In order to be able to find the counter parts, i.e., to know that the input of the ice accretion instance should be connected to the appropriate output of the scenario model instance, a dedicated XML file captures some additional information as shown in Figure 6. This way we can capture such interrelations without modifying the models. This data is used as follows: whenever the model "IceAccretionDynamics" is instantiated, bind its input port "aspeed" to the output ”port p_v”, which belongs to the instance of type ”ScenarioMissionProfile1”.

Whenever there will be another model that requires the same data, i.e., current aircraft speed, an additional client entry is added to the same mediator shown in Figure 6. Similarly, whenever there is another model that outputs this data, its corresponding element is referenced in a new provider entry. This approach in particular pays off as soon as there are several models that require or provide the same data. Their connection is then resolved whenever they are instantiated in a specific context model such as the one depicted in Figure 5.

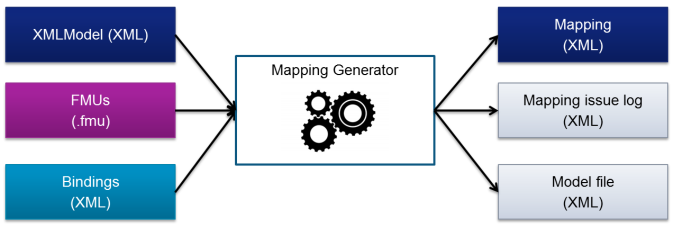

Figure 7. Mapping generator for simulation model composition

In our setting, the bindings specification XML file and the model XML file are application specific sources that are inputs to our generic mapping framework as depicted by Figure 7. The information read from these sources by the application specific parsers is put into the core module mappable database. Two rules, provided by Figures 9 and 10 in pseudocode, are used to query the mappable database to find suitable matches for each mappable. The matching results are then given to the resolver module. This module is aware of the bindings concept and is able to resolve chains of matches and generate a binding for each client and, if necessary, involve the user when an unambiguous mapping is not possible automatically.

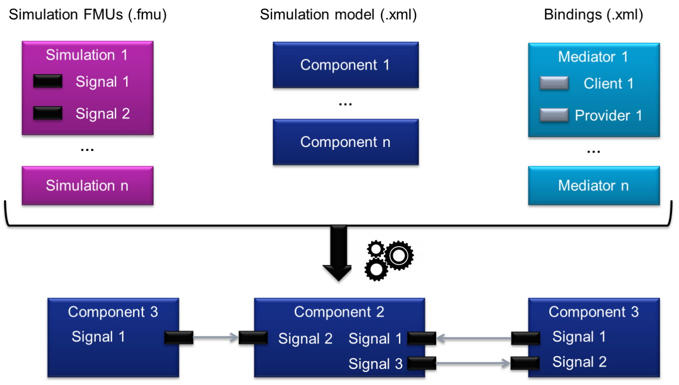

Figure 8. Input and output artefacts of simulation model composition mapper

In the end, the mapping framework uses a list of FMUs, a description of the simulation model consisting of instances of classes implemented in the FMUs and a description of the bindings in the form of an XML file. The output is then the complete simulation model with all the connections between the simulation instances as sketched by Figure 8.

B. Testbench setup

The testbench setup application case is driven by the needs of test engineers. They are given a hardware System under Test (SuT), a formal definition of the interfaces of the SuT and other equipment and a description of the specified logic of the SuT, which should be tested. Unfortunately, the formal interface definition has been finalized after the specification of the logic, which means that the signal names in the logic description and the signal names in the formal interface definition, which has been implemented in the SuT, do not match. Today, a significant amount of manual effort is required to discover the correct formal signal name for every logical signal. To ease this, GEMMA has been configured as shown by Figure 11.

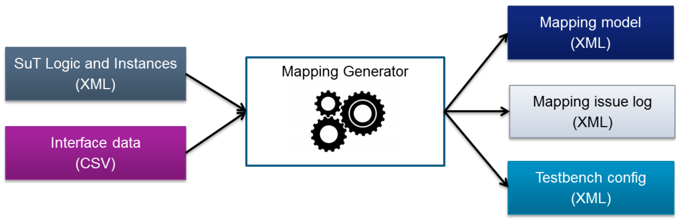

Figure 11. Mapping generator for testbench setup

The goal of the application case is to find a mapping between the name of a signal used in the description of the SuT logic and the corresponding formal interface signal name as shown by Figure 12.

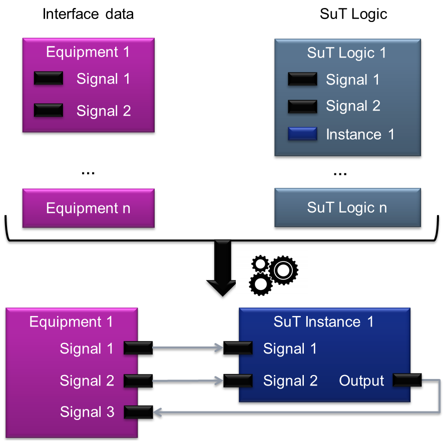

Figure 12. Input and output artefacts of testbench setup mapper

Since the names of the signals could be quite different, the testbench setup application case required the use of the structured rewriting rule type (see Section III-B). One of the rules for the testbench setup is depicted by Figure 13 in pseudo code. The rule defines a new local variable called *soughtName* whose content depends on some attributes of the mappable (enclosed in $$) and instead of searching for other mappables that have the same or a similar name as the original mappable, GEMMA searches now for mappables whose name is equal to the variable *soughtName*. If a mappable has the attributes *direction*, *type*, *BLOCKID* and *ID* with the respective values OUTPUT, SuT_Type1, 45 and 67 then *soughtName* would take the value *AB_BLOCK45_STATUS_67* and GEMMA will search for and map to another mappable in the database with that name.

The main challenge for this application case was the amount of data. Even for a small SuT, the mappable database

```
<mediator name="Current aircraft speed" requiredType="Real">
    <client mandatory="true" qualifiedName="IceAccretionDynamics.aspeed"/>
    <provider qualifiedName="ScenarioMissionProfile1.p_v"/>
</mediator>
```

Figure 6. Example of a binding specification entry in XML format

```
foreach Node:node from source BINDINGS
 foreach Mappable:mappable
  if mappable.detail(CAUSALITY) == INPUT || mappable.detail(CAUSALITY) == PARAMETER
   Search Mappable:otherMappables
    with otherMappables.detail(SOURCE) == BINDINGS
     and otherMappables.detail(CAUSALITY) == CLIENT
     and otherMappables.detail(NAME) == mappable.detail(QUALIFIED_NAME)
    Create Mapping from mapping to otherMappables
  endif
 endfor
endfor
```

Figure 9. First implemented rule for simulation model composition (in pseudo code)

```
foreach Node:node from source BINDINGS
 foreach Mappable:mappable
  if mappable.detail(TYPE) == PROVIDER
   Search Mappable:otherMappables
    with otherMappables.detail(SOURCE) == XMLMODEL
     and otherMappables.detail(CAUSALITY) == OUTPUT
     and otherMappables.detail(QUALIFIED_NAME) == mappable.detail(NAME)
    Create Mapping from mapping to otherMappables
  endif
 endfor
endfor
```

Figure 10. Second implemented rule for simulation model composition (in pseudo code)

```
foreach SuTLogicInstance:instance
 foreach Mappable:mappable
  if mappable.detail(DIRECTION) == INPUT
   Search Mappable:otherMappables
     with mappable.detail(NAME) == otherMappable.detail(NAME)
   Create mapping from mappable to otherMappables
   elseif mappable.detail(direction) == OUTPUT
    if instance.detail(type) == SuT_Type1
     soughtName = "AB_BLOCK$$mappable.detail(BLOCKID)$$_STATUS_$$mappable.detail(ID)$$"
    elseif instance.detail(type) == SuT_Type2
     soughtName = "BLOCK$$mappable.detail(BLOCKID)$$_STATUS_$$mappable.detail(ID)$$_CD"
    endif
    Search Mappable:otherMappables
     with mappable.detail(NAME) == soughtName
    Create mapping from mappable to otherMappables
  endif
 endfor
endfor
```

Figure 13. One implemented rule for testbench setup (in pseudo code)

contained 350000 mappables and matches had to be found for 2500 mappables. Nevertheless, the application proved to be successful. The total run time is around 30 seconds including the time for data import and export, and the average time per query is 4.5 ms on a standard PC.

V. Conclusion

In this paper, we introduce a new framework for generic mapping problems, GEMMA. It is geared towards high flexibility for dealing with a number of very different challenges. To this end it has an open architecture that allows the inclusion of application-specific code for reading and exporting data and the resolving of mapping results. Furthermore, it provides a generic rule-based mapping engine that allows users without programming knowledge to define their own mapping rules. So far, the evaluation in the two application cases described in this paper has been highly successful.

As said in Section II, as far as we know, there is currently no other tool with the same functionality as GEMMA. This prevents a direct comparison in terms of performance of GEMMA with other solutions. For our future work we also plan to compare GEMMA functionally to other solutions that rely on more formalized semantic information in the form of ontologies. Depending on the results of this comparison, this might lead to an extension of GEMMA, so that in addition to the matching based on the Lucene text database there will be the possibility to include the results from a semantic reasoner in the matching process.

From an implementation point of view, there is still a large to-do list:

- Definition of required mapping rules capabilities and format for representing mapping rules.
- Implementation of generic resolver module for post-processing.
- Implementation of generic rule-based mapper.
- Transform project into Eclipse product and use the Eclipse OSGi extension mechanism [14] for registering and instantiating modules.

At the same time, we are actively looking for further application cases to mature the framework.

REFERENCES

[1] H. L. Dunn, "Record linkage*," American Journal of Public Health and the Nations Health, vol. 36, no. 12, 1946, pp. 1412–1416.

[2] I. P. Fellegi and A. B. Sunter, "A theory for record linkage," Journal of the American Statistical Association, vol. 64, no. 328, 1969, pp. 1183–1210.

[3] Q. He, Z. Li, and X. Zhang, "Data deduplication techniques," in Future Information Technology and Management Engineering (FITME), 2010 International Conference on, vol. 1. IEEE, 2010, pp. 430–433.

[4] H. Koepcke and E. Rahm, "Frameworks for entity matching: A comparison," Data & Knowledge Engineering, vol. 69, no. 2, 2010, pp. 197 – 210.

[5] A. K. Elmagarmid, P. G. Ipeirotis, and V. S. Verykios, "Duplicate record detection: A survey," Knowledge and Data Engineering, IEEE Transactions on, vol. 19, no. 1, 2007, pp. 1–16.

[6] F. Giunchiglia, A. Autayeu, and J. Pane, "S-match: An open source framework for matching lightweight ontologies," Semantic Web, vol. 3, no. 3, 2012, pp. 307–317.

[7] Y. Hooi, M. Hassan, and A. Shariff, "A survey on ontology mapping techniques," in Advances in Computer Science and its Applications, ser. Lecture Notes in Electrical Engineering, H. Y. Jeong, M. S. Obaidat, N. Y. Yen, and J. J. J. H. Park, Eds. Springer Berlin Heidelberg, 2014, vol. 279, pp. 829–836.

[8] P. A. Hall and G. R. Dowling, "Approximate string matching," ACM computing surveys (CSUR), vol. 12, no. 4, 1980, pp. 381–402.

[9] V. I. Levenshtein, "Binary codes capable of correcting deletions, insertions, and reversals," in Soviet physics doklady, vol. 10, no. 8, 1966, pp. 707–710.

[10] W. Schamai, P. Fritzson, C. J. Paredis, and P. Helle, "ModelicaML value bindings for automated model composition," in Proceedings of the 2012 Symposium on Theory of Modeling and Simulation-DEVS Integrative M&S Symposium. Society for Computer Simulation International, 2012, p. 31.

[11] M. McCandless, E. Hatcher, and O. Gospodnetic, Lucene in Action. Manning Publications Co., 2010.

[12] A. Mitschke et al., "CRYSTAL public aerospace use case Development Report - V2," ARTEMIS EU CRYSTAL project, Tech. Rep. D208.902, 2015.

[13] T. Blochwitz et al., "The functional mockup interface for tool independent exchange of simulation models," in 8th International Modelica Conference, Dresden, 2011, pp. 20–22.

[14] R. Hall, K. Pauls, S. McCulloch, and D. Savage, OSGi in action: Creating modular applications in Java. Manning Publications Co., 2011.