Goals and Challenges of an Intelligent Discovery Service for Architecture Repositories

Michael Gerz, Francesco Bacchelli
Fraunhofer FKIE
Information Technology for Command & Control (ITF)
Fraunhoferstraße 20
53343 Wachtberg-Werthhoven
GERMANY

michael.gerz@fkie.fraunhofer.de
francesco.bacchelli@fkie.fraunhofer.de

ABSTRACT

Complex organisations such as the national armed forces run many studies, projects, and programmes that produce an increasing number of architecture descriptions as formal models. In order to avoid redundant efforts, it is important to make all models visible and accessible to interested parties.

A central architecture repository, in which all architecture models are stored, is a first step to publish existing products. However, the repository should be complemented by an intelligent discovery service that allows users to run a query for specific details in the architecture models stored in the repository. For illustration, a user might want to retrieve all information exchange relationships on the brigade level or to get an overview of the combat net radios deployed in national vehicles. Since the search results may come from many different models, the outcome must be presented in a way that allows the user to identify potentially relevant architecture models for further consultation.

In this paper, we present the motivation behind an intelligent discovery service for formal architecture models and describe classes of queries that users may want to run. We show that an architecture repository can deliver better results than when you query all architecture models individually. Thereafter, we give an overview of the challenges that you have to face when implementing such a discovery service. We also discuss technologies that can be adopted to implement such a service and to improve the quality of the query results. Finally, a system architecture for a discovery service is proposed.

1.0 INTRODUCTION

For any given enterprise, there is not just a single coherent architecture model that describes the enterprise in its entirety. Instead, there is a variety of different architectures ranging in the level of detail, the timeframe for validity etc. The NATO Architecture Framework (NAF), for instance, considers this by its NATO All View 1 (NAV-1), in which you have to define the scope and constraints of the architecture.

Complex organisations such as the national armed forces run many studies, projects, and programmes that produce a continuously growing number of architecture descriptions as formal models. In order to avoid redundant efforts in such a decentralized organisation, it is important to make all models visible and accessible to interested parties.

A first step is to set up a central architecture repository, in which all architecture models are stored. To facilitate initial retrieval capabilities, architecture models could be tagged with structured metadata that are extracted from the NAV-1. However, with this simplistic approach, it is still up to the user to identify and analyse potentially relevant architectures based on a limited set of metadata and textual descriptions.

That is why the repository should be complemented by an intelligent discovery service that allows users to run a query for specific details in the architecture models stored in the repository. Unlike Internet search services, such a discovery service does not operate on free-text but on formal models. Therefore, the output can also be presented in a structured manner, for instance, in tabular format, in a tree format or, better yet, as a graph representation. Since the search results may come from many different models, the outcome must be presented in a way that allows the user to identify potentially relevant architecture models for further consultation. For each result, the discovery service should also keep track of what led the service to provide it, making the whole process more understandable to the user. This way, the user can play an active role in his/her search by adjusting the search terms.

2.0 ARCHITECTURES

IEEE 1471 states that an architecture is “the fundamental organization of a system embodied in its components, their relationships to each other and to the environment, and the principles guiding its design and evolution”. In [1], it is argued that, as a consequence of that definition, an architecture and an architectural description are not the same thing, since the first one is a concept of a system (in a wide sense) while the second one is an artefact embodying some fundamental fact about the first one.

In this paper, for simplicity sake, we use the word “architecture” to describe both and depending on the context it will be clear which one is being referred to. The definition from IEEE 1471 also tells us what we are supposed to find in those architectures (architectural descriptions, of course): the components of a system and their relations. However, a significant distinction within those “components” is the one between *classes* and *instances*. Therefore, for the rest of this paper, we will consider that an architecture contains (at its lower level) classes, instances and relations.

The crucial point with regard to our goal is how *classes*, *instances* and *relations* will be represented in the descriptions. Since many years, for instance, UML has been a widespread standard for architectural descriptions but we cannot expect all architectures we want to deal with to be described in UML. In addition to that, the recent flourishing of architectural frameworks (such as MODAF and DoDAF) and other similar initiatives (like the Unified Profile for DoDAF and MODAF (UPDM) or the MODAF Ontological Data Exchange Mechanism (MODEM)) propose different views on architectural descriptions. The NATO Architecture Framework alone provides 50 different views (and sub-views) covering, among others, operational and technical aspects of an architecture. Accordingly, there are many types of information that a discovery service user may be interested in. Last but not least, all these standards provide support to the architects, not really restraining their expressive freedom. Therefore, on top of all these different standards, there is a user-related (or community-related) expressivity that must be taken into account.

The approach hereby presented aims at building a repository where such diversity is made transparent for an end-user searching the repository for architectural descriptions and their components.

3.0 USER QUERIES

The kind of queries that an end-user can submit will depend on the needs and knowledge that the user has. What is relevant, to best answer the user query, is how that query can be represented in terms of what is inside an architecture: classes, instances and relations. Especially the Relations of the repository play a pivotal role, in particular, with their properties: reflexive, symmetrical and transitive Relations, in fact, allow making important assumptions when examining the repository to satisfy user query. Last but not least, the fact that the user is querying a whole repository and not just individual architectural descriptions can be leveraged to provide better results.

3.1 Types of User Queries

To explain the importance that the representation of queries has in the whole process being described hereby, let us consider the following queries that could be submitted by an end-user:

- Find all the tanks that can run at more than 40 km/h;
- Find all the radios with ´80´ in their name;
- With whom does RC North in Afghanistan exchange information?
- With which tanks does RC North in Afghanistan exchange information?
- Find all the tanks with more than two radios.

From a technical perspective, all the queries the user can submit can be reduced to two main types and their composition.

The first type is the case when the end-user is searching directly for one "thing", be it a class, an instance, or a relation, basing the search over some attributes it may have; the first two queries above fall into this category.

The second type is the following: considering a triple (*X, Y, Z*), where *X* and *Z* are classes or instances (the user should be able to choose this when submitting the query) and *Y* is a relation between the two.

Figure 1: Generic Triple (X, Y, Z)

The end-user fixes two of them by applying a filter, as in the first type, and the system finds all the possible occurrences for the third one. The set of queries of this kind is extremely vast. One example is "*With whom does RC North in Afghanistan exchange information?*" In this example, *X* = "RC North in Afghanistan", *Y* = "Information Exchange", and *Z* is the variable.

Figure 2: Sample Query

Then, conditions of this kind can be composed in order to fulfil more complicated queries. For instance, consider a query like "*With which tanks does RC North in Afghanistan exchange information?*"; this is actually the combination of two conditions: in the first one, X_1 is the variable, Y_1 = "is-a" and Z_1 = "Class Tank"; in the second one, X_2 = "RC North in Afghanistan", Y_2 = "information exchange" and Z_2 includes all the X_1 matching the first condition.

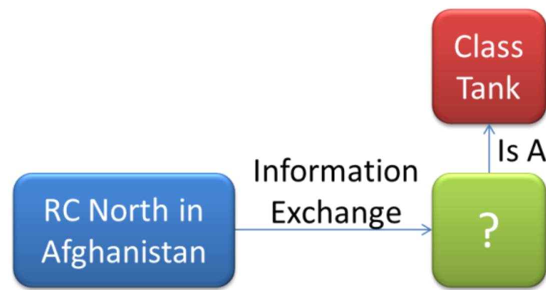

Figure 3: More Complex Sample Query

A further step is then represented by filtering results after the requested triples have been found. So, for instance a query like "*Find all the tanks with more than two radios*", can be seen as further composition of the above types where, in addition to the fact that *X* and *Z* are the result of some previous filtering, there are also conditions on the relation *Y* ("is part of", in this case), concerning its number of occurrences.

3.2 Navigating Relations

Certain queries imply that the result has to be obtained by exploring the graph of relations beyond the first step. Consider the example of the tank containing the radio. A first way of tackling the query is: if there is a relation "is part of" between an object that "is a" tank and an object that "is a" radio then return the tank, otherwise discard it. But then, what about a case where the radio is embedded into a combat system? The architect may have created three objects (the tank, the combat system, and the radio) and have established two "is part of" relations: one between the tank and the system and another one between the system and the radio.

In this example, if the user wants the embedded radio to be considered a radio, then the system must know that "is part of" is a transitive relation and deduce that if the tank contains the combat system and the combat system contains the radio, then the tank contains the radio.

In order to show the same problem with another relation, let us consider a query similar to that of Figure 3: "*Find all the vehicles with more than two radios*". The user will assume that tanks are vehicles and if the Leopard contained two radios the user would want it among the results of this query. Now, if an architecture contains a *Leopard* instance and a *Tank* class, there will be an "is a" relation between the two. But then, it will be the class *Tank* to have an "is a" relation with class *Vehicle*; therefore, to fulfil the user request, the system must know that "is a" is a transitive relation, and that the Leopard is a vehicle as well.

3.3 Beyond Querying a Single Architecture Model

As mentioned in chapter 1.0, each architecture shall encompass just the set of items of its interest, to the level of detail that was needed at the time of creation. For instance, an architecture describing reconnaissance missions with UAVs may include a detailed description of the *Predator* (stating, for instance, that it is equipped with an infrared sensor), but may omit the fact the Predator is a UAV, because it is taken for granted within the context. So if the end-user wants to query the repository for "*All UAVs equipped with infrared sensor*" the system will not be able to return the Predator instances from that architecture.

However, if another architecture existed in the repository, for instance a synthetic list of vehicles, this architecture could mention that among the UAVs there is the Predator, even though it may not describe its equipment. In such a case, when the system derives all the conditions necessary to implement the user query (as described in chapter 3.1), it should first check what items are UAVs (in the second architecture) and then which of the selected items have an infrared sensor (the information available in the first architecture).

4.0 CHALLENGES

Developing an intelligent discovery service is an ambitious task. There are several challenges that must be addressed to provide satisfactory results.

4.1 Formats and Languages

Architectural descriptions are available in many different formats and languages nowadays, also thanks to the plurality of software tools suitable for architectural design. The end products of the design process may range from self-designed Excel sheets (e.g., listing operational nodes and their information exchange relationships) to complex architectures developed with modelling tools such as ARIS, IBM System Architect, or Sparx Enterprise Architect. Also, the architectural descriptions can be contained in a single file or stored in a DBMS, or they can be exchanged via interchange formats like XMI and there are subtle differences in the output created by the different tools.

The repository must not be burdened by all these differences and store all the information contained in those descriptions in just one format, which can effectively be navigated to satisfy user queries.

4.2 Meta Models

The different modelling tools we mentioned previously are generally customized to meet the requirements of a given architecture framework (such as NAF) and/or national directives. In case of UML-based modelling, this tailoring can be achieved by UML profiles that specialise the semantics of UML. A popular standardized UML profile is OMG’s Unified Profile for DoDAF and MODAF (UPDM).

With regard to a discovery service, the main problem is that NAF concepts like *operational node* or *information element* are represented differently in the different architecture models. In order to provide an overarching discovery service, it is necessary to map the various meta-models onto a common, single meta-model that is the basis for the search engine. This mapping will be complex and possibly some information is lost in the process.

4.3 Architecture Objectives

Architectures are modelled with different objectives, at different times, by different people. This is reflected in the different types of architectures (overarching vs. reference vs. target vs. baseline architecture), their scopes, their granularity and their level of abstraction (see above example about UAVs).

Furthermore, an architecture may describe the current processes and systems, a projected state in the near or far future, or even multiple states as a migration path from the current to the planned situation.

The result of this diversity will be a set of architectural descriptions where the same concepts or items are represented differently, and partially.

4.4 Naming and Design Rules

Due to the lack of an all-encompassing supervision, architecture models are partly based on individual conventions. For instance, depending on the point of view of the modeller, a *video stream* or *intelligence information* can be considered as (operational) information elements, even though both terms are on different levels of abstraction.

In addition, there may be variations among the names:

- Full names vs. Acronyms?
- Composite terms with vs. without hyphen? (this is a problem in German, in particular)

- Noun vs. verbs – “Report IED” vs. “IED Report”?
- English terms vs. terms in mother language?

4.5 End-User Competence

The end-user querying the repository can be equipped with very diverse levels of competence and experience. There may be a high-level decision-maker, who ignores many details well-known to architects but there may also be an architect, querying a repository full of architectures built by others, at different times, for goals other than his own. In both cases, querying can be more successful (and satisfactory) if the user has the possibility to receive feedback by the system with respect to the query submitted.

4.6 Presentation of Results

Another important factor concerning the interaction of the discovery service with the human is related to the presentation of the obtained results. The very first step would be a textual list with a reference to the architecture and the name(s) of the elements found.

Two important elements would be missing in such a presentation: 1) a way to further expand/explore the results and 2) a clear evidence of why the result was returned. Both elements are important for the user to have a better comprehension of the query that took place and play an active role in improving the query submitted.

5.0 PROPOSED APPROACH

The approach that this paper will follow tries to tackle the problems described above: differences in format, differences in models, differences in architecture perspective and naming, differences in end-user capabilities to perform queries and support to the querying process.

5.1 A Unified Repository Format

The first step is the identification of a unifying format. It must be

- suitable for representing classes, instances and relations, and their attributes and properties,
- expressive enough to represent all those items with sufficient conciseness,
- simple enough to be manageable.

It must have

- mature software libraries and software tools that deal with it,
- querying languages to search the repository.

The language that we identified that satisfies all these requirement is OWL2 [2], where classes are called c*lasses*, instances are called *individuals* and relations are called *properties*. In particular, since this choice is not only for modelling but also (and mostly) for reasoning, we will restrain to OWL2 DL. Among the most relevant characteristics supported natively, the following ones are worth mentioning:

- Classes and Instances;
- Class Hierarchies;
- Property Hierarchies;
- “Domain” and “Range” Restrictions of a Property, i.e., the restriction over the classes that the endpoint of a relation can assume;

- Equality of Individuals;
- Property Restrictions and Property Cardinality Restrictions;
- Property Characteristics (symmetric, reflexive, transitive);
- Property Chains, i.e., the composition of more Relations in a single Relation;

The general idea is that every architectural description to be included in the repository will be transformed into one ontology. Once this ontology enters the repository, it will be integrated in just one big piece of knowledge, as described in chapter 5.3.

To transform an architecture model in an ontology, we will need a **Mediator**, i.e., a program capable of reading a specific file format of one of the modelling tools commonly available. It will read the architecture file and produce an ontology (a .OWL file). It will translate Instances (be them represented by *objects* or *elements*, depending on the specific source tool) into *instances*, Relations (be them *links* or *connectors*, depending on the specific source tool) into *properties* and classes… into *classes*.

Of course, it will not be a 1-to-1 translation, but it will depend on the specific convention used by the source format, e.g., in a format there could be just *elements* and *connectors*; then, a class will be a particular type of *element* and the "is a" relationship, coming from an *element* of that class, will be a special *connector*. The mediator will have to catch all these peculiar representations and create the right OWL components, e.g., a *class*, an *individual* and *ClassAssertion*.

5.2 Modelling the Repository

The native structures of OWL do not have the same expressiveness as modelling languages (like UML) and higher-level meta models (like UPDM). However, the higher-level, missing concepts can effectively be represented in OWL. Therefore, a complete structure of new *classes*, *instances*, and *properties* (in the OWL way) will have to be created in the repository order to express directly concepts like "stereotype", "diagram", "capability". Basically, all the concepts introduced by models and meta-models commonly used for architectural descriptions should be created in the repository ontology. It is beyond the scope of this paper to define precisely such ontological model.

Following the existence of such an ontological model, it is clear that the work of the **Mediator** introduced in the previous section cannot be restrained to the mere creation of OWL *classes*, *instances*, and *properties*. The **Mediator** (actually, all the mediators for all the different formats that have to be imported into the repository) will have to make use of that target ontological model.

5.3 Harmonizing the Repository

Through the job of the mediator, we have abstracted the repository from the specific format used in the architectural description. However, we have still to deal with the differences in architecture perspective and naming and design rules. To overcome that, it will be necessary to perform some reasoning over the repository, aimed at discovering equivalent items belonging to different architectures, and marking them as such. Therefore, a **Reasoner** should be in charge of Repository Harmonization.

The starting point could be a similarity analysis of terms/identifiers and methods adopted from computational linguistics. The most simplistic approach is to incorporate domain-specific dictionaries, a thesaurus, and a simple word translator. However, due to potential differences in naming (see 4.4), it is not possible to equate elements from different architectures purely based on their names. So, when evaluating and visualizing architecture information, it is necessary to take the original context into account. For instance, in a UML model such context could be given by the package structure.

Basically, all that is needed is a process of ontology alignment. This problem has been covered abundantly in literature for the last 10 years and more. It is beyond the scope of this paper to choose the ontology alignment approach most suitable to an architecture discovery service.

To further support repository harmonization, the repository should be complemented by a set of concepts called "Domain Knowledge". Basically, this means a further (and potentially very vast) set of *classes*, *instances*, and *properties* (in the OWL way) will be added to the repository (at "initialization time") to aid the alignment process (instead of aligning two architectures, it could be easier to align each of them to the Domain Knowledge) or to fill the gaps that the **Reasoner** could find when trying to align two concepts.

The result of the harmonization of the newly imported architecture within the repository will be a single ontology, representing the union of the previous repository and new architecture, plus the established alignments.

5.4 Assisted Querying

As stated in chapter 4.5, the effectiveness of a query cannot be left to the improvisation of the end-user. The discovery service has to support the process of querying with adequate tools. In chapter 3.1, we have shown how queries possibly submitted by an end-user can be translated into one or many conditions to be checked against the repository. We do not have natural language interpretation in mind or anything like that. Quite the opposite: the idea is that a dedicated **User Interface** will support the end-user during the whole search process, as we will describe now.

Let us suppose the interest of the end-user is to find "*All the addressees of orders issued by Regional Command North in Afghanistan*". First of all, how would "Regional Command North in Afghanistan" be called in the repository? "RCN AFG"? "RC North"? In addition, it may have been called differently by different architects. So the first step would be to identify the one (or better, the set of) "Regional Command North in Afghanistan"(s) that match the user interest. If the repository harmonization described in chapter 5.3 has successfully taken place, find one of them should mean finding all the equivalent ones as well. The **User Interface** should then represent this first set of results to the end-user.

The next step would be determining what the relation used to specify orders issuance would be. In that respect, there are different ways to help the user. Firstly, the system could present a list of all the relations coming out of the items found in the previous step but that could lead to an excess of information. Another path could be walked if the Capability of receiving orders is defined somewhere; then the set of possible relations of the Regional Command could be restricted to those going towards other classes/instances capable of receiving orders.

In this respect, the **User Interface** will receive input from the user and create the queries. Also, it will incorporate in those queries the possible options available to the user (such as "Search classes only" or "Search classes and instances"), translating them appropriately.

Submitting the queries, getting the result set, and applying further reasoning is the task of the **Search Engine**. In particular, in the example above (where the user first finds "Region Command North" and then imposes that the other end point must be capable of receiving orders) there will be more interactions between the User Interface and the Search Engine, in order to assist the user step-by-step. Another task of the **Search Engine** is the navigation of relations, as described in chapter 3.2. The decision of following transitive relations, for instance, only up to the *n*-th level, can make the difference between finding too many and too few results.

5.5 Graphical Representation of Results

Another major role of the **User Interface** is an effective presentation of the results to the user. Considering the requirements stated in chapter 4.6, the solution hereby proposed is composed by structured, text-based list and a graphical representation.

The structured, text-based list could actually be an expandable tree, where a node is a Class, an Instance or a Relation and expanding its sub-tree allows seeing other elements that have to do with that result. For instance, if it is a class, there could be the sub-tree of its instances or the sub-tree of its sub-classes; it may even make sense to have the result not as the root of a tree but at a certain level to show also its super class (or super classes, where applicable). As another example, if the result is an instance, than we could have the sub-tree of all the other Instances related to it, and on the way up the tree (as in the example before) we could have the class this instance belongs to and (further on) its super-classes. An additional plus would be the possibility of selecting another node and have the system load the tree for this new node, so the user can continue his virtually seamless exploration of the repository.

The graphical counterpart would essentially offer the same possibilities, i.e. it would show other elements up or down certain possible lines, such as super-class/sub-class or via relations. And, just as the text-based part, once a new element is selected, the other elements around are fetched according to this new centrality.

All these visual updates are obtained by continuous interactions between the **User Interface** and the **Search Engine**. In order to avoid waste of search and communication time, the data related to sub-trees to be opened or new elements to be focused on will be requested only as a consequence of a user request.

6.0 SYSTEM ARCHITECTURE FOR A DISCOVERY SERVICE

A system architecture implementing the approach described in chapter 5 is depicted in Figure 4.

- The *Mediators* (one for each different input format) translate from specific format to the ontological Architecture Warehouse.
- The *Reasoner* performs ontology alignment and all the other matching procedures aimed at harmonizing the repository.
- The *Domain Knowledge* (de facto in the Warehouse but conceptually separated) helps the Harmonization process.
- The *User Interface* supports the end-user by receiving input and providing output, possibly with the purpose of receiving more focused input for further search. It translates all the input provided by the end-user in queries for the Search Engine.
- The *Search Engine* submits the queries and processes the result for the User Interface.

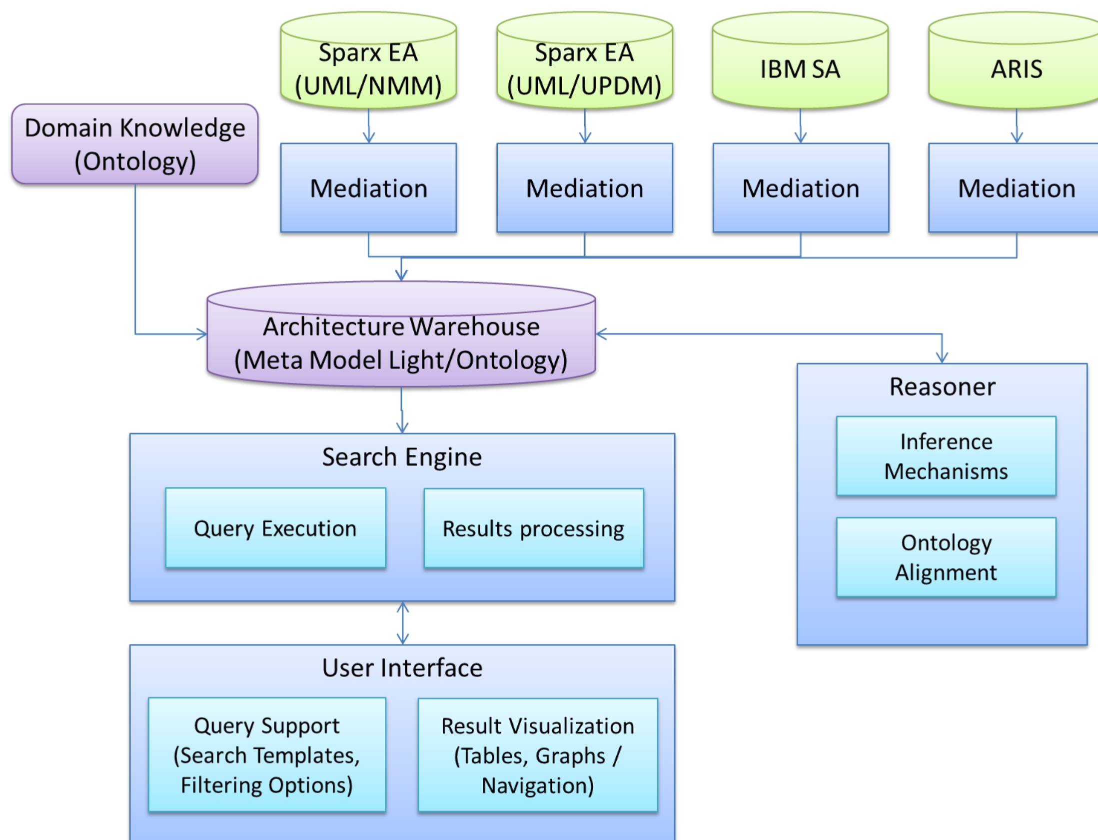

Figure 4: Architecture for an Intelligent Discovery Service

7.0 REFERENCES

[1] Mark W. Maier, David Emery, Rich Hilliard: "Software Architecture: Introducing IEEE Standard 1471", IEEE Computer volume 34, Issue 4, pages 107-109.

[2] http://www.w3.org/TR/owl2-primer, OWL 2 Web Ontology Language Primer (Second Edition)