Ontonuts: Reusable semantic components for multi-agent systems

Sergiy Nikitin, Artem Katasonov and Vagan Terziyan

Industrial Ontologies Group, University of Jyväskylä
{sergiy.nikitin, artem.katasonov, vagan.terziyan}@jyu.fi

Abstract

The volumes of data in information systems are growing drastically. The systems become increasingly complex in trying to handle heterogeneity of ubiquitous components, standards, data formats, etc. According to the vision of Autonomic Computing, the complexity can be handled by introducing self-manageable components able to “run themselves.” Agent Technology fits this vision, whereas interoperability among autonomic components can be tackled by Semantic Technologies. The problem of efficient heterogeneous data sharing, exchange and reuse within such systems plays a key role. We present an approach of constructing semantic capabilities (self-descriptive functional components) for software agents and a mechanism for distributed data management that applies these capabilities to build various industrial business intelligence systems.

1. Introduction

The volumes of data in information systems are growing drastically. The systems become increasingly complex in trying to handle heterogeneity of ubiquitous components, standards, data formats, etc. According to the vision of Autonomic Computing [1], the complexity of information systems can be handled by introducing self-manageable components, able to “run themselves.” In our opinion, Agent Technology fits this vision very well. Whereas the interoperability among autonomic components (agents) can be tackled by Semantic Technologies, efficient data sharing, exchange and reuse within such systems still play key roles. Semantic agent-driven systems cannot fully substitute e.g. high-performance industrial data storage, nor can they avoid physical distribution of data and services. In attempt to resolve the challenges stated, we are developing an agent platform of a new generation – called UBIWARE [2], [3]. Efficient data sharing, exchange and reuse determine the usability of the UBIWARE platform and its technological success in industry. In this paper we introduce a mechanism for distributed data management within the UBIWARE platform that allows platform users to build distributed industrial business solutions.

The paper is organized in the following way. In the next section, we present an industrial scenario for distributed querying and discuss problems that call for new ICT solutions. In Section 3, we briefly describe the UBIWARE platform. The fourth section presents a concept of semantic components called Ontonuts and shows how they are applied to the scenario of distributed querying. The discussion on related work is presented in Section 5. We conclude and propose future work in Section 6.

2. Distributed Querying Scenario in Paper Industry

The scenario we have selected is based on the real software infrastructure from process industry. There is a complex production line (e.g. paper producing machine) which is served by a number of control and diagnostic systems. The measurements taken from sensors are stored in the Alarm History Database. The Diary Database contains records about critical alarms and comments from maintenance workers. There are also comments on actions taken. The Scheduled Performance Monitoring database stores results of the analysis that is performed daily. The analysis includes all nodes with performance indices that indicate the condition of the node (Figure 1).

As an example, suppose a serious fault happened in the paper machine that has led to the subsequent alarm and maintenance actions. All events are recorded in the respective databases. However, in order to analyze the preceding events, e.g. during the week before the fault, an expert may need to query portions of data from all the databases, then change filtering parameters and query databases again and again. An expert may have interfaces to all the databases separately; however, the expert’s decision will be stored (if it would be at all) to a separate file or database.
Figure 1. The IT-infrastructure of the paper machine

Within the current IT infrastructure, it is hard to find previously made expert decisions on an immediate fault situation. Thus there is no integrated view on all the contents of the databases, nor on other sources of information about the paper machine operation. In the Semantic Web domain, it is called a proof when any knowledge is connected with the rules and facts that were used to infer it. The problem of an integrated view on the information is important in the industrial automation particularly because we perceive that many experienced experts in a majority of companies are going to retire during next 5-10 years without a proper knowledge transfer to new experts. On the other hand, semantic linking of the information will be in a great demand when the standardization efforts taken in companies will go beyond the companies’ boundaries and will call for a unified mechanism of distributed querying for knowledge and expertise exchange (see Figure 2).

Figure 2. Inter-company standardization in paper industry

In the ideal case, companies would be able to sell information and analytic services to each other seamlessly with little or no programming effort. The services sold would be easily integrated into the company environment with the guaranteed compatibility. Furthermore, service consumers (factories) could become service providers too. Industry might share data and use it as learning samples for analytic services. The role of providers of IT solutions would shift from the integration aspects to those of intelligence. We, therefore, foresee the need for tools and capabilities in the UBIWARE platform that will simplify distributed querying and information integration.

3. UBIWARE platform

In this section we briefly introduce the UBIWARE agent-driven middleware platform, its agent engine, and S-APL – a Semantic Agent Programming Language for programming of software agents within the platform.

3.1. UBIWARE Platform Architecture

Central to the core platform is the architecture of a UBIWARE agent, depicted in Figure 3.

There is a Live behavior engine implemented in Java, a declarative middle layer, and a set of Java components, known as Reusable Atomic Behaviors (RABs). RABs can be considered sensors and actuators, i.e. components sensing or affecting the agent’s environment, but are not restricted these. A RAB also can be a reasoner (data processor) if some of the logic needed is not efficient or possible to realize with the S-APL means, or if one wants to enable an agent to do some other kind of reasoning beyond the rule-based one.

The UBIWARE agent architecture implies that a particular UBIWARE-based software application will consist of a set of S-APL documents (data and
behavior models) and a set of specific atomic behaviors needed for this particular application. Since reusability is an important UBIWARE concern, it is reasonable that the UBIWARE platform provides some of those ready-made. Therefore, the UBIWARE platform, as such, can be seen as consisting of the following three elements:

- The Live behavior engine
- A set of “standard” S-APL models
- A set of “standard” RABs

The extensions to the platform are exactly some sets of such “standard” S-APL models and RABs that can be used by the developers to embed into their applications certain UBIWARE features.

3.2. S-APL platform language

In the UBIWARE Platform, behavior models are presented in a high-level, rule-based language, the Semantic Agent Programming Language (S-APL). S-APL is based on the RDF (http://www.w3.org/TR/1999/REC-rdf-syntax-19990222/) data model, i.e. the whole document can be seen as a set of subject-predicate-object triples. A behavior model specifies the initial beliefs (including knowledge, goals, commitments, and behavioral rules) of the agent in the role. Commitments and behavioral rules normally lead to adding/removing beliefs and executing various RABs. The notation that is selected for use in S-APL is a subset of Notation3 (http://www.w3.org/DesignIssues/Notation3.html). Notation3 was proposed by Tim Berners-Lee as an alternative to the dominant RDF/XML notation. There are namespaces in S-APL; in particular, the “sapl” namespace is used for the resources that are defined in the language’s ontology. The default namespace is used for all the other resources in this paper.

In S-APL every statement is a belief of the agent. Simple belief would look like:

`sapl:configuredAs{ ... sapl:Success sapl:add { { sapl:I sml:do ...} sapl:configuredAs { ... } } }

meaning that, upon successful execution of the first commitment, the enclosed one should be added.

However, the central construct of the language is the conditional commitment:

`
{sapl:I sml:do java:SendMail}
{sapl:configuredAs { ... } }
`

The interpretation is straightforward: Upon occurrence of a belief that satisfies the condition stated in the subject, the contents of the object are added to agent’s general context G. Another key construct is matching with variables (querying). The commitment for querying is defined as follows:

```
{ {:John :Loves ?x} :accordingTo ?y. ?x sapl:is :Girl } =>
{ sapl:I sml:do java:SendMail} sml:configuredAs {
p:receiver sapl:is ?x ...
}
```

which can be interpreted, then, as “If John loves ?x, according to someone’s opinion, and ?x is a girl, then send email to ?x”.

Yet one more construct is a behavior rule:

```
{ { ... } => { ... } } sapl:is sapl:Rule
```

The behavior rule differs from the commitment. Whereas a commitment is removed from the agent’s beliefs upon execution, the rule stays and executes in agent’s beliefs permanently. It must be removed explicitly.

In this section we have briefly introduced the core concepts of the UBIWARE. In the next section, we present an extension done beyond the core that makes an important step towards practical applicability of the platform in industrial applications.

4. Ontonuts Concept

We introduce here the concept of Ontonut to facilitate the presentation of modular scripts and plans within the UBIWARE platform. Ontonut is a semantic software component. Instances of the Ontonut concept generally represent a capability with known input and expected output. We then extend Ontonuts to solve the problem of distributed querying discussed in Section 2 of this paper.

4.1. Ontonuts in a nutshell

The Ontonuts technology is implemented as a combination of an S-APL script and RABs and, hence, can be dynamically added, removed or configured. Ontonuts allow componentization of S-APL code by introducing a semantic annotation to it. Such annotated
pieces of code are called *capabilities* (analog of function in procedural programming). The capabilities have S-APL descriptions with explicitly defined preconditions and effects:

\[
\text{Ontonut: } \{\text{script, precondition, effect}\}
\]

The capabilities can be dynamically combined further into plans and put into execution by the Ontonuts engine, which allows us to automatically compose the agent’s actions to achieve a specified goal. The script part of the capability in general has an S-APL code that produces the effect once the precondition is satisfied. The whole data model of the UBIWARE platform is triple-based; therefore, goals, preconditions and effects are defined as triple sets in S-APL. For example, we have an initial data set \{\text{A A A}\}, a goal \text{G1} defined as {\text{C C C}}, and we have two ontonuts \text{O1} and \text{O2}, defined as:

\[
\begin{align*}
\text{O1} & \text{ rdf:type :Ontonut} \\
& \text{ ont:precondition } \{\text{A A A}\} \\
& \text{ ont:effect } \{\text{B B B}\} \\
& \text{ ont:script } \{\{\text{A A A}\}=>...=>\{\text{B B B}\}\}
\end{align*}
\]

\[
\begin{align*}
\text{O2} & \text{ rdf:type :Ontonut} \\
& \text{ ont:precondition } \{\text{B B B}\} \\
& \text{ ont:effect } \{\text{C C C}\} \\
& \text{ ont:script } \{\{\text{B B B}\}=>...=>\{\text{C C C}\}\}
\end{align*}
\]

The appearance of the goal \text{G1} will activate the Ontonuts engine that will match the \text{G1} against available effects and then apply planning, which will result in an execution plan: \text{O1}=>\text{O2}=>\text{G1}.

Ontonuts reuse available scripts and RABs without any modifications to the platform. Architecturally, the Ontonuts engine consists of three main components: the Triggering Rule, Action Planner and Plan Executor (Figure 4). The Ontonuts Triggering Rule is a starting point of the engine work. The Triggering Rule is a MetaRule, i.e. it runs before other rules and commitments. On each iteration of the Live behavior, the rule checks whether there are any Ontonut calls to be handled and passes the activity to the Action Planner, which provides an execution plan for the Plan Executor.

Ontonut capabilities may include interaction with other agents or external resources, such as databases, files or web services. On the other hand, capabilities can perform local actions and do some computations on the data, e.g. statistical analysis.

**4.2. Invoking Ontonuts**

The Ontonuts engine supports three types of Ontonut calls:
- Explicit
- Goal-based
- Pattern-based
achieved, and, hence, the variable values in the left part of the commitment can be assigned, the Ontonuts engine produces the result (the right part) of the commitment using the variable values. This type of Ontonuts targets mainly the task of distributed querying that is discussed in Section 4.6.

4.3. Planning the execution

The planning is organized as a goal-driven process. We apply a backward chaining algorithm to build an action plan, which may involve other Ontonuts and Rules. The planner performs a semantic inference over the set of initial data before the actual plan generation starts. Therefore, the semantic annotations of Ontonuts, as well as the corresponding domain ontology, are key success factors of the Ontonuts-based applications. The planner acts in a straightforward way – it matches the goal against Ontonut annotations by subtracting (operation over sets) these annotations from the goal. If a goal can be fulfilled by the available initial data and Ontonuts, the planner starts to check whether the preconditions of these Ontonuts can be fulfilled. If the preconditions may need to use other Ontonuts, they are checked as well. In such an iterative manner, the planner builds a solution tree. The planner then chooses the preferable solution using different criteria, e.g. utility-based selection.

4.4. Handling the execution

The Ontonuts engine does not execute the plan as a whole; rather, it generates a plan that is run by the agent’s Live behavior engine. However, the plan is not straightforward: It includes additional handlers that allow the Ontonuts engine to observe the state of the execution and react if the execution cannot be successfully completed. The plan is sequential and therefore has steps or control points. At each control point, the plan produces the statements that represent the status of the execution. These statements are collected into a container that is attached to the plan:

```ont
:planid ont:execStatus {
  :01 ont:status ont:Success.
  …\n  :nn ont:status ont:NoResponse.}.
```

The engine then can use the status information for re-planning if the current plan did not succeed.

There are two classes of Ontonuts executed in different ways:
- Self-running
- Engine-running

The latter type has a built-in script that runs in the agent’s Live behavior as an independent code and returns the result to the G container. Meanwhile, the former is a description that is recognized and executed by the Ontonuts engine. The engine-running Ontonut calls are presented in the plan as explicit (see Section 4.2). In the current version, the engine supports one type of engine-running Ontonuts that simplify access to the databases.

4.5. Distributed querying with Ontonuts

There are two main viewpoints towards distributed querying in the UBIWARE: adapter-based and service-based. The former tackles the adaptation of the data sources that are external to the platform (databases, files, web services, etc.), while the latter deals with the agent-to-agent querying. Nevertheless, both target the same goal: to make distributed querying logic transparent (simple) to the UBIWARE agent (see Figure 5).

![Figure 5. Distributed querying in UBIWARE](image)

The agent-to-agent querying follows the servicing paradigm and, in particular, the data servicing discussed in [4]. The adaptation of external sources (e.g. RDF-based adaptation is discussed in [5]) resolves the problem of connectivity and interaction with those resources that are external to the platform, i.e. communicating with them in their native language.

However, from the business logic developer’s point of view, any remote resource should be transparent in order to keep business logic script as modular and clear as possible. Ontonuts become the wrapper, adapter and connector in one place.

In a distributed querying task, every Ontonut is an interface to the data source that has an associated data query pattern (effect) it replies to. The Ontonuts engine introduces an extension for the data source-based Ontonuts. The extension allows for the Ontonut developer to not implement all the RAB calls and S-APL transformations from the scratch, but rather to define a description of the data source and
transformation mappings. We call this subclass of Engine-running Ontonuts Donuts (Database Ontonuts). The engine distinguishes the Donuts and treats them in a different way. The user-defined query can match several Donuts; therefore, the Triggering Rule invokes Action Planner. The Action Planner distinguishes sub-queries from the initial query and produces a distributed query plan. The plan is then passed to the Plan Executor. The executor handles the intermediate results of sub-queries and modifies subsequent sub-queries accordingly.

The structure of the Donuts is defined by Donuts Ontology (see Figure 6).

The fragment of the ontology above describes the root classes Ontonut and DataSource, as well as their extensions for connectivity with relational databases (Donut, RDBDataSource). Similarly, other types of extensions will include type-specific facets in their descriptions.

The Plan Executor uses data source descriptions for fetching the sub-queries and applies mapping definitions to transform sub-query results into the semantic form.

4.6. An illustrative example

The example presented here is based on the usage scenario described in Section 2 of this paper. Suppose that a fault situation happened and the agent of the expert wants to extract the comment strings from the Expert’s Diary database and align them with the performance indices from the Performance Monitoring database. Then the alarm limits and alarm values are extracted from the Alarm History database, based on the node-to-tag mappings. The time interval used for filtering is 10 days before the fault. The agent prints the collected values to the command line. The resources involved in the query and their tables are shown in Figure 7. Each resource has an associated Ontonut in the agent’s beliefs.

Figure 6. A fragment of Donuts ontology

The Plan Executor uses data source descriptions for fetching the sub-queries and applies mapping definitions to transform sub-query results into the semantic form.

Figure 7. Sample database tables

The description of the Ontonut associated with the expert’s diary and the datasource object (an instance of RDBDataSource) are shown below:

```
:DiaryEntryNut rdf:type ont:Donut.
:DiaryEntryNut ont:dataSource :entrydb.

:DiaryEntryNut ont:SQLQueryBase "SELECT entryID, entryDate, author, title, description, position FROM diary.Entry".

:DiaryEntryNut ont:mapping {
  ?entryId ont:mapsTo {ont:sqlentity sapl:is "entryID"}.
  ?entryDate ont:mapsTo {ont:sqlentity sapl:is "entryDate"}.
  ?author ont:mapsTo { ont:sqlentity sapl:is "author"}.
  ?title ont:mapsTo { ont:sqlentity sapl:is "title"}.
  ?description ont:mapsTo {ont:sqlentity sapl:is "description"}.
  ?position ont:mapsTo {ont:sqlentity sapl:is "position"}.
}

:DiaryEntryNut ont:effect {
  ?entry :entryId ?entryId.
  ?entry :entryDate ?entrydate.
  ?entry :position ?position
  }.

:entrydb rdf:type ont:RDBDataSource.
:entrydb ont:hasDriver oracle.jdbc.OracleDriver.
:entrydb ont:hasUsername diaryuser.
:entrydb ont:hasPassword mypwd.
```

The ont:SQLQueryBase defines the SQL query that extracts the data that is used to produce the Ontonut instances. The mapping definitions use the column names from the SQLQueryBase property.

Other Ontonut descriptions are defined in a similar manner. The effect of the second Ontonut used in this example is defined as follows:

```
:PMAnalysisNut ont:effect {
```
The effect of the Ontonut for the Alarm History database is defined as:

```
:AHAlarmNut ont:effect {
  ?alarm :alimitHigh ?ahigh
  ?alarm :alimitLow ?alow
  ?alarm :value ?avalue}.
```

The commitment (query) in the agent's beliefs is shown below:

```
{?entry :entryDate ?edate.

 ?pos :mapsTo_1 ?nodeid.

 ?pmnode :analysisDate ?adate.


 ?edate = "31.12.2008".
 }=*

{ gb:I gb:do :Print } gb:configuredAs
```

The commitment does not explicitly refer to the type of the Ontonut by the rdf:type property, which would simplify the implementation of the triggering procedure, but we apply pattern-based matching, i.e. use subtract operation over available Ontonut effect patterns and the query.

In the particular query, the triple that matches the identifiers defined as:

```
?pos :mapsTo_1 ?nodeid.
```

It is a bridging property between the two property values of respective Ontonuts, which have physical data sources behind. It may belong to any of the Ontonuts it bridges or be an independent Ontonut: How it is modeled is domain-specific.

As soon as the matching with available Ontonuts has succeeded, the matchmaking rule passes the control to the Query Planner. In this particular case, the work of Query Planner is straightforward – to decide which Ontonut is to be queried first and apply the query parameters for the SQL query generation. The order of execution may depend e.g. on the average expected number of records of each independent sub-query. The methods of execution planning and optimization go beyond the scope of this paper. Further reading suggestions are in the next Section.

The result of the first executed sub-query is then used to limit the range of the variables in the subsequent sub-query. When all the query results are collected, they are printed to the command line (see Figure 8).

![Figure 8. Query results](image)

The table of results is compressed (duplicate rows are removed and repeating values substituted with dash sign) and arranged in a chronological order for a purpose of readability. The table allows e.g. an expert to analyze how the actual parameter values and performance indices were behaving before the fault happened.

### 5. Related work

The notion of semantic component composition is discussed in [10] and [11], but the authors focus on improving the programming paradigm as such, not the autonomic computing and semantic agent programming. The execution planning and optimization of queries are thoroughly researched in [6] and are a rather complementary part for our work dealing with query planning. The approach in [7] proposes a solution for management of corporate histories using multi-agent system and semantic data,
our work, in contrast, introduces an intra-agent feature, that simplifies the programming of an agent. A set of Semantic Web Service platforms and languages like OWL-S [8] and WSMF [9] externalize semantic components and allow planning and execution. However, the Ontonuts approach treats components as internal capabilities of an agent that can be externalized as semantic services. While the approach presented in [12] proposes an extension to the RDF language in order to allow modifications of the RDF content upon some triggered actions, we deal with the extension to the rule-based agent programming language, which allows componentization and data updates. Ontonuts allow semantic integration of both internal and external capabilities and stand for a semantic agent-driven workflow planning and execution engine.

6. Conclusions and future work

The approach presented in this work aims toward a quite specific target – to structure the S-APL code of the UBIWARE agent in order to simplify programming of the agent and allow automated goal-driven planning. Although the paper mainly covers a quite narrow domain of distributed querying, it involves the generic problems of agent planning and semantic programming. The emphasis of this paper is on the automated planning of distributed queries and so-called virtual graphs, when the graph being queried does not have RDF content beneath. The approach proposed uses patterns to define desired result and applies queries or any other code to fill the patterns requested.

At the moment, the first prototype is nearing its completion. We plan to extend functionality of the Ontonut engine by introducing more engine-running Ontonut types, e.g. for web services and agent services. Another important step to be taken is to perform scalability tests over large data volumes and compare the results with purely SQL-based implementation alternatives. The planning procedure should also be improved to keep alternative pathways on each stage of the execution. Furthermore, in theory, agents can share Ontonuts as self-containing executable modules.

7. Acknowledgement

This research has been performed in the UBIWARE project, funded by TEKES, and the industrial consortium of Metso Automation, Fingrid, Nokia and Inno-W.

8. References