Łukasz Faber

AGENT-BASED DATA INTEGRATION FRAMEWORK

Abstract *Combining data from diverse, heterogeneous sources while facilitating a unified access to it is an important (albeit difficult) task. There are various possibilities of performing it. In this publication, we propose and describe an agent-based framework dedicated to acquiring and processing distributed, heterogeneous data collected from diverse sources (e.g., the Internet, external software, relational, and document databases). Using this multi-agent-based approach in the aspects of the general architecture (the organization and management of the framework), we create a proof-of-concept implementation. The approach is presented using a sample scenario in which the system is used to search for personal and professional profiles of scientists.*

Keywords data integration, multi-agent systems

1. Introduction

The AgE platform[1] [6] is a general-purpose agent-based computation framework that has long been successfully used for soft-computing problems (e.g., [3]), more-general research such as the verification of functional integrity or testing stochastic features of systems of that kind [14], and simulations [6].

However, the platform has a potential to be successfully used in other scenarios. In this paper, we present an agent-based framework built upon it that makes it possible to build specialized, distributed systems for heterogeneous data integration. The general structure and functionality were highlighted before in [12, 2], and we present a short summary of these in Section 2.

Our goal is to take the AgE platform and try to exploit its best properties to obtain an extensible system structure. We show a system that can easily integrate different models of data.

Properties of agent systems are particularly suited to face the problems encountered in data mining [5]. Intrinsic decentralization and modularity provide us with a simple way to implement features expected in a data integration system. For example, an agent-based data integration system can easily adapt to the expectations of the user (learning about them and choosing the most important data), or to the structure of the sources. Moreover, multi-agent systems are naturally distributable and fault-tolerant. These properties are also welcomed in data-mining systems [4].

This paper begins with a short summary of previous papers and a presentation of AgE. We present its structure and agency mechanisms. Then, we briefly summarize selected systems built for integration of heterogeneous information. In Section 4, a detailed description of the hierarchical data integration and the processing framework is given. The above is illustrated by the sample system for building the personal profile of a scientist. As the general idea of the system was introduced in earlier papers ([12, 2]), we focus on a technical presentation of the framework.

2. General model of the system

The framework introduced in [12, 2] has three modes of operation: (i) item user-driven, (ii) item semi-automatic, (iii) item fully automatic, which are, in general, more and more independent versions of the same model, requiring less and less control from the end user. The first one was implemented and is presented in this paper.

The main goal of the framework is to provide domain decomposition to the user.

The core workflow element of the framework is *task*. Its responsibility is to transform an input to some output using a given configuration. Such tasks can be composed with regards to input and output type compatibility.

The work in this system is performed by agents forming a hierarchical structure of a tree. This is a result of a task delegation. When an agent (that is in the process

[1] `https://age.iisg.agh.edu.pl`

of obtaining data about a complex item) wants to gather more detailed information about one of the properties of the data, it can delegate this task to another agent located beneath it in the tree.

Data in the system is presented as objects comprising of a type and a set of simple or complex properties. These objects are entrusted to agents that can extend or modify them on the basis of tasks given by the user or other agents. There can be several agent types in the system: (i) item updaters, which update the data; (ii) simplifiers, which merge the data; (iii) verifiers, which verify the data; etc.

Agents can use *strategies* in order to obtain similar types of data from different sources. A strategy in this context is simply a software component that hides the implementation of data gathering from a service. For example, there may be different strategies of obtaining personal data from different sources (but the model of data is always the same).

3. AgE – Agent-based Computation Framework

AgE is a project developed as an open-source software at the Intelligent Information Systems Group of AGH University of Science and Technology. AgE is a framework for development of distributed agent-based applications.

In AgE, the user creates an instance of the system by providing a configuration file. The configuration specifies required node services, initial agents, and their behaviors. These services can extend the node with (e.g.,) distribution or monitoring facilities. After system startup, environment and agents are created, configured, and distributed among available nodes where they are executed.

The system is decomposed into agents that are responsible for performing a part of (or the whole) algorithm. In AgE, agents are structured into a tree composed of aggregate and simple agents, as shown in Figure 1.

Agents may be further decomposed into functional units according to strategy design pattern [7]. Strategies represent problem-dependent algorithm operators and may be exchanged without intruding on the agents' implementation. Their instances are usually shared between agents on a single node. In the case of data integration, a strategy may be responsible for obtaining currency rates, and its different implementations will be using different sources of such data.

Each agent has a unique address which identifies it across the whole system. We distinguish two kind of agents: threaded and simple ones. The former are implemented using separate threads. They can communicate and interact with their neighbors via asynchronous messages. The latter can also use queries and ask other agents to perform specific actions.

An increasing number of threaded agents would significantly decrease an application's performance due to frequent context switches. To prevent this, most of the computation is realized using simple agents. They are based on event-driven simulation, which results in the pseudo-parallel execution of tasks. In AgE, they are processed in two phases:

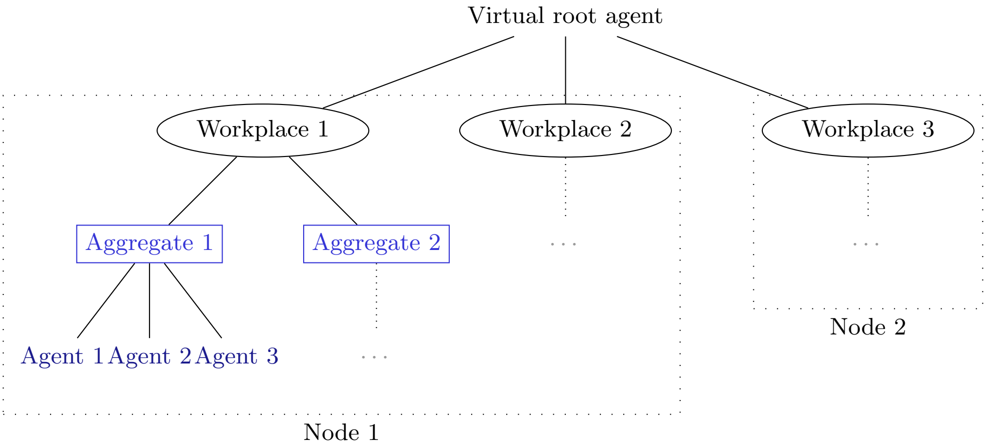

Figure 1. Agent tree structure in AgE. Workplaces are units of distribution among separate AgE nodes. Each workplace may consist of any number of aggregates and simple agents. The *virtual root agent* represents a common root for all workplaces which handles communication facilities on their level. Dotted rectangles represent separate physical nodes.

- Execution of computation semantics in the `step()` method. In case of an aggregate, all of its children perform their steps sequentially. While doing so, they can register various events which may indicate actions to perform by the parent aggregate.
- Processing of events registered in an event queue. Since events may be registered only in agents that possess children, this phase concerns only aggregate agents.

The described idea of processing ensures that, during execution of computational tasks of agents coexisting at the same level in the structure (agents with the same parent), the hierarchy remains unmodified; thus, the tasks may be carried out in any order. From the perspective of these agents, they are processed in parallel. All changes to the agent structure are made by aggregates during processing of the events indicating actions, such as the addition of a new agent, the migration or killing of an agent, etc. Results of these actions are visible for the agents during the next step.

Each agent exists in an environment defined by the parent agent. The environment (i.e., the parent agent) is responsible for communication between agents and for responding to queries from agents. It also determines the types of actions which may be ordered by child agents. An agent informs the environment of what it expects to be done (e.g., creation of a new agent), but it does not know how it will be done. The decision of how to execute the action is made by the parent agent.

Both strategies and agents can have named properties. They can be referenced during runtime by their names in order to access, modify, or monitor their values. For each class, a list of its properties may be retrieved, and each named property of the instance may be accessed in a uniform way. The properties may be of any type.

In our approach, all of the platform objects can be loosely coupled [15], which means that the dependencies and associations between them are realized through well-defined interfaces and not by concrete implementations. This hides realization details and allows for the exchange of dependent classes without code modifications. This is essential in terms of the flexible component-assembly framework. Properties of these components can also be configurable using the input configuration.

Aggregate agents (agent environments) have the ability to retrieve actions by resolving them by name or type during run-time. Moreover, environments can provide different implementations of the same action, depending on the level in the agent structure.

The platform has been implemented as a tool for a quick construction of high-performance, specialized systems. It does not need interoperability with other systems. It does not follow the standard, FIPA-compliant behaviors usually shown by multi-agent systems. This approach is justified by its planned usage and found validation in the past years.

4. Agent-based middleware for Data Integration

In this section, we provide a short review of selected data-integration software, and we present our own solution based on the multi-agent platform AgE. We explain the general structure of the system along with the description of clearly-separated agent roles and the type of system used to wrap processed data.

4.1. Related systems for Data Integration

Heterogeneous data integration is a theme important for very diverse research domains, from database warehouses for business intelligence to medical systems supporting health center operation. One on the main problems is how to resolve the semantic heterogeneity of the data sources. Several approaches exist for this; for example, data warehousing, federated database systems, and mediator-based approaches. We will discuss a few of them.

X-SIRD authors present an XML-based system for integrating heterogeneous relational data [9]. Its architecture comprises of a few elements. Interaction with data sources is performed using specialized wrappers. They translate source-specific schemas into a representation called *local Translated Schema*. These translated representations are integrated by a *Mediator*. The result of this process is a *global integrated schema*. The Mediator is also responsible for a translation of user queries into specialized relational representations.

A similar approach can be found in ACGC (Advancing Clinico-Genomic Trials on Cancer) [10]. The architecture of the system consists of a data access layer that is responsible for handling the heterogeneity of the used database systems. The authors use SPARQL for querying the data. Integration is performed by the *Semantic Mediator* and the *Master Ontology*.

An agent-based methodology was used in SMAMDD [13], for example - a platform aimed at knowledge extraction using *model integration* methods. In this system, a group of agents applies common machine learning algorithms to subsets of data. The system provides distribution, and the agents communicate using messages. The data processed by agents is used to generate sets of integration rules.

The concept of integrating services is shown, for example, in the system OntoMat-Service [1]. Services described with WSDL are mapped to the user-defined ontology. This system makes it possible to integrate much more diverse systems than SQL-based wrappers. However, the system is not automatic and requires manual specification in the way the sources should be integrated. Users need to annotate WSDL with terms from the ontology.

Another ontology-based approach can be found in X2R [11]. In this case, data is extracted from sources, transformed into RDF triplets, and than loaded into an RDF graph. The system works with XML, LDAP, and relational data. It creates a common view of these sources with a common ontology and with defined integrity constraints. The user does not have to care for the underlying data structure. The project uses the D2RQ software for relational-to-RDF conversion and its own solutions for XML and LDAP sources. The resultant RDF database can be queried using SPARQL queries.

An interesting approach using Artificial Neural Networks was presented in [8]. The authors discuss model for integrating electronic health records across multiple health care providers (e.g., hospitals) that use private clouds. To facilitate data integration, the authors introduce a central cloud comprised of an index of identifiers, a router service, a storage area, and an access log. Health records created in private clouds are then mapped into the global index that points to the originator of the record. Records can be modified by many health care providers; hence, there is a need to address their heterogeneity. An Artificial Neural Network is used for this purpose; for example, it learns data representations and unique features that can be used to map local parameters to the global schema.

4.2. System concept

The basic goal of the created system is to provide the data- and task-oriented workflow for collecting and integrating data from a wide range of diverse (in terms of the data model) services. The user is separated from the actual data providers by the abstract type system and agents that operate on it. The agent-based approach and AgE platform advantages are used extensively. They allow us to introduce a clear separation of concerns and make it possible to rapidly build topic-related versions of the system with little configuration. The framework is completely independent of the data it processes, and all scenario-related components are provided by the user during configuration. Data types are extended by the inheritance and data processing elements extensively use the *strategy* design pattern.

The notion of "integration" in the system involves the creation of a common view over all data sources and not creating copies of their data. All of the actions are performed *online*; i.e., on real sources and data, as opposed to a common pattern of doing an *offline* integration; i.e., collecting and merging data into one unified database for later querying. However, it does not mean that such a workflow is not possible. It is simply not a default way of work with the system.

The system consists of four loosely-coupled (see Figure 2) and well defined components:

1. services and data, i.e. the underlying data sources,
2. the middleware, i.e. the agent system being described in this paper,
3. the common type system, i.e. classes that wrap around the data from data sources,
4. views, i.e. graphical interfaces for users to interact with the system.

The user interacts with the system, providing it with tasks and initial data elements that are treated as parameters (e.g., for queries). The user interface additionally provides a means of control over the system.

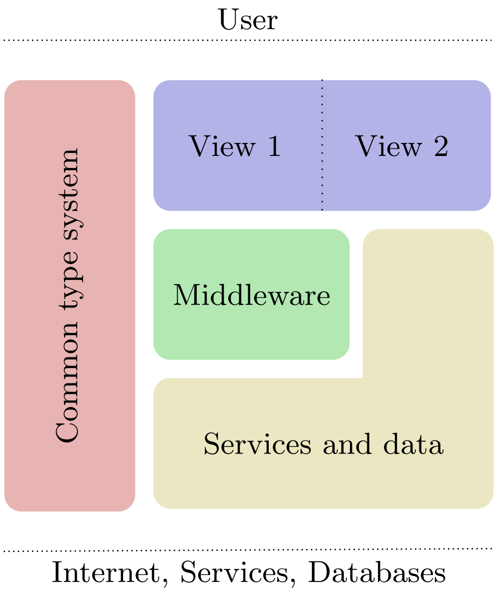

Figure 2. Layers in the system. The agent-based middleware is located in the center. It uses the same common type system (left) as the user interface (top) and services and data strategies (bottom).

Tasks created by the user are put into the agent system that performs two types of operations:

- the management of data (by inspecting data and tasks and delegating them to other agents), and

- the execution of demanded actions (including their selection, configuration, and fault recovery).

Some actions may be implemented in many ways and access diverse external resources. For example, one of the actions that performs look-ups of personal data may use LinkedIn API, but there may be another one that performs a search using Facebook. This dynamic and pluggable mechanism is implemented in the "Services and data" layer of the system.

All of these components use a common type system that offers uniform access to all data already obtained and stored in the system. It is thoroughly discussed later.

The system additionally allows us to divide processing into *issues*. An *issue* is intended to be an order for a separate part of data processing usually centered around some instance of a data object and data related to it. An issue is usually created by the user.

5. System implementation

The idea described in the previous section resulted in a working implementation of the data integration framework that follows the agent-based approach. The application has been built on top of AgE platform and is continuously being developed.

In this section, we explain the general structure of the system. We also clearly describe the separated agent roles and the type of system that is used to wrap the processed data.

5.1. Tasks, issues and data types

The diagram in Figure 3 shows how tasks, issues, and data types relate to each other. A task is described with four parameters:

- a function to perform (an action that is declared by some agent in the system; e.g., that it is able to search for personal data),
- configuration parameters to this function (e.g., names of source data providers),
- links to data to use during execution (e.g., the name of the person to look for),
- an issue identifier (assigned by the system).

The interfaces and classes that are used to describe tasks, issues and data are:

IssueIdentifier is a simple unique tag for an issue.

FunctionIdentifier is a definition of function offered by the system, `DefaultFunctionIdentifier` is a basic implementation of it.

DataElement is an interface that needs to be implemented by every type from the type system, `AbstractDataElement` provides its basic implementation with some additional utilities for interacting with data.

Task is a definition of a task ordered by the user and `DefaultTask` is its basic implementation.

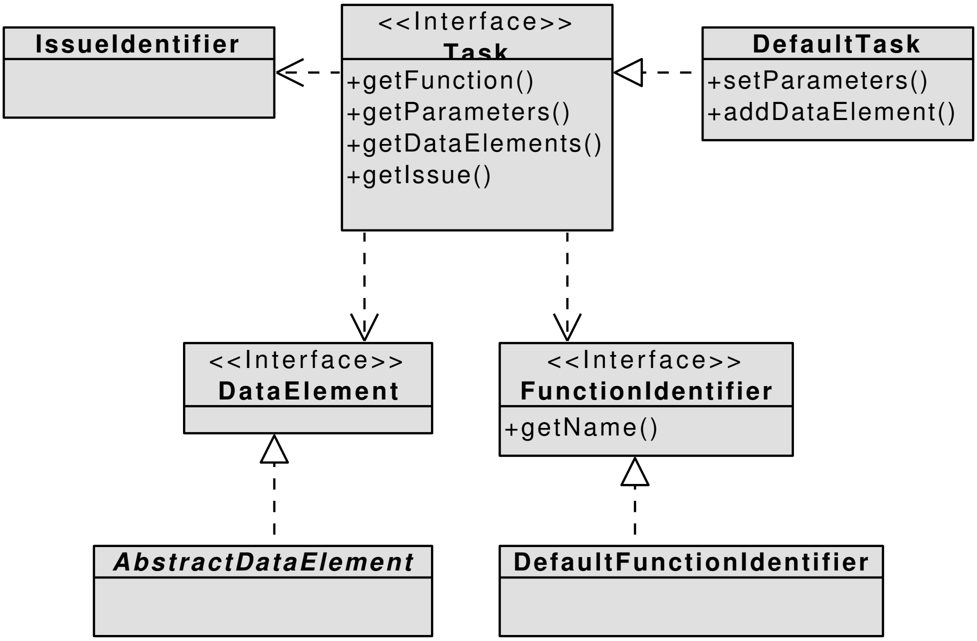

Figure 3. The class diagram of entities representing tasks and data. For a detailed description, see Section 5.1.

Example. Suppose, that we want to look for real estate offers in some *City A* matching the following criteria:

1. type of the estate is building,
2. minimum surface is 200 m^2.

Depending on the available agents, the task that would describe these requirements can be defined in various ways. For simplicity of explanation, we assume that there is an agent that understands all of these criteria. The task could appear as follows:

- the function is `search-estate`,
- in parameters we want to say that we are interested in all data sources, so we define the parameter *source* as `*`,
- as the link to data we create an object that matches the above criteria,
- the issue identifier will be automatically assigned.

5.2. Agents roles

The current implementation defines three possible functional roles for agents. Their interdependence is shown in Figure 4.

System agents: they provide a basic system functionality, like resolving new issues, error handling, monitoring (represented by `ControlAgent` in the diagram).

```
Estate estate = new Building();
estate.setSurface(new GreaterThan(200));
DefaultTask searchEstate = new DefaultTask(
        new DefaultFunctionIdentifier("search-estate"));
searchEstate.setParameter("source",
        new DefaultTask.Parameter("*"));
searchEstate.addDataElement(estate);
```

Listing 1. Creation of a task. First, the data element is created and put into `estate` variable. Then, it is filled with requirements. After preparing the data, the task is created with a name of the function to perform, a `source` parameter, and the data element.

Issue agents: responsible for keeping track of a single issue and delegating tasks to action agents on the basis of their capabilities. Such an agent retrieves a task and data from the pool, inspects it and then explicitly requests (with messages) a chosen action agent to perform an action specified in the task. They are created on-demand by the `ControlAgent`.

Action agents: they implement the actual executive part of the functionality. Upon receiving the task from an issue agent, they locate a strategy that can be used to fulfill it and then execute it over data bound to the task.
However, they are not constrained to only executing strategies. They can perform any action over the data they receive: merge, simplify, verify, etc. This flexibility and extensibility of action agents is the main reason why the system is well-suited for the task.

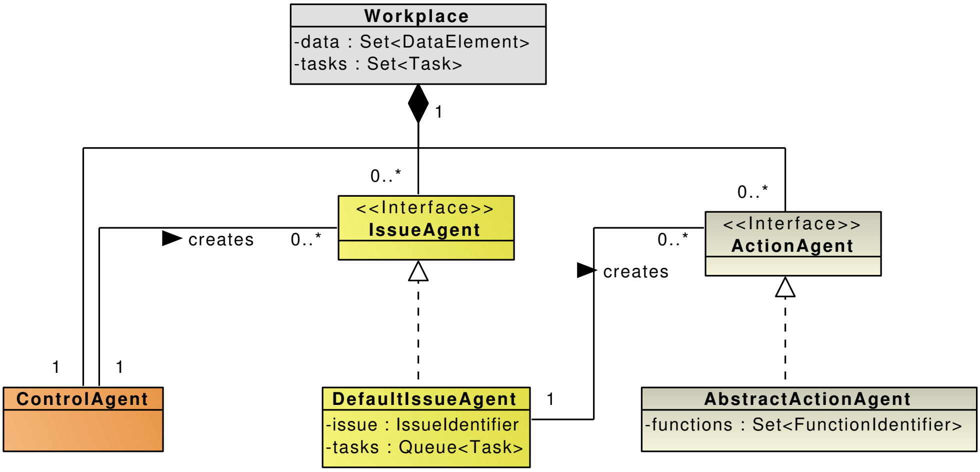

Figure 4. The class diagram of agents existing in the system. System agents are co loured in orange, issue agents – in yellow, and action agents – in gray-yellow. The workplace is a top-level agent required by the AgE.

Issue agents are identified by a runtime-generated issue identifier that represents a topic that they are taking care of. Action agents are described during creation (implementation) with tasks they can perform (called "capabilities") and data types they can operate on.

The sample action agent `PersonalDataAgent` could be described by the task *find personal data* and the data type `Person`. Figure 6 (presented later) shows that aspect of the system.

During the initialization of the system, only `ControlAgent` is required to be instantiated. All other agents are created on-demand when they are requested by the user or another agent.

Example of `ControlAgent`. Listing 2 shows the actions performed by the `ControlAgent` when new task is created. The agent starts with the creation of a new issue identifier, then instantiates a new `IssueAgent` that will be executing this task and adds it to the system.

```
void handleNewTask(Task task) {

        IssueIdentifier ii = new IssueIdentifier();
        task.setIssue(ii);
        IssueAgent issueAgent = createNewIssueAgent(ii);

        doAction(
                new SingleAction(new ParentAgentAddressSelector(),
                new AddAgentActionContext(issueAgent)));
}
IssueAgent createNewIssueAgent(
                IssueIdentifier issueIdentifier) {

        IssueAgent issueAgent =
                instanceProvider.getInstance(IssueAgent.class);
        AgentAddress address = new AgentAddress(
                "Issue\$" + issueIdentifier);
        if (getAgentEnvironment().registerAddress(address)) {
                issueAgent.setAddress(address);
        }

        issueAgent.setIssue(issueIdentifier);
        issueAgent.init();
        issueRegistry.put(issueIdentifier,
                issueAgent.getAddress());

        return issueAgent;
}
```

Listing 2. Sample actions performed by `ControlAgent` when new task is created. `handleNewTask` is called when a new task arrives and it performs an initialization of an issue and an issue agent. `createNewIssueAgent` contains the code used for initialization of this agent.

The instantiation of `IssueAgent` contains the following steps:

1. obtaining the instance from the instance provider,
2. creation, registration, and setting of the new address,
3. adding the issue to the agent,
4. calling post-construct initialization,
5. registration of the agent in *issue-to-agent* map.

5.3. Type System

This type of system is based on Java classes. Each system type is represented as a separate Java class, and all of its basic properties are implemented as fields in such classes available through standard getter and setter methods. The only requirement is to inherit from a provided base type. Figure 5 presents a fragment of a sample type *Person* and a corresponding Java class.

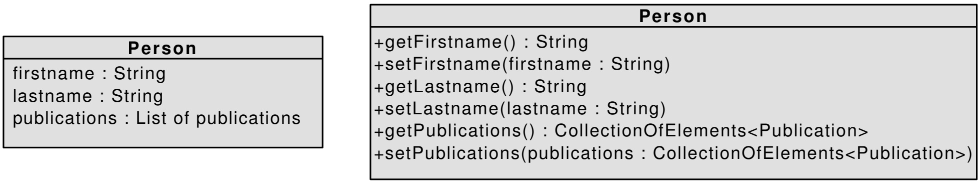
Person
firstname : String
lastname : String
publications : List of publications
Person
+getFirstname() : String
+setFirstname(firstname : String)
+getLastname() : String
+setLastname(lastname : String)
+getPublications() : CollectionOfElements<Publication>
+setPublications(publications : CollectionOfElements<Publication>)

Figure 5. Correspondence between a logical type (on the left) and its Java implementation (on the right). Parameters are simply converted into setter-getter pairs. Lists of elements are implemented as special collections.

However, the possibility that some strategies can offer more data than were predicted by the system during compilation time is also considered. Therefore, data types are *dynamic* in the sense of a possibility to extend them in the runtime by adding arbitrary properties of any type. This mechanism can also be used for providing some service-specific data to the user, like a profile photo. These properties can be easily obtained by any agent or user interface that is able to use the base type.

Moreover, a more-complex introspection mechanism is being planned as a way to allow an agent or a user to inspect or create completely new types in the runtime.

Data Type Example. Listing 3 shows how a sample type can be implemented.

6. Case Study

We present an implemented, sample use case (chosen for its simplicity) in which we want to collect information about scientists from popular, global, and organization-specific sites and databases. In this scenario, we can use data from (e.g.,) services providing general personal and professional information (LinkedIn) or sites offering strictly science-related data (like DBLP or internal university publication databases).

```
class Book extends AbstractDataElement {

        private String title;

        private CollectionOfElements<Author> authors;

        // Other parameters omitted.

        public Book() { super(); }

        public void setTitle(String title) {
                this.title = title;
        }

        public String getTitle() { return title }

        public void setAuthors(
                        CollectionOfElements<Author> authors) {
                this.authors = authors;
        }

        public CollectionOfElemeents<Author> getAuthors() {
                return authors;
        }

        // Other methods omitted.
}
```

Listing 3. Fragment of a sample implementation of a type describing a book.

Although this kind of a use case may look simple, there are enough interesting tasks and problems to observe, describe, and analyze the real behavior of the system.

6.1. Base Scenario

The base scenario (as seen from the user interface) consists of the following steps:

1. *Gathering general personal data about a person from all possible (known) sources.* At this point, the user feeds a query to the system. A query comprises only of a full name of some person; e.g., "John Smith". A result for such an action is usually a list of possible matches (because of the ambiguity of the provided data). There are several ways to solve the problem of multiple matches. In the simplest case, we require the user to choose one match (i.e., manual resolution). However, the choice could also be performed by an agent that can rate each result and select the best one. Alternatively, we try to perform searches for all returned matches to collect as much data as possible.
2. *Getting data about publications from the selected sources for the chosen person and merging it into one list.*

The user orders a task to obtain publications of the person. However, when using multiple heterogeneous sources, results must be merged to create a single registry. The merge can be performed in various ways: by joining lists (possibly leading to many duplicates) or in a more-sophisticated way, trying to recognize the same publication in different sources.

6.2. Scenario Implementation

Such a scenario translates into implementation in following way:

- We introduce types that are required to describe the domain (e.g., Person, Publication), see example in Listing 4.
- *Action agents* are implemented for every type: Personal Data Agent and Publications Agent. They perform operations on (related to) these types. Sample implementation of how agents perform these operations is shown in Listings 5 and 6:
  1. The agent obtains the parameters from the task.
  2. Then, it checks what sources it should consult. For the `*` it checks all available strategies, for other values it obtains them by the name.
  3. The `for` loop iterates over all data elements from the task and executes strategies on them. In this way, we get separate lists of publications for every `Person` object.
  4. Strategies are called (the agent does not care about a strategy implementation, only about its interface) and return results or throw exceptions. Results are collected by the agent and sent using messages to the agent that ordered the task.
- We may implement *merge* action in two ways:
  - either the Publications Agent may be able to merge publications, or
  - another agent can perform a general merge operation with help of a concrete strategy.
- For each external service that we use, we create strategies: Personal Data Search for (e.g.,) LinkedIn or DBLP. and Publications Search for DBLP and Google Scholar. These strategies may be implemented as separate classes for each service, separate classes for each type, or even as separate classes for each service-type pair.

Figure 6 presents some selected classes involved in the realization of the first step and their dependence on used types. A connection between a strategy and a type is declared simply by using a Java static-type system. On the other hand, action agents need a more complex way to confirm their ability to operate on data. For the sample operation `findPersonalData`, the two most-basic checks are: the recognition of the requested task identifier and the type correctness of all data provided by the user.

```
class Publication extends AbstractDataElement {

        private String title;

        private CollectionOfElements<Person> authors;

        // Other parameters omitted.

        public Publication(String title) {
                super();
                this.title = title;
        }

        public String getTitle() { return title; }

        public void setTitle(String title) {
                this.title = title;
        }

        public CollectionOfElements<Person> getAuthors() {
                return authors;
        }

        public void setAuthors(
                        CollectionOfElements<Person> authors) {
                this.authors = authors;
        }

        // Other methods omitted.
}
```

Listing 4. Fragment of the code for the Publication type.

6.3. Execution of Scenario

Figure 7 shows an actual execution of the first step. Its detailed description is presented below.

1 The user prepares a task specification that consists of two parts:

- a task identifier, represented in diagram as text `find-personal-data` – this is a required part,
- initial data to operate on (e.g., a name of the person).

The task is placed into the system in the `TaskAndDataPool`, which is responsible for communication between the user and the system.

1.1 Then, all issue agents are notified about it.

1.2 The agent responsible for this task obtains it from the pool (or, in case of a new issue, a new agent is created by `ControlAgent`).

```
void executeFindPublications(
                Task task, TaskMessage message) {

        Map<String, Parameter> parameters =
                task.getParameters();

        String source = parameters.get("source")
                .getContent(String.class);

        List<PublicationsSearchStrategy> strategies =
                new ArrayList<PublicationsSearchStrategy>();

        if ("*".equals(source)) {
                strategies.addAll(
                        findAllStrategies(
                            PublicationsSearchStrategy.class));
        } else {
                strategies.add(
                        findStrategyWithName(
                                PublicationsSearchStrategy.class,
                                "PublicationsSearchVia" + source));
        }
```

Listing 5. Fragment of the code for the `ActionAgent` type. The presented method is responsible for executing strategies that are able to find publications of the person in various sources. For the detailed description, see the main text.

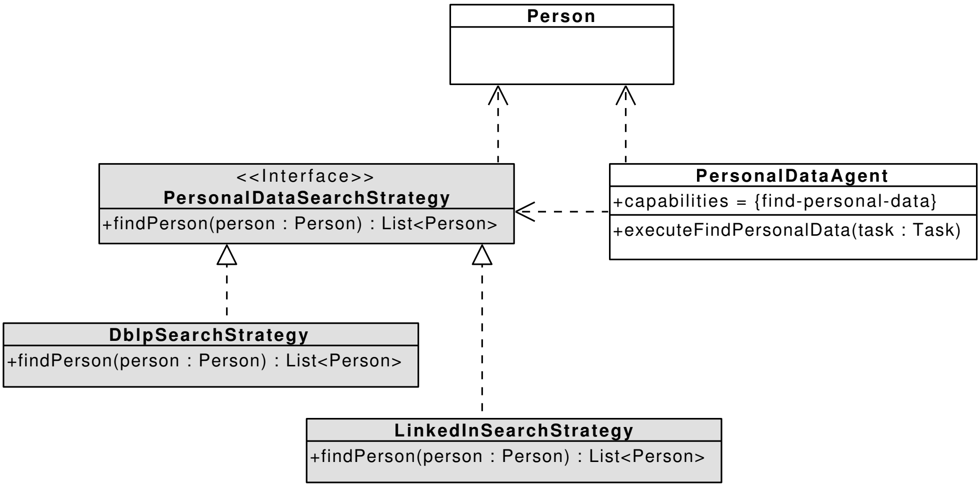
Person
<<Interface>>
PersonalDataSearchStrategy
+findPerson(person : Person) : List<Person>
PersonalDataAgent
+capabilities = {find-personal-data}
+executeFindPersonalData(task : Task)
DblpSearchStrategy
+findPerson(person : Person) : List<Person>
LinkedInSearchStrategy
+findPerson(person : Person) : List<Person>

Figure 6. A class diagram showing (selected) entities involved in the realization of the task. Strategies are in gray.

```
    for (Person person : task.getDataElements(Person.class)) {
        ListOfPublications publications =
            new ListOfPublications(new ArrayList<
                Publication>());

        try {
            for (PublicationsSearchStrategy strategy :
                strategies) {
                publications.addAll(
                    strategy.findPublications(
                        person));
            }
            person.setPublications(publications);

            if (publications.isEmpty()) {
                sendMessage(
                    new DataElementMessage(
                        getAddress(),
                        DataElementMessageType.
                            RESPONSE,
                        new NoResults(task)),
                    message.getSenderAddress());
            } else {
                sendMessage(
                    new DataElementMessage(
                        getAddress(),
                        DataElementMessageType.
                            RESPONSE,
                        person),
                    message.getSenderAddress());
            }

        } catch (StrategyException ex) {
            sendMessage(
                new DataElementMessage(
                    getAddress(),
                    DataElementMessageType.RESPONSE
                        ,
                    new Error(ex)),
                message.getSenderAddress());
        }
    }
}
```

Listing 6. Fragment of the code for the `ActionAgent` type (continued). The presented method is responsible for executing strategies that are able to find publications of the person in various sources. For the detailed description, see the main text.

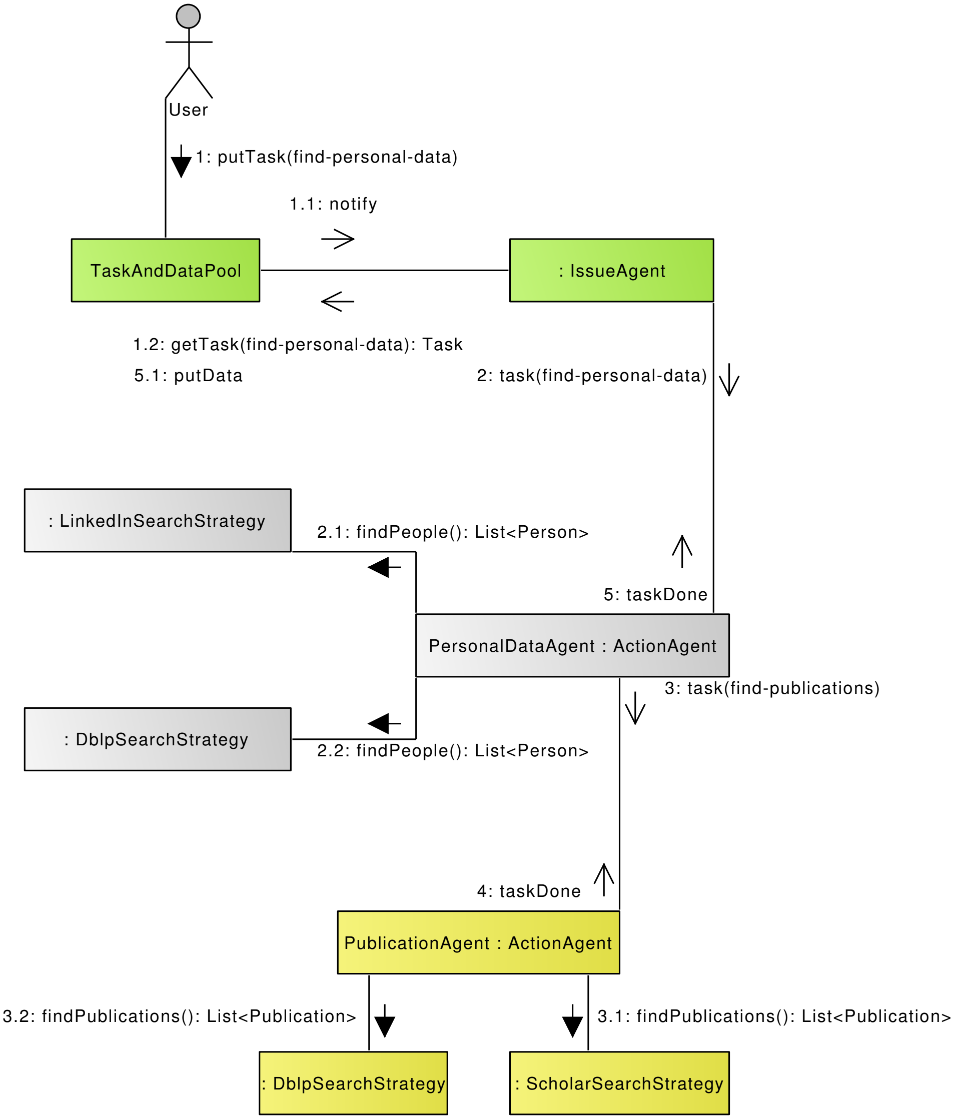
User
1: putTask(find-personal-data)
1.1: notify
TaskAndDataPool
: IssueAgent
1.2: getTask(find-personal-data): Task
5.1: putData
2: task(find-personal-data)
: LinkedInSearchStrategy
2.1: findPeople(): List<Person>
5: taskDone
PersonalDataAgent : ActionAgent
3: task(find-publications)
: DblpSearchStrategy
2.2: findPeople(): List<Person>
4: taskDone
PublicationAgent : ActionAgent
3.2: findPublications(): List<Publication>
3.1: findPublications(): List<Publication>
: DblpSearchStrategy
: ScholarSearchStrategy

Figure 7. A communication diagram showing operations needed to execute a task. The personal data search is shown in gray, the publications search – in yellow. The user interaction is shown in green.

2 The issue agent locates an action agent that can handle the specified task and delegates its execution to this agent. In this particular case, `IssueAgent` looks for `ActionAgent` that is capable of performing the `find-personal-data` task.

2.1 `PersonalDataAgent` inspects both the task specification and provided data and calls relevant strategies. First, `LinkedInSearchStrategy` is called.

2.2 The second one called is `DblpSearchStrategy`, which is able to obtain data from DBLP.

3 The type `Person` contains, for example, a list of publications. To fill this attribute, `PersonalDataAgent` creates a task `find-publications` and sends it to `PublicationAgent`. This task contains the `Person` object for which the publications should be found.

3.1 `PublicationAgent` executes the `ScholarSearchStratgy` that uses Google Scholar to find publications.

3.2 The second executed strategy is `DblpSearchStrategy`.

4 Lists of publications are merged and returned back to `PersonalDataAgent`.

5 After that, `PersonalDataAgent` merges all data and sends the results (a list of `Person` instances) to the requesting issue agent.

5.1 The issue agent finishes the realization of the task by putting the results to the pool.

At some of these points, several data conflict situations may occur when two or more different sources disagree on a piece of data; for example:

- dates of publications can vary,
- spelling of the surname can be distorted (e.g., lack of diacritics),
- etc.

In such situations, it is up to the agent implementation to make a decision on how to resolve these conflicts. However, what we used in test scenarios, is to multiply data for all possible conflicts (for example, when two versions of the same publication show up, we create two separate objects) and leave the conflict resolution for separate agents.

6.4. Extensions

As noted previously, this scenario can be extended in many ways. Some considered (and partially-realized) examples are:

- The user can spot another person in the list of authors of one publication. Data for this person can be retrieved, and then a list of publications common with John Smith can be generated. It should be noted that such processing that is independent of the first one is regarded as a good point to introduce a new issue into the system.
- There are cases when data in one service is outdated (e.g., a phone number has changed). A system can automatically perform (or with the user's help) a correction of such values on the base of data gathered from other services.

7. Conclusions

We have presented a proof-of-concept architecture and implementation of the agent-based framework for data mining. Introduced conceptually in [12], the framework has been implemented and, thus, verified in regards of suitability to the given task. The presented version operates on data related to personal and professional profiles of scientists. The sources for this kind of data are heterogeneous yet usually well-structured. We have only shown the version that operates in the first of the proposed "modes" of work – as a system fully controlled by the user. Thus, we have reached the goal of using AgE as a base of the data-integration framework.

Usage of the agency and component models from the AgE platform brought significant gains to the system. It is extensible, as we can easily add strategies specialized for new data sources without even recompilation. Both agents and data types can work with dynamic data that doesn't have a fully-known structure. An interesting property of the system is a mapping between the emerging tree of agents and the graph of mined data.

Although the system currently has limited functionality, extensions are planned for the future. For example, we want to extend the system with further strategies, other scenarios, and implement more-advanced modes of operation: semi- and fully-autonomous. Moreover, we also want to provide out-of-the-box strategies for data caching and local storage that would speed up querying actions and allow users to save the current state of the system for future use.

We have in mind other, more interesting scenarios in which such a system could be really useful. Two of these ideas are presented in the following paragraphs.

Weather forecasting data mining. This scenario would require implementation of agents and strategies for collecting data from meteorological equipment, such as thermometers, barometers, anemometers, or hygrometers. These agents would be simple (in terms of their performed functions). On top of them, there would be agents capable of more-sophisticated actions: historical data collection, statistical analysis, prediction, graphical presentation, etc. An issue in such a system could be, for example, "plot all lightnings in a given period of time on a map of Europe".

Criminal Analysis. This is another scenario with many different data sources. Suppose we want to have a functionality of tracking the behaviors of a particular person given some of their characteristics (e.g., a name or a photo of their face). For this, we would need agents that are able to look at standard, website- or database-based sources as well as different ones able to gather data from live or archival video sources, analyze them, and extract images of a particular individual. These would be agents responsible for obtaining the data. Other ones could be, for example, responsible for merging different facts (e.g., location of credit card usage), etc.

Acknowledgements

The research leading to these results received funding from research project No. O ROB 0008 01 "Advanced IT techniques supporting data processing in criminal analysis", funded by the Polish National Center for Research and Development.

The research presented in this paper was also partially supported by AGH University of Science and Technology statutory project.

References

[1] Agarwal S., Handschuh S., Staab S.: *Surfing the Service Web.* In: *International Semantic Web Conference 2003*, pp. 211–226. Springer, 2003.

[2] Byrski A., Kisiel-Dorohinicki M., Dajda J., Dobrowolski G., Nawarecki E.: Hierarchical multi-agent system for heterogeneous data integration. In: *Intelligent decision systems in large-scale distributed environments*. P. Bouvry, H. Gonzalez-Velez, J. Kolodziej, eds., Springer Verlag, 2011.

[3] Byrski A., Kisiel-Dorohinicki M., Nawarecki E.: Agent-Based Evolution of Neural Network Architecture. In: *Proc. of the IASTED Int. Symp.: Applied Informatics*, M. Hamza, ed. IASTED/ACTA Press, 2002.

[4] Cao L.: *Introduction to agent mining interaction and integration.* In: Data mining and multi-agent integration, pp. 3–36. Springer, 2009.

[5] Cao L., Gorodetsky V., Mitkas P. A.: *Agent mining: The synergy of agents and data mining. Intelligent Systems, IEEE*, vol. 24(3), pp. 64–72, 2009.

[6] Faber Ł., Piętak K., Byrski A., Kisiel-Dorohinicki M.: *Agent-Based Simulation in AgE Framework.* In: Advances in Intelligent Modelling and Simulation, pp. 55–83. Springer, 2012.

[7] Gamma E., Helm R., Johnson R., Vlissides J.: *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[8] Gul O., Al-Qutayri M., Vu Q. H., Yeun C. Y.: Data integration of electronic health records using artificial neural networks. In: *Internet Technology And Secured Transactions, 2012 International Conferece for*, pp. 313–317, 2012.

[9] Li S., Zhang D. H., Zhou J. T., Ma G. H., YangR H.: *An XML-Based Middleware for Information Integration of Enterprise Heterogeneous Systems. Materials Science Forum*, **vol. 532**, pp. 516–519, 2006.

[10] Martín L., Anguita A., Maojo V., Bonsma E., Bucur A. I. D., Vrijnsen J., Brochhausen M., Cocos C., Stenzhorn H., Tsiknakis M., Doerr M., Kondylakis H.: *Ontology Based Integration of Distributed and Heterogeneous Data Sources in ACGT.* In: HEALTHINF (1), L. Azevedo, A.R. Londral, eds., pp. 301–306. INSTICC – Institute for Systems and Technologies of Information, Control and Communication, 2008. ISBN 978-989-8111-16-6.

[11] Myłka A., Myłka A., Kryza B., Kitowski J.: Integration of Heterogeneous Data Sources in an Ontological Knowledge Base. *Computing & Informatics*, vol. 31 (1), 2012.

[12] Nawarecki E., Dobrowolski G., Byrski A., Kisiel-Dorohinicki M.: Agent-based integration of data acquired from heterogeneous sources. In: *Proc. of CISIS 2011, Seoul, Korea*. 2011.

[13] Paula A. C. M. P. d., Avila B. C., Scalabrin E., Enembreck F.: Multiagent-Based Model Integration. In: *Proceedings of the 2006 IEEE/WIC/ACM international conference on Web Intelligence and Intelligent Agent Technology*, WI-IATW '06, pp. 11–14. IEEE Computer Society, Washington, DC, USA, 2006. ISBN 0-7695-2749-3. `http://dx.doi.org/10.1109/WI-IATW.2006.96`.

[14] Pietak K., Wos A., Byrski A., Kisiel-Dorohinicki M.: Functional Integrity of Multi-Agent Computational System Supported By Component-Based Implementation. In: *Proc. of Holomas 2009, Linz, Austria (accepted for printing)*. 2009.

[15] Stevens W. P., Myers G. J., Constantine L. L.: Structured design. *IBM Systems Journal*, vol. 13 (2), pp. 115–139, 1974.

Affiliations

Łukasz Faber

AGH University of Science and Technology, Department of Computer Science, Krakow, Poland, `faber@agh.edu.pl`

Received: 23.04.2014
Revised: 29.07.2014
Accepted: 29.07.2014