Ontology-based Management of a Network for Distributed Control System

Dariusz Choiński, Michał Senik, Bartosz Pietrzyk
Silesian University of Technology
Faculty of Automatic Control, Electronics and Computer Science
Gliwice, Poland
e-mails:{dariusz.choinski@polsl.pl, michal.senik@polsl.pl, bartosz.pietrzyk@student.polsl.pl}

***Abstract*—In this paper, we propose an ontology-based analysis and management of a domain in which initially unspecified number of Industrial Automation (IA) data servers compatible with Open Platform Communication (OPC) specification are available for processing. The result of the performed analysis is an ontology-based Multi-Agent System (MAS) compatible with the Foundation for Intelligent Physical Agents (FIPA) specification that is capable of reliable management in a nondeterministic environment conditions in which not only data servers can change over time in number, but their hierarchical data structures as well. The first step of the presented analysis involves explicit knowledge formalization by means of a Unified Modeling Language (UML). The goal is to obtain complete, hierarchically structured, constrained, human readable ontology in a UML class diagrams format, representing both topology of the domain and its dynamic process control data. The second step is to translate the obtained UML class diagram into First Order Logic (FOL) expressions. The goal is to establish more detailed domain knowledge as well as to check consistency and assure confidence of the derived UML class diagram. The third step is to explicitly formalize the domain ontology in a machine interpretable extensible markup language (XML) schema (XSD), based on the derived UML class diagram. The goal is to have a possibility of automatic generation of hierarchically structured, class source code that, together with parent UML model and FOL expressions, will serve as a baseline for the domain integration system development.**

Keywords-Network management; Multi-Agent Systems; Ontology; FOL; FIPA; OPC; XML; hybrid systems; concurrent programming.

I. Domain Integration System Concept

In the presented domain integration systems, there is a predefined number of database servers, storing both historical and real-time hierarchical data and a nondeterministic number of OPC Data Access (DA) IA servers, serving as source of real-time, process control data. Each server can be accessed by numerous different client applications. OPC standard [1] was established as a method for efficient communication between automation devices and systems. One of the basic specifications is the OPC DA which defines the communication between the client and the server hosting the real-time process data. Data Access Clients have access to data from the automation system via Data Access Servers. The communication interface between the client and the server is completely independent of the physical data source. The OPC DA specification also defines two main structures for describing data shared by the server. These are the namespace (Namespace) and the OPC objects. The namespace provides hierarchically structured, control process data. The structure of the OPC objects is flat and is created by the client application. The OPC object, within the established structure, is attributed to identifiable characteristics, such as: the value of the corresponding variable, the time of measurement, the quality measurement and others [2][3]. Cooperation with OPC DA servers is established through the Java Native Interface (JNI) wrapper, which is based on the native OPC DA library. Cooperation with database servers is established through the NHibernate database entity framework wrapper [4].

The presented domain integration system concerns usage of partially autonomous, proactive and self-managed MAS as an integration solution. Resulting from such an approach, there is a number of different types of agents that have to cooperate together, performing their highly specialized set of activities, in order to perform integration tasks correctly [2][3][4]. The idea of cooperation, as communication between agents, using serialization of speech acts, requires the creation of an ontology that allows partitioning of the message by the agent, so that the intentions of the objectives are clear and unambiguous. At the same time, it should be a feature of the ontology which is easily processed by both man and machine. Therefore, the rationale for the MAS domain integration system development is an OPC-based ontology. The structure of the transmitted information is particularly important in the use of Multi-Agent technology in control and design of control systems. The set of concepts, which in this case is a description list of the data points, used and controlled variables in the whole process of designing the control system and its software is practically constant. What changes, mainly, is the structure of mutual connections and the structure of the information used for decision making and activities related to the control. An obvious advantage of the system of agents that communicate using messages based on a universal definition of a FIPA standard [5] is the ability to remotely boot services, regardless of the particular software implementation. Within the commonly used protocols in the domain, it is necessary to know the structure of instances of the individual objects performing services or storing information and for the MAS this knowledge is formalized by means of ontology.

The rest of this paper is organised as follows. The second section presents a high level, abstract description of an automation system analysis based on Petri Nets and ontologies. The third section proposes a solution to validate the obtained ontologies' UML class diagrams by means of additional FOL formulas. The fourth section presents general rules of conversion from UML class diagrams to FOL formulas. The fifth section presents a sample conversion from ontology-based, UML class diagrams to corresponding FOL formulas, proving the correctness of obtained ontology.

II. Dynamic Domain Events Response

Resources, as a part of the greater domain, are usually treated as a source of a variety of different pieces of information that need to be integrated and analyzed in order to be meaningful. These data are usually stored in a resource's hierarchical memory structures. The process of the fast data collection and analysis of those pieces of information is an essential activity for each integration system resulting in a proper and efficient domain control and maintenance. Both the data structure and the resource allocation are essentially dynamic, and thus, can change over time. The inability to synchronize automatically with each other causes confusion and various different data integrity problems. It is worth mentioning that the synchronization process always depends on various different reconfiguration and reorganization processes. To achieve such an autonomous and proactive domain integration system, additional detailed analysis of a domain from the perspective of each underlying integrated resource is required. The performed analysis ought to treat a resource as a finite state machine and result in explicitly defined state set that would cover its whole functionality. It is important to notice that those states must not lead to the integration system deadlock in any way as each state should have its transition conditions, predecessor and successor states clearly defined. To achieve this, a states reachability graph has to be created, and, since Petri Nets [6] is a widely accepted tool to perform such an analysis, it is a fairly easy and straightforward task (Figure 1).

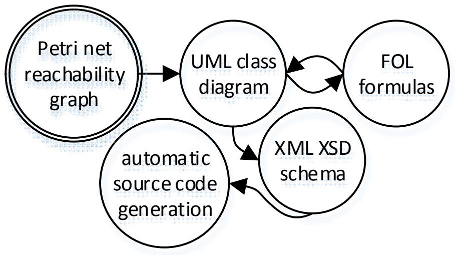
Petri net reachability graph
UML class diagram
FOL formulas
XML XSD schema
automatic source code generation

Figure 1. Iterative approach to domain integration system engineering.

However, this is only the initial step towards domain integration system creation. Since Petri Nets is still just a human only interpretable tool; a more sophisticated and expressive tool, such as ontology, is required. Ontology possesses enough capabilities to formalize the gathered domain knowledge in both human and machine readable format. It can describe both static properties of the system and its runtime states in which the system can reside, as well as the conditions of transitions between those states. Ontology can model anything starting from the tiniest domain element to vast hierarchies of domain resources. Based on the ontology, parts of the domain integration system can share their knowledge and cooperate to solve integrated system's problems [2][3]. Technically, ontology is a set of very simple and much more complex rules, such as concepts, predicates and actions, which can be created in many available programming languages, such as Java, C# or XML [2][3]. Each ontological expression can be organized in a hierarchical structure, which means that simpler entities can be nested in more complex ones.

III. Ontology-based Analysis and Engineering

In the proposed domain, integration system ontology is a fundamental element serving as a formal domain and its integration system specification, communication medium and domain integration system implementation base. Based on such an ontology, a detailed analysis can be performed, which allows to obtain peek quality of a domain integration system. Ontology-based analysis and engineering activities, however, require a good and comprehensive visual tool that could enhance ontology expressiveness and improve obtained results quality. For many experts in this field, UML and its class diagrams especially, have gained much credit. However, UML class diagrams along with their many useful features tend to contain also a lot of implicit knowledge that concerns declaration of classes and their methods, classes' generalization and accessibility level. Implicit, hidden knowledge always causes unnecessary problems during domain analysis and its integration system implementation. This is even more concerning, knowing that, in time, UML class diagrams, tend to grow uncontrollably fast in their size and complexity, which introduces a problem of domain knowledge maintenance. To eliminate or at least to reduce the impact of these problems, ontology-based UML class diagrams can be greatly enhanced with support of FOL [7]. By doing so, each single class and their hierarchies can be additionally supported with numerous FOL assertions.

Combined together, UML class diagrams and FOL assertions can produce a more complete, detailed and complementary description of a domain and its integration system. Such an approach to ontology engineering allows us to prove its soundness and check its consistency level, which in turn allows for the pre implementation domain integration system validation assuring its foundations. It is worth mentioning that the presented ontology-based analysis and engineering are essentially iterative activities, which also means that they end when enough confidence and knowledge about the integrated domain has been gathered and formalized explicitly (Figure 1).

However, for the ontology to be reused during domain integration system implementation and communication processes, it has to be defined in a machine interpretable format. A widely accepted solution to this problem would be to reuse XML XSD schema as it possesses enough expressive power to be not only analyzed by a machine but by a human as well. Because the ontology was modeled by means of UML class diagrams, it is also the most reasonable and convenient approach since translation to XML XSD schema is rather straightforward and well documented. However, the resulting XML XSD schema is still just a static representation of an ontology, which is just a static description of a domain and as such, XML XSD schema can only be reused as data validation tool during runtime. The true potential of reusing XML XSD schema is perhaps the possibility of automatic source code generation (Figure 1). In such a case, XML XSD schema stands as a template for the ontology class hierarchy source code creation, which can only improve consistency and interoperability amongst various different domain integration system elements.

For the sake of this paper and analysis of the domain integration system, it was decided to focus only on a simplified part of the system that involves various different OPC DA servers, host machines and operating agents.

IV. General Rules of UML to FOL transition

However, before any implementation, automatic source code generation, XML XSD schema creation and any FOL analysis, a set of general rules of conversion between UML class diagram model and FOL expressions has to be specified. Generally, in order to speak about FOL sentences predicates, such as class, argument, argument type, multiplicity, association and natural number has to be introduced (1).

$$NatNum, C, T, A, R, S, a, f, r \tag{1}$$

UML class diagrams, much alike the ontologies, allow for declarative modeling of the static structure of a domain, in terms of concepts and their relationship [7]. Single class in each UML class diagram denotes a set of objects with common features and in FOL it will be represented as a unary predicate C (1). Common features of each class are expressed by means of its underlying attributes. Each attribute is characterized by name, type and multiplicity and in FOL it is represented as a binary predicate a (1). To represent the type of an attribute in FOL a unary predicate T (1) will be used. Each attribute characterizes with multiplicity. Even single valued attribute has one. In order to express multiplicity, first of all, a natural number unary predicate NatNum (1) has to be specified [8].

$$\begin{gathered} NatNum(0) \\ \forall\ n.NatNum(n) \Rightarrow NatNum(S(n)) \\ \forall\ n.0 \neq S(n) \\ \forall\ m,n.m \neq n \Rightarrow S(m) \neq S(n) \end{gathered} \tag{2}$$

Natural numbers in FOL (2) can be characterized by means of one natural number constant symbol 0 and one successor unary predicate S (1) (2). For each n value, if n is natural number so is S(0), S(S(0))and S(…(S(n)). Natural number will help us in defining multiplicities [7] for each available class attribute for which the FOL expression (3) holds.

$$\forall\ x \exists\ y.C(x) \wedge a(x,y) \Rightarrow T(y) \tag{3}$$

An expression (3) states that for each x instance of class C there exists such value y of type T related to instance x by means of an a attribute. The multiplicity [7] of attribute a is denoted by FOL expression (4), which states that for each instance x of class C there exists such value y of type T that references to x by means of an attribute a that has at least i and at most j values.

$$\begin{gathered} \forall\ x,i,j,\exists\ y.C(x) \wedge T(y) \wedge a(x,y) \\ \wedge NatNum(i) \wedge NatNum(j) \\ \Rightarrow (i \leq \#\{y|a(x,y)\} \leq j) \end{gathered} \tag{4}$$

Hierarchical relationships between different classes are modeled by means of an association relationship which is very much alike attribute but differs in the level of accessibility. While an attribute models properties that are local to a class, an association models properties that are shared amongst different classes. An association [7] can be a binary or a n-ary relation, thus what holds for the binary association holds also for n-ary one. In FOL, association between various different classes without association class defined is represented as a n-ary predicate A (1) for which the FOL expression (5) holds. An expression (5) states that association instances have to be of correct classes.

$$\begin{gathered} \forall\ i,n,x_0,x_n.\ NatNum(n) \wedge NatNum(i) \\ \wedge A(x_0,x_n) \Rightarrow \wedge_{i=0}^{n} C_i(x_i) \end{gathered} \tag{5}$$

An association between different classes that has related association class is represented by a unary predicate A and n binary role names predicates $r_0 \ldots r_n$ (6)(7)(8).

$$\forall\ x,i \exists\ y.A(x) \wedge NatNum(i) \wedge r_i(x,y) \Rightarrow C_i(y) \tag{6}$$

$$\begin{gathered} \forall\ x,i \exists\ y,y'.A(x) \wedge NatNum(i) \\ \wedge r_i(x,y) \wedge r_i(x,y') \Rightarrow y = y' \end{gathered} \tag{7}$$

$$\begin{gathered} \forall\ x,x',i,n \exists\ y_i,y_{n.A}(x) \wedge A(x') \wedge NatNum(i) \\ \wedge NatNum(n) \wedge \wedge_{(i=0)}^{n}(r_i(x,y_i) \wedge r_i(x',y_i)) \\ \Rightarrow x = x' \end{gathered} \tag{8}$$

An assertion (6) types the association. It specifies that for each instance of association class A, instance value y that relates to x by means of role r_i is of specific class type C_i. An assertion (7) refers to the fact that there exists only one instance value y for each single role r_i of A. An assertion (8) specifies that there is only one instance of association class that faithfully represents one particular relation. Special kind of a binary association between two classes that corresponds to a containment relationship is an aggregation. It specifies that each containing class owns a set of contained classes instances (9). An aggregation [7] relationship does not need an additional association class, thus in FOL it can be specified as a binary relation (9).

$$\forall\ x \exists\ y.A(x,y) \Rightarrow C_1(x) \wedge C_2(x) \tag{9}$$

In the object-oriented domain, the notions of inheritance and polymorphism are very important. Both are often used interchangeably and both refer to the notion of generalization. Generally, class hierarchy is a result of a composition of several generalizations (10).

$$\forall\ x,i,n.\mathrm{NatNum}(i) \wedge \mathrm{NatNum}(n) \wedge \\ \wedge_{(i=0)}^{n}(C_i(x_i)) \Rightarrow C(x) \tag{10}$$

Class hierarchies usually form complex structures and usually during domain integration system analysis it is often required to formalize various different and smaller class subcategories. To do that, it is important to specify both class disjointness (11) and completeness (12) notions [7]. Disjointness refers to the notion that single object cannot be an instance of two different subclasses at the same time. Completeness refers to the fact that each subclass is also an instance of at least one superclass.

$$\forall\ x,i,j,n.\mathrm{NatNum}(n) \wedge \mathrm{NatNum}(i) \\ \wedge \mathrm{NatNum}(j) \wedge \wedge_{(i=0)}^{n} C_i(x) \\ \Rightarrow \wedge_{((j=0),(j\neq i))}^{n} \neg C_j(x) \tag{11}$$

$$\forall\ x,i,n.\mathrm{NatNum}(n) \wedge \\ \mathrm{NatNum}(i) \wedge C(x) \Rightarrow \vee_{(i=0)}^{n} C_i(x) \tag{12}$$

The methods, static or instance, are the executable part of each class. Each such method defines class competencies and the body of those methods represents how the class and its instances interact with a domain. Class methods are functions from the class or class instance to which a method is associated and possibly additional parameters to instances or simple values [7]. In FOL, a class method is represented by a n-ary predicate f (13) with up to $f_x+f_i+f_o$ arguments, where f_x is an instance of the parent class, f_i is the number of input parameters and f_o is the output of the method. Therefore, it is easy to deduce that an instance method without input parameters will be defined by a binary predicate with f_x+f_o number of arguments. FOL predicate f has to additionally satisfy a set of assertions that will explicitly specify method consistency (14) and correct typing of method's input and return parameters (13). Both assertions, as a result, allow capturing explicitly notions of overriding and overloading that in UML class diagrams are rather hidden from ones perspective.

$$f_{x,p_0,\ldots,p_m,r} \tag{13}$$

$$\forall\ x \exists\ p_0,\ldots,p_m,r.C(x) \wedge \mathrm{NatNum}(m) \wedge \\ f_{x,p_0,\ldots,p_m,r} \Rightarrow \wedge_{(i=0)}^{m}(P_i(p_i)) \wedge R(r) \tag{14}$$

$$\forall\ x \exists\ p_0,\ldots,p_m,r,r'.C(x) \wedge R(r) \wedge R(r') \\ \wedge \mathrm{NatNum}(m) \wedge f_{x,p_0,\ldots,p_m,r} \\ \wedge f(x,p_0,\ldots,p_m,r') \Rightarrow r=r' \tag{15}$$

V. DOMAIN INTEGRATION SYSTEM ANALYSIS

Based on the general rules of transition between UML class diagram and FOL, further domain integration system analysis can be performed that might reveal the flaws of the obtained model. The presented analysis results, however, were not generated automatically by means of any additional tool as no such tool is currently available. Each given FOL formula was manually derived during additional domain integration system UML class diagram analysis proving its soundness, correctness and consistency. Such an approach allows decomposing complex analysis problems into simpler ones; thus, it helps to describe integration issues in more details. Without such an approach, no scalable, open and reconfigurable domain integration system could be engineered. For the sake of better understanding, in the presented work, only a simple part of domain integration system was chosen. The part that was selected incorporates four different, hierarchically related to each other, concepts representing general domain characteristics (16), various different PC-based and IA-based workstations (17), industrial OPC DA servers (19) and intelligent, autonomous software agents (18).

MAXISConcept class (16) (Figure 2) is a general, domain ontology, integration system, abstract base class that defines a few different common methods (16) that have to be inherited and overridden on each ontology subclass level to meet specific ontology subclass requirements and to be used it has to be inherited.

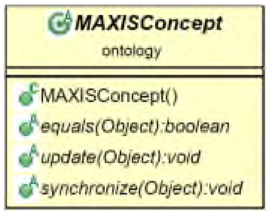

Figure 2. MAXISConcept class.

$$\begin{gathered} \mathrm{MAXISConcept}(x) \\ \mathrm{MAXISConcept} \\ \\ \mathrm{equals}_{x,\mathrm{Object},\mathrm{boolean}} \\ \mathrm{update}_{x,\mathrm{Object}} \\ \mathrm{synchronize}_{x,\mathrm{Object}} \end{gathered} \tag{16}$$

HostConcept class (17) (Figure 3) models a workstation machine on which OPC DA servers and agents can be deployed. It derives MAXISConcept class and overrides its methods to provide their specific implementation it also defines two members HostName and HostIp accessible through public getter and setter methods.

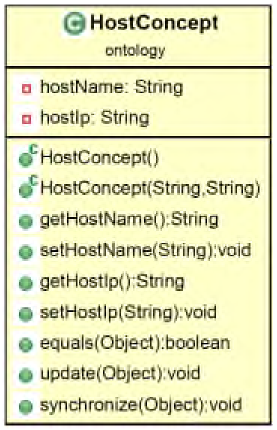

Figure 3. HostConcept class.

$$\begin{gathered} \mathrm{HostConcept}(x) \\ \mathrm{HostConcept}, \mathrm{HostConcept}_{\mathrm{String},\mathrm{String}} \\ \mathrm{SetHostName}_{x,\mathrm{String},\mathrm{void}} \\ \mathrm{GetHostName}_{x,\mathrm{void},\mathrm{String}} \\ \mathrm{SetHostIp}_{x,\mathrm{String},\mathrm{void}} \\ \mathrm{GetHostIp}_{x,\mathrm{void},\mathrm{String}} \\ \\ \forall\ x \exists\ y.\, \mathrm{HostConcept}(x) \wedge \mathrm{hostName}(x,y) \\ \Rightarrow \mathrm{String}(y) \\ \forall\ x \exists\ y.\, \mathrm{HostConcept}(x) \wedge \mathrm{hostIp}(x,y) \\ \Rightarrow \mathrm{String}(y) \\ \\ \forall\ x \exists\ y.\, \mathrm{HostConcept}(x) \wedge \mathrm{String}(y) \\ \Rightarrow 0 \le \{y \mid \mathrm{hostName}(x,y)\} \le 1 \\ \forall\ x \exists\ y_1, y_2.\, \mathrm{HostConcept}(x) \wedge \mathrm{String}(y_1) \\ \wedge\ \mathrm{String}(y_2) \wedge \mathrm{hostName}(x,y_1) \\ \wedge\ \mathrm{hostName}(x,y_2) \Rightarrow y_1 = y_2 \\ \\ \forall\ x \exists\ y.\, \mathrm{HostConcept}(x) \wedge \mathrm{String}(y) \\ \Rightarrow 1 \le \{y \mid \mathrm{hostIp}(x,y)\} \le 1 \\ \forall\ x \exists\ y_1, y_2.\, \mathrm{HostConcept}(x) \wedge \mathrm{String}(y_1) \\ \wedge\ \mathrm{String}(y_2) \wedge \mathrm{hostIp}(x,y_1) \\ \wedge\ \mathrm{hostIp}(x,y_2) \Rightarrow y_1 = y_2 \end{gathered} \tag{17}$$

AgentConcept class (18) (Figure 4) relates to the notion of intelligent, autonomous software agent that can cooperate with other different agents exchanging various different ontological messages under certain conditions in the given domain area solving assigned tasks according to designed functionality. It derives HostConcept class providing more specific implementation for each inherited method and is univocally characterized by its AgentLocalName class member.

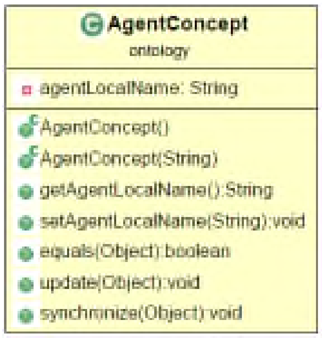

Figure 4. AgentConcept class.

$$\begin{gathered} \mathrm{AgentConcept}(x) \\ \mathrm{AgentConcept}, \mathrm{AgentConcept}_{\mathrm{String}} \\ \mathrm{SetAgentLocalName}_{x,\mathrm{String},\mathrm{void}} \\ \mathrm{GetAgentLocalName}_{x,\mathrm{void},\mathrm{String}} \\ \\ \forall\ x \exists\ y.\, \mathrm{AgentConcept}(x) \wedge \\ \mathrm{agentLocalName}(x,y) \Rightarrow \mathrm{String}(y) \\ \\ \forall\ x \exists\ y.\, \mathrm{AgentConcept}(x) \wedge \mathrm{String}(y) \\ \Rightarrow 1 \le \{y \mid \mathrm{agentLocalName}(x,y)\} \le 1 \\ \\ \forall\ x \exists\ y_1, y_2.\, \mathrm{AgentConcept}(x) \wedge \mathrm{String}(y_1) \\ \wedge\ \mathrm{String}(y_2) \wedge \mathrm{agentLocalName}(x,y_1) \\ \wedge\ \mathrm{agentLocalName}(x,y_2) \Rightarrow y_1 = y_2 \end{gathered} \tag{18}$$

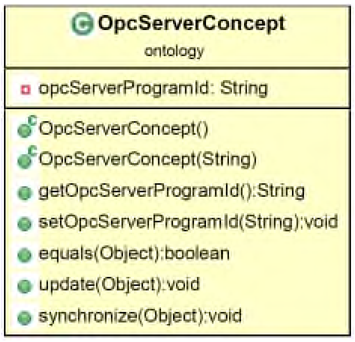

Figure 5. OpcServerConcept class.

OpcServerConcept class (19) (Figure 5) relates to the OPC DA industrial server that provides various different real-time data from an underlying process. Similar to the

AgentConcept class, OpcServerConcept class also derives HostConcept class providing more specific implementation for each inherited method. OpcServerConcept class defines OpcServerProgramId member to univocally distinguish operating OPC DA server.

$$\begin{gathered}\mathrm{OpcServerConcept}(x)\\ \mathrm{OpcServerConcept}_{\mathrm{String}}\\ \mathrm{SetOpcServerProgramId}_{x,\mathrm{String},\mathrm{void}}\\ \mathrm{GetOpcServerProgramId}_{x,\mathrm{void},\mathrm{String}}\end{gathered}$$

$$\begin{gathered}\forall\ x\,\exists\ y.\,\mathrm{OpcServerConcept}(x)\wedge\\ \mathrm{opcServerProgramId}(x,y)\Rightarrow\mathrm{String}(y)\end{gathered}$$

$$\begin{gathered}\forall\ x\,\exists\ y.\,\mathrm{OpcServerConcept}(x)\wedge\mathrm{String}(y)\\ \Rightarrow 1\le\{y\,|\,\mathrm{opcServerProgramId}(x,y)\}\le 1\end{gathered}$$

$$\begin{gathered}\forall\ x\,\exists\ y_1,y_2\,\mathrm{OpcServerConcept}(x)\wedge\mathrm{String}(y_1)\\ \wedge\,\mathrm{String}(y_2)\wedge\mathrm{opcServerProgramId}(x,y_1)\\ \wedge\,\mathrm{opcServerProgramId}(x,y_2)\Rightarrow y_1=y_2\end{gathered}\qquad(19)$$

Class hierarchy always imposes numerous constraints, which are implicit in UML class diagram model (Figure 6), however, to improve readability, each such constraint can be defined explicitly (20).

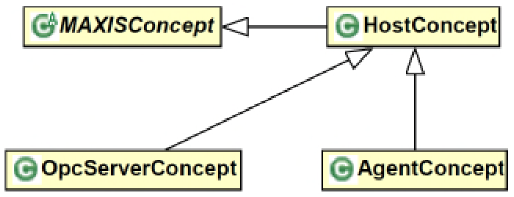

Figure 6. Simple domain integration system class hierarchy.

$$\begin{gathered}\forall\ x.\mathrm{HostConcept}(x)\Rightarrow\mathrm{MAXISConcept}(x)\\ \forall\ x.\mathrm{AgentConcept}(x)\Rightarrow\mathrm{HostConcept}(x)\\ \forall\ x.\mathrm{OpcServerConcept}(x)\Rightarrow\mathrm{HostConcept}(x)\end{gathered}$$

$$\begin{gathered}\forall\ x.\,\mathrm{AgentConcept}(x)\Rightarrow\neg\mathrm{OpcServerConcept}(x)\\ \forall\ x.\mathrm{OpcServerConcept}(x)\Rightarrow\neg\mathrm{AgentConcept}(x)\end{gathered}$$

$$\begin{gathered}\forall\ x.\mathrm{HostConcept}(x)\Rightarrow\mathrm{AgentConcept}(x)\\ \vee\,\mathrm{OpcServerConcept}(x)\end{gathered}\qquad(20)$$

VI. CONCLUSION

UML class diagrams are fundamental during engineering of each domain integration system. However, each such diagram tends to hide many simple but important facts therefore, an additional tool that can precisely capture such implicit knowledge has to be introduced. The presented analysis focuses on a precise, detailed specification and hierarchical relationship of each domain element, showing an easy way to produce a comprehensive and complementary description of an integrated domain using both UML class diagrams and FOL expressions. Such analysis is not a trivial task as it requires multiple iterations to obtain satisfactory results and thus confidence about integrated domain details. However, the goal of this analysis is not simply to obtain multiple FOL assertions and better UML class diagrams. The main goal is to produce a most detailed description of a scalable, open and reconfigurable domain integration system that can faithfully operate under nondeterministic conditions. To achieve this, resulting ontology-based UML class diagram has to be transformed into a machine interpretable format. The most reasonable solution to this problem is to translate the UML class diagram into XML XSD schema, which in turn can be straightforwardly reused during both implementation and runtime.

ACKNOWLEDGMENT

This work was supported by the National Science Centre under grant No. 2012/05/B/ST7/00096 (D. Choinski) and by the Human Capital Operational Programme co-financed by the EU from the financial resources of the European Social Fund-project no. POKL. 04. 01. 02-00-209/11 (B. Pietrzyk)

HUMAN CAPITAL
NATIONAL COHESION STRATEGY
EUROPEAN UNION
EUROPEAN
SOCIAL FUND

REFERENCES

[1] F. Iwanitz and J. Lange, OPC–Fundamentals, Implementation and Application. Huthig Verlag Heidelberg, 2006.

[2] D. Choinski and M. Senik, 2010. “Collaborative Control of Hierarchical System Based on JADE.” In: Y. Luo (ed.), CDVE 2010, LNCS. vol. 6240, Springer, Heidelberg, pp. 262-269.

[3] D. Choinski and M. Senik, 2011. “Multi-Agent oriented integration in Distributed Control System.” In: J. O'Shea et al. (eds.), KES-AMSTA 2011, LNAI. Vol. 6682, Springer, Heidelberg, pp. 231-240.

[4] D. Choinski and M. Senik, “Multi-Agent System for Adaptation of Distributed Control System.” ICINCO 2012, pp. 206-211.

[5] F. Bellifemine, G. Caire, and D. Greenwood, Developing Multi-Agent Systems with JADE. John Wiley & Sons, Chichester, 2007.

[6] J. L. Peterson, Petri net theory and the modeling of systems. Prentice Hall, 1981.

[7] D. Berardi, D. Calvanese, and D. G. Giacomo, “Reasoning on UML class diagrams.” Artif. Intell. 168(1-2): pp. 70-118, 2005.

[8] S. Russel and P. Norwig, Artificial Intelligence a Modern Approach, 3rd ed. Prentice Hall, 2010.