Proceedings of the International Multiconference on
Computer Science and Information Technology pp. 149–154
ISBN 978-83-60810-27-9
ISSN 1896-7094

Usage of reflection in .NET to inference of knowledge base

Marek Vajgl
Ostravská Univerzita v Ostravě,
Přírodovědecká fakulta, katedra
informatiky a počítačů. 30. dubna
22, Ostrava, Czech Republic.
Email: marek.vajgl@osu

***Abstract*—This document describes how information generated by integrated development environment (namely Visual Studio 2008) can be used to generate and usage of knowledge base. The main aim is to explain, how the approach of extension of currently implemented software project can achieve knowledge representation, and how already created and implemented data types can be used to create knowledge bases using modern language's development environment and behavior. The article is aimed to the Description Logic formal system, but can be applied to any formal deduction mechanism.**

I. Introduction

Today, lot of information systems are built in non-research area and most of them use for data or knowledge storage standard relational databases, object databases or other storage places, like XML files.

This, often very large structures, contain a lot of information about company (or any targeted subject), which can be manipulated as knowledge and shared. Information systems or applications from design development phase through fully implementations contains well considered, well organized, and well described and documented schemas (like database models, class models), which are not taken and mentioned as knowledge base and does not contains any kind of inference mechanism. Nevertheless a lot of that information can be taken as concepts/roles/instances and used to create full knowledge base with inference mechanism. Moreover, this process can be done automatically due to source format – implemented diagrams, classes in object oriented programming, etc. However, only minor part use some kind of inference or deduction mechanism, like expert system, or some kind of formal deduction based over knowledge bases.

Although the main reason is not sufficient spread of knowledge mechanism approaches (like expert systems are mostly oriented in technological industry or medicine), one of the another reasons is inability to check, test, or prove those systems on created project, that will simply show the advantages (and disadvantages) of those approaches. If aimed on knowledge bases, and one of the most today's formal system, description logic, the existing implementations often requires to install third party runtime environment (e.g. Java Virtual Machine for .NET users), or are integrated as server applications listening on special ports with special syntax requested for input and output for test. True is, that most of those systems are supported by today's standard for knowledge base/ontology editor and designer Protégé [6], there is again a load on a developer to 1. Find, install and open knowledge base system, 2. Install and execute Protégé system, 3. Learn how to insert data from existing information system and 4. Learn how to profit from those combinations against the classical approaches mentioned above. A lot of other approaches are built on client-server principles, where information system as client has to send its requests to the server realizing inference over knowledge base. Those approaches again includes requirement to create and use any implementation of format of communication protocol(s) between client and server.

This approach is for "common" developer almost impassable, so there are open doors for other approaches, which brings those formal systems into the existing system of developers. The main idea is to extend developer's existing project. The added part will consist of a solution, which can (partially) automatically analyze existing project, generate knowledge base and offer a mechanism to inheritance or deduction over gained data.

Lot of integrated development environments (like Microsoft Visual Studio) offers mechanism providing automatic generating of the content into some universal format (like XML files), moreover, modern programming languages contains mechanism allowing runtime analysis of any compiled code, called "Reflection". Those two built stones open a way how can (and should) knowledge system use those data to offer a developer a ways to use knowledge representational approach in application development.

Here presented solution – XReasoner system – manages three main parts of operation with data presented: a) creation of the knowledge base – T-Box and A-Box, b) managing knowledge base, c) sharing information with other sources based on other modern phenomenon, web services. This article describes first part of this behavior. (System is built over description logic DL1 [1][3] and uses semantic tableau algorithm for inference. Idea of approach presented in this article is mainly commonly applicable over other systems; at least, for description logic (above common constructors) is required constructor of negation (to implement exclusiveness of derived classes), value restriction (for properties and relations) and sometimes quantified number restriction (for "property" behavior in .NET languages) –$\mathcal{ALCQ}$).

One more condition is important. Developer should not (at least in testing phase) adapt or rewrite the existing code. Therefore, the introduced solution, where able, presents two

approaches: first one with changes to origin source code; second one which remains origin source code intact.

II. Description logic

At the beginning of this contribution, only the short introduction of the formal system of description logic will be mentioned. This formal system is very popular today, for its simplicity, expressivity. Moreover, we prefer it due to its close approach to the object oriented paradigm of programming.

Description logic (presented according to [7]) is a formalism which arises from concept languages. Concept languages are built on description of abstract objects of real world – called concepts, and roles, representing relations between those concepts. In description logic, the concepts are ordered into hierarchical structures. Knowledge base of description logic consists of definition of concepts and their roles (both those items represent intensional knowledge) and instances of the concepts (instances represents extensional knowledge about concrete object of the real world).

From the viewpoint of formal systems for knowledge representation is description logic built on formalism supporting frames, associative networks and object oriented knowledge representation.

Knowledge base of description logic is formally divided into two parts. First part is called T-Box (terminological box) and contains terminological intensional knowledge in the form of concepts and definition of relations between those concepts. Second part is called A-Box (assertional box) and consists of facts about instances – individuals – where each instance is related to the concrete concept or role. Each instance's domain is one or more concepts in T-Box.

Moreover, as mentioned before, concepts in T-Box are automatically ordered into hierarchical structure according to theirs definition. They are classified and placed (subsumed) into taxonomy – this behavior is called *subsumption*.

III. IDE, Reflection and Automatization

As mentioned before, the main aim is to utilize the behavior of modern programming languages and modern integrated development environment. The contribution is aimed on Microsoft's .NET technology (and examples will be presented in C# programming language), because it is in close development with the XReasoner tool – that is description logic implementation over the .NET platform, created to support this behavior. The two main tools of the modern software development in .NET bring opportunity to automatic knowledge base generation from existing application or information system.

The first construct is automatization. The modern IDE (like mentioned Visual Studio) offer tools to generate some universal data (like xml files) from existing classes into structures also called *DataSets*. IDE is able to capture classes, relation between classes (with occurrences count), inheritance, interface implementations into the xml files. This information is not only generated from classes, but databases or source XML files can be used too. So, IDE offers easy way how to extract terminological part of software's "knowledge base" into the xml file, which can be later analyzed.

The second construct is more powerful. It does not rely on development environment, but is part of programming language, and is called *reflection*.

Reflection [5] is ability of modern programming language (including languages used to development over the Microsoft .NET platform) to achieve, process and invoke compiled source code dynamically during application run. Simple said, for example, during application run can a programmer choose library, load a type from it, create an instance and invoke type members over this instance. In both most used programming languages, C# (.NET) and Java uses in compilation mechanism, which includes all information about data types, its members and relations in the compiled file (e.g. .dll assembly file in .NET platform). Due to this behavior reflection does not need any additional information from other files (like C++, where the .h files are required). This part of information are saved in the *manifest* of the assembly (assembly is .NET compiled file). In .NET, assembly is simply loaded (on request by programmer), its manifest is analyzed and the runtime environment "knows" which types can be found in the library, which members can be called and with which parameters. When the object oriented programming is taken into account, there must be mentioned that reflection can access not only public types and members, but also protected, private and internal (means visible only in current assembly) ones are visible and accessible with this mechanism.

When aimed to the behavior of modern programming, one more very powerful and important feature (accessible during reflection) should be mentioned. It is called *attributes*.

Attributes allows a developer to add some special kind of declaration (!) information to a class definition. This information can be later used by compiler or other executing code to adjust its behavior against marked class. For example, class declared this way:

```
[Serializable()]
public class X
{
   public int A;
   [NonSerialized()]
   public int B;
   …
}
```

Class X will be during serialization process successfully stored to a target location; moreover, the field B will be skipped and will not be saved.

Attributes are not built-in part of the language, it the view of language they are special cases of classes. So, the developer can create its own attributes and use them with classes. Other developers can then use those attributes to mark their classes.

This mechanism is now used in coordination with persistence into relational database systems (RDBS); each class has specified into which table, with which properties and

which data-types, is stored/loaded. The same way can be used to specify which classes and which its properties can be used to knowledge base generating.

IV. Creation of Knowledge Base

Generally, the mechanism uses set of assemblies containing XReasoner system used to manage description logic knowledge base and set of assemblies containing attributes and classes used to generate knowledge base by adjusted properties. As mentioned in the introduction, two approaches can be used: external, which does not affect code of the existing code, and internal, which affects this code. External approach uses xml file to define requested behavior, however, sometimes internal behavior can be more readable and understandable. Both approaches can be used together, in case of conflict internal approach has higher priority and will be used.

Physically, implementation requires adding references to assemblies of XReasoner into current project (or creating new project and adding references to both, analysed project and XReasoner). If external approach is used, moreover xml file needs to be created and passed as parameter for XReasoner system.

The mechanism of generating knowledge base including concepts, roles and also instances will be divided and introduced in two parts: a) creation of T-Box, b) adding instances into A-Box.

A. T-Box - introduction

The main contribution of reflection technology is its "knowledge" of project classes and relations between them. This information can be presented by reflection simply by analysis of existing data types, their fields, properties and type inference hierarchy.

T-Box creation request to answer three questions: a) what data are available through reflection, b) which data are relevant and public and should be included, c) how to identity and identify different definitions representing the same object.

The first step is to retrieve all defined data types in selected assembly or assemblies. As mentioned before, reflection can provide (and provides) mechanism that returns simple listing of all data types defined in assembly. In .NET, there are a lot of data types, but only classes (reference types) and structures (value types) will be taken into account (that is important, i.a. interfaces are also omitted).

Next step is to analyze inference hierarchy between those data types. .NET languages are strictly object oriented; and therefore inheritance usage is very common. Information about data type also contains references to all its predecessors. With this information tree of hierarchy can be created.

Last part of T-Box analysis aims to properties and fields of the data types. Fields are not very often used to present some information value directly, because it in contrast with encapsulation principle of object oriented paradigm. Therefore properties are used instead. Properties contain information about their data-type and define strictly binary relations between two data types.

All those information can be used to define concepts and roles between them in T-Box. Basically, data-types will represent concepts in description logic knowledge base. Subsumption of concepts will be equal to data-type inheritance hierarchy. This creates only "simple subsumption" in meaning that there is only "Concept A is subsumption of concept B" relation, without any additional conditions. Programming language and object oriented paradigm ensure there will be no cyclic dependencies in definition.

As mentioned before, properties or fields define binary relations between two concepts. Those relations represent roles between concepts. First data type (the one containing the property or field) defines concept which is domain of the created role. Second data type (the one which is property or field data type) represents range of created role. Domain and range can be the same type; the behavior of properties or fields ensures that only binary roles are created.

B. T-Box - implementation

Before implementation description, one more behavior must be explained. Previous paragraph explains that roles always create relation between two objects. In C# language, this is correct. But in knowledge base a slightly different behavior should be required. Let's state an example. We have class "Person" and it has property representing a set of e-mail addresses. Data type of this set will be some collection (in C# language probably a list of strings (List<string>)). The relation is that person has its own list of strings, which contains his e-mail addresses. But the aim of knowledge base is aim to capture the knowledge "person and his e-mails", not "person and his list, containing emails". Therefore this decomposition can be made into multiple roles realizing relation between concrete person and one of its e-mail addresses. But, in this behavior the exact mapping between origin C# code and created knowledge base is lost. Therefore, developer can choose which one of those behavior he will prefer.

As same as in object oriented paradigm not all data types are public and visible to everybody, developer can make decision which of the source data-types will be used to create concept added into knowledge base. Some types can be omitted due to their irrelevance; some may be too complex and will make the knowledge base unnecessarily difficult. Therefore, developer has to mark which types he would like to include in the knowledge base.

As mentioned in the introduction, one of the aims is to have the source code intact. Therefore there are two ways how to achieve that. First one, external, uses xml file to define requested behavior, second one inserts requested info directly into source code. Internal approach is more intuitive and will be explained as first.

This approach uses predefined attributes to define which classes and how will be transformed into concepts; and also, which of its properties will be used. XReasoner defines two attributes:

- EKnowledgeClassAttribute – only classes with this attribute will be converted into concepts. This attributes also defines three properties: a) name – which defines

name of the created concept (default name created from class name will be created if not specified); b) IdentityProperties – string defining which properties uniquely define the instances of the concept (s.t. like primary key in databases) – it is used later, in A-Box instantialization; c) URIs – string defining the unique resource identifier(s) for the project – it will be used in knowledge base sharing and is expected if two different concepts (from different sources) have the same URI, then they represents the same object in the real world.

- EKnowledgePropertyAttribute – only properties marked with this attribute will be used to represents roles of the parent class/concept. Again, it has two properties: a) IsIdentity – with the same meaning as above, but is specified for property directly; b) URIs with the same meaning as above, but for role.

Simple example of this approach follows (irrelevant parts of file are dotted):

```
using ENG.XReasoner.AssemblyAnalyser.Attributes;

namespace Solution
{
  [EKnowledgeClass(
    Name="Person",
    IdentityProperties="BornNumber",
    URIs="http://.../Person")]
  public class Person
  {
    [EKnowledgeProperty(IsIdentity=true)]
    public string BornNumber { get; set; }
    [EKnowledgeProperty()]
    public string Name { get; set; }
  }
}
```

This, intensional, approach is more difficult, because all required classes/concepts have to be marked with the attribute, but a developer has exact control over the process.

Extensional approach uses xml file to define requested behavior. Upper file can be created accordingly to DTD file (supplied with XReasoner solution), and (briefly) may looks like this (irrelevant parts of file are dotted):

```
<?xml version="1.0" encoding="utf-8" ?>
<!DOCTYPE …>
<EKnowledge>
  <TBox>
    <Assembly nameRgx=".+">
      <Behavior
        conceptNaming="fullClassName"
        includePropertiesRgx=".*"
        expandEnumerations="false" />
      <Concepts>
        <Concept
          typeName="Person" explicitName="Person" >
          <Property name="BornNumber"
          isIdentity="true" />
          <Property name="Name" isIdentity="false" />
          <Property name="Addresses" isIdentity="false" />
        </Concept>
        <ConceptRgx typeNameRgx=".*" >
        </ConceptRgx>
      </Concepts>
      <Others></Others>
    </Assembly>
  </TBox>
…
</EKnowledge>
```

Simple explanation of presented elements follows.

EKnowledge element covers the whole behavior of the XReasoner system, including not only how the knowledge base is defined, but also how is shared via web services (if any).

TBox element defines how the T-Box is created. This element contains of definition for one or more assembly.

Assembly element defines how the assembly will be treated. The “nameRgx” attribute defines which assemblies will be processed with this element’s setting. It is regular expression, which is matched against full assembly name. If assembly name matches more than one definition, first suitable is taken.

In the assembly, default behavior can be set. This element specifies, how the concepts will be named (is used only for classes/concepts without explicit name specification). Options are to use class name without namespace (here can arise problem with two classes with the same names, which cannot be later treated anyway!), or full class name used, including preceding namespaces. Second attribute, “includePropertiesRgx”, defines which properties will be included as roles of concept (if there is no explicit definition for concept). Again, this is regular expression which will be matched against property name (those names are distinct within class). Last parameter refers to the first paragraph of this chapter and defines, if enumerations (or collections) are expanded into special concept instance, or if multiple roles are created for each element of enumeration (collection). This dilemma has been explained above.

Next part contains definition for concept and conceptRgx elements, both nested under Concepts element.

Concept element describes behavior of one exact class/concept. It defines name of type (obligatory), explicit name for created concept (if any, default will be used), may contains behavior specification (same as explained before) and set of elements “Properties”. Those elements explicitly define which properties will be used to define role.

ConceptRgx element is definition which will be applied on all concepts matching “typeNameRgx” regural expression. Again, expression is matched against full type name. Moreover, ConceptRgx element can contain behavior specification (in the same format as mentioned for assembly element), which is expected to be more specific than any upper one.

There is implementation in the XReasoner solution, the classes “Analyser” and “Converter”. First one takes into its

public static function "Analyse" two parameters: 1) name of assembly/list of names of the assemblies, which will be proceeded; 2) name of the xml file containing the definition mentioned in explicit approach explanation. The second one can be omitted (can be "null"). The result of the process will be set of some intermediate info defines which classes and how will be transformed. Second class, "Converter", contains static function "Convert", which accepts one argument in the input. It is the result of the "Analyse" function. This transformation produces a set of formulas (in the set of data-types of XReasoner solution), which can be directly added into XReasoner knowledge base instance.

C. A-Box introduction

A-Box creation is the process where the instances of classes of the project are taken and transformed into instances of concepts and relations between those instances. From the previous part of transformation, there are defined concepts and roles in the terminological box of the knowledge base.

Obtaining instances of classes is a lot tricky, because it may include many different sources (database, xml files, multiple collections in the memory), in which may be the same instances (that means, instances representing the same real object) multiple, sometimes with changed properties, or may contain a lot of unnecessary individuals. Therefore it is required to involve the developer in this process. He has to specify how the instances are achieved and passes for the XReasoner. Its task is to achieve the instances only, analysis and parsing of the instances is made by XReasoner's classes and is done according to created concepts and roles in the previous part of process.

Physically, the task for the developer is to create a class containing two static methods. They have a simple purpose, but, of course, implementation can be difficult.

First method is used to return names for all instances of current concept. The input for the method is name of the concept (this name must match with the created concept name in the previous part, obviously). The reason for this method is really only to return instance names (that means identifiers) only, it is not necessary to load those objects fully with their properties. This method is used to create list of all instances of current concept.

Second one is used to return all roles for the current instance. It accepts one parameter, defining the name of the instance, and returns a fully loaded object, which will be converted into the instances and roles for the knowledge base.

There is one major issue which is very important and must be taken into account. The presented mechanism, created by the developer, has to always assign the same identifier to the same object, regardless of the state or time of the program. That means, if e.g. person "Michal" changes its name (between calling those two methods) to "Marek", it have to return still the same identifier. There are two intuitive ways how to assure this: a) create identifier by combination of identity properties (or primary keys). This can assure that instance can have recognizable identifier, bud brings more work for the developer, because he needs implement this for every class/concept in the project. Second approach profits (in the Microsoft .NET technology) from the method derived from highest abstract class "Object" to all its descendants (that are all classes). Every class has method "GetHashCode()", which should be unique for every instance and therefore can be used to return the unique identifier. However, hash codes are quite long and absolutely unreadable for humans, so person analyzing the knowledge base will not see what every instance is on the first sight.

D. A-Box implementation

The main principle was already explained in previous part of the article.

The implementation of the developer (the created class) has not to implement any specific interface or derive from any class. Therefore, this class can be included in original project. However, developer can inherit from interface presented in XReasoner solution to ensure he uses correct parameters and return types.

As the implementation suggests, XReasoner mechanism again uses reflection to invoke defined methods with requested parameters and achieve requested information. It is done by class named "InstanceExtractor". This class offers some methods to create knowledge base. According to invoked method and passed parameter it is able to return instances' names only, as well full instance definition with all roles. The result of this methods are again processed by the "Converter" class, which creates set of formulas (defining instances in the set of data-types of XReasoner solution), which can be directly added into XReasoner knowledge base.

V. Conclusion

There are a lot of information systems, or at least programs, which does not use any of knowledge representation approaches. There can be a motivation for developers to extend already implemented systems, or include this approach to currently developed systems. This article brings simple way how the existing solution can be extended to offer knowledge representation without any penetrative changes in the origin source code. What is not covered by this article, this extension can later be used to next deduction over created knowledge base, and also for sharing partial or full knowledge base using web services and definitions of URIs for selected concepts.

References

[1] Vajgl, M., Lukasová, A. A, *Semi-Dedicable Semantic Tableau Proof System in a Description Logic DL1*. Proceedings of CSE 2008 International Scientific Conference on Computer Science and Engineering. Košice: Department of Computers and Informatics FEEI TU of Košice, 2008. s. 277-284. [2008-09-24]. ISBN 978-80-8086-092-9

[2] Vajgl, M., Lukasová, A. *RDF-model as Associative Network*. DATAKON 2009 - Sborník databázové konference. Praha: Vysoká škola ekonomická v Praze, 2009. s. 95-103. [2009-10-10]. ISBN 978-80-245-1568-7

[3] Lukasová, A., Vajgl, M. *Genzen-like Proofs in Description Logic DL1*. Proceedings of the Tenth International Conference on Informatics 2009. Košice: Technical Univerzity of Košice, 2009. s. 160-166. [2009-11-23]. ISBN 978-80-8086-126-1

[4] W3C. Extensible Markup Language. [Cited 13. 6. 2010] Web resource. http://www.w3.org/XML/

[5] Microsoft – *System Reflection namespace*. [Cited 5. 4. 2010]. Web resource. http://msdn.microsoft.com/en-us/library/136wx94f.aspx
[6] The Protégé *Ontology Editor and Knowledge Acquisition System* – [Cited 5. 4. 2010] Web resource. http://protege.standford.edu
[7] F Baader et col. (eds.): *The Description Logic Handbook – Theory, Interpretation and Applications*. Cambridge University Press, 2003. ISBN 0-521-78176-0.