Towards a Knowledge-Aware AI
A. Dimou et al. (Eds.)
© 2022 The Authors.
This article is published online with Open Access by IOS Press and distributed under the terms of the Creative Commons Attribution License 4.0 (CC BY 4.0).
doi:10.3233/SSW220011

Assigning Systems to Test Environments Through Ontological Reasoning

Petar PARADZIKOVIC [a], Ralph HOCH [b] and Hermann KAINDL [b]
[a] *petar.paradzikovic@gmail.com*
[b] {*ralph.hoch, hermann.kaindl*}@*tuwien.ac.at*
Institute of Computer Technology, TU Wien

Abstract. In the automotive industry, testing for reliability and safety is very important but costly. Due to the deployment of an increasing number of features within these systems, mapping them to compatible test environments becomes more and more complex. In this paper, we present a use case for applying ontological reasoning in the automotive industry for supporting testers while making the selection of test environments. The given task has been to map the software under test together with test cases to test environments through ontological reasoning. To this end, we defined an ontology of test environments. It can be used for ontological reasoning, both by applying instance classification and subsumption reasoning, to assign test environments. This approach is prototypically implemented in Stardog, in combination with OWL2 and SPARQL. It is deployed alongside existing software at our industry partner's premises and provides a user interface, which supports testers while selecting test environments and executing tests.

Keywords. Ontology, subsumption, test environments, automotive software

1. Introduction

In the automotive industry, a variety of test environments for software-based automotive systems are used for testing components and functions of automotive systems. Usually, the choice of the "right" test environment for testing a component and/or its subcomponents is performed manually by an experienced tester. Since a test environment can be used for different test cases, the goal is to reduce time and costs by reusing test environments for different functions or components, while at the same time maintaining the testing quality, such that functional safety is always achieved as required. Specific test cases sometimes present false positives when executed on a test environment that is not compatible with these test cases.

For automating the selection of a test environment, we developed an approach based on ontological reasoning, see [1], which this paper is based upon. It requires a classification of test systems. For this purpose, we created an ontology of test environments within the scope of automated software testing in the automotive industry.

Figure 1 shows the main entities that had to be specified, which play different roles within the test environments. One test environment is a combination of one Vehicle-Model (MDL) and one Hardware-in-the-Loop (HIL) with a connected electronic control unit (ECU). The HIL with its connected ECU may also be simulated by a Software-in-the-Loop (SIL) system. A software under test is one Program Version (Software) in combination with one Dataset.

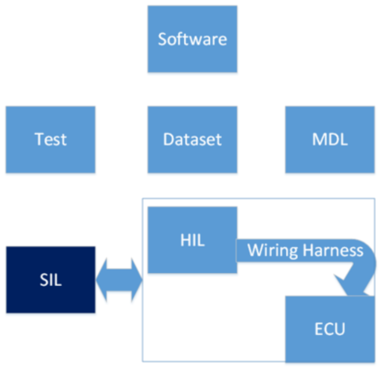


Figure 1. Main entities of test environments.

Since there are a lot of interdependencies between the necessary entities of test environments, we analyzed these dependencies and represented them formally in the ontology. We modeled it using Protégé, whereas for the integration and usage of the ontology the knowledge graph application Stardog was used. Stardog provides an integrated HTTP/REST API and the possibility for storing data from heterogeneous sources in a unified way. Furthermore, it allows querying and manipulation of the knowledge graph using SPARQL and supports built-in reasoning over ontologies.

The overall task has been to map the software under test together with test cases to test environments through automated reasoning. The use case has been defined as follows: given a specific *test case* and attributes of a specific *software under test*, a mapping to a specific *test environment* is to be found that is compatible in the sense that it is configured to be able to correctly perform the test with the software under test.

For supporting the given use case, we provided automated reasoning based on the ontology, both by applying instance classification and subsumption reasoning. For proving this approach to ontological reasoning to the testers, we deployed it at our industry partner's premises and provided a user interface through a REST API. The resulting application features prepared queries, which are called from a separate tool for test automation that outputs their results.

The remainder of this paper is organized in the following manner. First, we sketch some background material on the system test environments in the automotive domain, and on ontological reasoning, in order to keep this paper self-contained. We then describe the developed ontology and, building upon it, we show how ontological reasoning is applied and integrated with tool support. Finally, we discuss our approach and findings as well as related work, before we conclude.

2. Background

We provide here background material on the system test environments in the automotive domain, the ontology language used for their representation, and ontological reasoning as it is used in this work for mapping a specific test case and software under test to a specific test environment. Finally, we present a short introduction of the Stardog tool, which we used for deploying the ontological reasoning approach.

2.1. System Test Environments

Testing hardware and software of Electronic Control Units (ECUs) for automobile engines is essential, and specific test environments are needed for that. In particular, ECUs are running certain software, which is programmed by a development team according to customer requirements. This software is released to storage systems and the information about the attributes of the software is stored in a specific database. The ECU hardware, for testing specific ECU variants of different automobile engine types, has a unique serial number and a company-internal name, which is also stored in specific databases. To be able to test a specific ECU, a suitable software version has to be flashed to the ECU and the test environment components have to be configured for testing that specific ECU. Hence, there are certain dependencies of the hardware and software with the configurations of the test environments.

Test environments are specific workplaces, which usually are preconfigured for testing specific ECU types and functionalities. They are typically shared among testers and maintained by specific persons. Such a test environment consists of a test environment computer, a Real-Time-PC (RTPC), mostly a Hardware-in-the-loop and other hardware. The test environment computer runs a simulation program of the vehicle, where different simulations can be performed using Vehicle-models. This simulation of the vehicle-model is loaded by the RTPC and controls the hardware of the vehicle through the connection of the computer to a Hardware-in-the-loop. The RTPC together with the Hardware-in-the-loop simulate the controlled hardware. Figure 2 roughly depicts an example of a test environment setup.

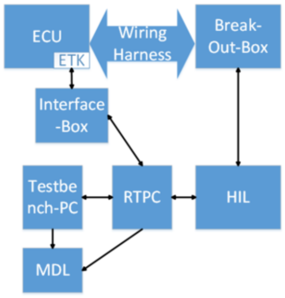


Figure 2. Example of test environment setup.

2.2. Ontology Language

Description Logics (DL) describe a family of logics for knowlegde modeling that are essentially fragments of first order predicate logic [2]. In ontology languages based on DL, semantics are expressed with rules, which relate to a semantically predefined vocabular. For building an ontology based on description logics, concepts, sets of objects, and roles, denoting binary relations between instances of those concepts, are used as semantic entities, which can be atomic or complex. Complex concepts and roles are created using constructors.

OWL (Web Ontology Language), a standard from the World Wide Web Consortium (W3C), is a language for representing an ontology. Currently there are OWL1, where OWL1 DL is the language based on Description Logics, and OWL2, which is an updated version of OWL1 DL.[1] The language builds open XML (Extensible Markup Language) and RDF (Resource Description Framework) and uses a triple structure where a triple consists of subject, predicate and object. Such a triple can be seen as a Node-Edge-Node construct, and a set of triples builds an RDF graph [3].

Using description logics, Gaševic et al. [4] describe developing a knowledge base in two parts. The TBox of a knowledge base contains the terminology, which is the vocabulary of the application domain. It defines concepts / classes, semantic relationships and properties. Inside a TBox, concepts are defined in terms of other, previously defined concepts [5]. The ABox contains assertions about concrete instances, using the vocabulary from the TBox.

2.3. Ontological Reasoning

There is a basic distinction in ontological reasoning between subsumption reasoning over concepts and instance checking over instances. Subsumption reasoning is the main reasoning service in the Tbox, and instance checking is the main reasoning service in the ABox.

An ontology represented formally, e.g., in OWL2 enables logical reasoning over concepts using *subsumption* as the basic reasoning technique for the TBox. Subsumption in DL languages is typically written as $C \sqsubseteq D$ [6]. This means, that all objects described by concept C are also objects described by D. The formalism is based on semantics from the mathematical set theory. This encompasses *inheritance* in terms of object orientation.

Reasoning services in the ABox can be defined in terms of instance checking [6], which checks instances for their properties one by one and if the properties are equivalent to the object properties of the defined class they are retrieved as instances of that defined class. For example, "realization" finds the most specific concept of an instance. Querying over ABox data is a core task of DL ontologies [7].

2.4. Stardog

Stardog[2] is an enterprise knowledge application using a graph database with integrated reasoning capabilities and several database connectors for data import. It has built-in features like path querying, where all paths (nodes and edges) between two nodes can be queried. Furthermore, Stardog allows for querying and manipulating the RDF data representing an ontology over the command line interface or an HTTP/REST API. There is also an editor tool called Stardog Studio for creating and manipulating ontologies saved in Stardog.

Stardog uses a *Services Layer* that represents the interfaces to the enterprise applications. In the *Graph Database Layer* are the query engine for processing SPARQL queries and the support for declarative models, which allows the creation of knowledge graphs

[1] https://www.w3.org/TR/owl2-new-features/
[2] https://www.stardog.com

without coding. The *Ingest Layer* of Stardog handles different types of data. Structured data is stored in the Stardog storage, semi-structured data is handled with virtual graphs and unstructured data is stored with *BITES*, which is a document storage system for unifying unstructured data like images, voice, etc. Stardog was chosen, because the industry partner is already familiar with it and uses it for other applications.

3. The Ontology

For enabling ontological reasoning, we had to develop an ontology of the domain, i.e., the system test environments and their attributes. It is represented in OWL2, and Figure 3 depicts a top-level view for illustration purposes. The classes shown were specified as main top-level classes. Some of them classify specific segments of expert knowledge like different wiring harnesses, whereas others are more general classes like different databases or repositories containing the actual data, or serving more general purposes. All these classes are subclasses of the owl:Thing class and are further subclassified. Additionally, some classes have object property relationships to other classes on a high level. Since this is very specific domain knowledge, we did not use any upper ontology, but modeled it to our best understanding during knowledge acquisition together with domain experts.

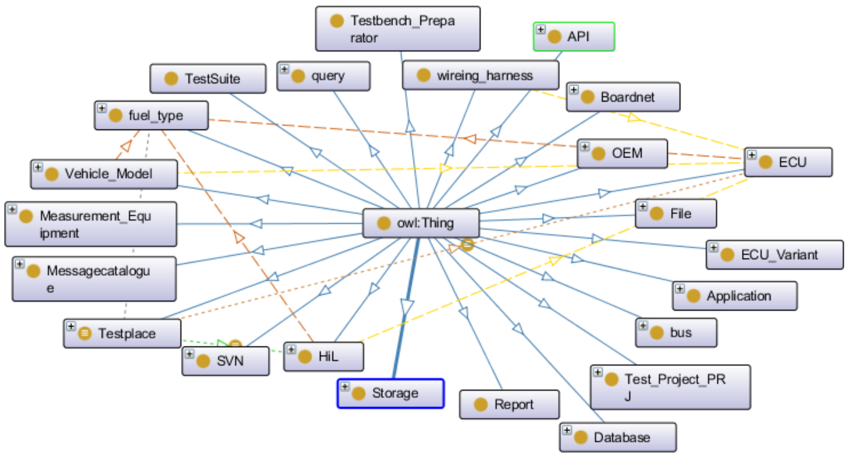


Figure 3. Top-level view of the ontology developed.

Overall, this ontology currently consists of 215 classes, 39 object properties and 3,721 instances. Furthermore, the metrics of Protégé [3] state that there are 14,685 axioms within the ontology.

4. Ontological Reasoning Applied

Based on this ontology, we show how ontological reasoning using instance classification and subsumption can be applied in our use case of assigning a compatible test environ-

[3] https://protegewiki.stanford.edu/wiki/Main_Page

ment to the software under test together with test cases. The reasoner performs instance checking over instances, and subsumption reasoning over classes.

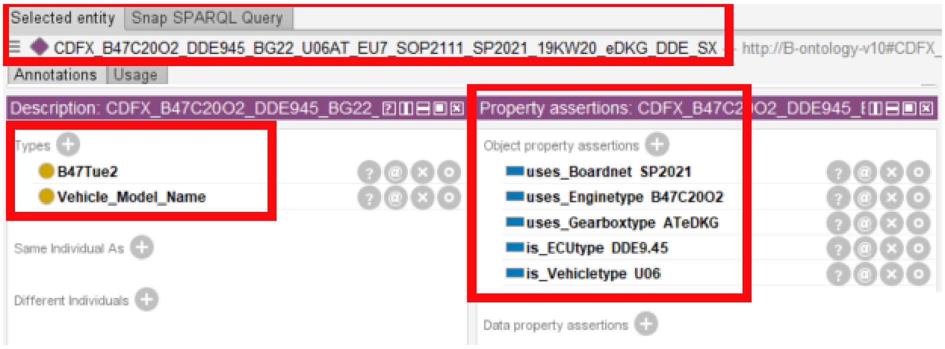
4.1. Using Instance Checking

The reasoner infers instances through instance checking with specified rules and retrieves them under a defined class, such that those queried instances can be combined with each other. The information needed for the reasoning consists of the known attributes of an entity, which need to be matched through the same attributes to another entity in the ontology.

Let us illustrate the application of this instance classification approach to our use case. There are two instances given, one instance of a Vehicle-Model and one instance of a Dataset. Their properties are mapped to properties of the test environment, i.e., the properties of its HIL and ECU (or its SIL, respectively).

Furthermore, there is a class "Vehicle_Model_to_Software_Mapping" defined with "equivalent" class axioms, making it a *defined class*. In the following example, the two instances are retrieved as instances of this defined class according to the axioms of the class.

Figure 4 shows an instance of a Vehicle-Model as selected entity in Protégé. The name of the instance is a long identifier and begins with "CDFX_...". The identifier of the selected instance is marked in the top red square, having a purple diamond shape to the left of the instance's name. It has five object property assertions (blue squares), shown in the bottom right red square, and is from class types "Vehicle_Model_Name" and "B47Tue2" (yellow circles), shown in the bottom left red square.


Selected entity | Snap SPARQL Query

CDFX_B47C20O2_DDE945_BG22_U06AT_EU7_SOP2111_SP2021_19KW20_eDKG_DDE_SX — http://B-ontology-v10#CDFX_

Annotations | Usage

Description: CDFX_B47C20O2_DDE945_BG22_

Types
- B47Tue2
- Vehicle_Model_Name

Same Individual As

Different Individuals

Property assertions: CDFX_B47C2O2_DDE945_

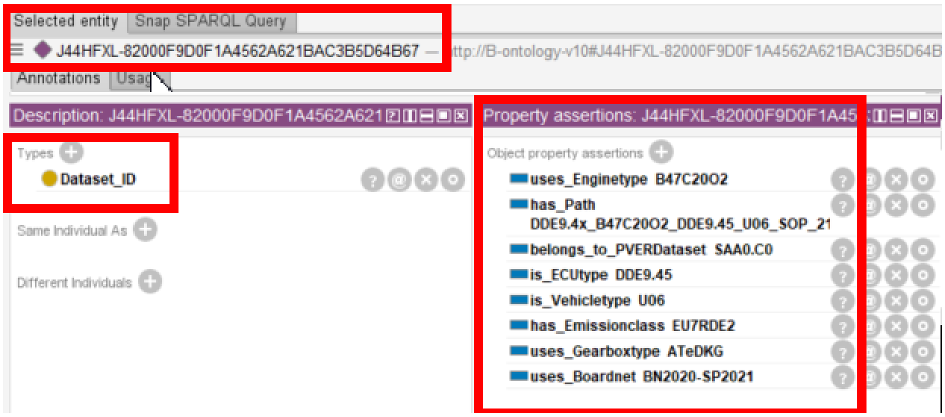
Object property assertions
- uses_Boardnet SP2021
- uses_Enginetype B47C20O2
- uses_Gearboxtype ATeDKG
- is_ECUtype DDE9.45
- is_Vehicletype U06

Data property assertions

Figure 4. An instance of the class Vehicle-Model with its properties.

Figure 5 shows a selected instance of a Software Dataset with an instance identifier beginning with "J44HFXL...". The identifier of the selected instance is marked in the top red square of the figure. The class type, shown in the bottom left red square, is "Dataset_ID" and there are eight asserted object properties, shown in the bottom right red square.

A defined class "Vehicle_Model_to_Software_Mapping" is shown in Figure 6. The identifier of the selected defined class is marked in the top red square. The selected defined class has an "Equivalent To" field, marked in the middle red square of the figure. This field contains "equivalent" class axioms denoting the classes and properties that are equivalent to the selected class, and thus defining the class. The instances that are retrieved by the semantic reasoner as instances of the selected defined class, are shown in the bottom red square of the figure. In fact, the reasoner retrieves the instances shown in Figures 4 and 5.



Selected entity | Snap SPARQL Query

J44HFXL-82000F9D0F1A4562A621BAC3B5D64B67 — ttp://B-ontology-v10#J44HFXL-82000F9D0F1A4562A621BAC3B5D64B

Annotations | Usag

Description: J44HFXL-82000F9D0F1A4562A621

Types
Dataset_ID

Same Individual As

Different Individuals

Property assertions: J44HFXL-82000F9D0F1A45

Object property assertions
- uses_Enginetype B47C20O2
- has_Path DDE9.4x_B47C20O2_DDE9.45_U06_SOP_21
- belongs_to_PVERDataset SAA0.C0
- is_ECUtype DDE9.45
- is_Vehicletype U06
- has_Emissionclass EU7RDE2
- uses_Gearboxtype ATeDKG
- uses_Boardnet BN2020-SP2021

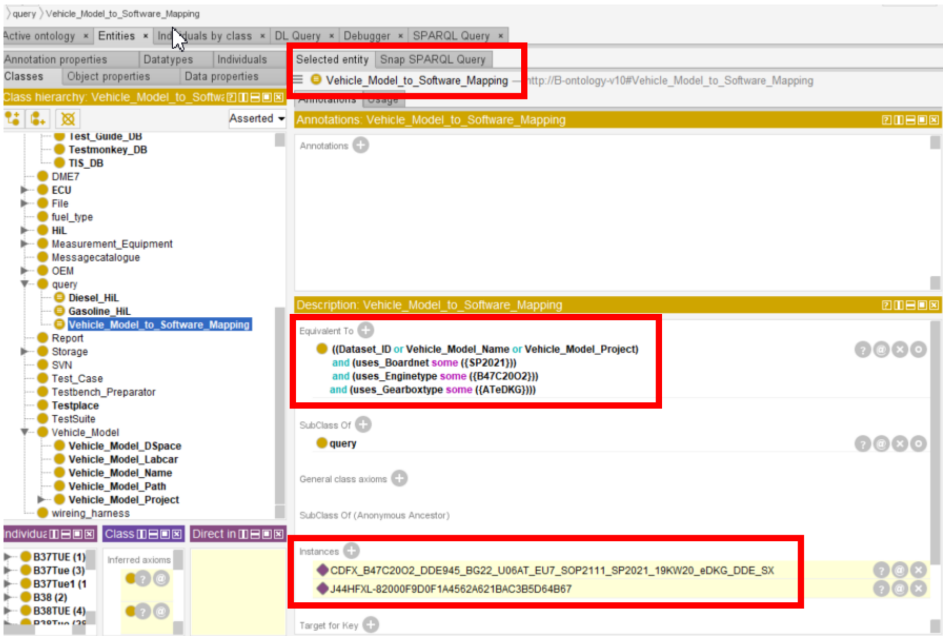
Figure 5. An instance of a Software Dataset with its properties.

Since the properties of the software under test are known for this use case, the defined class should retrieve all instances with these properties to find a match and limit the compatible test environments to use. Therefore, the defined class is set as equivalent to a set of classes, in combination with a set of asserted properties, which define the instances. Thus, under “Equivalent To” this statement is defined:

$$\begin{aligned}((textscDataset\_ID\ or\ Vehicle\_Model\_Name\ or\\ Vehicle\_Model\_Project)\ and\\ (uses\_Boardnetsome(\{SP2021\}))\ and\\ (uses\_Enginetype\ some\ (\{B47C20O2\}))\\ and\ (uses\_Gearboxtype\ some\ (\{ATeDKG\})))\end{aligned} \tag{1}$$

In such a statement, “or” defines the union of concepts, whereas “and” defines an intersection of concepts. The first part of the statement, (Dataset_ID or Vehicle_Model_Name or Vehicle_Model_Project), defines the union of the classes Dataset_ID, Vehicle_Model_Name and Vehicle_Model_Project. Assuming that this part of the statement was given by itself under “Equivalent To”, it would denote, that an instance of “Vehicle_Model_to_Software_Mapping” has to be an instance of this union of classes.

The first part of the statement is further intersected with three “equivalent” class axioms, each having an object property, an existential role and a fixed instance. The existential role is defined by the keyword “some” (which can also be read as “at least one of”), i.e., the class expression syntax for an existential restriction. The curly brackets {} describe a class of specific individuals, in this case a single instance. Hence, the first class axiom, “uses_Boardnet some ({SP2021})”, defines that every instance of the



```
Equivalent To
((Dataset_ID or Vehicle_Model_Name or Vehicle_Model_Project)
 and (uses_Boardnet some ({SP2021}))
 and (uses_Enginetype some ({B47C20O2}))
 and (uses_Gearboxtype some ({ATeDKG})))

SubClass Of
query

General class axioms

SubClass Of (Anonymous Ancestor)

Instances
CDFX_B47C20O2_DDE945_BG22_U06AT_EU7_SOP2111_SP2021_19KW20_eDKG_DDE_SX
J44HFXL-82000F9D0F1A4562A621BAC3B5D64B67

Target for Key
```

Figure 6. A defined class Vehicle_Model_to_Software_Mapping with queried instances.

class axiom has to have *at least* one object property assignment of uses_Boardnet to an instance {SP2021}. Therefore, the intersection of the first part of the statement with the three "equivalent" class axioms defines the class "Vehicle_Model_to_Software_Mapping" to be the "type of" all instances that are "type of" any of the classes in the union and are connected to fixed instances through the defined object properties.

The semantic reasoner checks and retrieves the instance of a Vehicle Model from Figure 4 and the software under test instance from Figure 5 as instances of this defined class "Vehicle_Model_to_Software_Mapping" from Figure 6, and infers in this way, that this software under test can be tested by the test environment that uses this Vehicle Model.

4.2. Using Subsumption Reasoning

The semantic reasoner also builds a hierarchy and infers information that can be used for subsumption reasoning in the ontology. More specifically, to determine the possible test environments on which an ECU software can be tested, subsumption reasoning can be applied in our use case. We present an example for subsuming a vehicle-model project below an IO Test gasoline project, and other projects under diesel projects, respectively.

The basic idea is to create two classes, each with a special meaning, and define them with the same axioms. These classes will subsume other classes that are subclasses of classes defined by those axioms and, therefore, give them the same special meaning. Since the classes are defined with the same axioms, they are also inferred to be equivalent to each other.

Figure 7 shows a simple presentation of how the vehicle-model class B47Tue is subsumed. On the left hand side of the arrow, the classes are depicted as circles surrounding their class names. Below the circles are the class axioms defining the classes. Only the class B47Tue is not exactly defined by class axiom given below, but it is a subclass of a class defined by that axiom. On the right hand side of the arrow, the inferred result after subsumption reasoning is shown, i.e., that classes IO_Test_for_GasolineProject and Vehicle_Model_GasolineProject subsume the class B47Tue.

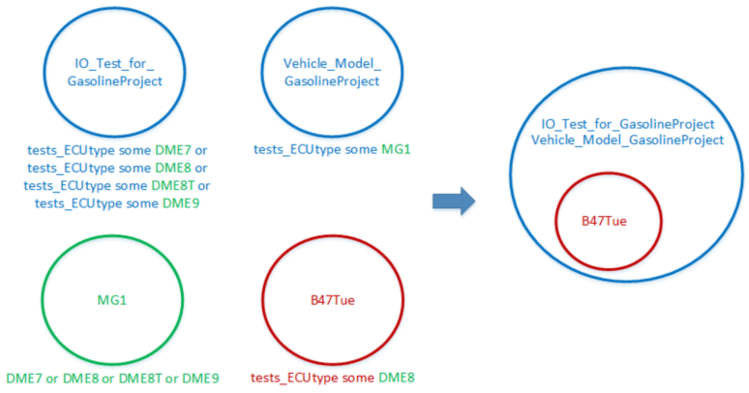
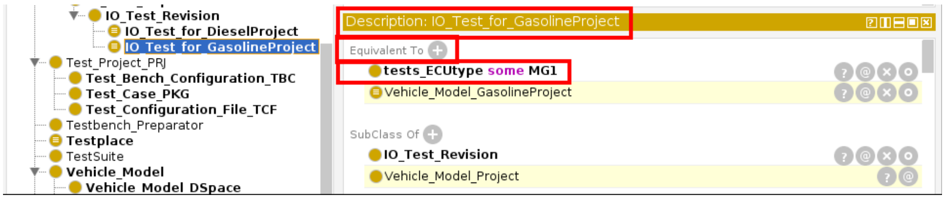

IO_Test_for_ GasolineProject
tests_ECUtype some DME7 or tests_ECUtype some DME8 or tests_ECUtype some DME8T or tests_ECUtype some DME9
Vehicle_Model_ GasolineProject
tests_ECUtype some MG1
MG1
DME7 or DME8 or DME8T or DME9
B47Tue
tests_ECUtype some DME8
IO_Test_for_GasolineProject Vehicle_Model_GasolineProject
B47Tue

Figure 7. Simplified presentation of how class B47Tue is inferred as a subclass of classes IO_Test_for_GasolineProject and Vehicle_Model_GasolineProject.

From a high-level perspective, the example shows a Vehicle-Model class, which is a subclass of a class axiom (building a superclass for the Vehicle-Model class). This class axiom is specified together with other class axioms as equivalent to other classes in the ontology and *defines* them. The defined classes are an IO Test gasoline project and a vehicle model gasoline project in this example. They represent gasoline projects, whereas other classes are defined for diesel projects, respectively.

Through subsumption reasoning, these classes subsume the vehicle-model class according to their superclass axiom. The IO Tests for gasoline projects are subsumed in the same way. Both, the gasoline IO Tests and the gasoline vehicle-model projects, are subsumed under these defined classes, which declare that they are gasoline projects and are possibly compatible. In other words, the reasoner maps a set of IO Tests – from an IO Test gasoline software project – to a gasoline vehicle-model project running on a test environment that is able to test gasoline ECU types.

Figure 8 shows the class IO_Test_for_GasolineProject, which defines all IO Test projects that test gasoline ECU types.



IO_Test_Revision
IO_Test_for_DieselProject
IO_Test_for_GasolineProject
Test_Project_PRJ
Test_Bench_Configuration_TBC
Test_Case_PKG
Test_Configuration_File_TCF
Testbench_Preparator
Testplace
TestSuite
Vehicle_Model
Vehicle_Model_DSpace

Description: IO_Test_for_GasolineProject

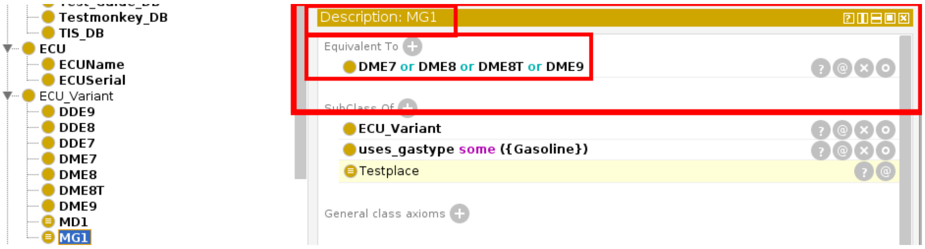
Equivalent To
tests_ECUtype some MG1
Vehicle_Model_GasolineProject

SubClass Of
IO_Test_Revision
Vehicle_Model_Project

Figure 8. A class IO_Test_for_GasolineProject with its properties in Protégé.

The class is defined through necessary and sufficient conditions in the "Equivalent To" field, marked in the red square. These "equivalent" class axioms specify a class (yellow circles denote classes in Protégé) and define the properties of the class "IO_Test_for_GasolineProject": tests_ECUtype some MG1.

This class specification consists of an asserted axiom, having an object property, an existential role and a class. The class axiom "tests_ECUtype *some* MG1" denotes, that an instance of the class with this "equivalent" class axiom tests *at least* one ECU of type "MG1". Figure 9 shows the class "MG1", which is defined through the "equivalent" class axioms: DME7 or DME8 or DME8T or DME9

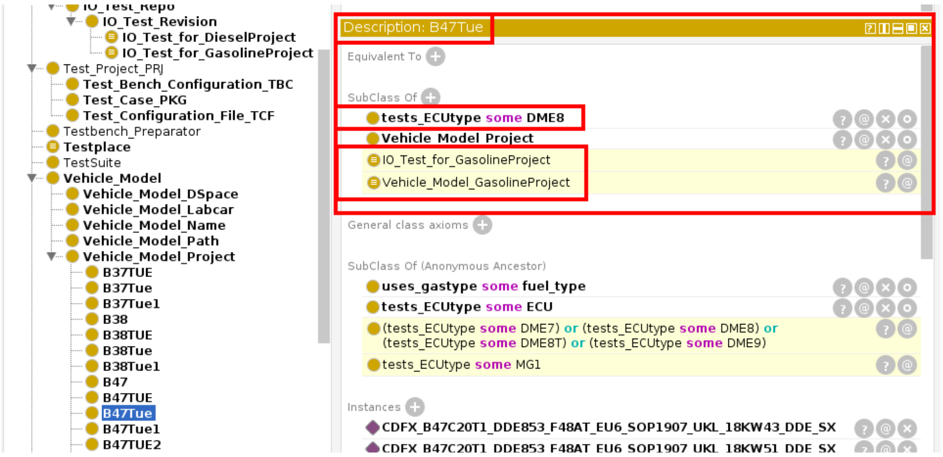

Testmonkey_DB
TIS_DB
ECU
ECUName
ECUSerial
ECU_Variant
DDE9
DDE8
DDE7
DME7
DME8
DME8T
DME9
MD1
MG1

Description: MG1
Equivalent To
DME7 or DME8 or DME8T or DME9
SubClass Of
ECU_Variant
uses_gastype some ({Gasoline})
Testplace
General class axioms

Figure 9. A class MG1 with its properties.

In the class "MG1", these four axioms are linked with the "or"-operator. This operator enables the union (in terms of set theory) of all classes that are defined with one or more of these four axioms.

During the process of hierarchy building through the semantic reasoner, it subsumes the vehicle-model class B47Tue under both the classes IO_Test_for_GasolineProject and Vehicle_Model_GasolineProject, the result can be seen in Figure 10. In Protégé, the inferred reasoning result is shown with yellow background color. This subsumption reasoning, marked in the red square in Figure 10, happens, because class B47Tue was asserted as subclass of "DME7 or DME8 or DME8T or DME9" (equivalent to MG1), and therefore a subclass of IO_Test_for_GasolineProject.


IO_Test_Revision
IO_Test_for_DieselProject
IO_Test_for_GasolineProject
Test_Project_PRJ
Test_Bench_Configuration_TBC
Test_Case_PKG
Test_Configuration_File_TCF
Testbench_Preparator
Testplace
TestSuite
Vehicle_Model
Vehicle_Model_DSpace
Vehicle_Model_Labcar
Vehicle_Model_Name
Vehicle_Model_Path
Vehicle_Model_Project
B37TUE
B37Tue
B37Tue1
B38
B38TUE
B38Tue
B38Tue1
B47
B47TUE
B47Tue
B47Tue1
B47TUE2

Description: B47Tue
Equivalent To
SubClass Of
tests_ECUtype some DME8
Vehicle_Model_Project
IO_Test_for_GasolineProject
Vehicle_Model_GasolineProject
General class axioms
SubClass Of (Anonymous Ancestor)
uses_gastype some fuel_type
tests_ECUtype some ECU
(tests_ECUtype some DME7) or (tests_ECUtype some DME8) or (tests_ECUtype some DME8T) or (tests_ECUtype some DME9)
tests_ECUtype some MG1
Instances
CDFX_B47C20T1_DDE853_F48AT_EU6_SOP1907_UKL_18KW43_DDE_SX

Figure 10. Class B47Tue was inferred as subclass of classes IO_Test_for_GasolineProject and Vehicle_Model_GasolineProject.

Since the vehicle-model project classes use the same ECU type classification (gasoline or diesel) as the IO Test project classes, they are subsumed as well. For instance, the subsumed IO Test projects and vehicle-models in the class “B47Tue” are possibly compatible to each other, but definitely incompatible to diesel software projects and diesel vehicle-models, respectively.

5. Integration and Usage with Tool Support

This section describes the integration of the ontological reasoning approach and its implementation using Stardog and SPARQL. It is deployed at our industry partner alongside existing applications and extends the backend database with an ontology that is queried using Stardog’s HTTP API.

Stardog allows configuring the “reasoning type” according to OWL2 *profiles*, which are syntactic subsets of OWL2, offering syntactic restrictions to the language. Any axiom outside the selected type will be ignored by the reasoner. These restrictions result in a trade-off between the language’s expressive power and implementational and/or computational benefits. “SL” is a Stardog-specific profile representing a combination of RDFS, QL, RL, and EL axioms (plus SWRL rules) from other OWL2 profiles. The option “SL” was chosen as standard “reasoning type” for the ontology. Another configuration option was “sameAs reasoning”, which can be set to ON/OFF. The OWL2 “sameAs” directive denotes that an instance is equivalent to another instance. While other reasoning is performed in a “lazy” (late-binding) way at query-time, “sameAs reasoning” inferences are computed and indexed eagerly, such that these inferences can be used directly at query-time. Since there was a bug with “sameAs reasoning” in Stardog, relatively simple queries timed out with this option turned on. Therefore, “sameAs reasoning” was turned off and for certain cases a work-around had to be implemented using an object property “same_as”.

Additionally, an option was configured that enables Stardog to approximate axioms that are outside the supported profile and normally ignored. Some database configurations can be performed with an active database and others require a restart of the database.

For identifying ontologies and their elements, OWL2 uses Internationalized Resource Identifiers (IRI), which can be very long strings. Stardog allows replacing them using a namespace prefix binding configuration. For example, the IRI “http://B-ontology-v10#U06” is the unique identifier for the vehicle-type “U06”. The prefix of this IRI (“http://B-ontology-v10#”) is used in all other elements and it makes sense to define a prefix binding for it. The prefix “B:” was configured for the IRI “http://B-ontology-v10#” and Stardog replaces this prefix with the IRI in the background. Both can be used in a query, the full IRI and the element with prefix binding.

Interaction with the system is supported through a frontend application, which queries the Stardog API. Predefined SPARQL queries are prepared and stored on Stardog and can be directly invoked. These queries accept input variables (fetched from databases) and allow selecting specific outputs from the ontology, while reasoning is activated. Listing 1 shows an example of a prepared SPARQL query that accepts a Program Version (PVER) and a Dataset as input variables, which define the software under test. This query outputs a Vehicle-Model that has the same dependencies.

```
1  PREFIX B: <http://B-ontology-v10#>
2  SELECT  DISTINCT ?proj ?model  ?dsvehicletype
3    ?boardnet ?enginetype ?gearbox
4  WHERE {
5    $pver B:belongs_to_project ?proj .
6    ?proj rdf:type ?mproj .
7    ?mproj rdfs:subClassOf B:Vehicle_Model_Project .
8    $dataset B:belongs_to_PVERDataset $pver .
9    $dataset B:is_Vehicletype ?dsvehicletype ;
10       B:uses_Boardnet ?boardnet ;
11       B:uses_Enginetype ?enginetype ;
12       B:uses_Gearboxtype ?gearbox .
13   ?model rdf:type B:Vehicle_Model_Name .
14   ?model B:is_Vehicletype ?dsvehicletype ;
15       B:uses_Enginetype ?enginetype ;
16       B:uses_Gearboxtype ?gearbox .
17 }
```

Listing 1: Sample stored SPARQL query with input variables.

The prepared query is stored with a specific name on Stardog and is called with this name through the API via HTTP GET or POST request. In Listing 1, a PREFIX is set at the beginning (line 1). The sample query accepts two input variables ($pver and $dataset in lines 5 and 8) and selects six output variables (?proj, ?model, ?dsvehicletype, ?boardnet, ?enginetype and ?gearbox in lines 2 and 3). Basically, the query retrieves instances of PVER and Dataset and outputs the corresponding software project (?proj), vehicle-model (?model), vehicle-type (?dsvehicletype), boardnet (?boardnet), engine-type (?enginetype) and gearbox-type (?gearbox). A triple with the predicate "belongs_to_project" is produced. The Subject is a given PVER in the variable $pver and the object is the corresponding software project in the variable ?proj.

Line 6 shows a triple with the software projects in variable ?proj connected to Vehicle-Model Projects in variable ?mproj through the predicate "rdf:type". The reason is that a Vehicle-Model Project in the ontology is defined as a superclass of RQONE software projects. The next produced triple in line 7 denotes that ?mproj has to be subclass of "Vehicle_Model_Project".

The input variable $dataset is used in the produced triple in line 8 with the predicate "belongs_to_PVERDataset" and the variable $pver. Then, in lines 9-12 a *Predicate List* is produced with $dataset as Subject and some referenced Predicates and Objects.

In line 13, the variable ?model is declared as type of "Vehicle_Model_Name". Finally, a *Predicate List* is produced with Objects ?dsvehicletype, ?enginetype and ?gearbox, which were used above in the Predicate List referencing to $dataset.

The result is serialized in a form defined in the HTTP Header. In this case, JSON was chosen as exchange format. Example results with specific input variables passed to the query in Listing 1 are shown in Listing 2.

```
{
  "head": {
    "vars": [
      "proj",
      "model",
      "dsvehicletype",
      "boardnet",
      "enginetype",
      "gearbox"]},
  "results": {
    "bindings": [
    {
      "dsvehicletype": {
        "type": "uri",
        "value": "http://B-ontology-v10#U06"
      },
      "boardnet": {
        "type": "uri",
        "value": "http://B-ontology-v10#BN2020-SP2021"
      },
      "proj": {
        "type": "uri",
        "value": "http://B-ontology-v10#Project_RQONE01071608"
      },
      "model": {
        "type": "uri",
        "value": "http://B-ontology-v10#CDFX_B47C20O2_DDE945_BG22_U06
    AT_EU7_SOP2111_SP2021_19KW20_DDE_SX"
      },
      "gearbox": {
        "type": "uri",
        "value": "http://B-ontology-v10#ATDKG"
      },
      "enginetype": {
        "type": "uri",
        "value": "http://B-ontology-v10#B47C20O2"
      }}]}
  }
```

Listing 2: Example Results in JSON format

The frontend-application, which is completely independent of Stardog, implements the services through passing the prepared query's name and input variables as parameters in the POST request body. Furthermore, it passes the authentication credentials to be able to access the Stardog API. Usually, the frontend-application obtains this from specially created database tables, where the developers try to fill the data accordingly. The Stardog SPARQL queries are used as verification support for different parts of the defined use case. The example above shows how it can be checked which Vehicle-model can test which PVER and Dataset. Another query that works similarly, deals with identifying which IO-Test Project can be tested on which Vehicle-model.

6. Related Work

In technical sciences and industry, ontologies can be used for formal specification of domain knowledge. We provide here an overview of applications of ontologies in the automotive industry.

Alvares-Coello and Gómez [8] discuss how ontologies can benefit vehicle architectures in the automotive industry. They propose an ontology-based approach for integrating vehicle-related data with applications. To this end, application-specific data is annotated with well-defined semantic models and combined with vehicle-related data for facilitating more stable queries over the lifetime of applications.

In [9], an ontology- and rule-based approach using F-Logic [10] is specified and a small prototype is described. The prototype analyses HiL test data with additional formalized rules that are derived from interviews with experts. The test data is recorded during test runs on a HiL system and imported as instances into the ontology. Test instances are checked for rules that are defined in concepts in the ontology and errors are detected and highlighted. In contrast to our work, the authors do not consider using subsumption reasoning nor the use case of mapping test cases to test environments.

An ontological approach for collecting incidents of car breakdowns and connecting them to repair instructions is implemented in [11]. For searching breakdown documents of different vehicle types, a semantic search engine using OWL has been developed.

In [12] and [13], context information is modeled using OWL ontologies and Protégé, for helping drivers handling the car based on contextual information.

7. Discussion

Through specification of knowledge in the ontology, the expert knowledge necessary to assign systems to test environments is gathered at a central place. The landscape of interdependent entities in this domain of interest is both larege and complex.

The granularity of the entity specification can be refined. For instance, test-system components can be further analyzed and fragmented into smaller parts, which can have further dependencies to other parts. This would lead to more detailed dependency management and allow for different use cases within the context of automotive software and hardware testing.

Regularly automated import of relevant data to the ontology is a practical challenge. We have already analyzed import possibilities as well as tooling for gathering and importing of relevant data from heterogeneous sources. Technically, the automated import from such tools can be accomplished by interfacing Stardog.

8. Conclusion

In this paper, we present an approach for improving automated testing in the automotive industry using an ontology and ontological reasoning. We chose to specify the expert knowledge in an ontology and to model dependencies between systems as well. While there are several use cases where an ontology could provide dependency management and decision support, one specific use case was defined for the scope of this paper: for given test cases and software under test, a compatible test environment is found, by taking into account the attributes of the test environment, software and test cases.

For accomplishing this automated mapping, we propose an approach based on ontological reasoning. First, we use instance classification where we infer for specific instances, in our application test environments, to which defined classes they belong, essentially enabling the automatic combination of those queried instances. Secondly, we use subsumption reasoning to determine the test environments compatible with given test cases of a software under test.

We deployed our ontological reasoning approach using Stardog and SPARQL. Stardog provides the means for knowledge graph representation and allows for querying and manipulating the ontology, which is imported and stored as a knowledge graph. Furthermore, reasoning mechanisms that are supported by Stardog were used for the purpose of ontological reasoning.

By doing so, we were able to demonstrate our approach for the defined use case with our prototype deployed at our industry partner. The prototype showed that ontological reasoning can support the process of automated software testing in the automotive industry.

Acknowledgments

Part of this research has been carried out in the VerASoS project (No. 861210), funded by the Austrian Federal Ministry of Transport, Innovation and Technology (BMVIT) under the program "ICT of the Future" of the Austrian FFG.

References

[1] Paradzikovic P. Defining and using an ontology of test environments [Master Thesis]. TU Wien; 2019.

[2] Hitzler P, Krötzsch M, Rudolph S, Sure Y. Semantic Web: Grundlagen. Berlin: Springer; 2008.

[3] Klyne G, Carroll JJ. RDF Resource Description Framework Concepts and Abstract Syntax. W3C; 2004. https://www.w3.org/TR/rdf-concepts/#section-data-model.

[4] Gaševic D, Djuric D, Devedžic V. Model Driven Engineering and Ontology Development. Berlin, Heidelberg: Springer Berlin Heidelberg; 2009. Available from: https://doi.org/10.1007/978-3-642-00282-3_2.

[5] Gruber TR. A Translation Approach to Portable Ontology Specifications. Knowl Acquis. 1993 Jun;5(2):199-220. Available from: http://dx.doi.org/10.1006/knac.1993.1008.

[6] Baader F, Calvanese D, McGuinness DL, Nardi D, Patel-Schneider PF. The Description Logic Handbook: Theory, Implementation and Applications. 2nd ed. New York, NY, USA: Cambridge University Press; 2010.

[7] Pascal H, Markus K, Sebastian R. Foundations of Semantic Web Technologies. 1st ed. Chapman and Hall/CRC; 2009.

[8] Alvarez-Coello D, Gómez JM. Ontology-Based Integration of Vehicle-Related Data. In: 15th IEEE International Conference on Semantic Computing, ICSC 2021, Laguna Hills, CA, USA, January 27-29, 2021. IEEE; 2021. p. 437-42. Available from: https://doi.org/10.1109/ICSC50631.2021.00078.

[9] Syldatke T, Chen W, Angele J, Nierlich A, Ullrich M. How Ontologies and Rules Help to Advance Automobile Development. In: Paschke A, Biletskiy Y, editors. Advances in Rule Interchange and Applications. Berlin, Heidelberg: Springer Berlin Heidelberg; 2007. p. 1-6.

[10] Kifer M, Lausen G, Wu J. Logical Foundations of Object-Oriented and Frame-based Languages. J ACM. 1995 Jul;42(4):741-843. Available from: http://doi.acm.org/10.1145/210332.210335.

[11] Reymonet A, Thomas J, Aussenac-gilles N. Ontology Based Information Retrieval: an application to automotive diagnosis. In: Linköping University, Institute of Technology; 2009. p. 914.

[12] Madkour M, Maach A. Ontology-Based Context Modeling for Vehicle Context-Aware Services. Journal of Theoretical and Applied Information Technology. 2011 12;31.

[13] Lüddecke D, Bergmann N, Schaefer I. Ontology-Based Modeling of Context-Aware Systems. In: Dingel J, Schulte W, Ramos I, Abrahão S, Insfran E, editors. Model-Driven Engineering Languages and Systems. Cham: Springer International Publishing; 2014. p. 484-500.