SoftwareX 12 (2020) 100571

Contents lists available at ScienceDirect

SoftwareX

journal homepage: www.elsevier.com/locate/softx

Original software publication

OWL2Go: Auto-generation of Go data models for OWL ontologies with integrated serialization and deserialization functionality

Stefan Dähling [a,*], Lukas Razik [b], Antonello Monti [a]

[a] Institute for Automation of Complex Power Systems at E.ON Energy Research Center of RWTH Aachen University, Mathieustr. 10, 52074 Aachen, Germany

[b] Institute of Energy and Climate Research, Energy Systems Engineering of Forschungszentrum Jülich GmbH, 52425 Jülich, Germany

ARTICLE INFO

Article history:
Received 2 March 2020
Received in revised form 10 July 2020
Accepted 10 July 2020

Keywords:
Ontology
OWL
Go
SAREF

ABSTRACT

The Web Ontology Language (OWL) is a formal language for the description of terms and their relationship in a certain domain. It enables information exchange among heterogeneous applications and devices in a machine-readable format. However, in software development the usage of data models is common. In order to facilitate the usage of ontologies encoded in OWL also in software development we present OWL2Go. OWL2Go is a code-generator that parses an OWL ontology and generates a Go package implementing a data model compliant with the ontology as well as a serializer and deserializer for conversion between the Go data model and Turtle or JSON-LD documents. We demonstrate the generation process and the usage of the resulting Go package with the Smart Appliances REFerence (SAREF) ontology.

Code metadata

Current code version	v1.1.0
Permanent link to code/repository used for this code version	https://github.com/ElsevierSoftwareX/SOFTX_2020_76
Code Ocean compute capsule	
Legal Code License	Apache License, Version 2.0 or MIT License
Code versioning system used	git
Software code languages, tools, and services used	Go
Compilation requirements, operating environments & dependencies	imports module github.com/piprate/json-gold v0.3.0
If available Link to developer documentation/manual	https://git.rwth-aachen.de/acs/public/ontology/owl/owl2go
Support email for questions	SDaehling@eonerc.rwth-aachen.de

1. Motivation and significance

Semantic interoperability is important for the integration of distributed and heterogeneous applications. Ontologies enable the formal description of a domain, the objects it contains and their relationship. The Web Ontology Language (OWL) is the most prominent language for expressing ontologies [1,2]. An emerging field of application for ontologies is the Internet of Things (IoT) [3,4].

OWL is based on the Resource Description Framework (RDF) and RDF Schema (RDFS). RDF can be used to describe resource relations in form of triples, consisting of subject, predicate and object [5]. Resources are identified by their unique International Resource Identifiers (IRIs). In order to give further structure to the knowledge expressed by RDF, it is complemented by RDFS [5,6]. RDFS adds the concept of classes. OWL builds upon RDFS and adds the concept of restrictions [5]. A restriction can be used to add properties to a class, restricting the values that can be assigned to that property.

In software development data models are used to describe the problem that the software aims to solve, together with its domain. Data models are usually specific to a certain implementation and depend on the chosen programming language or data storage mechanisms (databases). While many modeling features of OWL also exist in object-oriented programming languages, e. g., inheritance, some constructs cannot be directly mapped due to design characteristics of a specific programming language, e. g.,

* Corresponding author.
E-mail address: sdaehling@eonerc.rwth-aachen.de (S. Dähling).

https://doi.org/10.1016/j.softx.2020.100571

static typing or lack of support for multiple inheritance [7]. In order to enable interoperability and data exchange among applications within the same domain, two challenges have to be overcome:

- A mapping between a given OWL ontology and a data model is needed.
- Deserialization of a document, containing individuals, to instances of the data structures defined by the mapping must be enabled. Additionally, the inverse process of serializing a set of instances to a document is required.

Different solutions for the mapping of OWL to a programming language have been proposed in literature, typically for class-based programming languages like JAVA [3,8–10]. An extensive overview and classification can be found in [11]. The authors distinguish three categories of methods for mapping OWL to object-oriented programming languages. Applying this classification, OWL2Go belongs to the second category of active–static mappings. Software in this category translates the ontology to code statements, once before compilation. Subsequently data can be accessed and manipulated by means of an application programming interface (API). Other concepts in that category are for example [3,9,10]. These tools generate proxy classes according to class definitions in an OWL ontology. The proxy classes offer an API to manipulate underlying RDF triple sets or graph structures. In case of [3] other third party tools are reused for that. While this approach minimizes the code to be generated, it introduces an additional procedure for interaction between the proxy classes and the underlying triple store. OWL2Go eliminates this additional effort. In [8] a mapping from OWL to JAVA is presented. Based on an OWL ontology, code for JAVA classes and interfaces is auto-generated. In contrast to the tools analyzed before, no underlying graph or triple store is required. In OWL2Go the data is stored directly in the generated data model, as well. However, OWL2Go targets the programming language Go.

Go is a modern open-source programming language. It especially targets high performance applications for system level and networking. That is why it is widely used in the area of cloud computing and the IoT. One main difference between Go and many other object oriented programming languages, is that Go does not have a class and inheritance model [12]. OWL2Go offers a methodology to mimic a class-like behavior, in order to enable a mapping from the class-based OWL to Go. The importance of Go in the domain of web applications and the possibility to map OWL classes to Go, as described in this paper, motivate its use as target programming language. A more detailed introduction of Go and its relevance as well as its suitability for ontology-based applications is discussed in Section 2.1.

Serialization and deserialization enable the exchange of data in standardized format. Thereby, applications can interact with each other and share knowledge, even if the data models specific to each application differ. Previous implementations introduced before tackle this challenge in different ways. In [8] the corresponding feature is not discussed at all. In [3] the implementation relies on third-party tools which implement a RDF triple store and also provide serialization and deserialization. In OWL2Go this functionality is directly supported by the generated Go package. Therefore, no third-party tool is necessary which reduces dependencies and hence, eases the use of the generated package.

Regarding the challenges mentioned before, our contributions are:

- a methodology for a mapping from the class-based OWL to the non class-based Go, as well as the automatic generation of a Go package which
- implements a data model compliant with the OWL ontology and
- implements serialization / deserialization to / from the Turtle (ttl) / JSON-LD document format.

Therefore, OWL2Go is a tool to ease and thereby further increase the usage of ontologies in software development. OWL2Go enables software developers and researches to facilitate the use of ontologies. In order to do so they do not have to interact with the description logic of OWL, i.e., the triples. Instead they use the structures and methods, i.e., the API, generated by OWL2Go. Therefore, the hurdle to develop ontology-based applications is substantially decreased. This is identified as important technical requirement for the development of ontology-based applications [13].

One specific use case that has motivated the development of OWL2Go is the Smart Appliances REFerence (SAREF) ontology [14,15] which is formalized in OWL. SAREF was designed to allow interoperability among heterogeneous IoT-devices in the household area, supporting energy management at system level. The European Telecommunications Standards Institute (ETSI) has released SAREF and further extensions as technical specification [16–19]. Despite the novelty of SAREF already some applications of it can be found in literature [4,20,21].

While the methodology presented in this paper works for any OWL ontology (subject to limitations described in Section 2.4), we explain and demonstrate our concept using the SAREF ontology.

2. Software description

2.1. The Go programming language

Go (not to be confused with GO!, the multi-paradigm language used for knowledge representation tasks [22]) is a compiled language which was developed especially for networked servers [12,23] and therefore is also suitable for the IoT and cloud computing. The syntax is C-like but it provides features of many other languages, e.g., native support for concurrency or automatic memory management.

In the cloud computing area, popular tools such as Kubernetes [24] and Docker [25] are developed in Go. With cloud computing and the IoT being emerging fields of application for ontologies, the relevance of Go as popular programming language in this field is evident.

The main challenge for mapping OWL to Go is the absence of classes and inheritance in Go which is a key feature of OWL and also most other object-oriented programming languages. Go provides *structs* to define custom types. A similar effect as in class-based inheritance can be achieved by embedding of structs. A struct that embeds another struct automatically inherits the fields defined by the embedded struct. As an orthogonal concept Go offers *interfaces*. An interface consists of a set of method declarations. Every struct (or native type) that implements these methods implicitly also implements the interface. Interfaces can be used to achieve a similar behavior as polymorphism in other programming languages (e.g. C++). Also interfaces can be embedded. An interface that embeds another interface declares the methods of the embedded interface.

The mapping of the class-based OWL to Go is the main contribution of this paper. In the following we present how Go structs, interfaces and the concept of embedding can be used to implement classes and inheritance as defined by OWL.

2.2. Software architecture

Fig. 1 shows the general workflow of the code generation process for a given OWL ontology. In a first step, the OWL class and property information has to be extracted. The ontology encoded

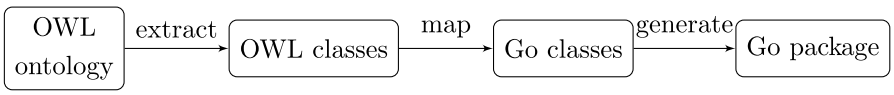

Fig. 1. Workflow of OWL2Go code generation.

Table 1
Mapping of OWL constructs to Go.

OWL construct	OWL axiom (owl:, rdf(s):)	Go construct
Class	Class A	`interface A;` `struct sA implements A`
	A subClassOf B	`interface A extends interface B`
	A intersectionOf B and C	`interface A extends interfaces B and C`
	A unionOf B and C	`interfaces B and C extend interface A`
	oneOf	enumeration: create class and individuals
Restriction (of class A) on Property B	allValuesFrom C,D someValuesFrom hasValue xCardinality	C,D: allowed types E := base type of C,D F := common type of C,D A ={… B() []E SetB ([]E) error AddB (…E) error DelB(…E) } sA = {… b map[string]F}
Property A	domain B	add property A to class B as for restrictions
	range B	add B to allowed types of property
	inverseOf B	add function call to AddB()and DelB() in AddA() and DelA() functions and vice versa
	SymmetricProperty	add function call to AddA()and DelA() in AddA() and DelA() functions
Indvidual	type	create individual when creating model
Ontology	imports	get imported ontology and execute OWL2Go for it

in ttl is used as an input for this step. Subsequently, these classes and properties have to be mapped to Go. The final step is the code generation for the Go package according to the previously performed mapping. The resulting architecture and application workflow of the generated package can be seen in Fig. 2. In this section we will describe the single application parts while Section 2.3 explains the functionalities. Section 2.4 states the limits of our approach regarding the expressiveness of OWL.

Ontologies expressed in OWL specify three kinds of objects: classes, properties and individuals. Ontologies describing classes and properties are referred to as T-box ontologies. A T-box ontology describes the terminology to describe a domain. OWL2Go maps classes and properties defined by a T-box ontology into a Go data model. Individuals are part of a A-box ontology. A-box ontologies describe instances or individuals and their relationships. They typically use the concepts introduced by a T-box ontology. The Go package that is generated by OWL2Go can be used to create instances of the resulting Go data model. Serialization of these instances to ttl or JSON-LD yields a A-box ontology compliant with the original T-box ontology.

The mapping for different OWL constructs is summarized in Table 1. Some important constructs are described in more detail below. One important aspect of OWL is the reuse of existing ontologies. OWL ontologies can import other ontologies and built upon their classes and properties. This is supported by OWL2Go. In case an ontology is imported, its ttl document is retrieved and the generation process is also executed for the imported ontology. For each ontology a specific prefix is used for the naming of classes and properties.

The SAREF ontology is used as an example to demonstrate the mapping and the code generation. OWL2Go automatically generates the code for SAREF as well as for the imported Time ontology [26]. The resulting Go package is provided as open source.[1]

2.2.1. Definition of an ontology model

In order to describe a set of objects and their relations to each other, first the introduction of a `Model` datatype is necessary. The definition of the `Model` type can be seen in Listing 1. It holds one map for each defined class, mapping an IRI to the related object. An object can be stored in multiple maps if its type is a derived class. For example an object of type `SarefAppliance` would be stored in `mSarefAppliance`, `mSarefFunctionRelated` and `mSarefDevice`. Additionally, all objects are stored in `mThing`.

A `New` function for each class is defined for the `Model` datatype. This function creates a new instance of the class and adds this instance to the maps corresponding to the class and all parent classes (see Section 2.2.4). Moreover, functions to search for specific objects and to delete them are implemented for the `Model` type.

Ontologies often predefine standard individuals that are available to the user of the ontology. In order to automatically populate the model with these individuals, the `Model` type offers the

[1] https://git.rwth-aachen.de/acs/public/ontology/owl/saref.

method `CreateIndividuals` to create them (Listing 1 line 9).

```
1  type Model struct {
2      mThing map[string]Thing
3      mSarefDevice map[string]SarefDevice
4      mSarefFunctionRelated map[string]
           SarefFunctionRelated
5      mSarefAppliance map[string]SarefAppliance
6      ...
7  }
8
9  func (mod *Model) CreateIndividuals() {
10     mod.NewSarefTask("https://w3id.org/saref#Washing
           ")
11     ...
12 }
```

Listing 1: Model data type.

The base class of all classes, generated with the concept described in the next section, is the `Thing` class. It is implemented as an interface that is embedded by every generated class. The `Thing` class allows the execution of common functions, e. g., serialization and deserialization, on all objects irrespective of their specific type.

2.2.2. Classes

To mimic inheritance in Go, each OWL class is expressed as a combination of one Go interface and one Go struct implementing the interface. This combination is now referred to as *Go class*. The interface declares methods for manipulating the class properties. Every class inherits from the `Thing`class. The interface of a Go class has to extend all the interfaces of its parent classes, i. e., it has to declare the same methods. This is achieved by embedding the interface of a parent class. This way, instances of a Go class, i. e., its struct, can be used as instances of parent classes.

An example for inheritance using Go interfaces and structs is provided in Listing 2. Class `saref:FunctionRelated` is a subclass of `saref:Device`. `saref:Device` has no parent class specified by SAREF and hence, inherits from the `Thing` class. All classes are expressed as one interface (Listing 2 lines 1, 5, 11). Each interface embeds the interface of its parent classes (Listing 2 lines 6, 12). Listing 2 also shows the struct for the `saref:FunctionRelated` class (Listing 2 line 15). This struct implements all methods declared by the corresponding interface and thereby, automatically implements the interfaces of parent classes (Listing 2 lines 16–19). As a result, `*sSarefFunctionRelated` can be used as `SarefFunctionRelated`, `SarefDevice` and `Thing`.

```
1  type Thing interface {
2      ThingMethod()
3  }
4
5  type SarefDevice interface {
6      Thing
7      SarefDeviceMethod()
8      IsSarefDevice()
9  }
10
11 type SarefFunctionRelated interface {
12     SarefDevice
13     IsSarefFuntionRelated()
14 }
15 type sSarefFunctionRelated struct {/*
       FunctionRelated properties */}
16 func (res *sSarefFunctionRelated)
       IsSarefFunctionRelated() {/*...*/}
17 func (res *sSarefFunctionRelated) IsSarefDevice() {
       /*...*/}
18 func (res *sSarefFunctionRelated) SarefDeviceMethod
       () {/*...*/}
19 func (res *sSarefFunctionRelated) ThingMethod() {/*
       ...*/}
```

Listing 2: Inheritance using Go interfaces and structs.

One special case is the definition of a class as an enumeration of individuals (`owl:oneOf`). In this case the code for the class is generated as described before. Additionally, the enumerated individuals are created in the `CreateIndividuals` methods as explained in Section 2.2.1.

2.2.3. Properties

The struct of a Go class contains one field for each property of the OWL class. Classes contain properties that are assigned to it either by means of a restriction or in a `owl:domain` axiom of the property. The interface of the class declares methods to manipulate these properties. For properties with a cardinality of one, only a `Get` and a `Set` function are declared. For properties with greater cardinality `Add` and `Delete` functions are declared additionally. For an `owl:ObjectProperty` wit cardinality greater than one, the field in the belonging struct is of type `map[string]PropertyType`. The key in the map is the IRI of the stored object. An `owl:DatatypeProperty` has a literal type. These types are mapped to primitive Go types such as `string` or `int`. In case the multiplicity is greater than one slices of these types are used.

```
1  type SarefDevice interface {
2      SarefHasTypicalConsumption() []SarefProperty
3      SetSarefHasTypicalConsumption([]SarefProperty)
           error
4      AddSarefHasTypicalConsumption(...SarefProperty)
           error
5      DelSarefHasTypicalConsumption(...SarefProperty)
6      ...
7  }
8
9  type sSarefDevice struct {
10     sarefHasTypicalConsumption map[string]
           SarefProperty
11     ...
12 }
13
14 func (res *sSarefDevice) SarefHasTypicalConsumption
       () (out []SarefProperty) {
15     out = make([]SarefProperty, len(res.
           sarefHasTypicalConsumption))
16     index := 0
17     for i := range res.sarefHasTypicalConsumption {
18         out[index] = res.sarefHasTypicalConsumption[
               i]
19         index++
20     }
21     return
22 }
23
24 func (res *sSarefDevice)
       SetSarefHasTypicalConsumption(in []
       SarefProperty) (err error) {
25     res.sarefHasTypicalConsumption = make(map[string
           ]SarefProperty)
26     err = res.AddSarefHasTypicalConsumption(in...)
27     return
28 }
29
30 func (res *sSarefDevice)
       AddSarefHasTypicalConsumption(in ...
       SarefProperty) (err error) {
31     for i := range in {
32         if v, ok := in[i].(SarefEnergy); ok {
33             res.sarefHasTypicalConsumption[v.IRI()]
                   = v
34             continue
35         } else if v, ok := in[i].(SarefPower); ok {
36             res.sarefHasTypicalConsumption[v.IRI()]
                   = v
37             continue
38         }
39         err = errors.New("Wrong Property type.
               Allowed types are
40             [Energy Power]")
41     }
42     return
43 }
44
```

```
45 func (res *sSarefDevice)
       DelSarefHasTypicalConsumption(in ...
       SarefProperty) {
46     for i := range in {
47         delete(res.sarefHasTypicalConsumption, in[i
               ].IRI())
48     }
49     return
50 }
```

Listing 3: Excerpt from Device class

OWL properties can be of multiple types. However, in most programming languages a variable can only have one specific type. Moreover, in OWL the type of a property can differ from the type of the same property in the parent class. To cope with these problems, each property is assigned three types during the mapping: a *base type*, a *common type* and (possibly multiple) *allowed types*. Listing 3 provides an example using the `saref:HasTypicalConsumption` property of the `saref:Device` class. The corresponding excerpt of the SAREF ontology can be seen in Listing 4.

```
1  saref:Device rdf:type owl:Class ;
2  rdfs:subClassOf [ rdf:type owl:Restriction ;
3      owl:onProperty saref:hasTypicalConsumption ;
4      owl:allValuesFrom [ rdf:type owl:Class ;
5          owl:unionOf ( saref:Energy saref:Power ) ]
6  ] .
7
8  saref:Property rdf:type owl:Class .
9
10 saref:Energy rdf:type owl:Class ;
11     rdfs:subClassOf saref:Property .
12
13 saref:Power rdf:type owl:Class ;
14     rdfs:subClassOf saref:Property .
```

Listing 4: SAREF excerpt for Device class and hasTypicalConsumption property.

The allowed types are all types specified within restrictions or in `owl:range` statements of the OWL property. In case of the example property, allowed types are `saref:Energy` and `saref:Power` (Listing 4 line 5).

The common type is the common parent class of all allowed types. The common parent class of `saref:Energy` and `saref:Power` is `saref:Property` (Listing 4 lines 11, 14). It is used in the related struct to define the property field (Listing 3 line 10).

The base type of a class's property corresponds to the type of a property in parent classes. Since the `saref:Device` class has no parents, the base type in the example is also `saref:Property`. All interface functions for property manipulation (Listing 3 line 2–5) use this type in order to be compatible with the interfaces of parent classes.

`SarefProperty` is the interface of the Go class that corresponds to `saref:Property`. `SarefEnergy` and `SarefPower` are the interfaces of their Go classes. Since both classes inherit from `saref:Property`, their interfaces embed `SarefProperty`.

As `sarefHasTypicalConsumption` is of type `SarefProperty` also other types that inherit from `SarefProperty` could be inserted. However, within the `Add`-function, type assertions are used to ensure that only allowed types are stored for that property (Listing 3 line 34 and 37). In case, a wrong type is used, an error is returned.

2.2.4. Individuals

OWL individuals are instances of classes. In order to create instances each Go class has a `New` function that takes an IRI and creates a new object of that class. Listing 5 shows the `NewSarefDevice` method. It is defined for the `*Model` data type. It ensures that no other resources with the same IRI exists (Listing 5 line 2) and creates a new `*sSarefDevice` (Listing 5 line 6). This object is added to maps of the `*Model` variable for the `SarefDevice` class and all its parents (in this case only `Thing`) (Listing 5 line 7–8). Lastly, all property maps are created (Listing 5 line 9–10).

```
1  func (mod *Model) NewSarefDevice(iri string) (ret
       SarefDevice, err error) {
2      if mod.Exist(iri) {
3          err = errors.New("Resource already exists")
4          return
5      }
6      res := &sSarefDevice{iri: iri}
7      mod.mSarefDevice[res.IRI()] = res
8      mod.mThing[res.IRI()] = res
9      res.sarefHasTypicalConsumption = make(map[string
           ]Property)
10     // ... (make all property maps)
11     ret = res
12     return
13 }
```

Listing 5: New method for Device.

2.3. Software functionalities

The main functionality of the generated package is the provision of an ontology-compliant data model and serialization as well as deserialization for interoperable data exchange among applications. A graphical description of both processes is depicted in Fig. 2.

For the deserialization process, a ttl or JSON-LD encoded document containing the description of individuals, i.e., a A-box ontology, is the input and parsed into a graph structure. Within this graph, the edges are searched for the keyword `rdf:type`. If the object of the triple, that corresponds to the edge, is one of the classes defined by the T-box ontology, the triple expresses the creation of an individual. All individuals that can be found are created in the model and filled with the properties as specified in the document.

For the serialization process, the steps are executed inversely. First, one graph node for every individual is created. For this purpose, the model's map containing all `Things` is used (see Listing 1 line 2). An edge with name `rdf:type` is added to these nodes. The object is the type of the individual. Additionally, one edge with name `rdf:type` and object `owl:NamedIndividual` is created. After this, one edge for each property of that individual is added to the graph. Finally the ttl / JSON-LD document is created from the resulting graph.

2.4. Limitations of OWL2Go

OWL2Go targets OWL-DL ontologies. OWL-DL is an extension to OWL-Lite and adds further constructs, e.g., enumerated classes or property values. Another extension of OWL-DL is OWL-Full, which allows that a class can also be an individual or property. A detailed description of the different OWL flavors can be found in [27].

Axioms stating equality, inequality or uniquness (equivalentClass, equivalentProperty, sameAs, differentFrom, AllDifferent, disjointWith) are not addressed by OWL2Go. The reason for this is, that such equivalence statements cannot be directly treated in object-oriented languages since different classes or structs are disjoint by design [11]. Hence, this is not a specific limitation of OWL2Go but rather a limitation of the equivalence between OWL and object-oriented programming languages. However, the use of such axioms in an ontology does not cause an error if applied to OWL2Go. Instead these axioms are simply ignored

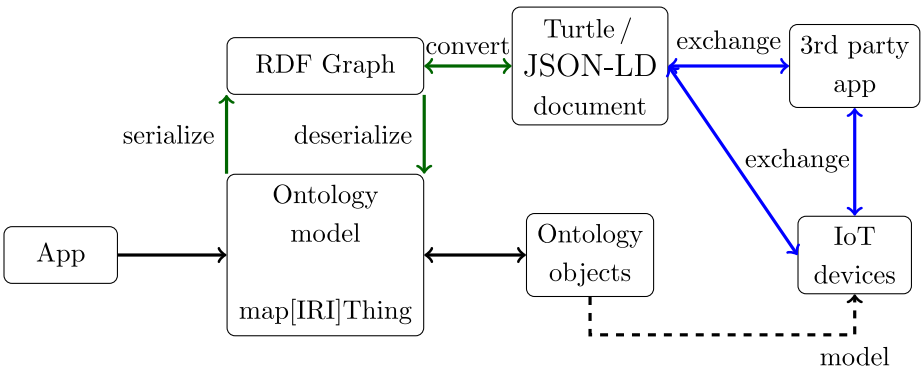

Fig. 2. Serialization and deserialization with generated Go package.

by OWL2Go. For example, in case two classes are related by an `owl:equivalent` axiom, both classes would be translated to Go independently.

Another limitation of the proposed mapping is the interpretation of the open-world assumption, that is typically associated with OWL ontologies. The open-world assumptions says, that assumptions cannot be made based on the absence of knowledge. This is different in data modeling in programming languages. If an entity (instance) is not present in the data model, it is assumed to be not existent in general. Regarding OWL2Go this difference is of importance in case a property is lacking an `owl:range` axiom. While this is in line with the open-world assumption, it is not in line with the fact that for the mapping to Go a specific type is required for the property. In this case OWL2Go would use the `Thing` class as type for this property. Thereby, it is possible to assign an instance of any class, that is defined by the ontology, to this property.

Another case in which the open-world assumption cannot be directly mapped to Go is the lack of the `owl:domain` axiom for a property. OWL2Go will add the respective property only to classes that are related to the property by means of a restriction. If no class contains a restriction regarding this property, the property will be ignored by OWL2Go as it cannot be linked to any defined class.

Due to the given limitations the generated Go model and the OWL ontology used for generation are in general not semantically equivalent. OWL ontologies can be more expressive than their corresponding Go data model. However, the Go data model does not contradict with the ontology. Therefore, it represents a subset of representable knowledge compared to the OWL ontology. It should also be noted that the limitations listed above might not be relevant for a specific ontology. For example, the SAREF ontology does not use axioms that are not handled by OWL2Go and all properties are clearly linked to classes by means of range, domain axioms and/or restrictions. In addition to SAREF we have successfully tested OWL2Go with the NASA Air Traffic Management Ontology (atmonto) [28], which consists of five T-box ontologies.

3. Illustrative examples

The following example using the SAREF ontology comprises all relevant features of the mapping and code generation process explained above. The example demonstrates the entire process of using a OWL2Go-generated Go package. The first step is the deserialization from a ttl document, containing a A-box ontology compliant with SAREF. This document could be retrieved from another application. Afterwards, the instances created by the deserialization process, are manipulated using the API generated by OWL2Go. In a last step the instances are serialized to a JSON-LD document.

In the first step, a ttl-encoded SAREF document is deserialized.

```
@prefix saref:  <https://w3id.org/saref#> .
<http://example.com#dev>    a   saref:Appliance ;
    saref:accomplishes  <http://example.com#task> ;
    saref:hasTypicalConsumption <http://example.com#
        pow> ;
    saref:measuresProperty  <http://example.com#pow>
        .
<http://example.com#pow>    a   saref:Power .
<http://example.com#task>   a   saref:Task ;
    saref:isAccomplishedBy  <http://example.com#dev>
        .
```

Listing 6: Input ttl file

Listing 6 shows the input ttl document with the definition of a `saref:Appliance`, a subclass of `saref:Device`, and a `saref:Power` which is a subclass of `saref:Property`. The device and the property are related by `saref:measuresProperty` and `saref:hasTypicalConsumption`. These relations are defined by the `saref:Device` class. Hence, the example shows that inherited properties are properly used by derived classes. Also the correct usage of `saref:hasTypicalConsumption`, which can be of two distinct types (i. e. `saref:Power` and `saref:Energy`), is demonstrated. Additionally, the example also contains a `saref:Task` object, `task`. This is related to `dev` by both, the `saref:accomplishes` and the inverse `saref:isAccomplishedBy` property. The corresponding model state after deserialization is presented in Fig. 3.

The model is extended by adding a `saref:Measurement`. Relations among the objects are added in conformance to the specification. The measurement relates to the power property, has a value and a unit. Moreover, the `task` object is deleted from the model to demonstrate proper deletion of both relations with `dev`.

The code for the given procedure is provided in Listing 7. Fig. 4 highlights the new model state.

```
package main

import (
    "git.rwth-aachen.de/acs/public/ontology/owl/
        saref/pkg/ontology"
    "os"
)

func main() {
    var mod *ontology.Model
    var file *os.File
    // deserialization
    file, _ = os.Open("input.ttl")
    mod, _ = ontology.NewModelFromTTL(file)
    file.Close()
    // manipulation
    meas, _ := mod.NewSarefMeasurement("http://
        example.com#meas")
    meas.SetSarefHasValue(3.14)
    dev := mod.SarefDevice("http://example.com#dev")
```

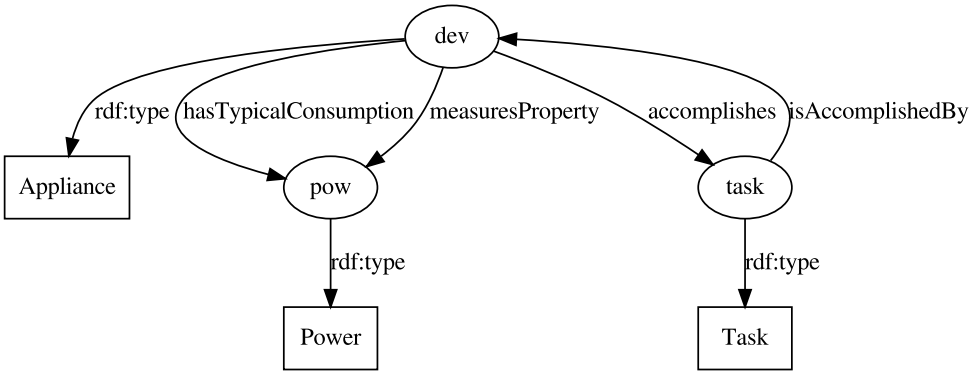

Fig. 3. State of model after deserialization.

```
19      dev[0].AddSarefMakesMeasurement(meas)
20      pow := mod.SarefPower("http://example.com#pow")
21      meas.AddSarefRelatesToProperty(pow[0])
22      pow[0].AddSarefRelatesToMeasurement(meas)
23      unit, _ := mod.NewSarefPowerUnit("http://example
            .com#kilowatt")
24      meas.AddSarefIsMeasuredIn(unit)
25      task := mod.SarefTask("http://example.com#task")
26      mod.DeleteObject(task[0])
27      // serialization
28      file, _ = os.Open("output.jsonld")
29      mod.ToJSONLD(file)
30      file.Close()
31  return
32  }
```

Listing 7: Code for example application.

The last step is serialization of the new model. Listing 8 provides the resulting JSON-LD document.

```
1  [{
2       "@id":"http://example.com#dev",
3       "@type": ["https://w3id.org/saref#Appliance",
4       "http://www.w3.org/2002/07/owl#NamedIndividual"]
            ,
5       "https://w3id.org/saref#hasTypicalConsumption":
6           [{"@id":"http://example.com#pow"}],
7       "https://w3id.org/saref#makesMeasurement":
8           [{"@id":"http://example.com#meas"}],
9       "https://w3id.org/saref#measuresProperty":
10          [{"@id":"http://example.com#pow"}]
11 },{
12      "@id":"http://example.com#kilowatt",
13      "@type": ["https://w3id.org/saref#PowerUnit",
14      "http://www.w3.org/2002/07/owl#NamedIndividual"]
15 },{
16      "@id":"http://example.com#meas",
17      "@type": ["https://w3id.org/saref#Measurement",
18      "http://www.w3.org/2002/07/owl#NamedIndividual"]
            ,
19      "https://w3id.org/saref#hasValue":
20          [{"@type":"http://www.w3.org/2001/XMLSchema#
                double","@value":"3.14"}],
21      "https://w3id.org/saref#isMeasuredIn":
22          [{"@id":"http://example.com#kilowatt"}],
23      "https://w3id.org/saref#relatesToProperty":
24          [{"@id":"http://example.com#pow"}]
25 },{
26      "@id":"http://example.com#pow",
27      "@type": ["https://w3id.org/saref#Power",
28      "http://www.w3.org/2002/07/owl#NamedIndividual"]
            ,
29      "https://w3id.org/saref#relatesToMeasurement":
30          [{"@id":"http://example.com#meas"}]
31 }]
```

Listing 8: Output JSON-LD file.

3.1. Performance evaluation

A performance evaluation is necessary in order to ensure that the use of OWL2Go does not impose a bottleneck. This is done in two steps. First the performance of OWL2Go itself is evaluated. While OWL2Go has to be executed only once to generate the package for a given ontology, its execution time should remain within reasonable boundaries. In a second step the performance of the generated package is evaluated. We use the package for the SAREF ontology again. This package is used during runtime of the ontology application. Hence, execution times for common tasks, such as serialization and deserialization, are of importance. The following assessment identifies the main factors which influence execution times in both steps.

For the performance evaluation of OWL2Go four different input T-box ontologies are created. These ontologies define an increasing number of classes. Within each ontology every class is related to one other class using a property. OWL2Go is executed for all four ontologies. The resulting execution times are depicted in Fig. 5. OWL2Go maps each OWL class to a Go class and generates a corresponding interface and struct as well as methods defined for the interface, as described in Section 2. Hence, the main influence for the complexity of the problem is the number of classes. Since the same procedure has to be executed for each class, a linear increase of the execution time results.

For the performance evaluation of the generated SAREF package four different input A-box ontologies containing a varying number of individuals are used. All individuals are of type `SAREF :Device` and relate to one other device by means of the `SAREF :consistsOf` property. A simple Go program is implemented which deserializes the input A-box ontology and subsequently serializes it back to a ttl document. The execution times for all four ontologies is shown in Fig. 5. In the deserialization process an instance of a Go struct is created and the properties, i.e, the relations to other individuals, are added. For the serialization process each Go instance is traversed and the triples corresponding to the individual creation and its relation to other individuals are created. As a result both processes depend linearly on the number of individuals.

The underlying problem for both, OWL2Go itself and the use of the resulting Go package for deserialization and serialization, is of linear complexity and also the measured execution times increase accordingly. Therefore, both steps of the performance evaluation indicate that our approach does not add any unnecessary overhead.

4. Impact

The general mapping from OWL to a modern programming language in this work can inspire research activities in the Informatics area. This can affect mappings of ontologies to programming languages as already done for OWL in case of Java [8]

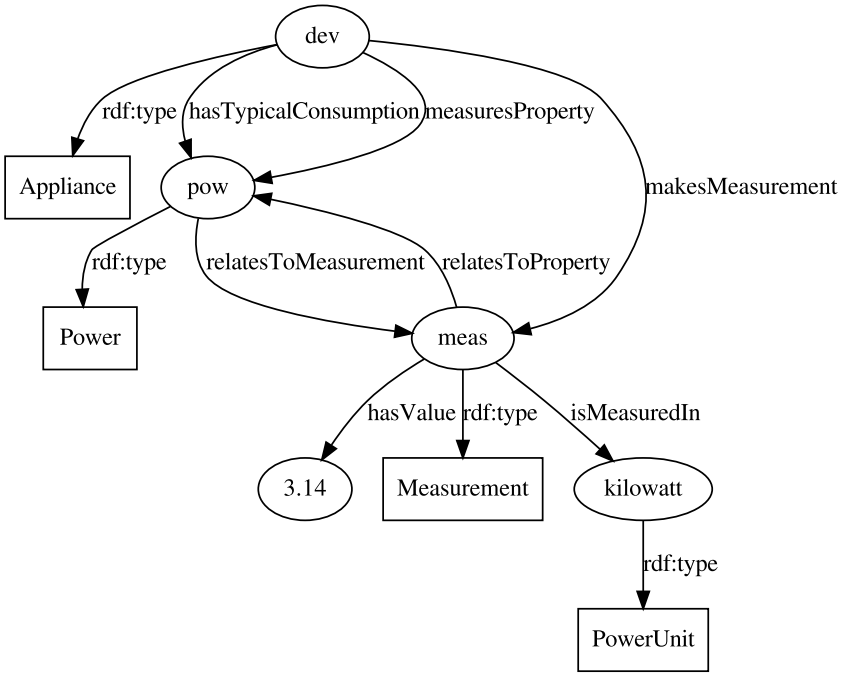

Fig. 4. State of model after manipulation.

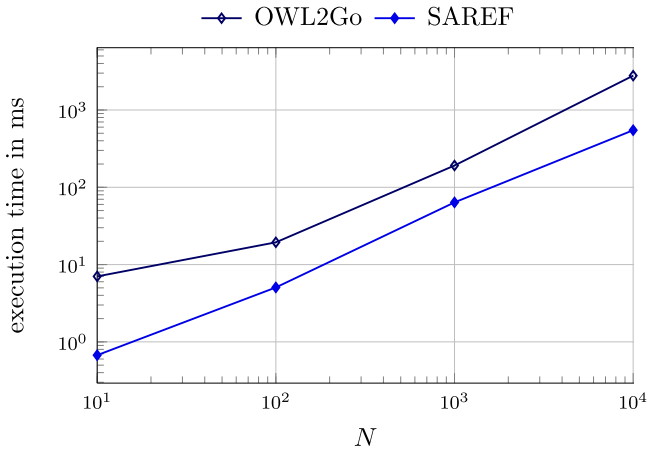

Fig. 5. Performance of OWL2Go (N: number of classes) and the SAREF package (N: number of individuals).

but also for other ontologies [29]. Also general questions could be investigated. For example, how ontology languages can be integrated in programming languages avoiding the need of generators producing source code for handling machine-representable ontologies. Furthermore, the generated code for common ontologies, e. g., SAREF, as open-source software is expected to help other researchers to write interoperable applications. One possible use case from the authors' background is energy management systems in Smart Grids using IoT. However, also many other IoT use cases exist [30]. Consequently, research progress between multiple entities on a common data basis can be accelerated as results of different tools being comparable.

The auto-generated code for SAREF and the imported Time ontology comprises more than twenty five thousand lines of code. A manual implementation of that size would be error prone and time consuming. Moreover, the code generation process can simply be repeated in case the specification changes. No additional manual rework would be necessary.

The code generation for SAREF is used in the German research project ENSURE [31]. There a cloud-based platform for the management of flexibility in electrical distribution grids is developed. The concept of this platform is based on a multi-agent system approach [32]. SAREF is used to describe the flexible devices and their capabilities. Multi-agent systems especially rely on communication. For information exchange among agents a shared ontology is crucial. The package generated by OWL2Go enables the agent software developer to use SAREF without the need to understand OWL. Instead the developer manipulates objects of a data model and exchanges its current state by means of automatic serialization / deserialization to / from a standardized serialization format, as shown in the example before.

Conclusions

We present a methodology for the mapping of OWL to the programming language Go. Based on this methodology we implemented OWL2Go which auto-generates a Go data model compliant with a given ontology. That data model can be modified by simple Go instructions without OWL knowledge. Additionally, Go objects can be serialized to ttl / JSON-LD and deserialized from ttl / JSON-LD to objects.

OWL2Go eases the use of ontologies by software developers and researchers. Thereby, it contributes to increase the utilization of ontologies for knowledge expression.

In future work, other serialization formats than ttl and JSON-LD, such as RDF/XML, could be supported by OWL2Go.

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Acknowledgments

This work was supported by the German Federal Ministry of Education and Research in the project ENSURE (Grant No. 03SFK1C0) and by the German Federal Ministry of Economic Affairs and Energy in the project AGENT (Grant No. 03ET1495A).

References

[1] Horrocks I, Patel-Schneider PF, van Harmelen F. From SHIQ and RDF to OWL: the making of a web ontology language. J Web Semant 2003;1(1):7–26.

[2] W3C OWL Working Group. OWL 2 web ontology language document overview (second edition). 2012, https://www.w3.org/TR/owl2-overview/.

[3] Kaed CE, Ponnouradjane A. A model driven approach accelerating ontology-based IoT applications Development. In: Semantic interoperability and standardization (SIS-IoT) workshop part of the 13th semantics conference; 2017.

[4] Daniele L, Solanki M, den Hartog F, Roes J. Interoperability for smart appliances in the IoT world. In: The semantic web – ISWC 2016. Lecture notes in computer science, 2016, p. 21–9.

[5] Allemang D, Hendler J. Semantic web for the working ontologist: Effective modeling in RDFS and OWL. United States: Elsevier; 2011.

[6] Brickley D, Guha R. RDF Schema 1.1. 2014, https://www.w3.org/TR/2014/REC-rdf-schema-20140225.

[7] Goldman NM. Ontology-oriented programming: Static typing for the inconsistent programmer. In: The semantic web - ISWC 2003. Lecture notes in computer science, Springer; 2003, p. 850–65. http://dx.doi.org/10.1007/978-3-540-39718-2_54.

[8] Kalyanpur A, Pastor DJ, Battle S, Padget J. Automatic mapping of OWL ontologies into Java. In: Proceedings of the sixteenth international conference on software engineering & knowledge engineering; 2004. p. 98–103.

[9] Völkel M, Sure Y. RDFReactor - from ontologies to programmatic data access. In: international semantic web conference (ISWC); 2005.

[10] Chevalier F. Autordf - using OWL as an object graph mapping (OGM) specification language. In: The semantic web. 2016, p. 151–5.

[11] Baset S, Stoffel K. Object-oriented modeling with ontologies around: A survey of existing approaches. Int J Softw Eng Knowl Eng 2018;28:1775–94.

[12] Donovan AA, Kernighan BW. The Go programming language. United States: Addison-Wesley Professional; 2015.

[13] Gyrard A, Serrano M, Patel P. Building interoperable and cross-domain semantic web of things applications. In: Managing the web of things. Boston: Morgan Kaufmann; 2017, p. 305–24. http://dx.doi.org/10.1016/B978-0-12-809764-9.00014-7.

[14] Daniele L, den Hartog F, Roes J. Created in close interaction with the industry: The smart appliances reference (SAREF) ontology. In: Formal ontologies meet industry: 7th international workshop, vol. 225; 2015.

[15] SAREF Ontology documentation. 2016, http://ontology.tno.nl/saref/.

[16] ETSITS 103 264 v211. SmartM2M; smart appliances; reference ontology and oneM2M mapping. 2017, https://www.etsi.org/deliver/etsi_ts/103200_103299/103264/02.01.01_60/ts_103264v020101p.pdf.

[17] ETSITS 103 410-1. SmartM2M; smart appliances extension to SAREF; part 1: energy domain. 2017, https://www.etsi.org/deliver/etsi_ts/103400_103499/10341001/01.01.01_60/ts_10341001v010101p.pdf.

[18] ETSITS 103 410-2. SmartM2M; smart appliances extension to SAREF; part 2: environment domain. 2017, https://www.etsi.org/deliver/etsi_ts/103400_103499/10341002/01.01.01_60/ts_10341002v010101p.pdf.

[19] ETSITS 103 410-3. SmartM2M; smart appliances extension to SAREF; part 3: building domain. 2017, https://www.etsi.org/deliver/etsi_ts/103400_103499/10341003/01.01.01_60/ts_10341003v010101p.pdf.

[20] Moreira J, van Sinderen M, Pawłowski W, Daniele L, Wasielewska K, Ganzha M, et al. Towards IoT platforms' integration: Semantic translations between W3C SSN and ETSI SAREF. In: Workshop semantic interoperability and standardization in the IoT; 2017.

[21] Esnaola-Gonzalez I, Díez FJ, Berbakov L, Tomasevic N, Štorek P, Cruz M, et al. Semantic interoperability for demand-response programs: RESPOND project's use case. In: 2018 global internet of things summit (GIoTS); 2018. p. 1–6.

[22] Clark KL, McCabe FG. Go! for multi-threaded deliberative agents. In: Declarative Agent Languages and Technologies. Lecture Notes in Computer Science, Berlin, Heidelberg: Springer; 2004, p. 54–75. http://dx.doi.org/10.1007/978-3-540-25932-9_4.

[23] The Go programming language. https://golang.org/.

[24] The Kubernetes Authors. What is Kubernetes? – Kubernetes. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/.

[25] Boettiger C. An introduction to docker for reproducible research. SIGOPS Oper Syst Rev 2015;49(1):71–9, http://doi.acm.org/10.1145/2723872.2723882.

[26] Hobbs JR, Pan F. An ontology of time for the semantic web. ACM Trans Asian Lang Inf Process 2004;3(1):66–85.

[27] W3C OWL Working Group. OWL Web ontology language overview. 2004, https://www.w3.org/TR/owl-features/.

[28] Keller R. The NASA air traffic management ontology (atmonto). 2018, https://data.nasa.gov/ontologies/atmonto/index.html.

[29] Razik L, Mirz M, Knibbe D, Lankes S, Monti A. Automated deserializer generation from CIM ontologies: CIM++—an easy-to-use and automated adaptable open-source library for object deserialization in C++ from documents based on user-specified UML models following the common information model (CIM) standards for the energy sector. Comput Sci - Res Dev 2018;33(1):93–103.

[30] Vermesan O, Friess P, et al. Internet of things-from research and innovation to market deployment, vol. 29. River publishers Aalborg; 2014.

[31] ENSURE Geschäftsstelle Kopernikus-Projekt. Kopernikus Projekte: New network structures. https://www.kopernikus-projekte.de/en/projects/new-network-structures.

[32] Dähling S, Kolen S, Monti A. Swarm-based automation of electrical power distribution and transmission system support. IET Cyber-Physical Syst Theory Appl 2018;3(4):212–23. http://dx.doi.org/10.1049/iet-cps.2018.5001.