

A semantic-based approach for domain specific language development

Eman Negm, Akram Salah, Soha Makady

Department of Computer Science, Faculty of Computers and Artificial Intelligence, Cairo University, Giza, Egypt

Article Info

Article history:

Received Jan 25, 2024

Revised May 30, 2024

Accepted Jun 4, 2024

Keywords:

Domain specific languages

Ontology

Projectional-editing

Reasoning

Software Engineering

ABSTRACT

A domain specific language (DSL) ties the business and technical models, by letting technical developers write programs with the business domain properties. Yet, DSLs are not used due to the cost of developing them. Such cost stems from the needed expertise within both the domain knowledge and language development technicalities for any DSL engineer who would design such a language. This paper proposes a semantic-based DSL development approach that utilizes an ontology as a formal way for domain representation. The domain ontology is semi-automatically transformed into a DSL. Then, an ontology reasoning algorithm provides reasoning services on the DSL structure and the programs developed using such DSL by application developers. Such reasoning services can automatically detect flaws in the DSL design like possible inconsistency or the presence of unsatisfiable or redundant classes thus serving the DSL engineer. The reasoning services can also discover inconsistency or redundant classes in programs built using the designed DSL, thus serving the application developer. The proposed approach was implemented within a language workbench using projectional-editing and was evaluated on two different ontologies from varied domains. The results show correct transformation of the input ontology, valid instantiation of designed application, and efficient reasoning services.

This is an open access article under the [CC BY-SA](https://creativecommons.org/licenses/by-sa/4.0/) license.



Corresponding Author:

Soha Makady

Department of Computer Science, Faculty of Computers and Artificial Intelligence, Cairo University

5 Dr Ahmad Zewail St., Dokki, Giza, Egypt

Email: s.makady@fci-cu.edu.eg

1. INTRODUCTION

Domain specific languages (DSLs) [1] are languages that are designed to model a specific class of problems. Such a class represents the domain of the language. DSLs have many advantages over general purpose languages (GPLs) for representing a specific domain. On one hand, DSLs provide higher abstractions for the given domain which raises the productivity and the quality of the development process, by making the developer focus on the domain-specific modeling, rather than the low-level programming details [2]–[4]. DSL also provides better validation and verification for the output programs since it utilizes domain specific constraints. Such constraints verify that the generated program is meaningful within its corresponding domain. For example, an internet of things (IoT) DSL does not allow configuring the same sensor in two different locations at the same time. The error messages are also more meaningful since the messages utilize the domain concepts. On the other hand, the use of domain concepts within DSLs allows more involvement for the domain expert in the development process since the program itself becomes readable for the domain expert [5], [6] thus enhancing the final products' quality [7], [8].

Problem statement: despite the advantages mentioned above, DSLs suffer from a problem due to the cost needed to develop a new DSL [9]. Developing a DSL is an extremely difficult task that requires a mix between domain knowledge and language development expertise [9]. Language workbenches [5] are comprehensive environments that aim to reduce the cost of developing a DSL by providing high level tools for building different DSL aspects [10]. However, the language engineer who would be using the language workbench to build a new DSL, may lack the comprehensive domain knowledge to properly cover the different aspects of such DSL.

A crucial phase of building a new DSL is the domain analysis phase which determines the concepts, relations, and constraints inside the concerned domain [9]. Such a phase is essential in the DSL development since a wrong analysis will lead to a DSL that incorrectly models the domain. The output of such a phase is the representation of the given domain. Most language workbenches do not use standard and formal ways for the domain representation. Furthermore, most of the DSLs depend on one general scenario instead of multiple use-cases for analyzing the domain [11]. Such scenario is transformed manually into a DSL design by the language engineer. The informal domain representation leads to the absence of any formal validation for the generated DSL. Such formal validation could have been utilized to detect problems in the DSL design like inconsistency. Alternatively, an ontology provides a formal conceptualization for a given domain that determines the concepts, attributes, relations, and constraints of this domain [12]. A lot of effort has been done to develop valid ontologies for different domains [13]–[15]. Accordingly, building a DSL from such ontologies will generate a valid DSL and reduce the DSL development time. In addition, the DSL could utilize existing ontology reasoning algorithms to provide semantic reasoning services within DSL editors.

To evaluate the feasibility of building DSLs from ontologies, we previously proposed an approach to automatically generate *OntIoT* DSL [16], which models only parts of the IoT domain. The structure of *OntIoT* was generated from the semantic sensor network (SSN) ontology [17]. However, the proposed approach had several limitations as follows. First, the generation of aDSL from an ontology was only limited to mapping a subset of the ontology (the structure information) but did not cover the constraints nor the instance information. Second, the approach did not support the reasoning services. Third, the applicability of the approach to different domains, other than the IoT domain, was not examined. Fourth, there was no quantitative evaluation for the correctness of transforming input ontologies to their corresponding DSL, nor for the correctness of generating the domain-specific programs.

Proposed solution: in this paper, we extend our semantic DSL generation approach to support various domains in a generic way and provide support for reasoning services on domain-specific applications that utilize the generated DSL. The paper makes the following contributions: i) utilizes an ontology as a formal representation for the domain, ii) provides a semi-automated generation for the DSL from a domain ontology that covers structure, constraints, and instance information present within a formal ontology, iii) provides reasoning services for both the language engineer and the developer, and iv) evaluates the different phases of the proposed approach, as well as its applicability to different domains in a quantitative manner.

Results: we evaluated the different phases of our proposed solution on two applications from different domains, to ensure that applicability of the proposed approach to various domains. The evaluation proved: i) the proper transformation of an ontology to a DSL with proper object and data properties; ii) the correctness of generating an instance ontology for a DSL program; and iii) the ability of the reasoning service to detect problems in the developed DSL programs.

The rest of the paper is organized as follows: section 2 describes the different phases of the proposed approach. Section 3 presents the evaluation of the approach on two different domains. Section 4 reviews the current research for utilizing an ontology in developing DSLs and illustrates the differences between the current research and the proposed approach. Finally, section 5. concludes the work and discusses our future work.

2. METHOD

We propose a semantic-based DSL development approach that is not restricted to a specific domain or ontology. The proposed approach is based on using ontology as a domain model in the domain analysis phase. The ontology generated from the domain analysis phase is used to make a semi-automatic transformation from the domain analysis phase into the design and implementation phases. The transformation is not fully automated since some DSL aspects, like the editor aspect, cannot be auto-generated from the input ontology.

Figure 1 depicts the different phases of our approach. The input to the approach is the ontology that has been developed in the domain analysis phase. The language engineer can utilize one of the already published ontologies or build a new one from scratch according to his requirements. The approach mainly consists of three phases, which are briefly explained here, then detailed within the following subsections.

a. Transformation phase: which is responsible for transforming the input ontology into a DSL abstract representation. Two manual actions are done by the language engineer and the developer after the

- transformation phase: i) The language engineer creates the final DSL editor based on the generated representation through a projectional language workbench; and ii) The developer to build a DSL application using the created DSL editor.
- b. Instance phase: which is responsible for traversing the built DSL application and translating it into web ontology language (OWL) constructs to generate an instance ontology that represents the given DSL application.
 - c. Reasoning phase: the final phase which provides the reasoning services.

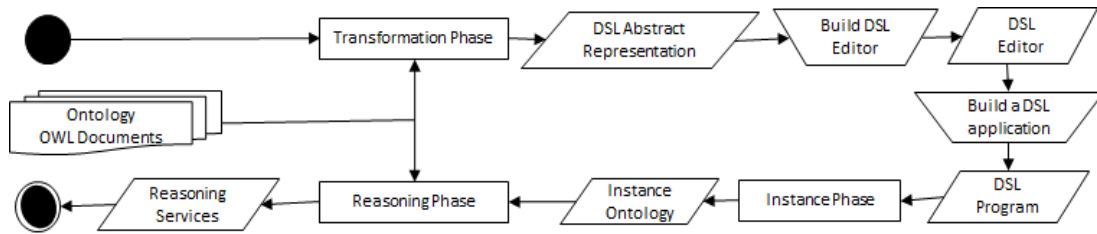


Figure 1. The process of applying our proposed approach

2.1. Transformation phase

This phase uses the domain ontology as an input, to generate the DSL abstract representation which is the core representation in the projectional editing approach. The abstract representation is a tree representation that encapsulates the semantic entities and relations in the DSL domain. The nodes of the tree represent the concepts of the domain and the references among nodes represent the relations. Figure 2 depicts the internal structure for the transformation phase. The core component for this phase is the transformation engine that utilizes the ontology handler component to extract the concepts, attributes, and relations from the input ontology. Then, the transformation engine uses a set of mapping rules to generate the different components of the DSL abstract representation. In the following subsections, we describe the components of the DSL abstract representation and how the transformation engine generates these components from the given ontology.

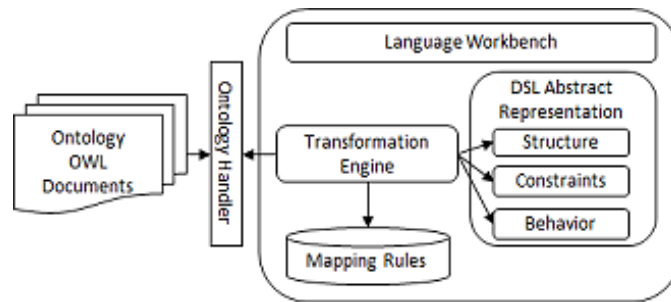


Figure 2. Transformation phase architecture

2.1.1. DSL structure

The structure component of the DSL abstract representation includes the structure of the tree. It contains the nodes of the tree and the references among these nodes. The transformation engine translates the TBox statements in the domain ontology into a tree structure. It utilizes the mapping rules defined in Table 1 for applying the above transformation. The root node of the tree is mapped to the ontology itself. Each class in the OWL ontology is mapped to a node in the tree representation. The class hierarchy in OWL (i.e., *rdfs:subClassOf* and *rdfs:superClassOf*) is translated into extension relations between the nodes of the tree. A property in OWL is a directional binary relation between source entities called domain and target entities called range. OWL defines three types of properties: object property, datatype property, and annotation property. The object properties are translated into references among the nodes of the tree. The domain of the property is the source node and the range is the target node. On the other hand, the datatype properties are

added as attributes for the tree nodes. The data type of the generated node's attribute is the same as the data type of the property. Finally, the annotation property is added also as a node attribute with string data type.

A property restriction in OWL represents a constraint on a specific property. There are two types of property restrictions: value constraints and cardinality constraints. A value constraint adds a constraint on the range of the property while a cardinality constraint adds a constraint on the number of values for a given property. *owl:allValuesFrom(P C)* is a value constraint that restricts the range of the property *P* to a class *C*. This restriction is used to determine the target node for the reference of the property *P*. The second type of OWL property restrictions is the cardinality constraints that are used to construct the constraints for the DSL abstract representation as described in subsection 2.1.3.

- a. Structure mapping limitations: some OWL constructs are not currently covered in the above mapping since the abstract representation could not support it up to now:
- Multiple inheritance: OWL supports multiple parents for the same class which is not supported in the DSL abstract representation as the tree node could extend only one parent.
 - Multiple property ranges for the same domain: OWL property is mapped to a node reference from the domain node to the range node. Accordingly, the same reference cannot contain multiple targets in the DSL abstract representation.
 - Restriction *owl:someValuesFrom(P C)*: OWL property restriction *owl:someValuesFrom(P C)* is a value constraint. It means that the range of the property *P* should contain some values from class *C*. The DSL abstract representation contains one specific target node for each node reference. Accordingly, the mapping does not support *owl:someValuesFrom(P C)* restriction up till now.

Table 1. Structure mapping rules

OWL 2 Construct	Abstract representation
<i>owl:Ontology</i>	Root node
<i>owl:Class</i>	Node
<i>rdfs:subClassOf</i>	Node extension
<i>rdfs:superClassOf</i>	Node extension
<i>owl:ObjectProperty</i>	Node reference
<i>owl:DatatypeProperty</i>	Node attribute
<i>owl:AnnotationProperty</i>	Node attribute
<i>owl:Restriction (owl:allValuesFrom)</i>	Node reference target

2.1.2. DSL behavior

The behavior component of the DSL abstract representation includes the methods attached to each node in the tree that represents the behavior of this node. We use this component to set the value of the annotation properties (i.e. *owl:annotationAssertion*). As mentioned in the previous section, the annotation properties are added as attributes for the specified node. A constructor is generated for each node to set the value of the annotation property as a default value for the specified attribute.

2.1.3. DSL constraints

The constraints component of the DSL abstract representation includes constraints on the relations among nodes. The constraints of each node are validated every time a new node is created or updated. The transformation engine translates the cardinality constraints in OWL ontology into tree constraints. OWL defines three types of cardinality constraints. For a class *C* the following constraints could be defined:

- *Owl:ObjectMinCardinality(C P n)*: means class *C* should contain at least *n* values for property *P*.
- *Owl:ObjectExactCardinality(C P n)*: means class *C* should contain exactly *n* values for property *P*.
- *Owl:ObjectMaxCardinality(C P n)*: means class *C* should contain maximum *n* values for property *P*.

The above restrictions are mapped to constraints for the source node *C* on the number of references of the property *P*. Table 2 shows the mapping among the above OWL property restrictions and the abstract representation constraints. $\# \{C.P\}$ represents the number of references for property *P* from node *C*. Every time a reference is added or removed for class *C*, the constraints are validated. The editor displays an error message to the developer in case one of the constraints failed.

Table 2. Constraints mapping rules

OWL 2 construct	Abstract representation constraint
<i>Owl:ObjectMinCardinality(C P n)</i>	$\# \{C.P\} \geq n$
<i>Owl:ObjectExactCardinality(C P n)</i>	$\# \{C.P\} = n$
<i>Owl:ObjectMaxCardinality(C P n)</i>	$\# \{C.P\} \leq n$

2.2. Instance phase

In the previous phase, the DSL abstract representation was generated from the input ontology. The language engineer utilizes this representation to create a DSL editor. This editor is used by a developer to write a concrete DSL program. The DSL program represents an instance from the language meta-model. In the instance phase, the program, written by the developer, is mapped to an instance ontology. The instance ontology is an extension of the input ontology with ABox statements. The input of the instance phase is the DSL program and the output is the instance ontology. Figure 3 depicts the internal structure for the instance phase. Since we depend on the projectional editing approach, the DSL program itself is represented by an abstract tree representation. The first step in this phase is traversing this tree representation. Our approach utilizes Breadth-First traversing algorithm. The second step is extracting the instances, properties values, and relations among instances. The final step is using the instance mapping rules to create the instance ontology through the ontology handler.

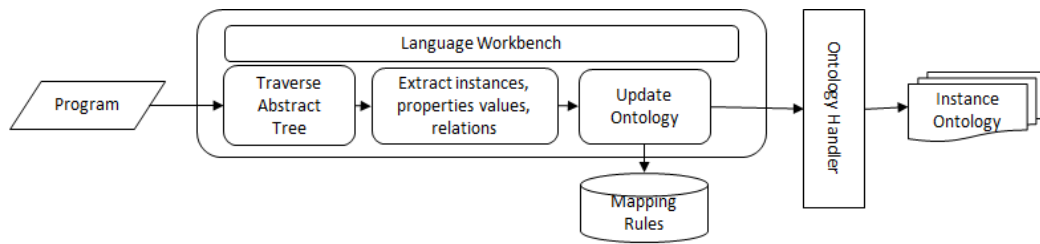


Figure 3. Instance phase architecture

Table 3 shows the instance mapping rules that are utilized to map a DSL program abstract representation into an instance ontology. Unlike structure mapping rules that translate OWL TBox constructs into DSL abstract representation elements, instance mapping rules translate program abstract representation elements into OWL ABox constructs. Each instance in the program is mapped into an individual in the instance ontology. This is done by adding *owl:classAssertion*. References among instances are translated into *owl:-objectPropertyAssertion*. OWL could define the inverse property for a given property. Our approach adds *owl:-objectPropertyAssertion* for the property and its inverse property if exist. The values of the attributes are added as values for the dataType properties using *owl:dataProperty-Assertion*.

Table 3. Instance mapping rules

Abstract representation	OWL 2 construct
Class instances	<i>owl:classAssertion</i>
Instance's references	<i>owl:objectPropertyAssertion</i>
Attribute values	<i>owl:dataPropertyAssertion</i>

2.3. Reasoning phase

The final phase is the reasoning phase. The goal of this phase is to use the instance ontology and the original input ontology to provide reasoning services for the language engineer and the developer. Ontology reasoning solves problems like checking satisfiability, check consistency, and find implicit facts that are not explicitly defined in the ontology in the TBox or ABox statements. Figure 4 shows the internal structure of the reasoning phase. The core component is the reasoning engine which provides three reasoning services through the ontology handler: i) consistency checking, ii) satisfiability checking, and iii) equality checking. While the language engineer benefits from the three services, the developer utilizes the consistency checking and equality checking services only. The reasoning engine depends on the standard pellet reasoner [18] for applying the reasoning services. We describe those three services in detail.

2.3.1. Satisfiability checking

For the ontology domain, satisfiability checking checks if the ontology contains unsatisfiable classes. Unsatisfiable classes are classes that could not have any individuals (i.e., instances) due to some constraints in the ontology definition. An ontology that contains at least one unsatisfiable class is called incoherent ontology, but it remains a consistent ontology since it has no contradictions. Any model of an incoherent ontology that contains an instance of an unsatisfiable class is an inconsistent model.

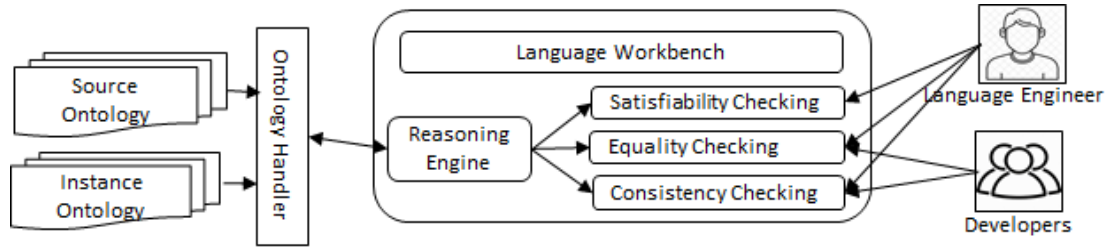


Figure. 4. Reasoning phase architecture

For the DSL domain, satisfiability checking provides the language engineer with a list of unsatisfiable concepts that should not be instantiated by the developer due to some constraints defined in the ontology. These classes should be removed from the DSL structure or updated by the language engineer since if the developer could create instances of them, then the program becomes inconsistent. On the other hand, if the editor prevents the developer from instantiating those classes, such classes will become useless and need to be removed. Since our approach maps the DSL structure into an ontology, the approach utilizes ontology reasoning to check if the DSL structure contains unsatisfiable concepts. In addition, the approach provides language engineers with a list of unsatisfiable concepts to fix them.

2.3.2. Consistency checking

An ontology is consistent if it has no contradiction. Consistency checking is a precondition for any other reasoning service. If the ontology is inconsistent, then no reasoning algorithm can run on this ontology. Our approach checks consistency on the DSL level and the program level. The language engineer can check the consistency of the DSL structure and the developer can check the consistency of the DSL program. This is done by checking the consistency of the mapped ontology. The source ontology is used to validate the consistency of the DSL structure and the instance ontology is used to validate the consistency of the DSL program. One reason for inconsistency is the initialization of an instance from an unsatisfiable concept.

2.3.3. Equality checking

Equality checking extracts the equal classes and individuals from a given ontology. The equality check is done on the DSL level and the program level. The language engineer can retrieve a list of equivalent classes. These classes may be redundant classes in the DSL structure that should be removed or may indicate missing fields or methods that should be added to differentiate between those equivalent classes. On the program level, the developer can retrieve a list of equivalent objects which should be removed or updated by the developer.

2.4. Implementation

The approach is implemented as a new component for meta programming system (MPS) workbench [19]. The component is integrated with MPS as a new plugin. MPS is an open-source DSL workbench. It follows the projectional editing approach for the DSL development by providing a set of meta-languages that allow a language engineer to define the different aspects of a DSL. New actions are added to the MPS workbench as follows: i) loading language structure from ontology, ii) creating instance ontology, and iii) running reasoning services. Ontology handler utilizes OWL API for OWL operations. It is a JavaAPI for handling OWL 2 ontologies. The reasoning engine depends on the standard Pellet reasoner [18].

3. EVALUATION AND RESULTS

This section discusses two case studies that demonstrate how our approach could be used to build a DSL for the IoT domain and the e-commerce domain. The case studies evaluate the three phases of the approach. The first case study see section 3.1 involves an IoT application, whereas the second case study see section 3.2 utilizes an e-commerce application. Such variance is intentional to illustrate that our proposed approach can generalize to different domains. Each study explains the evaluation steps, and the corresponding results based on some collected metrics.

3.1. Case study 1: *OntIoT*

In this case study, we show how the proposed approach can be utilized to build *OntIoT* DSL. The *OntIoT* DSL is a DSL that models IoT domain based on the semantic sensor network (SSN) ontology [17].

The case study evaluates the results generated from the transformation and instance phases. In addition, an evolution scenario for *OntIoT* is proposed to explain the benefits of the proposed reasoning services for the language engineer and the developer. SSN ontology is an ontology to model the domain of sensors, actuators, and samplers with their observations and actuations. The current version of SSN ontology is developed by world wide web consortium (W3C) and open geospatial consortium (OGC).

3.1.1. Transformation phase evaluation

In this phase, the language engineer will auto-generate the abstract representation of the *OntIoT* DSL from the SSN ontology. The language engineer is not required to model each IoT concept manually. Additionally, it reduces the error probability that *OntIoT* has missing or wrong domain relations or concepts. Finally, *OntIoT* extends the IoT knowledge that is encapsulated in the SSN ontology. Table 4 shows the mapping between *OntIoT* structure and SSN ontology. All SSN ontology classes are successfully mapped into *OntIoT* concepts plus three utility concepts: first, the *MainScript* which is an empty concept that should be updated by the language engineer to determine which concepts should be included in the editor main script. Second, the *OntConcept* which is the parent concept for all the generated ontology concepts. *OntConcept* contains the internationalized resource identifier (IRI) of the concept in the ontology. Third, the *DateTime* concept which represents the *DateTime* RDF datatype as it not a default datatype in MPS. All restrictions are covered in *OntIoT* (i.e., *ObjectAllValuesFrom*, *DataExactCardinality*, *ObjectExactCardinality*, and *ObjectMinCardinality*).

Table 4. *OntIoT* structure evaluation

	SSN ontology	<i>OntIoT</i> structure
Classes	22	25
Restriction: object all values from	48	48
Restriction: data exact cardinality	3	3
Restriction: object exact cardinality	9	9
Restriction: object min cardinality	13	13
Object properties	35	35
Data properties	3	3
Annotation properties	75	75
Super class relation	7	6

All object properties and data properties that are defined by a domain and a range are mapped successfully in the *OntIoT* structure. All annotation properties are mapped to concept attributes in the *OntIoT* structure. The default values for these attributes are mapped successfully to the values defined in SSN ontology. *OntIoT* is missing one superclass relation since it is a multiple inheritance relation which is not supported in our approach up to now.

- a) Creating *OntIoT* editor: an editor was created manually to design the projection process of the above structure. *OntIoT* editor is a textual editor created by MPS workbench using the MPS editor meta-language.
- b) Writing IoT script: SSN ontology provides a set of scenarios for using the ontology terms [20]. For each scenario, it provides an OWL file that represents the given scenario. We utilized scenario (B.3 Apartment 134 [20]) for evaluating *OntIoT* instance ontology. In the following sections, we will refer to the OWL provided by SSN as “SSN instance ontology”. The scenario includes two sensors and one actuator that are deployed in apartment no.134. The first sensor (s 926) observes the electric consumption of the apartment. It observed the electric consumption on 15 April 2017 00:00:00 with a value of 22.4. The other sensor (s 23) measures the temperature of the apartment. There is a window closer actuator (a 987) that is responsible for closing the window of the apartment. The actuator changed the status of the window to close on 18 April 2017 17:24:00 and return true. To simulate the manual action that should be done by the developer to write a script that encodes the above scenario, an IoT script has been written to represent the above scenario using the *OntIoT* editor. Figure 5 shows the IoT script and the created *OntIoT* textual editor.

3.1.2. Instance phase evaluation

The instance ontology, that maps to the above IoT script, has been generated using our MPS plugin. Table 5 shows a comparison among the individuals and the properties included in the *OntIoT* instance ontology and the SSN instance ontology. The *OntIoT* instance ontology does not miss any individuals or properties from the given SSN instance ontology.

```

IoT Script ApartmentScript {
  Feature Of Ineterset:
  {
    Feature Apartment_134 has properties electricConsumption temperature
    Feature W_104 has properties state
  }
  Sensors:
  {
    Sensor S_926 Observes electricConsumption
    Sensor S_23 Observes temperature
  }
  Observations:
  {
    Observation Obv_235714 made by Sensor ( S_926 )
    for property electricConsumption of Apartment_134 with value 22.4 at time 2017 / 4 / 16 0 : 0 : 0
  }
  Actuators:
  {
    Actuator A_987 for Property state
  }
  Actuations:
  {
    Actuation Act_188 done by actuator ( A_987
    on property state of W_104
    with result true at time 2017 / 4 / 18 17 : 24 : 0
  }
}

```

Figure 5. IoT script using *OntIoT* editorTable 5. *OntIoT* instance ontology evaluation

	SSN instance ontology	<i>OntIoT</i> instance ontology
Individuals	9	12
Property assertions	17	29

Table 6 shows the analysis for the results provided in Table 5 by listing the extra entities that are generated in the *OntIoT* instance ontology and do not exist in the SSN instance ontology. apartment 134 is added as an individual in the *OntIoT* instance ontology which is not included in the SSN instance ontology (I1). Two extra individuals are generated in the *OntIoT* instance ontology that map to the *Result* class with values (22.4, true) (I2, I3).

Inverse property (P1) is missing in the SSN instance ontology. In addition, the SSN instance ontology is missing the relation between window W 104 and the actuation Act 188. Accordingly, two extra properties are generated in the *OntIoT* instance ontology (P2, P3). The remaining extra properties are generated in the *OntIoT* instance ontology due to adding the new individuals (P4-P12). The SSN instance ontology contains property *hasSimpleResult(Act 188, true)* which maps to *hasResult(Act 188, true)* in the *OntIoT* instance ontology. Consequently, the property *hasResult(Act 188, true)* is not included in Table 6. This evaluation proves that *OntIoT* instance ontology models the given script successfully and includes some entities that are missing in the manual modeling of the scenario.

Table 6. *OntIoT* instance ontology extra entities

Entity type	Code	Entity name	Entity values
Individuals	I1	<i>FeatureOfInterest</i>	<i>apartment 134</i>
	I2	Result	22.4
	I3	Result	true
Properties	P1	<i>isObservedBy</i>	(<i>temperature, S 23</i>)
	P2	<i>hasFeatureOfIneterset</i>	(Act 188, W 104)
	P3	<i>isFeatureOfIneterset</i>	(W 104, Act 188)
	P4	<i>hasFeatureOfIneterset</i>	(<i>Obv 235714, apartment 134</i>)
	P5	<i>isFeatureOfIneterset</i>	(<i>apartment 134, Obv 235714</i>)
	P6	<i>isResultOf</i>	(22.4, <i>Obv 235714</i>)
	P7	<i>isResultOf</i>	(true, Act 188)
	P8	<i>hasResult</i>	(<i>Obv 235714, 22.4</i>)
	P8	<i>isPropertyOf</i>	(<i>electricConsumption, apartment 134</i>)
P10	<i>isPropertyOf</i>	(<i>temperature, apartment 134</i>)	
P11	<i>hasProperty</i>	(<i>apartment 134, electricConsumption</i>)	
P12	<i>hasProperty</i>	(<i>apartment 134, temperature</i>)	

3.1.3. Reasoning phase evaluation

In the reasoning phase, the language engineer and the developer would utilize the reasoning services of the *OntIoT* DSL. In this section, we will assume an evolution scenario for the previous *OntIoT* structure to describe the value of the provided reasoning services. The evolution scenario is shown in Table 7. Figure 6 depicts the final structure for *OntIoT* V1.3. In the following paragraphs, we will show how reasoning services will help the language engineer and the developer to detect problems in the *OntIoT* version 1.3.

Table 7. *OntIoT* evolution scenario

Version	Type	Modification
V1.0	-	Same as SSN ontology
V1.1	Add	<i>SimpleTemperatureSensor</i> Extends <i>Sensor</i>
V1.2	Add	Observes <i>TemperatureObservation</i> <i>SimpleCiscoTemperatureSensor</i> Extends <i>SimpleTemperatureSensor</i> Hosted by <i>CiscoPlatform</i>
V1.3	Add	<i>AdvancedTemperatureSensor</i> Extends <i>Sensor</i> Hosted by <i>AdvancedCiscoPlatform</i> Observes <i>AdvancedTemperatureObservation</i>
Update		<i>TemperatureObservation</i> can be observed only by <i>AdvancedTemperatureSensor</i>
Add		<i>SimpleTemperatureSensor</i> and <i>AdvancedTemperatureSensor</i> are disjoint concepts

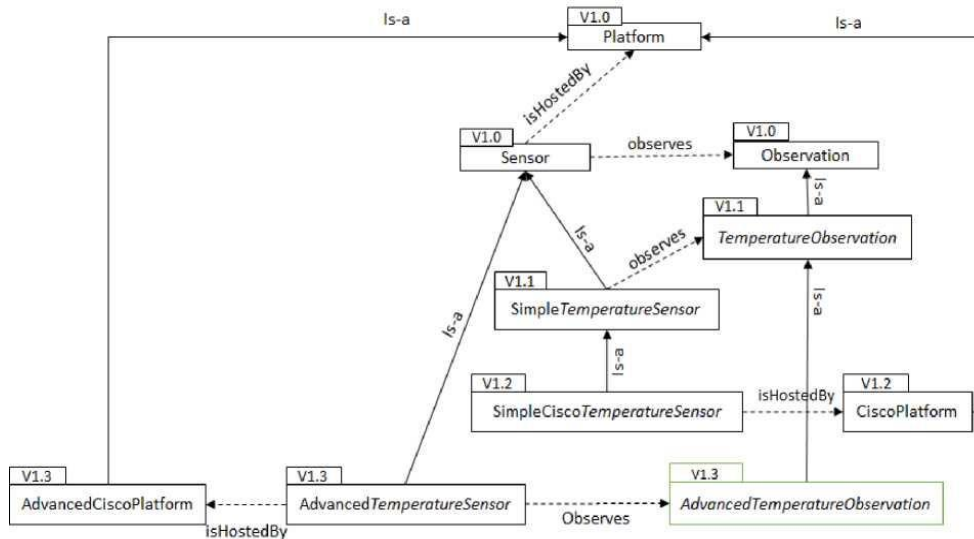


Figure 6. *OntIoT* structure V 1.3

a. DSL level

The language developer will run the reasoning services on *OntIoT* V1.3. The consistency checking detects that it is a consistent DSL that has no contradictions. Figure 7 shows that *OntIoT* V1.3 contains unsatisfiable classes. *SimpleTemperatureSensor*, *SimpleCiscoTemperatureSensor*, and *CiscoPlatform* are unsatisfiable classes. *SimpleTemperatureSensor* requires at least one *TemperatureObservation* while the *TemperatureObservation* can be observed only by *AdvancedTemperatureSensor* as per V1.3 updates. Accordingly, no instances can be created from *SimpleTemperatureSensor*. *SimpleCiscoTemperatureSensor* extends *SimpleTemperatureSensor*. Consequently, any instantiation for *SimpleCiscoTemperatureSensor* requires instantiation for *SimpleTemperatureSensor*. As a result, *SimpleCiscoTemperatureSensor* is also an unsatisfiable class. The same case for *CiscoPlatform* that requires at least one *SimpleCiscoTemperatureSensor*.

For the class equality checking, Figure 7 shows that *TemperatureObservation* and *AdvancedTemperatureObservation* are equivalent classes. The changes done in V1.3 make all temperature observations are done using *AdvancedTemperatureSensor*, so no need to differentiate between *TemperatureObservation* and *AdvancedTemperatureObservation*. The language engineer will get a list of the previous unsatisfiable and equivalent classes that should be updated in the *OntIoT* V1.3 which may not be

detected by the language engineer in the normal approaches.

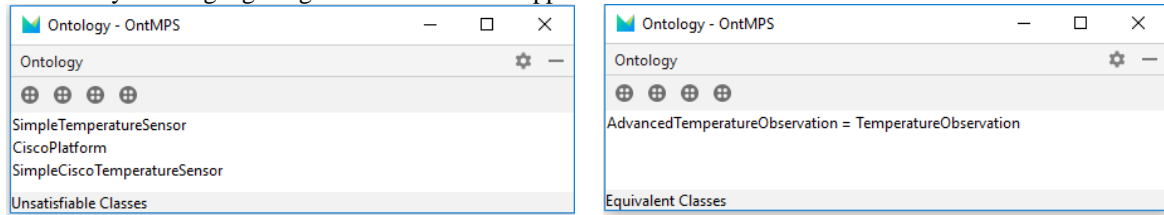


Figure 7. *OntIoT* V1.3 unsatisfiable classes and *OntIoT* V1.3 equal classes

b. Program level

Assuming that the developer used the new release from *OntIoT* editor to write the IoT script that is depicted in Figure 8. The consistency checking service detects that the script is inconsistent since it contains an instance (*cisco_S_124*) of unsatisfiable class (*SimpleCiscoTemperatureSensor*). The developer should not initialize any *SimpleCiscoTemperatureSensor* starting from *OntIoT* V 1.3.

```
Simple Cisco Sensor:
{
  Simple Cisco Sensor cisco_S_124 Observes temperature
}
Advanced Temperature Sensor:
{
  Sensor S_926 Observes electricConsumption
  Sensor S_23 Observes temperature
}
Observations:
{
  Observation Obv_235714 made by Sensor ( S_926 )
  for property electricConsumption of Apartment_134
  with value 22.4 at time 2017 / 4 / 16 0 : 0 : 0
  Observation Obv_235715 made by Sensor ( S_926 )
  for property electricConsumption of Apartment_134
  with value 22.5 at time 2017 / 4 / 16 0 : 0 : 0
  Observation Obv_235722 made by Sensor ( S_23 )
  for property temperature of Apartment_134
  with value 22.5 at time 2017 / 4 / 16 0 : 0 : 0
}
Actuators:
{
  Actuator A_987 for Property state
}
Actuations:
{
  Actuation Act_188 done by actuator ( A_987
  on property state of W_104
  with result true at time 2017 / 4 / 18 17 : 24 : 0
  Actuation Act_132 done by actuator ( A_987
  on property state of W_104
  with result true at time 2017 / 4 / 18 17 : 24 : 0
}
```

Figure 8. *OntIoT* V1.3 consistency on program level

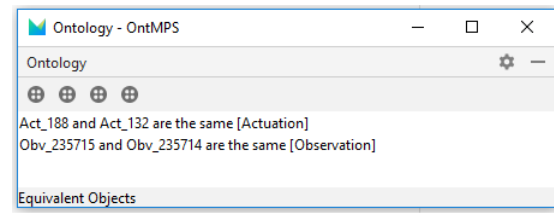
For simulating the equivalence among objects, the OWL constructs shown in Figure 9 is added to the SSN ontology. Two observations are equivalent if the two observations are made by the same sensor at the same time for the same feature of interest. A similar construct has been added to actuations. After the developer removed the reason for inconsistency by removing the *cisco_S_124* object, the developer can run the object equality service. Observations *Obv_235714* and *Obv_235715* are displayed as equivalent observations as shown in Figure 10. The developer should validate if these observations are redundant or there is a mistake in their data. In addition, the service shows that actuations *Act_188* and *Act_132* are equivalent actuations.

```

sosa:Actuation rdf:type owl:Class ;
               owl:hasKey ( sosa:actsOnProperty
                             sosa:hasFeatureOfInterest
                             sosa:madeByActuator
                             ssn:ssntimeForResult
                             ) .
sosa:Observation rdf:type owl:Class ;
                 owl:hasKey ( sosa:hasFeatureOfInterest
                               sosa:madeBySensor
                               sosa:observedProperty
                               ssn:ssntimeForResult
                               ) .

```

Figure 9. Modified SSN equality constructs

Figure 10. *OntIoT* V1.3 equal individuals

3.2. Case study 2: *OnteCom*

The proposed approach is a generic approach that does not depend on a specific domain or ontology. In this case study, the proposed approach is evaluated on the e-commerce domain. The approach is utilized to build *OnteCom* DSL. The *OnteCom* DSL is a DSL that models the e-commerce domain based on the *GoodRelations* ontology [21]. The case study proves the generalization of the mapping rules utilized in the transformation and instance phase. The generalization of the reasoning phase is extended from the standardization and generalization of the pellet reasoner.

GoodRelations ontology is an ontology to model the e-commerce aspects like companies, products, services, opening hours, and offers. The ontology is successfully used to semantically annotate the services, products and offers provided by business entities like Best Buy to enhance its visibility in search engines. *GoodRelations* ontology is supported by Google and Bing search engines. In 2012, *GoodRelations* ontology is considered as the official e-commerce core for schema.org. *Schema.org* [22] is a vocabulary to model structured data of web pages and emails. It is sponsored by Google, Microsoft, Yahoo.

The semantic model of the *GoodRelations* ontology consists mainly of three classes: i) *BusinessEntity*: models business organizations or persons. *BusinessEntity* has a set of attributes like name, address, and branches; ii) *Offering*: models the offers that are provided by the business entities. An offer includes attributes like *validFrom* and *validThrough*; and iii) *ProductOrService*: models the products and services that are included in offers.

3.2.1. Transformation phase evaluation

As shown in Table 8, *OnteCom* covers all the classes, object properties, data properties, and annotation properties of the *GoodRelations* ontology. There are three extra classes that are described in subsection 3.1.1. *OnteCom* is missing three super-class relations due to multiple inheritance which is not supported by the proposed approach up to now.

Table 8. *OnteCom* structure evaluation

	<i>GoodRelations</i> ontology	<i>OnteCom</i> structure
Classes	37	40
Object properties	93	93
Data properties	150	150
Annotation properties	100	100
Super class relation	19	16

- Creating *OnteCom* editor: as done for *OntIoT*, an editor was created manually to design the projection process of the above structure. *OnteCom* editor is a textual editor that supports tabular and textual editing for the opening hours of the branches.
- Writing e-commerce script: for evaluating the instance phase, we mix the examples provided by Semantic Web and (E-Business+Web Science) Research Group at Bundeswehr University Munich. We refer to the ontology provided by the above two sites as “*GoodRelations* Instance Ontology”. The script includes defining a business entity called *Electron.com*. *Electron.com* has an offer for one TV set with price 200 USD. *Electron.com* has one branch called *myshop* with a set of predefined opening hours.

Figure 11 shows the e-commerce script for the above example with a textual representation for the opening hours of the *myShop* branch. Figure 12 shows a tabular editor of the opening hours concept for the same abstract representation. The *OnteCom* editor shows the benefits of the projectional editor since the editor could mix textual and tabular notations. In addition, it supports multiple projections for the same abstract representation.

```

Electron.cssCom is a business entity with Legal name Electron.css.com Ltd.
Offer Offering_1 to Sell items:
  Type and Quantity TypeAndQuantityNode_1 = Amount: 1 of TVset Unit of Measure C62
  with Price Specification UnitPriceSpecification_1 = 200 USD Unit of measurement: C62
Branches: myShop open
  Saturdays From: 8 : 0 : 0 To: 14 : 0 : 0 Days Saturday
  Workdays From: 8 : 0 : 0 To: 18 : 0 : 0 Days Monday Tuesday Thursday Friday Wednesday
    
```

Figure 11. *OnteCom* editor with textual notations

```

Electron.cssCom is a business entity with Legal name Electron.css.com Ltd.
Offer Offering_1 to Sell items:
  Type and Quantity TypeAndQuantityNode_1 = Amount: 1 of TVset Unit of Measure C62
  with Price Specification UnitPriceSpecification_1 = 200 USD Unit of measurement: C62
Branches: myShop open


|           |           |            |
|-----------|-----------|------------|
| Saturday  | 8 : 0 : 0 | 14 : 0 : 0 |
| Monday    | 8 : 0 : 0 | 18 : 0 : 0 |
| Tuesday   | 8 : 0 : 0 | 18 : 0 : 0 |
| Thursday  | 8 : 0 : 0 | 18 : 0 : 0 |
| Friday    | 8 : 0 : 0 | 18 : 0 : 0 |
| Wednesday | 8 : 0 : 0 | 18 : 0 : 0 |


```

Figure 12. *OnteCom* editor with tabular notations

3.2.2. Instance phase evaluation

Table 9 shows a comparison between the individuals and properties generated in the instance ontology for the script depicted in Figure 11 and the *GoodRelations* instance ontology provided by Semantic Web and (E-Business+Web Science) research group. The *OnteCom* instance ontology does not miss any individuals from the *GoodRelations* instance ontology. All properties of *GoodRelations* instance ontology are covered in *OnteCom* instance ontology plus an extra 15 properties. These extra properties are data properties that represent a *name* data property for each individual which are not assigned in the *GoodRelations* instance ontology.

Table 9. *OnteCom* instance ontology evaluation

	<i>OnteCom</i> instance ontology	<i>GoodRelations</i> instance ontology
Individuals	15	15
Properties	39	24

4. DISCUSSION

In this section, we consider a few remaining issues. We discuss whether ontology has previously been used during the DSL development process, or not. Such discussion holds for both grammar-based languages, and model-based software development. We also discuss how our proposed approach compares to previous work within the same topic.

4.1. Has ontology been used during the DSL development process?

4.1.1. Ontology and grammar-based languages

One of the research directions to merge ontology and DSL development process is to utilize the ontology generated from the domain analysis phase to construct the target DSL grammar. Tairas *et al.* [23] investigate the use of ontology in the domain analysis to define the concepts and the relations of the domain. The developed ontology is translated into a class diagram then into a grammar. Ceh *et al.* [24], [25] proposed using ontology in the domain analysis phase and introduced *Ontology2DSL* tool that utilizes such ontology in the DSL domain analysis phase to model the domain. Then, *Ontology2DSL* uses the given ontology to automatically generate the grammar of the DSL in the implementation phase. Alternatively, Pereira *et al.* [26] translates an ontology that is used in the DSL domain analysis phase, into an attribute grammar. The generated grammar is written in the syntax ANTLR. As a result, the parser could be generated directly from this grammar. Semantic rules could be added manually to the generated grammar to get the desired behavior that is not defined in the source ontology. The generated attribute grammar contains attributes and semantic evaluation rules. Unlike our approach, the role of the ontology is terminated after generating the DSL

grammar in all aforementioned research [23]–[26]. As a result, the language engineers and developers cannot benefit from ontology reasoning services.

4.1.2. Ontology and model-based software development

To use a standard model in the model based soft-ware development (MBSD) process, an ontology can be merged with MBSD to utilize ontology as a domain model [27], [28]. Walter *et al.* [29] integrate Ecore metamodel [30] and OWL2 at the M3 layer. The integrated MetaMetaModel is used by the language engineer to create the DSL metamodel at the M2 layer. Such work was further extended provide a unified ontology-based metamodeling language to support both language engineers and domain engineers [31]. In [32], the authors propose an ontology-based framework for domain specific modeling. The framework provides a set of reasoning services like consistency, satisfiability, and classification checking. The limitations of this framework are the scalability and the usability, the time needed to perform semantic reasoning services is increased with the model's size. Moreover, the meta-language, used by the DSL engineer and the developer, integrates the OWL syntax and KM3 syntax [33]. Accordingly, the DSL engineer and the developer still should deal with the OWL syntax which is not user friendly. Jafer *et al.* [34] studied the ability of automatic transformation from OWL 2 to Ecore meta-model. Jafer *et al.* [34] provide mapping rules from OWL to Ecore. However, no concrete implementation is provided up to now for the above transformation rules.

4.2. How does our approach compare to alternative approaches for DSL development?

To our knowledge, no current research is done to integrate ontology and projectional editing. Table 10 displays a comparison between our approach and the work done in the grammar-based approach and model-based approach. The comparison includes four criteria: i) integration with an existing workbench, ii) providing reasoning services, iii) hiding OWL syntax, and iv) supporting projectional editing. While discussed research translates the ontology developed in the domain analysis phase into grammar in the implementation phase, our approach translates the ontology into an abstract model for the target DSL. This allows our approach to support textual and graphical DSLs, unlike grammar-based approaches. Furthermore, our approach provides reasoning services for a language engineer and a developer which is not supported in the grammar-based approaches.

The model-based approaches vary from our approach as follows: i) They lack support for the projectional editing; ii) Only one approach supports reasoning services for the language engineer and the developer [29], [31], [32], but it depends on an integrated metamodeling language that integrates OWL and Ecore/KM3. Hence, the language engineer should be aware of the OWL syntax. Our approach uses transformation rules to translate OWL into the DSL abstract representation. As a result, there is no need for the language engineer to be aware of the OWL syntax; and iii) None of the previous research is integrated with one of the current DSL workbenches. Our approach is integrated with meta programming system (MPS) workbench.

Table 10. Comparison with related work

Approach	Research	Integration with work-bench	Reasoning services	Hide OWL Syntax	Projectional editing
Grammar based	Tairas <i>et al.</i> [23] (2009)			√	
	Ceh <i>et al.</i> [24] (2010) and [25] (2011)			√	
	Pereira <i>et al.</i> [26] (2016)			√	
Model based	Walter <i>et al.</i> [29] (2009)		√		
	Walter <i>et al.</i> [31] (2010)		√		
	Walter <i>et al.</i> [32] (2014)		√		
	Jafer [34] (2017)			√	
	Ojamaa <i>et al.</i> [35] (2017) and Haav and Ojamaa [36] (2015)			√	
	Our Approach (2020)		√	√	√

5. CONCLUSION AND FUTURE WORK

A semi-automatic generation for a DSL from a domain ontology is proposed based on a set of mapping rules. As a result, the DSL development time and the manual work required from a language engineer are reduced. The proposed approach utilizes ontology to provide a set of reasoning services for the language engineer and the developer. There is a big similarity between the ontology structure and the semantic abstract representation for DSLs due to the similarity between the nature of an ontology and the nature of a DSL. Both concepts are used to model a domain but for different purposes. Consequently, the

projection editing approach which depends on the abstract representation to represent the DSL is more suitable for ontology integration than grammar-based approaches. Additionally, utilizing ontology in the DSL development can reduce the DSL development time since it allows automatic transformation from the domain analysis phase into the design and implementation phase. In addition, it allows formal validation for the DSL and the domain program which is missing in the current language workbenches.

Future research should also consider extending the reasoning phase by adding more reasoning and querying services. In addition, we plan to extend the mapping between OWL and DSL abstract representation to cover more OWL constructs. Further work is needed to provide an ontology and a DSL co-evolution. The changes made on the DSL should be reflected on the corresponding ontology and vice versa. Moreover, future studies could investigate utilizing ontology to allow semantic composition among DSLs. In addition, the applicability of extending OWL as a standard representation for the DSLs to allow DSLs interoperability might prove an important area for future research.




REFERENCES

- [1] R. Lämmel, "A story of a domain-specific language," in *Software Languages*, Cham: Springer International Publishing, 2018, pp. 51–86.
- [2] R. B. Kieburtz *et al.*, "A software engineering experiment in software component generation," in *Proceedings of IEEE 18th International Conference on Software Engineering*, 1996, pp. 542–552, doi: 10.1109/ICSE.1996.493448.
- [3] J.-P. Tolvanen, J. Sprinkle, and J. Gray, "The 6th OOPSLA workshop on domain-specific modeling," in *Companion to the 21st ACM SIGPLAN conference on Object-oriented programming systems, languages, and applications - OOPSLA '06*, 2006, pp. 707–708, doi: 10.1145/1176617.1176647.
- [4] A. N. Johanson and W. Hasselbring, "Effectiveness and efficiency of a domain-specific language for high-performance marine ecosystem simulation: a controlled experiment," *Empirical Software Engineering*, vol. 22, no. 4, pp. 2206–2236, Aug. 2017, doi: 10.1007/s10664-016-9483-z.
- [5] M. Fowler, "Language workbenches: the killer-app for domain specific languages," *martinfowler.com*, 2005. <https://martinfowler.com/articles/languageWorkbench.html> (accessed Feb. 03, 2023).
- [6] J. Gray, K. Fisher, C. Consel, G. Karsai, M. Mernik, and J.-P. Tolvanen, "DSLs: the good, the bad, and the ugly," in *Companion to the 23rd ACM SIGPLAN conference on Object-oriented programming systems languages and applications*, Oct. 2008, pp. 791–794, doi: 10.1145/1449814.1449863.
- [7] T. Kosar, M. Mernik, and J. C. Carver, "Program comprehension of domain-specific and general-purpose languages: comparison using a family of experiments," *Empirical Software Engineering*, vol. 17, no. 3, pp. 276–304, Jun. 2012, doi: 10.1007/s10664-011-9172-x.
- [8] T. Kosar, S. Gaberc, J. C. Carver, and M. Mernik, "Program comprehension of domain-specific and general-purpose languages: replication of a family of experiments using integrated development environments," *Empirical Software Engineering*, vol. 23, no. 5, pp. 2734–2763, Oct. 2018, doi: 10.1007/s10664-017-9593-2.
- [9] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, Dec. 2005, doi: 10.1145/1118890.1118892.
- [10] E. Negm, S. Makady, and A. Salah, "Survey on domain specific languages implementation aspects," *International Journal of Advanced Computer Science and Applications*, vol. 10, no. 11, 2019, doi: 10.14569/IJACSA.2019.0101183.
- [11] A. Barišič, M. Goulão, and V. Amaral, "Domain-specific language domain analysis and evaluation: a systematic literature review," Universidade Nova da Lisboa, 2015.
- [12] T. R. Gruber, "Toward principles for the design of ontologies used for knowledge sharing?," *International Journal of Human-Computer Studies*, vol. 43, no. 5–6, pp. 907–928, Nov. 1995, doi: 10.1006/ijhc.1995.1081.
- [13] N. MaduraiMeenachi and M. Sai Baba, "A survey on usage of ontology in different domain," *International Journal of Applied Information Systems*, vol. 4, no. 2, pp. 46–55, Sep. 2012, doi: 10.5120/ijais12-450666.
- [14] V. Dimitrieski, G. Petrović, A. Kovačević, I. Luković, and H. Fujita, "A survey on ontologies and ontology alignment approaches in healthcare," in *International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems*, 2016, vol. 9799, pp. 373–385, doi: 10.1007/978-3-319-42007-3_32.
- [15] I. Szilagyí and P. Wira, "Ontologies and semantic web for the internet of things-a survey," in *IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society*, Oct. 2016, pp. 6949–6954, doi: 10.1109/IECON.2016.7793744.
- [16] E. Negm, S. Makady, and A. Salah, "Towards ontology-based domain specific language for internet of things," in *Proceedings of the 2020 9th International Conference on Software and Information Engineering (ICSIE)*, Nov. 2020, pp. 146–151, doi: 10.1145/3436829.3436833.
- [17] A. Haller *et al.*, "The modular SSN ontology: a joint W3C and OGC standard specifying the semantics of sensors, observations, sampling, and actuation," *Semantic Web*, vol. 10, no. 1, pp. 9–32, Dec. 2018, doi: 10.3233/SW-180320.
- [18] E. Sirin, B. Parsia, B. C. Grau, A. Kalyanpur, and Y. Katz, "Pellet: a practical OWL-DL reasoner," *Journal of Web Semantics*, vol. 5, no. 2, pp. 51–53, Jun. 2007, doi: 10.1016/j.websem.2007.03.004.
- [19] F. Campagne, *The MPS language workbench: volume I*. Fabien Campagne. CreateSpace Independent Publishing Platform, 2014.
- [20] OGC, "Semantic sensor network ontology," W3C, 2017. <https://www.w3.org/TR/vocab-ssn/> (accessed Feb. 03, 2023).
- [21] M. Hepp, "Goodrelations: an ontology for describing products and services offers on the web," in *Knowledge Engineering: Practice and Patterns: 16th International Conference, EKAW 2008, Acitrezza, Italy, September 29-October 2, 2008. Proceedings 16*, vol. 5268, A. Gangemi and J. Euzenat, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 329–346.
- [22] R. V. Guha, D. Brickley, and S. Macbeth, "Schema.org: evolution of structured data on the web," *Communications of the ACM*, vol. 59, no. 2, pp. 44–51, Jan. 2016, doi: 10.1145/2844544.
- [23] R. Tairas, M. Mernik, and J. Gray, "Using ontologies in the domain analysis of domain-specific languages," in *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28-October 3, 2008. Reports and Revised Selected Papers 11*, vol. 5421, M. R. V. Chaudron, Ed. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 332–342.
- [24] I. Čeh, M. Črepinšek, T. Kosar, and M. Mernik, "Using ontology in the development of domain-specific languages," *INForum*, pp. 185–196, 2010.
- [25] I. Čeh, M. Črepinšek, T. Kosar, and M. Mernik, "Ontology driven development of domain-specific languages," *Computer Science*




- and Information Systems*, vol. 8, no. 2, pp. 317–342, 2011, doi: 10.2298/CSIS101231019C.
- [26] M. J. V. Pereira, J. Fonseca, and P. R. Henriques, “Ontological approach for DSL development,” *Computer Languages, Systems & Structures*, vol. 45, pp. 35–52, Apr. 2016, doi: 10.1016/j.cl.2015.12.004.
- [27] S. Staab, T. Walter, G. Gröner, and F. S. Parreiras, “Model driven engineering with ontology technologies,” in *Reasoning Web. Semantic Technologies for Software Engineering*, 2010, pp. 62–98.
- [28] H. M. Haav, “A comparative study of approaches of ontology driven software development,” *Informatica*, vol. 29, no. 3, pp. 439–466, 2018, doi: 10.5555/ios.INF1185.
- [29] T. Walter, F. Silva Parreiras, and S. Staab, “OntoDSL: an ontology-based framework for domain-specific languages,” in *Model Driven Engineering Languages and Systems*, 2009, pp. 408–422.
- [30] D. Steinberg, F. Budinsky, E. Merks, and M. Paternostro, *EMF: eclipse modeling framework*. Pearson Education, 2008.
- [31] T. Walter, F. S. Parreiras, S. Staab, and J. Ebert, “Joint language and domain engineering,” in *Modelling Foundations and Applications*, 2010, pp. 321–336.
- [32] T. Walter, F. S. Parreiras, and S. Staab, “An ontology-based framework for domain-specific modeling,” *Software & Systems Modeling*, vol. 13, no. 1, pp. 83–108, Feb. 2014, doi: 10.1007/s10270-012-0249-9.
- [33] F. Jouault and J. Bézuvin, “KM3: a DSL for metamodel specification,” in *Formal Methods for Open Object-Based Distributed Systems*, 2006, pp. 171–185.
- [34] S. Jafer, B. Chhaya, and U. Durak, “OWL ontology to Ecore metamodel transformation for designing a domain specific language to develop aviation scenarios,” in *Proceedings of the symposium on model-driven approaches for simulation engineering*, 2017, pp. 1–11.
- [35] A. Ojamaa, H.-M. Haav, and J. Penjam, “Semi-automated generation of DSL meta models from formal domain ontologies,” in *Model and Data Engineering*, 2015, pp. 3–15.
- [36] H. M. Haav and A. Ojamaa, “Semi-automated integration of domain ontologies to DSL meta-models,” *International Journal of Intelligent Information and Database Systems*, vol. 10, no. 1/2, 2017, doi: 10.1504/IJIDS.2017.086198.

BIOGRAPHIES OF AUTHORS






Eman Negm    received the B.Sc. and M.Sc. degrees in computer science from the Faculty of Computers and Information, Cairo University in 2008, and 2014 respectively. She received her Ph.D. degree in computer science from the Faculty of Computers and Artificial Intelligence, Cairo University in 2021. Currently, she is an assistant lecturer at the Department of Computer Science, Faculty of Computers and Artificial Intelligence, Cairo University. Her research interests include software engineering, software modeling, ontologies, and domain specific languages. She can be contacted at email: e.negm@fci-cu.edu.eg.



Akram Salah    graduated from mechanical engineering and worked in computer programming for 7 years before he got his M.Sc. (85) and Ph.D. degrees from University of Alabama at Birmingham, USA in 1986 in computer and information sciences. He taught in the American University in Cairo, Michigan State University, Cairo University, before he joined North Dakota State University where he designed and started a graduate program that offers Ph.D. and M.Sc. in software engineering. Dr. Salah’s research interest was in data knowledge, and software engineering. He had over than 100 published papers. He was an associate professor in the Faculty of Computers and Artificial Intelligence, Cairo University. He can be contacted at email: akram.salah@fci-cu.edu.eg.



Soha Makady    received her B.Sc. and M.Sc. in computer science with an honorary degree from the Faculty of Computers and Information, Cairo University in 2002 and 2005 respectively. She received her Ph.D. in software engineering from the University of Calgary, Canada in 2015. Her main research interests include software evolution, software architecture, and software testing. She is currently an associate professor in the Faculty of Computers and Artificial Intelligence, Cairo University, Egypt. She has supervised 2 M.Sc. students and one Ph.D. student. She is currently supervising 3 Ph.D. students and 3 M.Sc. students. She has 10 referred research papers in international journals and conference proceedings. She can be contacted at email: s.makady@fci-cu.edu.eg.