Formalizing Ontologies for AI Models Validation: from OWL to Event-B

Mohamed Ould Bah*†1, Zakaryae Boudi*†, Mohamed Toub*†, Abderrahim Ait Wakrime‡, and Ghassane Aniba†
*TrouveTaVoie, Paris, France
Emails: bah@trouvetavoie.io, boudi@trouvetavoie.io, toub@trouvetavoie.io
†Mohammadia School of Engineers, Mohammed V University in Rabat, Morocco
Email: ghassane@emi.ac.ma
‡Computer Science Department, Faculty of Sciences, Mohammed V University in Rabat, Morocco
Email: abderrahim.aitwakrime@um5.ac.ma

***Abstract*—Quality data is of decisive importance for controlling critical cyber-physical systems. Most common real-life systems are driven by unstructured, decentralized, and growing amounts of data, while validation requires coherent data, that is well structured, consistent, and without ambiguities. Often, ontologies are well-suited for capturing domain knowledge data, deriving requirements, providing analysis, and developing applications. Still, ontological representations fall short when it comes to formal verification and validation, especially for large complex systems. In this research, we suggest a fully automated approach to transform ontology axioms, expressed in the Web Ontology Language (OWL), to Event-B predicates. We show, in a practical example, how bridging OWL to Event-B can scale the validation of ontology-driven AI systems.**

***Index Terms*—Event-B, Formal Methods, Ontology, OWL, AI**

I. Introduction

With very large volumes of data exchanged every day, providing automated processing solutions is prerequisite for developing scalable systems and applications. However, such automation - and data - at scale must be subject to rigorous control and validation processes, especially within evolving and domain-specific environments, where safety and quality standards are high priority.

Putting control on living domain data is key to any data-driven system such as artificial intelligence (AI) implementations. Here, ontologies stand out for capturing the data in programmable ways. Ontologies are representations of domain knowledge that define concepts and their relationships. They can be used both alone or with other ontologies and offer interoperability solutions to data heterogeneity problems, providing a factorization benefit to knowledge so it can be reused in various projects. [1], [2]. For that, several languages have been introduced in the literature, including the Web Ontology Language (OWL), which is one of the most used and popular ontology representation languages [3].

But capturing data is just one part of the solution. Building valid systems with the data needs engineers and designers to prove that specific data and functional requirements hold. That brings formal methods to the scope. Formalization of ontologies allows the derivation of formal requirements and their integration into system development cycles, resulting in better system specifications and bridging the gap between domain information - that is updated through an ontology - and system developers who use the domain ontology's formalized requirements in their specifications or programs. In the case of safety-critical systems (SCS), such domain knowledge integration is of core importance to limit potential risks [4].

Formal methods are about using mathematical techniques for describing system properties, enabling systematic specification, development, and validation frameworks [5]–[7]. Our research suggests that advancing ontology integration with formal methods is a key step towards scaling robust and safe AI systems, and this contribution presents a fully automated transformation framework for OWL ontologies to Event-B (B-method), which is a formal method for modeling and analyzing systems at different levels of abstractions, including property verification with mathematical proofs [8]. Our approach defines rules to each OWL/XML ontological axiom, without the need to model the relationships between concepts prior to their transformation,

The following section recalls some related works on OWL combinations with the B-method, and section 3 details the transformation approach. A concrete example with open-source code are also

[1] Corresponding author.

provided next.

II. Related Work

Ontologies and formal methods have been extensively investigated in the literature, separately. Only few researchers showed interest in studying both areas combined, and the integration of formal ontology-derived requirements in system modeling is still an open subject. For example, authors in [9] presented a transformation approach for domain knowledge formalization in nuclear systems, through a specific ontology, to be used in the system's formal model expressed in Event-B. The model actions are transformed into an OWL ontology and sent back to the domain experts. But this approach has a drawback : it is focused on nuclear domain knowledge and cannot apply to general use cases.

Another work, in [10], proposed an approach to derive formal requirements from a given ontology that takes OWL/XML representations as input, transforms them to Attempto Controlled English (ACE) using OWL verbalizer, and then maps the generated ACE texts into Event-B notations. ACE is a controlled natural language which is a first-rate logical language expressed in English that can be interpreted by humans and machines [11]. It was used in the OWL verbalizer project as a result of the transformation of OWL/XML ontologies [12]. OWL verbalizer is not able however to verbalize all OWL axioms, which considerably limits automation options at scale.

Authors in [13] suggest the use of the ontological languages OWL and PLIB [14], to create a Universal Modeling Languages (UML) meta-model and combine it with SysML/KAOS, which is an engineering method representing system requirements as a hierarchy of goal, where each level corresponds to a goal refinement [15]. The proposed meta-model aims to overcome the lack of expressiveness in KAOS representations with few adjustments as they are not suited for SCS. This approach is however very complex for implementation, and requires manual changes in the meta-model for any evolution in the knowledge representation to keep it up to date.

Finally, the study in [16] proposes a less complex approach that transforms OWL ontologies to Event-B, by introducing transformation rules related to all ontological concepts. It takes place in three stages: formalization, transformation, and annotation. First, the system to be developed must be formalized in Event-B to model its behavior. Next, the ontology is transformed into an Event-B context using sets, constants, and axioms. The constraints are then integrated into the context. The transformation phase is also split into three parts : Input model, Pivot model (PM), and Output model. The input model takes in the ontology, extracts the ontological concepts to be processed, and hands them to the Pivot model. The PM contains the definitions of the generic ontological concepts that can be processed and transforms the specific concepts received into generic ones. The result of this transformation is sent to the Output model containing the transformation rules. The approach provides a transformation rule for each ontological concept. An updated version was published in [17], extending the PM and adding the option to generate a PM file. Much like the approach in reference [13], changes in the OWL standard will require an update of the PM as well as adding new transformation rules. The framework presented here is also platform-dependent, since it works only as a plugin for Rodin, which is a tool that supports Event-B modeling [18]. Also, the transformation does not consider the ontology instances and their relationships in the transformation.

This work falls within the framework of the research project around building valid ontologies and AI systems in the HR landscape [19], and can be viewed as a continuation of previous model transformation approaches explored in [20]–[24], [24]–[29], which focus on enhancing the verification oversight of complex systems by combining formal methods such as Petri nets and the B-method. Our contribution lies in the fact that modelling concepts is not needed, only a set of transformation rules are required. Compared to the works cited before [16], [17], our approach includes instances in the transformation, and only the concepts' transformation rules are updated, with no model to worry about in case an update is applied to the transformation framework.

III. Approach

A. Transformation scheme

To explain the overall scheme of our transformation approach, a general view is presented in Fig.1, which highlights the positioning of this work in the entire process of formal ontology integration.

The ontology must be expressed in OWL/XML format before providing it as input to the tool. The tool generates a file containing the transformation of ontology axioms. The generated Event-B axioms are used as axioms for Context components in Event-B modeling environment.

B. Transformation methodology

The input OWL/XML ontology undergoes two major transformations, as shown in Fig.2. The first

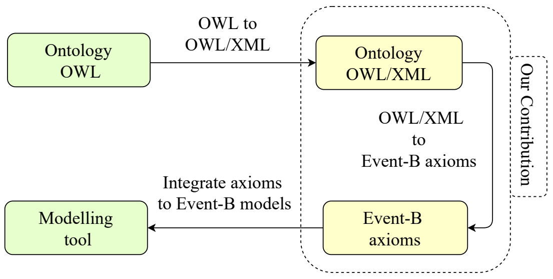


Fig. 1. Overview of the ontology formalization for AI model validation

transformation has an intermediary role and expresses each axiom as a predicate, in a format similar to OWL FFS. This transformation allows the axioms to be interpreted as predicates using Swi-Prolog - which is a Programming in Logic (Prolog) language mostly used for processing natural languages and artificial intelligence applications [30]. The intermediary transformation is followed by a second transformation that translates each axiom to an equivalent Event-B notation.

The transformation of OWL/XML axioms into Swi-Prolog predicates enables the categorization of axioms into three different types, each type requiring a different processing technique and with a specific complexity level. To illustrate this we will use the OWL/XML transformation of the well-known Pizza ontology [31]. Table I presents examples of the three axiom types.

C. Intermediary transformation

This part of the transformation is done in two separate phases as well: loading the OWL/XML ontology into a list structure and transforming its axioms into predicates. The Swi-Prolog built-in predicate **load_structure** loads an XML structure into a list while keeping its hierarchy and all elements unchanged. Each element of this generated list is transformed into another predicate. The resultant list of predicates is at this stage ready to undergo the second and final transformation.

As mentioned in the previous section, three major types of axioms were distinguished. Type 1 axioms can be directly translated into Event-B axioms. They either have a value as argument such as **IRI(”pizza.owl#isToppingOf”)** or an entity such as **Declaration(Class(”Veneziana”))**. Type 1 axioms have the lowest complexity.

Type 2 axioms can take 2 or 3 arguments, which can be axioms, entities or values. **EquivalentClasses([Class(Spiciness), ObjectUnionOf([Class(Hot), Class(Medium), Class(Mild)])])** is a good example of type 2 axioms, since it has an entity as its first argument but has another axiom as its second argument. For type 2, we provide a transformation for all possible combinations of arguments for each axiom: $Entity \times Entity,\ Entity \times Axiom,\ Axiom \times Entity,\ and\ Axiom \times Axiom.$ Type 2 axioms' complexity depends on the type of arguments they have. Their complexity can be as low as a type 1 axiom, or as high as a recursive function.

Type 3 axioms are axioms that can take any number of arguments as a list. From The Pizza ontology, there is an example of type 3 axiom **ObjectUnionOf([AnchoviesTopping, ... ,TomatoTopping])**. These arguments can be a value, an entity, or an axiom. To process a type 3 axiom we defined the transformation for the axiom with 2 arguments using type 2 definition. Then we use a call back of the transformation for every 2 elements of the list. Generally, Type 3 axioms have a recursive complexity.

D. Final transformation

In this step, the OWL/XML axioms are in predicate forms and will be interpreted as such by Swi-Prolog. These predicates are mapped to Definite Clause Grammar (DCG) rules for each OWL/XML axiom. DCG is a Prolog formalism, for artificial and natural language analysis and processing [32]. DCG rules results are gathered in a list which will be processed element by element to construct the ontology's transformation output. The notation **Predicate/Arity** will be used to refer to DCG rules. **Predicate** is the name of the rule, and **Arity** is

TABLE I
AXIOM TYPES

Type 1 axiom	**OWL FSS form**	Axiom(:X)
	Predicate form	Axiom(X)
	Example	Declaration(Class(Veneziana))
Type 2 axiom	**OWL FSS form**	Axiom(:X :Y) and Axiom(:X :Y :Z)
	Predicate form	Axiom(X,Y) and Axiom(X,Y,Z)
	Example	EquivalentClasses(Class(Spiciness),ObjectUnionOf([Class(Hot),Class(Medium),Class(Mild)]))
Type 3 axiom	**OWL FSS form**	Axiom(:X_1 :X_2 ...:X_n)
	Predicate form	Axiom([X_1,X_2,...,X_n])
	Example	ObjectUnionOf([AnchoviesTopping, ...,TomatoTopping])

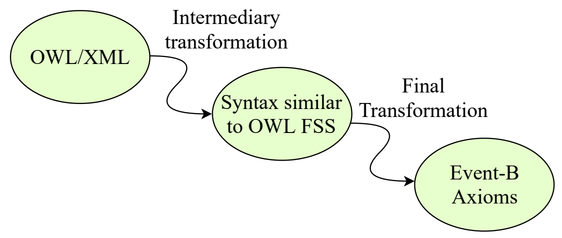


Fig. 2. OWL to B-method transformation approach schematic

the number of arguments. The DCG rules for the different types are presented in Table II.

An example of type 1 DCG rules is **'Declaration'/1**, with **'Class'(X)** as argument. By applying this rule on **Declaration(Class(Veneziana))**, we get the result as a list of elements constructing the axioms translation. The list is then computed element by element to get the axiom as a string. An example of type 1 DCG rules is provided in Listings 1.

Applying type 2 DCG rule to an axiom will test that axiom for each possible written arguments variations of the rule. In the example provided in Listings 2, we have an example type 2 rule **'SubClassOf'/2** and **'ObjectIntersectionOf'/2**. For a given **SubClassOf(Class(Boy), ObjectIntersectionOf(Class(Male),Class(Young)))**, the transformation is going to follow the **'SubClassOf'('Class'(X),AxiomY)** rule.

In this Example of Type 3 DCG rules, we showcase the **'ObjectUnionOf'(List)** example. As mentioned earlier for type 3 axioms, we first define a type 2 rule, **'ObjectUnionOf'/2**, then we define a type 3 rule, **'ObjectUnionOf'/1**, which calls the type 2 rule for every two arguments of the list. **'ObjectUnionOf'/1** is browsing through its list of arguments by a step of two. Lets take, for instance, the axiom **ObjectUnionOf([Class(Hot),Class(Medium),Class(Mild)])**, the transformation is done according to the rule **'ObjectUnionOf'/1**. An example for type 3 DCG rules is provided in Listings 3.

'Declaration'('ObjectProperty'(R)) and **'Declaration'('DataProperty'(R))** axioms, among others, require special rules to be transformed correctly. Since properties' declarations depend on their range and domain, they need to be collected if they exist in the ontology and are used in the DCG rule definition. Let's take **'ObjectProperty'**, for instance, every **'ObjectPropertyDomain'(R, Dom)** axiom is going to be tested if it has the same **IRI** as the property **R** in the declaration, its domain **Dom** is collected. If no match is found, the domain is going to be replaced by the class **Thing**. The same procedure is performed to collect the range **Ran** of the property **R**. Once the range and property are collected, the DCG rule **property(R, Dom, Ran)** is called to generate the Event-B axiom. The DCG rule **property/3** is called using the built-in predicate **phrase/2**, which collects the result in the second argument, as shown in listings 4.

IV. Case study

A. *Ontology : Employees & Skills*

For simplicity, a sample ontology containing the three axiom types is used to test and validate our approach. This ontology presents a group of employees linked to their set of skills. The class "employee" contains a group of employees, each defined by their name and linked to one or more skills. The Class "Competency" contains sub-classes, "Web_Development" and "Software_Development", each grouping a more refined set of skills. Each employee is linked to one or more skills by the "HasCompetency" relationship.

Fig. 3(a) shows the relationships between the ontology instances and the property "HasCompetency", while Fig. 3(b) shows the ontology entities and the rest of the relationships.

The ontology, in this case study, was created using Protégé and exported in OWL/XML format. Protégé is one of the well-known tools for ontology creation and editing that imports/exports ontologies in various formats [33]. Note that if the ontology was exported in OWL format, it can be converted to OWL/XML using the service OWL Syntax Con-

TABLE II
DCG rules types

Type 1 DCG rule	**Axiom Example**	Declaration(Class(Veneziana))
	DCG rule result	['Veneziana',' <: Thing']
	Output	Veneziana <: Thing
Type 2 DCG rule	**Axiom Example**	SubClassOf(Class(Boy),ObjectIntersectionOf(Class(Male) ,Class(Young)))
	DCG rule result	['Boy',' <: (','Male', ' /\\','Young',')']
	Output	Boy <: (Male /\Young)
Type 3 DCG rule	**Axiom Example**	ObjectUnionOf([Class(Hot), Class(Medium),Class(Mild)])
	DCG rule result	['(','Hot', ' \\/ ', 'Medium',') \\/ (','Mild',')']
	Output	(Hot \/ Medium) \/ (Mild)

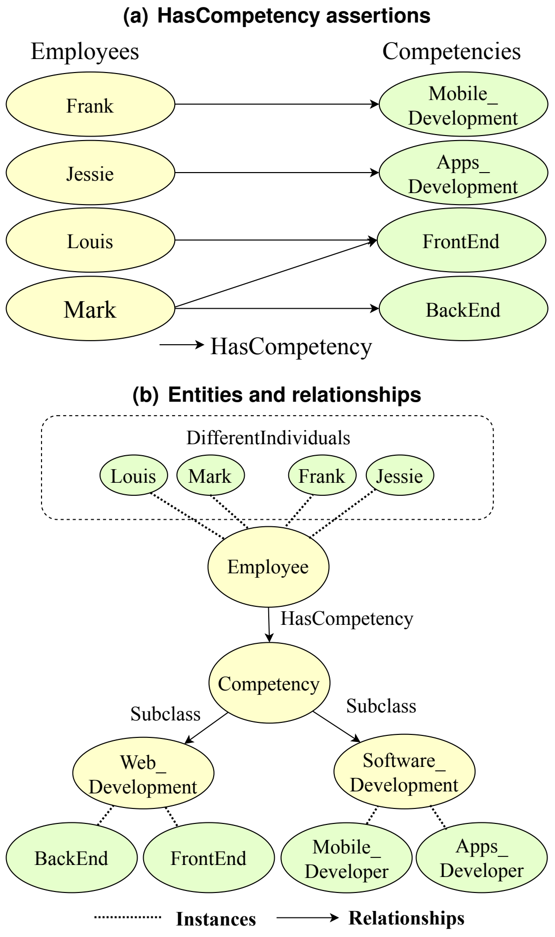
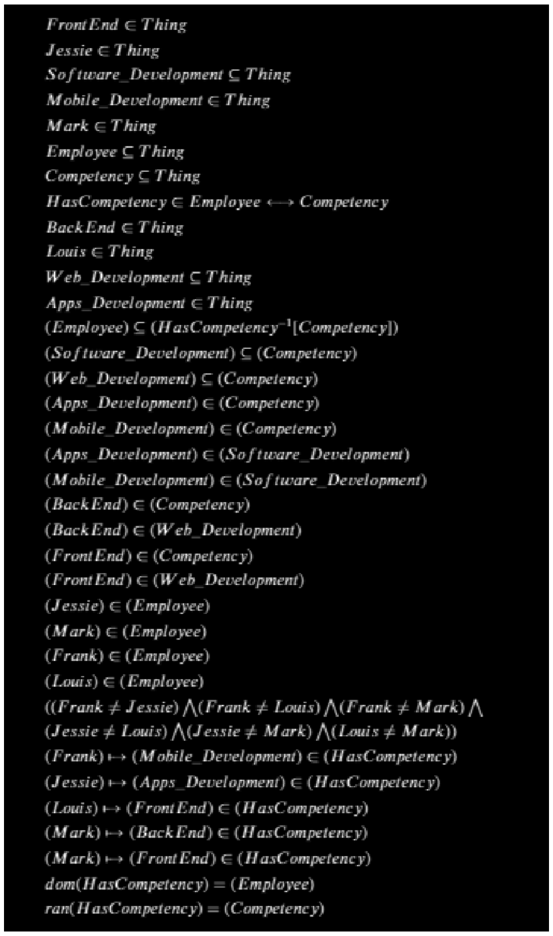


Fig. 3. Case Study Ontology: Employees & Competencies.



$FrontEnd \in Thing$

$Jessie \in Thing$

$Software\_Development \subseteq Thing$

$Mobile\_Development \in Thing$

$Mark \in Thing$

$Employee \subseteq Thing$

$Competency \subseteq Thing$

$HasCompetency \in Employee \longleftrightarrow Competency$

$BackEnd \in Thing$

$Louis \in Thing$

$Web\_Development \subseteq Thing$

$Apps\_Development \in Thing$

$(Employee) \subseteq (HasCompetency^{-1}[Competency])$

$(Software\_Development) \subseteq (Competency)$

$(Web\_Development) \subseteq (Competency)$

$(Apps\_Development) \in (Competency)$

$(Mobile\_Development) \in (Competency)$

$(Apps\_Development) \in (Software\_Development)$

$(Mobile\_Development) \in (Software\_Development)$

$(BackEnd) \in (Competency)$

$(BackEnd) \in (Web\_Development)$

$(FrontEnd) \in (Competency)$

$(FrontEnd) \in (Web\_Development)$

$(Jessie) \in (Employee)$

$(Mark) \in (Employee)$

$(Frank) \in (Employee)$

$(Louis) \in (Employee)$

$((Frank \neq Jessie) \bigwedge (Frank \neq Louis) \bigwedge (Frank \neq Mark) \bigwedge$

$(Jessie \neq Louis) \bigwedge (Jessie \neq Mark) \bigwedge (Louis \neq Mark))$

$(Frank) \mapsto (Mobile\_Development) \in (HasCompetency)$

$(Jessie) \mapsto (Apps\_Development) \in (HasCompetency)$

$(Louis) \mapsto (FrontEnd) \in (HasCompetency)$

$(Mark) \mapsto (BackEnd) \in (HasCompetency)$

$(Mark) \mapsto (FrontEnd) \in (HasCompetency)$

$dom(HasCompetency) = (Employee)$

$ran(HasCompetency) = (Competency)$

Fig. 4. Transformation results.

verter [34], which converts OWL to different formats including OWL/XML. Furthermore, we used Protégé's reasoner which not only reveals inconsistencies, but infers all implicit relationships, as well. Protégé's reasoner usage is completely optional. But the inferred relationships provide a more complete ontology.

B. Ontology transformation

The transformation framework takes the ontology expressed in OWL/XML format and converts it into equivalent Event-B axioms as shown in Fig. 4. To make the generated Event-B axioms easily adaptable for different Event-B modeling platforms, we chose to express each Event-B construct in ASCII form.

C. Requirements integration

The transformed axioms can be integrated into an Event-B modeling tool as contexts. Here we are using the Rodin tool, supporting analysis and verification functionalities that can be expanded by additional plug-ins [18]. Note that other tools such as Atelier B can be used as well.

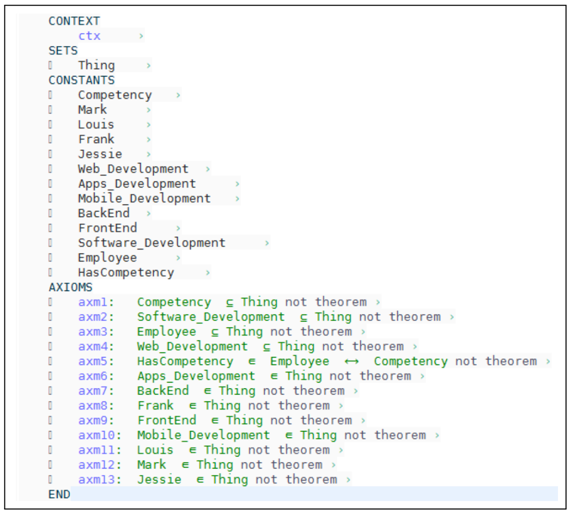
For clarity purposes, we separated the context into two sub-contexts. The first one is *ctx* (Fig. 5(a)), and the second one *ctx_1* (Fig. 5(b)) extending from *ctx*. And we divided the **DifferentIndividuals** equivalent generated axiom (Fig. 5(b)), into two axioms : axiom16 and axiom17.

Let's suppose we would like to undertake some internal adjustments in our organization, by removing all employees with ”FrontEnd” skills and relocating them elsewhere. To get all our ontology requirements, we'll use *ctx_1* (extends *ctx*) in our machine model. The machine *mach* (Fig. 5(c)), groups all employees with the ”FrontEnd” development skills in a new set, and removes them from the employees set.

V. Conclusion & Perspectives

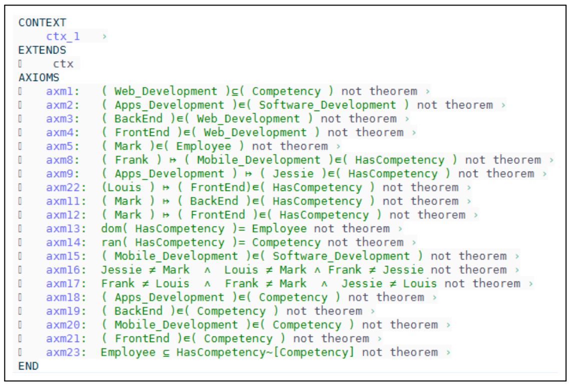
In this paper, we presented an approach automating the transformation of concepts from an OWL ontology into Event-B axioms. Each axiom of the

(a) Context *ctx*



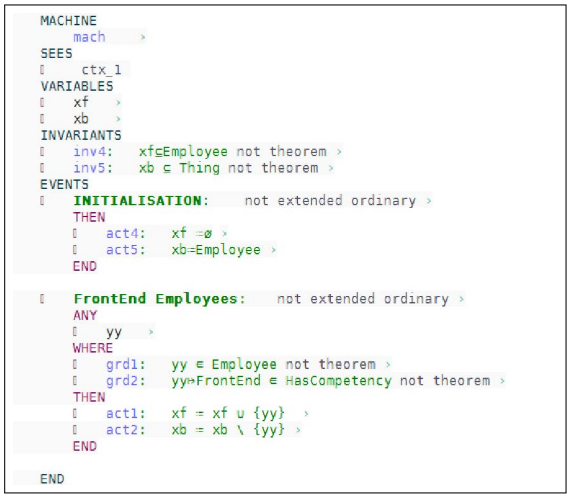
```
CONTEXT
    ctx
SETS
    Thing
CONSTANTS
    Competency
    Mark
    Louis
    Frank
    Jessie
    Web_Development
    Apps_Development
    Mobile_Development
    BackEnd
    FrontEnd
    Software_Development
    Employee
    HasCompetency
AXIOMS
    axm1:   Competency  ⊆ Thing not theorem
    axm2:   Software_Development  ⊆ Thing not theorem
    axm3:   Employee  ⊆ Thing not theorem
    axm4:   Web_Development  ⊆ Thing not theorem
    axm5:   HasCompetency  ∈  Employee  ↔  Competency not theorem
    axm6:   Apps_Development  ∈ Thing not theorem
    axm7:   BackEnd  ∈ Thing not theorem
    axm8:   Frank  ∈ Thing not theorem
    axm9:   FrontEnd  ∈ Thing not theorem
    axm10:  Mobile_Development  ∈ Thing not theorem
    axm11:  Louis  ∈ Thing not theorem
    axm12:  Mark  ∈ Thing not theorem
    axm13:  Jessie  ∈ Thing not theorem
END
```

(b) Context *ctx_1* (extends *ctx*)



```
CONTEXT
    ctx_1
EXTENDS
    ctx
AXIOMS
    axm1:   ( Web_Development )⊆( Competency ) not theorem
    axm2:   ( Apps_Development )∈( Software_Development ) not theorem
    axm3:   ( BackEnd )∈( Web_Development ) not theorem
    axm4:   ( FrontEnd )∈( Web_Development ) not theorem
    axm5:   ( Mark )∈( Employee ) not theorem
    axm8:   ( Frank ) ↦ ( Mobile_Development )∈( HasCompetency ) not theorem
    axm9:   ( Apps_Development ) ↦ ( Jessie )∈( HasCompetency ) not theorem
    axm22:  (Louis ) ↦ ( FrontEnd)∈( HasCompetency ) not theorem
    axm11:  ( Mark ) ↦ ( BackEnd )∈( HasCompetency ) not theorem
    axm12:  ( Mark ) ↦ ( FrontEnd )∈( HasCompetency ) not theorem
    axm13:  dom( HasCompetency )= Employee not theorem
    axm14:  ran( HasCompetency )= Competency not theorem
    axm15:  ( Mobile_Development )∈( Software_Development ) not theorem
    axm16:  Jessie ≠ Mark  ∧  Louis ≠ Mark ∧ Frank ≠ Jessie not theorem
    axm17:  Frank ≠ Louis  ∧  Frank ≠ Mark  ∧  Jessie ≠ Louis not theorem
    axm18:  ( Apps_Development )∈( Competency ) not theorem
    axm19:  ( BackEnd )∈( Competency ) not theorem
    axm20:  ( Mobile_Development )∈( Competency ) not theorem
    axm21:  ( FrontEnd )∈( Competency ) not theorem
    axm23:  Employee ⊆ HasCompetency~[Competency] not theorem
END
```

(c) Machine *mach* (sees *ctx_1*)



```
MACHINE
    mach
SEES
    ctx_1
VARIABLES
    xf
    xb
INVARIANTS
    inv4:   xf⊆Employee not theorem
    inv5:   xb ⊆ Thing not theorem
EVENTS
    INITIALISATION:    not extended ordinary
    THEN
        act4:   xf ≔∅
        act5:   xb≔Employee
    END

    FrontEnd Employees:    not extended ordinary
    ANY
        yy
    WHERE
        grd1:   yy ∈ Employee not theorem
        grd2:   yy↦FrontEnd ∈ HasCompetency not theorem
    THEN
        act1:   xf ≔ xf ∪ {yy}
        act2:   xb ≔ xb ∖ {yy}
    END

END
```

Fig. 5. Ontology-derived Context & Machines.

input ontology goes through two major transformations. The first transformation, which is intermediary, transforms the axiom into a Swi-Prolog predicate form. The second transformation translates it to an Event-B axiom using predefined DCG rules. The framework includes instances as well as relationships in the transformation process, which maintains the integrity of the ontology knowledge.

Our approach requires only the definition of DCG transformation rules for each given axiom facilitating the addition, modification and the deletion of transformation rules. The framework's generated Event-B axioms are in ASCII form, making them easily adaptable for different modeling platform.

Future work will focus on implementing such a transformation into a further deep reinforcement learning framework, combining OWL, B-method and CPN formalisms, with the purpose of introducing a preliminary approach to controlling AI systems in a formal setup, using ontology as a requirement representation.

ACKNOWLEDGEMENT

Financial support from the National Center for Scientific and Technical Research (CNRST) is gratefully acknowledged.

REFERENCES

[1] Bruno Bachimont, Antoine Isaac, and Raphaël Troncy. Semantic Commitment for Designing Ontologies: A Proposal. In Asunción Gómez-Pérez and V. Richard Benjamins, editors, *Knowledge Engineering and Knowledge Management: Ontologies and the Semantic Web*, Lecture Notes in Computer Science, pages 114–121, Berlin, Heidelberg, 2002. Springer.

[2] Thomas R Gruber. The Role of Common Ontology in Achieving Sharable, Reusable Knowledge Bases. page 4.

[3] Deborah L. McGuinness and Frank Van Harmelen. OWL web ontology language overview. *W3C recommendation*, 10(10):2004, 2004.

[4] John C. Knight. Safety critical systems: challenges and directions. In *Proceedings of the 24th International Conference on Software Engineering*, ICSE '02, pages 547–550, Orlando, Florida, May 2002. Association for Computing Machinery.

[5] Edmund M. Clarke and Jeannette M. Wing. Formal methods: State of the art and future directions. *ACM Computing Surveys (CSUR)*, 28(4):626–643, 1996. Publisher: ACM New York, NY, USA.

[6] Abderrahim Ait Wakrime, J. Paul Gibson, and Jean-Luc Raffy. Formalising the requirements of an E-voting software product line using Event-B. In *2018 IEEE 27th International Conference on Enabling Technologies: Infrastructure for Collaborative Enterprises (WETICE)*, pages 78–84. IEEE, 2018.

[7] Abderrahim Ait Wakrime, Souha Boubaker, Slim Kallel, and Walid Gaaloul. A SAT-Based Formal Approach for Verifying Business Process Configuration. In *International Conference on Big Data Innovations and Applications*, pages 47–62. Springer, 2019.

[8] Jean-Raymond Abrial. *Modeling in Event-B: system and software engineering*. Cambridge University Press, 2010.

[9] Jihun Kim and Moon-Ghu Park. Formal development of an operation monitoring and control system for nuclear reactors using Event-B method. *International Journal of Energy Research*, 2020. Publisher: Wiley Online Library.

[10] Eman H. Alkhammash. Derivation of Event-B models from OWL ontologies. In *MATEC Web of Conferences*, volume 76, page 04008. EDP Sciences, 2016.

[11] Norbert E. Fuchs, Kaarel Kaljurand, and Tobias Kuhn. Attempto Controlled English for Knowledge Representation. In Cristina Baroglio, Piero A. Bonatti, Jan Małuszyński, Massimo Marchiori, Axel Polleres, and Sebastian Schaffert, editors, *Reasoning Web: 4th International Summer School 2008, Venice, Italy, September 7-11, 2008, Tutorial Lectures*, Lecture Notes in Computer Science, pages 104–124. Springer, Berlin, Heidelberg, 2008.

[12] Kaarel Kaljurand and Norbert E. Fuchs. Verbalizing owl in attempto controlled english. 2007. Publisher: University of Zurich.

[13] Steve Tueno, Régine Laleau, Amel Mammar, and Marc Frappier. Integrating Domain Modeling Within a Formal Requirements Engineering Method. In *Implicit and Explicit Semantics Integration in Proof-Based Developments of Discrete Systems*, pages 39–58. Springer, 2020.

[14] Guy Pierra. Context-explication in conceptual ontologies: the PLIB approach. In *ISPE CE*, pages 243–253, 2003.

[15] Steve Tueno, Régine Laleau, Amel Mammar, and Marc Frappier. The SysML/KAOS domain modeling approach. *arXiv preprint arXiv:1710.00903*, 2017.

[16] Linda Mohand-Oussaïd and Idir Aït-Sadoune. Formal modelling of domain constraints in Event-B. In *International Conference on Model and Data Engineering*, pages 153–166. Springer, 2017.

[17] Idir Ait-Sadoune and Linda Mohand-Oussaid. Building Formal Semantic Domain Model: An Event-B Based Approach. In *International Conference on Model and Data Engineering*, pages 140–155. Springer, 2019.

[18] Michael Jastram and Prof Michael Butler. Rodin User's Handbook: Covers Rodin v. 2.8. 2014. Publisher: CreateSpace Independent Publishing Platform.

[19] Zakaryae Boudi, Abderrahim Ait Wakrime, Mohamed Toub, and Mohamed Haloua. Building Valid Career Ontologies with B-CPNs. In Mohamed Hamlich, Ladjel Bellatreche, Anirban Mondal, and Carlos Ordonez, editors, *Smart Applications and Data Analysis*, Communications in Computer and Information Science, pages 33–46, Cham, 2020. Springer International Publishing.

[20] Zakaryae BOUDI, El Miloudi El Koursi, and Simon Collart-Dutilleul. Safety critical software construction using CPN modeling and B method's proof. In *SESA 2014, Software Engineering and Systems Architecture*, page 4p, Tétouan, Morocco, December 2014.

[21] Zakaryae BOUDI, El Miloudi El Koursi, Simon Collart-Dutilleul, and Moha KHADDOUR. High level Petri net modeling For railway safety critical scenarios. In *10th FORMS-FORMAT symposium, Formal Methods for Automation and Safety in Railway and Automotive Systems*, pages p65–75, Braunschweig, Germany, September 2014.

[22] Zakaryae Boudi, El Miloudi El Koursi, and Simon Collart-Dutilleul. Colored Petri Nets formal transformation to B machines for safety critical software development. In *2015 International Conference on Industrial Engineering and Systems Management (IESM)*, pages 12–18, October 2015.

[23] Zakaryae Boudi, El Miloudi El Koursi, and Simon Collart-Dutilleul. HCPN Modeling for ERTMS Requirements Specification. In *2015 IEEE 16th International Symposium on High Assurance Systems Engineering*, pages 271–272, January 2015. ISSN: 1530-2059.

[24] Zakaryae Boudi, El Miloudi, El Koursi, and Simon Collart-Dutilleul. From Place/Transition Petri nets to B abstract machines for safety critical systems. *IFAC-PapersOnLine*, 48(21):332–338, January 2015.

[25] Zakaryae Boudi, El Miloudi El Koursi, and Simon Collart-Dutilleul. An HCPN Pattern for Railway Safety Critical Scenarios Formal Modeling. American Society of Mechanical Engineers Digital Collection, June 2015.

[26] Zakaryae BOUDI, El Miloudi El Koursi, and Mohamed Ghazel. The New Challenges of Rail Security. *Journal of Traffic and Logistics Engineering*, page 5p, January 2016.

[27] Zakaryae Boudi, Rahma Ben-Ayed, El Miloudi El Koursi, Simon Collart-Dutilleul, Thomas Nolasco, and Mohamed Haloua. A CPN/B method transformation framework for railway safety rules formal validation. *European Transport Research Review*, 9(2):13, June 2017.

[28] Zakaryae Boudi, Abderrahim Ait Wakrime, Simon Collart-Dutilleul, and Mohamed Haloua. Petri Nets to Event-B: Handling Mathematical Sequences Through an ERTMS L3 Case. In El Hassan Abdelwahed, Ladjel Bellatreche, Djamal Benslimane, Matteo Golfarelli, Stéphane Jean, Dominique Mery, Kazumi Nakamatsu, and Carlos Ordonez, editors, *New Trends in Model and Data Engineering*, Communications in Computer and Information Science, pages 50–62, Cham, 2018. Springer International Publishing.

[29] Zakaryae Boudi, Abderrahim Ait Wakrime, Simon Collart-Dutilleul, and Mohamed Haloua. Introducing B-Sequenced Petri Nets as a CPN Sub-class for Safe Train Control:. In *Proceedings of the 14th International Conference on Evaluation of Novel Approaches to Software Engineering*, pages 350–358, Heraklion, Crete, Greece, 2019. SCITEPRESS - Science and Technology Publications.

[30] Alain Colmerauer and Philippe Roussel. The birth of Prolog. In *History of programming languages—II*, pages 331–367. Association for Computing Machinery, New York, NY, USA, January 1996.

[31] Nick Drummond, Matthew Horridge, Robert Stevens, Chris Wroe, and Sandra Sampaio. Pizza ontology. *The University of Manchester*, 2, 2007.

[32] Fernando CN Pereira and David HD Warren. Definite clause grammars for language analysis—a survey of the formalism and a comparison with augmented transition networks. *Artificial intelligence*, 13(3):231–278, 1980. Publisher: Elsevier.

[33] Holger Knublauch, Matthew Horridge, Mark A. Musen, Alan L. Rector, Robert Stevens, Nick Drummond, Phillip W. Lord, Natalya Fridman Noy, Julian Seidenberg, and Hai Wang. The Protege OWL Experience. In *OWLED*, 2005.

[34] OWL Syntax Converter, http://mowl-power.cs.man.ac.uk:8080/converter/.

```
'Class'(X) --> [X].

'Declaration'('Class'(X)) --> 'Class'(X),[' <: Thing'].
```

Listing 1. Type 1 DCG rules: Swi-Prolog code snippet

```
'Class'(X) --> [X].

'SubClassOf'('Class'(X),'Class'(Y)) --> 'Class'(X),[' <: '],'Class'(Y).
'SubClassOf'('Class'(X),AxiomY) --> 'Class'(X),[' <: ('],AxiomY,[')'].
'SubClassOf'(AxiomX,'Class'(Y)) --> ['('],AxiomX,[') <: '],'Class'(Y).
'SubClassOf'(AxiomX,AxiomY) --> ['('],AxiomX,[') <: ('],AxiomY,[')'].

'ObjectIntersectionOf'('Class'(X),'Class'(Y)) --> 'Class'(X),[' /\\ '],'Class'(Y).
'ObjectIntersectionOf'('Class'(X),AxiomY) --> 'Class'(X),[' /\\ ('], AxiomY,[')'].
'ObjectIntersectionOf'(AxiomX,'Class'(Y)) --> ['('],AxiomX,[') /\\ '],'Class'(Y).
'ObjectIntersectionOf'(AxiomX,AxiomY) --> ['('],AxiomX,[') /\\ ('],AxiomY,[')'].
```

Listing 2. Type 2 DCG rules: Swi-Prolog code snippet

```
'Class'(X) --> [X].

'ObjectUnionOf'('Class'(X),'Class'(Y)) --> 'Class'(X),[' <: '],'Class'(Y).
'ObjectUnionOf'('Class'(X),AxiomY) --> 'Class'(X),[' <: ('], AxiomY,[')'].
'ObjectUnionOf'(AxiomX,'Class'(Y)) --> ['('],AxiomX,[') <: '], 'Class'(Y).
'ObjectUnionOf'(AxiomX,AxiomY) --> ['('],AxiomX,[') <: ('],AxiomY,[')'].

'ObjectUnionOf'([])--> [].
'ObjectUnionOf'([X])--> X.
'ObjectUnionOf'([X,Y])--> 'ObjectUnionOf'(X,Y).
'ObjectUnionOf'([X,Y|Tail])--> ['('],'ObjectUnionOf'(X,Y),[') \\/ ('],'ObjectUnionOf'(Tail),[')'].
```

Listing 3. Type 3 DCG rules: Swi-Prolog code snippet

```
dcg('Declaration'('ObjectProperty'(R)),AM,T):-propDom(T,R,Dom),propRan(T,R,Ran),
    phrase(property(R,Dom,Ran),AM),!.

'Declaration'('ObjectProperty'(R))-->R,[' : Thing <-> Thing'],!.

propDom([],_,_):-!.
propDom(['ObjectPropertyDomain'('ObjectProperty'(R),Dom)|_],R,Dom).
propDom(['DataPropertyDomain'('DataProperty'(R),Dom)|_],R,Dom).
propDom([_|T],R,Dom):-propDom(T,R,Dom).

propRan([],_,_):-!.
propRan(['ObjectPropertyRange'('ObjectProperty'(R),Ran)|_],R,Ran):-!.
propRan(['DataPropertyRange'('DataProperty'(R),Ran)|_],R,Ran):-!.
propRan([_|T],R,Ran):-propRan(T,R,Ran).


property(R,[],[])-->'Declaration'('ObjectProperty'(R)).
property(R,Dom,[])-->R,[' : '],Dom,[' <-> Thing'],!.
property(R,[],Ran)-->R,[' : '],['Thing <-> '],Ran,!.
property(R,Dom,Ran)-->R,[' : '],Dom,[' <-> '],Ran,!.
```

Listing 4. ObjectProperty DCG rules: Swi-Prolog code snippet