Science of Computer Programming 121 (2016) 100–127

Contents lists available at ScienceDirect

Science of Computer Programming

www.elsevier.com/locate/scico

Making explicit domain knowledge in formal system development ☆

Yamine Ait-Ameur [a], Dominique Méry [b]

[a] IRIT, INPT-ENSEEIHT, France
[b] LORIA, Université de Lorraine, France

ARTICLE INFO

Article history:
Received 15 April 2015
Received in revised form 7 December 2015
Accepted 10 December 2015
Available online 17 December 2015

Keywords:
System design models
Explicit vs. implicit semantics
Ontologies and ontology engineering
Models verification and validation
Domain knowledge

ABSTRACT

Modeling languages are concerned with providing techniques and tool support for the design, synthesis and analysis of the models resulting from a given modeling activity, this activity being usually part of a system development model or process. These languages quite successfully focused on the analysis of the designed system exploiting the expressed semantic power of the underlying modeling language. The semantics of these modeling languages are well understood by the system designers and/or the modeling language users i.e. implicit semantics.

In general, modeling languages are not equipped with resources, concepts or entities handling explicitly domain engineering features and characteristics (domain knowledge) in which the modeled systems evolve.

Indeed, the designer has to explicitly handle the knowledge issued and/or mined from an analysis of this application domain i.e. explicit semantics. Nowadays, making explicit the domain knowledge inside system design models does not obey to any methodological rule validated by the practice. The modeling languages users introduce through types, constraints, profiles, etc. these domain knowledge features.

Our claim is that ontologies are good candidates for handling explicit domain knowledge. They define domain theories and provide resources for uniquely identifying domain knowledge concepts. Therefore, allowing models to make references to ontologies is a modular solution for models to explicitly handle domain knowledge.

Overcoming the absence of explicit semantics expression in the modeling languages used to specify systems models will increase the robustness of the designed system models. Indeed, the axioms and theorems resulting from the ontologies, thanks to references, can be used to strengthen the properties of the designed models.

The objective of this paper is to offer rigorous mechanisms for handling domain knowledge in design models. This paper also shows how these mechanisms are set up in the cases of static, dynamic and formal models.

☆ This work was supported by grant ANR-13-INSE-0001 (The IMPEX Project http://impex.gforge.inria.fr) from the Agence Nationale de la Recherche (ANR).
E-mail addresses: yamine@enseeiht.fr (Y. Ait-Ameur), dominique.mery@loria.fr (D. Méry).
URLs: http://yamine.perso.enseeiht.fr/ (Y. Ait-Ameur), http://www.loria.fr/~mery/ (D. Méry).

http://dx.doi.org/10.1016/j.scico.2015.12.004
0167-6423/

1. Introduction

Problem statement A lot of efforts have been devoted to the study of ontologies and their applications in the area of semantic web and information retrieval. In these areas, studied objects are usually corpuses of documents like texts, videos, or images with *weak* design models (characters, signals, pixels) on which it is very hard to perform accurate and rich analyses. Several approaches for describing, designing, formalizing ontologies for these application domains have been produced by many authors. Moreover, these approaches focused on establishing formal links between these ontologies and the studied domain objects. The majority of these approaches paid a lot of attention to the use of ontologies in order to make explicit the semantics of these objects and interpret them. In these approaches, the objectives are to allow users 1) to interpret and to understand what is the semantics carried out by these objects, and 2) to query these corpuses of documents and extract relevant information. Here, terms and patterns are fundamental to perform such analyses and reasoning capabilities offered by the ontologies play an important role.

In system engineering, design models are richer than the ones associated with texts, images or videos. Engineers define, using a modeling language, complex design models. Several design models may be defined for the same system. They perform different analyses of these models according to the modeling language and techniques they are using.

Although engineers use complex design descriptions to design their models, they still miss some relevant information related to the domain of interest. In most of the cases, this is due to

1. the fact that engineers use the available modeling languages to hard encode, on the fly, specific domain properties with their own point of view, using the semantics of the available modeling languages (implicit semantics). Such a process may lead to incomplete and/or inconsistent descriptions, especially in the case of model composition, decomposition, abstraction and/or refinement;
2. and to the absence of resources allowing an engineer to make explicit the domain knowledge of interest (explicit semantics). The main problem is related to the absence, in the design modeling languages, of resources supporting such knowledge domain descriptions.

Objectives The objective of this paper is to show that domain knowledge formalized by ontologies on the one hand and design models on the other hand can be integrated. Once this integration is realized, new properties issued from the domain knowledge enrich the design models and improve their verification and validation. This integration has the merit to strengthen design models.

Our approach advocates the use of domain ontologies in order to make explicit the domain knowledge and to enrich design models with relevant domain properties expressed in these domain ontologies. For this purpose, we propose to formalize links between domain ontologies and design models using annotations.

Our contribution In this paper, we give an overview of ontologies when used for in other domains than semantic web. Indeed, we study domain ontologies for modeling specific domains particularly in system engineering domain modeling with a specific focus on engineering. By focus on engineering, we mean the use of ontologies to increase the quality of formal design models developed for system engineering.

We report on the use of domain knowledge models expressed by ontologies to annotate design models. By annotation, we want the definition of specific mechanisms to make explicit domain knowledge in design models. The proposed approach shall show how domain knowledge information can be exploited by the design models in order increase their quality and expressiveness. Axioms and theorems carried out by ontologies, modeling knowledge domains, shall be exploited by system design models. We illustrate the defined annotation-based approach on two different cases of formal modeling.

1. First, we consider the comparison of behavioral models expressed by labelled transition systems (*lts*). Traditional techniques use simulation or bi-simulation relationships [1] to handle such comparisons of *lts* sharing a same set of labels. The need to compare *lts* with different sets of labels may occur in different situations like semantic-web services [2–4] or plastic human computer-interfaces [5,6]. Relations borrowed from ontologies are used to rewrite labels in order to obtain *lts* with the same set of labels. Bi-simulation properties can be checked on the obtained *lts*.
2. The second situation is illustrated by proof and refinement based formal methods. We show how the Event-B [7] method handles the definition and the exploitation of ontologies in order to enrich classical models with ontological invariants described with domain ontologies. New properties issued from domain knowledge expression which enrich the Event-B models become verifiable.

Structure of the paper This paper is structured as follows. Next section discusses semantic aspects and gives our interpretation of making explicit domain knowledge in design models. It discusses *implicit* and *explicit* semantics. Section 3 overviews the main characteristics of ontologies when defined for characterizing engineering knowledge domains. We focus on the definition and on the use of the ontologies in system and software design. Then, section 4 discusses the existing separation of domain models and design models and the need to integrate them when designing a system. Our methodology to handle the semantic of a domain expressed by an ontology in a design model is presented in section 5. The use of formal methods is discussed. Finally, sections 6 and 7 are devoted to the deployment of our methodology in the case of two formal

methods: model checking on labelled transition systems and proof and refinement-based formal methods. Last section gives conclusion and future research directions.

2. Implicit versus explicit

In general, the definition of a modeling language is associated to the notion of formal system which defines three main elements.

1. A *syntax* giving the syntactic representation of the language constructs.
2. A *semantics* given by a satisfaction relationship, usually denoted by $\models$ expressing the possible interpretations of the syntactic constructs expressed within the modeling language. This relationship describes the logical notions of interpretation and model.
3. A *proof system* defined by an entailment relationship denoted by $\vdash$ expressing the capability to deduce new theorems from axioms and already proved theorems.

Modeling languages are used for engineering *software and/or systems* and it is necessary to have a clear correct and complete meaning of the adjectives *implicit* and *explicit*, in a formal framework.

2.1. Implicit semantics

When considering the term *implicit* in modeling languages (for instance, Event-B [8], or labelled transition systems [9]), we refer to underlying semantic as well as pragmatic *features*, which are expressing the semantics of modeling elements (sets, axioms, data structures, state-variables, safety properties, events, ...). The understanding of those underlying features becomes *tacit* through the learning of the modeling language, which is supported by specific consistent semantics-based techniques and pragmatic tools (for instance, Rodin [10] is an environment for developing Event-B models, or the CADP toolbox [11] equipped with a model checker for analyzing automata-based models).

2.2. Explicit semantics

In general usage, *explicit* means *fully revealed* or *expressed without ambiguity*, whilst *implicit* means *implied* or *expressed indirectly* or *tacit*. The meaning of these two adjectives is related to semantical features in a current context and it should be noted that there is some inconsistency, within the computer science and software engineering communities, regarding the precise meaning of these adjectives. For example, in logic and belief models [12] *a sentence is explicitly believed when it is actively held to be true by an agent and implicitly believed when it follows from what is believed*. However, in the semantic web [13] or in system engineering, *Semantics can be implicit, existing only in the minds of the humans [...]. They can also be explicit and informal, or they can be formal.* The Explicit Semantic Analysis (ESA) [14] interprets semantics of unrestricted natural language texts and *represents meaning in a high-dimensional space of concepts derived from Wikipedia, the largest encyclopedia in existence*. The meaning of any text is *explicitly* represented *in terms of Wikipedia-based concepts*. The requirements engineering community uses the terms to distinguish between declarative (descriptive) and operational (prescriptive) requirements [15], where they acknowledge the need for *a formal method for generating explicit, declarative, type-level requirements from operational, instance-level scenarios in which such requirements are implicit*.

2.3. Separation of concerns

Nowadays, several research projects and approaches [16–18] aim at formalizing mathematical theories that are applicable in the formal developments of systems. These theories are helpful for building complex formalizations, expressing and reusing proof of properties. The need for handling domain knowledge has been identified so far by the software engineering community [19–21]. Following the *Domain Engineering Dogma*, D. Bjørner and A. Eir [22] emphasize the role of *understanding the domain*, when designing a software system. They describe structures using an ontology aspect. An *ontology consists of entities of four kinds of specification types: simple entities, operations, events and behaviors*. Usually, these theories are defined within contexts, that are imported and and/or instantiated. They usually represent the implicit semantics of the systems and are expressed by types, logics, algebras, etc. based approaches. Domain engineering [16–18] adequately addresses the formal description of domains expressing the semantics of the universe in which the developed systems run and their integration in the formal development process. However, this domain information is usually defined in an *ontology* [23].

Several relevant properties — safety, security, liveness, for instance — are checked using formal methods. These properties are defined in the *implicit* semantics associated to the formal technique being used. When considering these properties in their context with the associated explicit semantics, these properties may be no longer respected or may have a different *meaning*. According to D. Bjørner and A. Eir [22], *... The domain description, ..., is (best)expressed when both informally narrated and formally specified*. The formalization is based on a list of *explicit* properties defined as *axioms*. The identification of *properties* as *axioms* is a crucial step when dealing with domain engineering, since it is based on engineering knowledges and expertises.

We believe that *without a more formal software and/or system engineering development approach, based on separation of implicit and explicit semantics*, the composition of software and/or system components in common contexts risks compromising correct operation of the resulting system. This is a significant problem when software and/or systems are constructed from heterogeneous components [24] that must be reliable in unreliable contexts [25].

2.4. Need for integrating the implicit and explicit concepts

We are concerned with the separation of concerns when reasoning about properties of models. Although the concerns need to be cleanly separated, the models need to be tightly integrated: achieving both is a significant challenge.

Allowing formal methods users and developers to integrate — in a flexible and modular manner — both the implicit semantics, offered by the formal method semantics, and the explicit semantics, provided by external formal knowledge models like ontologies, is a major challenge. Indeed, the formal models should be defined in the formal modeling language being used, and explicit reference and/or annotation mechanisms must be provided for associating explicit semantics to the formal modeling concepts. Once this integration is realized, the formalization and verification of several properties related to the integration of heterogeneous models becomes possible. The most important properties that need to be addressed relate to interoperability, adaptability, dissimilarity, re-configurability and identification of degraded modes. Refinement/instantiation and composition/decomposition could play a major role for specifying and verifying these properties. Currently, no formal method nor formal technique provides explicit means for handling such an integration.

In the context of formal methods, it is well known that several formal methods for system design and verification have been proposed. These techniques are well established on a solid formal basis and their domain of efficiency, their strengths and weaknesses are well acknowledged by the formal methods community. Although, some ad-hoc formalization of domain knowledge [16–18] within formal methods is possible, none of these techniques offers a built-in mechanism for handling explicit semantics.

Regarding ontologies and domain modeling, most of the work has been achieved in the large semantic web, information retrieval, natural language processing, ... research communities. There, the problem consists of annotating web pages and/or documents with semantic information available in ontologies. Thus, ontologies have mainly been used for assigning meanings and semantics to terms occurring in documents. Once, these meanings are assigned, formal reasoning can be performed on the ontology side due to their formal descriptive logics basis. In general, however, the documents to be annotated do not conform to any model (or conform to weak models, usually based on XML descriptions expressing document structures) and the domain associated to the documents is not fixed. Therefore, ontologies behave like a model associated to the resources that are annotated. As a consequence, ontologies, as a model, avoids implicit semantics alignment between the semantics of the ontology and of the annotated resources modeling languages.

We propose *to integrate both worlds*. On the one hand, formal methods facilitate prescriptive modeling whereas, on the other hand, ontologies provide mechanisms for explicit descriptive semantics. We conclude by noting that, in most cases, the formal models are usually defined in a fixed and limited application domain well understood by the developers and in main case studies, expertises in a given domain as avionics, transportations, medecine or energy, ... are required for providing a clear, correct and complete model of the system under engineering.

3. Ontologies are good candidates for domain knowledge modeling

As mentioned above, a lot of efforts have been devoted to the study of ontologies and their applications in the area of semantic web and information retrieval. Several approaches for describing, designing and formalizing ontologies for these application domains have been proposed by many authors. Models [26–31], browsers like Protégé[1] [32,33] or PlibEditor,[2] reasoners [34–37], annotators [38,39], XML-based translators [40,41] have been developed to engineer such ontologies and establish formal links with the studied domain objects like texts, images, videos, signals, ... Most of these approaches paid a lot of attention to the use of ontologies to interpret these objects and/or provide classifications of these interpreted objects. Next subsection recalls basic definitions of ontologies and their characteristics. It gives our view of domain ontologies and their characteristics with a specific focus on system engineering.

3.1. Domain modeling: ontologies

3.1.1. Definition

Gruber defines an ontology as *an explicit specification of a conceptualization* [23]. Another definition relies on the notion of dictionary, where [42] considers a domain ontology as a *formal and consensual dictionnary of categories and properties of entities of a domain and the relationships that hold among them*. Here, an entity represents any concept belonging to the considered domain. The term *dictionary* entails two major concepts. First, it makes explicit the existence, through a constructive definition or declaration, of entities in the domain under consideration and second that any entity or relationship described in this

[1] http://protege.stanford.edu/.
[2] http://www.plib.ensma.fr/.

ontology is directly referencable, for any purpose and from any context, independently of the other entities or relationships. Reference is carried by a symbol defining an identifier. This identification symbol may be either a language-independent identifier, or a language-specific set of words. However, whatever this symbol is, and unlike in linguistic dictionary, it directly denotes a domain entity or relationship. Each *description* of each entity or relationship is formally *stated* using an ontology modeling language equipped with a formal semantics. It allows automatic reasoning and consistency checking.

3.1.2. Some fundamental characteristics

A domain ontology is an explicit conceptualization of domain entities and relationships [23]. Ontology definitions will fulfil three fundamental criteria [42].

1. **Formality.** An ontology is a conceptualization expressed within a modelling language. It has an underlying formal semantics and may offer reasoning.
   Similar to modeling languages, the semantics of ontology modeling languages is expressed using a satisfaction and an entailement relationships. If an ontology *Ont* is defined in the ontology modelling language O, then
   - the satisfaction relationship ($\models_O$) offers interpretation capabilities. It is usefull to check whether an instance or an individual of a given concept of the ontology *Ont* belongs to the model (set of all its instances) defined by the considered ontology;
   - the entailment relationship ($\vdash_O$) handles proofs. Entailment expresses the sequence of deductions (sequents) that leads to a given theorem (expressed as a statement in the ontology modelling language O) can be deduced from axioms and theorems defined by the ontology *Ont*.

   Automatic or semi-automatic reasoning techniques are associated to an ontology modelling language O. They support instance checking ($\models_O$) and reasoning ($\vdash_O$). As a consequence, checking properties expressed over the ontology-defined concepts and individuals, becomes possible, thanks to the associated theory, either by automatic or semi-automatic reasoning techniques.
2. **Consensuality.** Agreement on the conceptualization defined by an ontology needs to be reached for a large community of users. This community is not restricted to users or to developers of a specific application: it gathers all the potential users and developers of other applications related to the conceptualized domain. Consequently, an ontology will be shared by several applications and design models. For example, ISO 13584-compliant (PLIB) [31,30] product ontologies are defined according to a formal standardization process. They are published as ISO and/or IEC international standards. This criterion excludes conceptual models defined for a specific application.
3. **Capability to be referenced.** As stated in the previous definition, each concept defined in an ontology is associated to an identifier provided to allow applications to refer this concept from any environment. Moreover, this concept can be referenced whatever is the ontology model set up to describe this concept.

3.2. Ontology models

Several ontology modeling languages were developed during last ten years, as for example, Ontolingua [43] for artificial intelligence applications, DAML+OIL [26], RDFS [27] and OWL [28,29] for Web applications, and PLIB [30,31] for engineering applications.

In [44], authors have identified some relevant characteristics associated to ontology modeling languages. These characteristics have been extended with new ones in order to handle the system engineering aspects.

- **Words and concepts.** Ontology models offer the capability to describe words and concepts. Various relationships are offered by these languages: between words, between concepts and between words and concepts. The presence of such relationships leads to two ontologies design processes [42]: from words to concepts (e.g. semantic web) or from concepts to words (e.g. engineering catalogues).
- **Strong typing.** Ontology modeling languages are equipped with a type system characterizing classes, properties, relationships and domain values. According to the modeling language, this type system may be more or less a strong type system. For example, the PLIB ontology modeling language uses a strong typing system (e.g. unit types are built-in types) while description logics do not require strong typing. Types are useful for indexation, and thus for the definition of persistent frameworks like semantic databases [45–50] to sore both ontologies and their instances.
- **Algebraic operators** may be associated to the types available in the ontology language. Operators on classes like union, intersection, etc. are available in most of the ontology modeling languages. For example, operators on reals, are available in the PLIB ontology model. They make it possible to express property derivation (e.g. diameter equals two times a ray). These defined algebraic operators define abstract data types and give complete definition of the data types discussed above.
- **Constraints description** constructs are offered by the ontology modeling language to define constraints on classes, properties or on whole ontology. In the engineering domain, the more the constraint description language is rich, the more the expressed concepts of the ontologies are precisely described. Checking these constraints depend on the used constraint solving techniques associated to the ontology modeling language.

- **CWA vs. OWA.** Closed world assumption (CWA) implies that a complete knowledge is known and, if a fact is not a consequence of the ontology model, then its negation is, while in open world assumption (OWA) this reasoning is no longer available. In general, CWA is used in system engineering, while semantic web considers OWA.
- **Context modeling.** In the engineering domain, the context in which a property is defined is important [51]. At the ontology level, the domain of a context dependent property is not only its class, but it is also a context description (usually a class). For example, the definition of the lifetime (property) of a tyre (class) depends on the average temperature of use (context of use). Note that PLIB offers built-in constructs for such properties.
- **Inheritance and instantiation.** Classes may be linked by single or multiple inheritance relationships. Inheritance helps to factorize objects with the same properties, it also contributes to the definition of the subsumption relationship. Instances of a class represent the individuals, and an individual may belong to a single class (mono-instantiation) or to several classes (multi-instantiation). Ontology modeling languages offer different forms of inheritance and instantiation. For example OWL supports multiple inheritance and multi-instantiation while PLIB supports single inheritance and mono-instantiation.
- **Reasoning.** In ontology engineering, reasoning essentially concerns subsumption (e.g. to link ontologies classes in case of integration), class membership checking (e.g. for migration of instances from one ontology class to another one) and classification (e.g. to build new class hierarchies according to some criteria). Other logical aspects of reasoning concern the specific properties of the underlying logic like symmetry, reflexivity, equivalence etc are useful for knowledge inference. Different reasoning techniques and tools have been defined. One can cite [34–37], running in central memory have been defined. Like for model checking, these approaches may face the problems of memory saturation or space exploration. Other reasoning technique more commonly used by formal methods have also been set up to handle proof of properties in ontologies. These approaches, which proved scalable, use of theorem provers like COQ with [52,53] or Event-B [24] to infer ontologies properties.
- **Exchange formats**. All the ontology modeling languages offer exchange formats based on the XML language. When expressed in this exchange format, classes and their instances can be interpreted in the in different contexts of use.

3.3. Ontologies in engineering

Our work does not address semantic web applications. In our study of design models, we have been involved in the engineering area. We focus on domain ontologies where the whole knowledge on the domain is described in the provided ontology. Due to the system engineering targeted application domain, we use ontologies conforming to the PLIB ontology model [54–57]. This ontology model advocates the use of strong typing with a rich type system (similar to the one of a programming language more specific types like units), property derivation with algebraic operators corresponding to the defined types, first order logic and set theory as a constraint language, CWA and context dependent properties.

Like in usual engineering practices and unlike OWL, additional models may be added to a technical object description. Indeed, a set of different functional models, each one representing a particular discipline-specific representation (e.g., safety, real time, energy consumption, geometry procurement, simulation, etc.) can be associated to a given technical object described within the PLIB ontology model.

Finally, a number of domain ontologies based on this model already exist. Examples are the ISO 13584 and ISO 15926 (e.g. mechanical fasteners, measure instruments, cutting tools) and IEC 61360 (e.g. electronic components, process instruments) series of ontologies developed within international standardization organizations (e.g. ISO, IEC) or national ones (e.g. JEMIMA,[3] CNIS[4]) that cover progressively all the technical domain.

3.4. Ontologies and changes

Ontologies may evolve in time. The *continuous ontological principle* requires that any evolution does not contradict the previous ontology version. Several evolution scenarios fulfilling this principle have been defined like

- *extending by subsumption a concept* by another one. A new concept is subsumed by an existing one (inheritance),
- *defining new equivalent concepts* where new concepts declared as equivalent to another one is introduced,
- *introducing new properties* by adding new attributes to given concepts,
- ...

After evolutions, new ontology versions are issued. The versioning mechanisms define a version management protocol with a current reference version and old versions are preserved. Specific configuration attributes are associated to each ontology concept in order to record the different evolutions. Finally, suppression of concepts is not allowed in order to preserve the reference to ontologies concepts mechanism (annotation for example). In case of suppression, obsolete concepts are introduced and the associated configuration management attribute is set to the obsolete value.

[3] Japan Electric Measuring Instruments Manufacturers Association.
[4] Chinese Institute for Standardization.

4. System modeling versus domain modeling

The previous section has presented an overview of the fundamental characteristics of ontologies and ontology modeling languages. We have shown that different ontology languages can be set up to describe domain ontologies. Application domains, semantics, constraints expressiveness, reasoning capabilities, assumption on the universe of discourse etc. are some of the characteristics to be assessed before designing a domain ontology. Ontologies are used to describe domain models. This section is devoted to the comparison of domain models and design models.

4.1. System modeling: design models

Design models address the definition of models of systems to be realized. They are described within modeling languages and they correspond to abstractions of the considered system. Semantics of the modeling language is given by a satisfaction relationship ($\models_M$) checking if a model is satisfiable and an entailment relationship ($\vdash_M$) defining a proof system where properties can be proved from axioms, theorems and proof rules applications). Some examples of such formal modeling languages are [8,58–61]. Obviously, other modelling languages are available, and we can add all the languages derived from the UML modelling language [62,63].

Various analyses, properties checking, models manipulations, etc. are performed on such models depending on the provided modeling language, its semantics, its associated verification procedure and on the abstraction level where models are defined. As a consequence, different models of a same system may be produced along the design process. These models may record abstract or concrete levels (top–down or bottom–up design approaches) or they may, at a given design step, describe different models corresponding to different views of a the system under design. Thus, several heterogeneous models expressed with different modeling languages are produced. This heterogeneity may lead to ambiguities in the interpretation of the system characteristics and/or behavior in each produced design model.

4.2. Domain modeling: ontologies versus system modeling: design models

In [42], authors have proposed a comparison of ontologies and design models. Modeling languages are required to express both ontologies and design models. These modeling languages offer different verification techniques according to their semantics. As such, both ontologies and design models are models. They define a conceptualization of a part of the subject concerned by through models defined within modeling languages. So, one may guess that from this point of view, ontologies and design models are similar, since they share a common goal, namely modeling. However, if we consider the three criteria identified above in section 3.1.2, we can identify some significant differences.

Design models are equipped with formal semantics In this sense, they fit the *formality* criterion. They are grounded on rigorously defined semantics and associated to property verification systems which use reasoning and logical theories. However, according to [42], design models are closely related to the system under design. In other words, a design model *prescribes* and *enforces* which system characteristics *will* be available in the model to perform a specific analysis or treatment. Indeed, as mentioned above, a single system may have different design models (safety oriented model, real time model, energy consumption model and an architectural model for example). From this observation, we can conclude that the *consensuality* criterion is not (or partly) fulfilled by design models. Finally, if we consider naming processes in design models and design modeling languages, there is no rule requiring a single identification of entities (variables, constants, states, events, etc.) manipulated by design models. A typical example of such a naming rule relates to the description of point coordinates, where a pair of floats (v_1, v_2) may be interpreted differently in a model referring to polar coordinates (r, θ) and in another model referring to cartesian coordinates (x, y). Engineering abounds in such examples. The entity identification is unique in the context of the considered design model, but it may be used with a different semantics in another model. So, the *capability to be referenced* criterion is not fulfilled by design models.

Previously identified differences do not constitute a drawback. The simultaneous use of both ontologies and design models in an engineering context, strengthens the modeling processes.

The subject of this paper is not to oppose ontologies and design models. The previously identified differences advocate for the use of both ontologies and design models in a single framework. Ontologies carry relevant information usually handled implicitly in design models. For example, the previously defined points can be defined either in an absolute, or in a relative coordinate system. When ontologies and design models are integrated (or composed), domain properties may be explicitly used in the design models and in the associated system development processes.

4.3. Ontologies and annotations

One of the main use of ontologies is annotation. Let us consider a set of entities available in a given corpus. These entities may be words or sentences in a document, images or videos, entities of a design model, etc. By annotation, we denote the link that may exist between an ontology concept (class, instance, property, etc.) and an entity of the considered corpus. The annotation process consists in defining and running a set of rules leading to the production of annotations. This process may be completely automated, semi-automatic with user validation or completely interactive. Automatic annotation

proved powerful in the area of the semantic web and natural language processing where the entities of the corpus are words appearing in texts. Several tools (or annotators) have been developed for various ontologies and different natural languages [64,65,38,39,66]. Other approaches to annotate images and multi-media documents have also been developed [67].

In the area of system design, the objective of model annotation is to increase model interoperability. Consensual domain ontologies are shared by different system models corresponding to different engineering views. Annotations allow the designer to link different entities of different system models to ontology concepts. Reasoning at the ontology level makes it possible to check some domain properties. Model annotations have been produced using semi-automatic and/or interactive approaches. Automatic annotation is not recommended in such application domains. For example, model annotations have been produced for product life management (PLM) models in [68], for petroleum engineering models in [69,70] or for aircraft system modeling in [71]. All these examples used controlled annotation techniques being either semi-automatic or interactive.

4.4. Semantic mismatch

The separation of concerns principle followed when designing systems that exploit domain models expressed by ontologies may lead to semantic heterogeneity. Indeed, when system design models and ontologies are modeled with different modeling languages, this can lead to different semantic interpretations. Moreover, the capability to verify the relevant properties corresponding to system requirement depends on the proof system associated to the modeling language. This problem can be expressed in a more formal way as follows. When an ontology modeling language (with its own satisfaction $\models_O$ and entailment $\vdash_O$ relationships) is used to describe the domain model associated to the design of a given system which is expressed in a design modeling language (with its own satisfaction $\models_M$ and entailment $\vdash_M$ relationships) there is a need of *semantic alignment* due to different semantics of these two modeling languages. This topic is outside the scope of this paper.

5. Embedding ontologies into system design formal models

5.1. What if the ontology and the design model were linked?

Usually, design models do not handle, in an explicit manner, the knowledge of the application domain or context where models are designed. Therefore, some useful properties available in the domain knowledge are not considered by the design models, more, these properties could be violated by the design model. For instance, the nose gear velocity system measures the velocity on the ground of an aircraft. To do so, it uses a 16-bit register variable to store the number of cycles of the wheels (to compute the distance travelled and the aircraft speed). Therefore, it is important to *know* the maximal length of the runway on earth in order to check that the chosen size of the register variable (16-bit) is correct. In terms of system engineering, the determined size of this register comes from flight mechanics, more precisely from the lift of the aircraft. Without an explicit definition of this knowledge, engineers would not be able to set up such a value for the register size.

So, linking knowledge domains, expressed by ontologies, with the design models strengthens the designed models and support more verifications, since the properties expressed in the ontologies will be part of the designed models. Moreover, thanks to the capability to reference domain entities, it becomes possible to avoid ambiguous definitions of the same entity in two different models.

Model annotation is the mechanism classically set up to link domain ontologies with design models. It consists in defining specific relationships to relate ontology concepts with models entities. The annotation mechanism extends the one that has been defined by the semantic web community to annotate web pages [64,65,38,39,66] or images [67].

5.2. Modeling languages

Different modeling languages may be used for building both ontologies, design models and annotation models. These languages may have the same semantics and verification procedure, but these may be different. Two situations occur. First, as mentioned in section 4.4, the semantics and the verification procedure can be expressed in a single modeling language. In this case, there is no semantic mismatch and both design models, ontologies and annotation can be formalized in the same modeling language. The second option considers different semantics of both modeling languages. This case is out of the scope of this paper, it requires semantic alignment. Several approaches to align semantics have proposed in the literature, they are based on the definition of institutions as models [72–74].

5.3. Our approach

Our approach advocates the exploitation of domain knowledge, carried out by ontologies, in design models. We propose a stepwise methodology, composed of four steps, to establish a formal link between these two models. The approach is based on the definition of an annotation mechanism that represents this link. The definition of this mechanism depends on the used modeling languages for both ontologies and design models. Fig. 1 shows the overall schema of the approach.

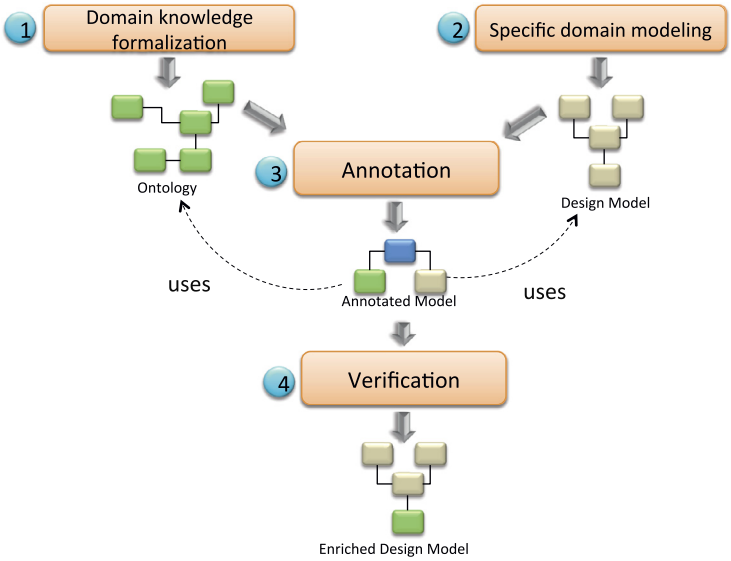

Fig. 1. A four steps methodology for integrating domain knowledge and design models.

1. **Formalization of domain knowledge.** Domain information are formalized in an ontology modeling language. Concepts, entities, relationships, constraints, rules, etc. are explicitly defined. The result is a formal ontology expressed in the chosen ontology modeling language. The semantics of this language and the associated verification techniques are used to establish properties of the ontology. The expressive power of this language has an impact on the defined ontologies (e.g. different constraint description languages may be used). Finally, the ontology shall be defined independently of any context of use. It may also be built from already existing ontologies (e.g. standard ontologies).
2. **Definition of design models.** Any formal modeling language is used to describe design models. Within this formal modeling language, users define and formalize specific design models corresponding to a given specification. Different analyses allowed by the modeling language and its associated verification technique may be performed on the designed model.
3. **Annotation of design model by references to ontologies.** Using specific mechanisms available in the formal modeling language, annotation of design models are explicitly described. Annotation consists of defining specific relationships between design models entities and ontology concepts. Different relationships are available, they have their specific properties. For example, the *Is_a* relationship can be used to assert that a given design model entity *is a*n ontology concept (annotation by *subsumption*). Annotation is made explicit in the design models thanks to the use of these relationships.
4. **Expression and verification of properties.** Once design models are annotated by domain ontologies, the proof context of the design models is enriched by the domain properties expressed in the ontology. It becomes possible to check on the one side the consistence of the design models already established before annotation (they may be no longer correct after annotation) and on the other hand other properties that emerged after annotation.

At the end of this process, a new design model enriched with new information of the domain knowledge is obtained. This model makes an explicit representation of domain concepts and properties borrowed from the ontology thanks to the annotation.

5.4. Some comments

1. It is important that the specified and used ontologies are defined in a consensual manner by the stakeholders involved in the system under design. Moreover, they should have relationships with the domain of the design model.
2. Steps 1 and 2 of the previous methodology are independent. They may be run in parallel. Ontologies may be defined prior to the design model or they may preexist.
3. In the semantic web area, lot of efforts are devoted to the definition of automatic annotation mechanisms [64,65,38,39,66,67]. There the annotated models are documents in general and ontologies usually exploit terms rather than concepts. The definition of the annotations may be realized either by manual, semi-automatic or automatic processes [68–71,24]. In this paper, we are concerned with formal design models targeting system design. Therefore, our approach relies on an interactive annotation performed by the designer.
4. The situation where a design model is an extension (by specialization of the ontology or the design model is subsumed by the ontology) of the defined ontology may occur. This situation is ideal, it occurs when the design model is an extension of the domain ontology. In this case, reasoning by subsumption is possible. However, other situations may

occur. For example in case the specific design model is cross to several domain ontologies (i.e. multi-domain model) and uses its own concepts, such a specialization is not straightforward. More complex mappings (e.g. algebraic expressions, or other structural relationships) are required. The reasoning capabilities offered by subsumption may be lost and more complex reasoning mechanisms may be needed.

5.5. Associated theory

Following the previously defined stepwise methodology to make explicit domain knowledge expressed by ontologies in design models, we propose a general formal setting in which such a methodology can be deployed for specific formal methods.

As we do not address heterogeneous semantics, the rest of this paper considers that the same satisfaction $\models$ and entailment $\vdash$ relationships are used for both of the ontology modeling language and for the design modeling language.

1. **Formalization of Domain Knowledge.** An ontology is described with an ontology modeling language. It defines axioms $A_{O_1}, \ldots, A_{O_m}$ and proof deduction rules from which properties i.e. theorems $T_{O_1}, \ldots, T_{O_n}$ may be deduced.
   - Ontologies shall be sound (healthiness of the ontology). This means that there exists a model M_O that satisfies the axioms of the ontology. We write $M_O \models A_{O_i}$ for $1 \leq i \leq m$
   - Each theorem can be deduced from the axioms and the other theorems. We write $A_{O_1}, \ldots, A_{O_m} \vdash T_{O_1}$ and $A_{O_1}, \ldots, A_{O_m}, T_{O_1}, \ldots, T_{O_{i-1}} \vdash T_{O_i}$ for all $2 \leq i \leq n$
2. **Definition of design models.** The studied systems are described in the chosen modeling language. If the modeling language supports properties verification, then properties may be expressed and checked. Axioms $A_1, \ldots, A_k$ and theorems $T_1, \ldots, T_l$ describing the model properties are defined.
   - Described system models shall be sound (healthiness of the design model). This means that there exists a model M_D that satisfies the axioms defined for the system model. We write $M_D \models A_i$ for $1 \leq i \leq k$
   - Each theorem expressing properties on the design model can be deduced from the axioms and the other demonstrated theorems. We write $A_1, \ldots, A_k \vdash T_1$ and $A_1, \ldots, A_m, T_1, \ldots, T_{i-1} \vdash T_i$ for all $2 \leq i \leq l$
3. **Annotation of design model by references to ontologies.** Annotation consists in integrating domain knowledge expressed by ontologies in the design model.
   - Integrated axioms define a sound annotation (healthiness of the annotated model). There exists a model M satisfying the axioms of both the ontology and the design model. We write $M \models A_1 \wedge \ldots \wedge A_k \wedge A_{O_1} \wedge \ldots \wedge A_{O_m}$.
4. **Expression and verification of properties.** The properties $T_1, \ldots, T_l$ shall be re-proved again once the model has been enriched by ontologies. Moreover, new emerging properties $P_1, \ldots, P_t$ may be inferred from the annotated model.
   - The properties of the design model before annotation need to be re-proved again. Indeed, the ontology may have brought relevant information that falsify 0 or more properties. We write $A_1, \ldots, A_k, A_{O_1}, \ldots, A_{O_m} \vdash T_1$ and $A_1, \ldots, A_k, A_{O_1}, \ldots, A_{O_m}, T_{O_1}, \ldots, T_{O_n}, T_1, \ldots, T_{i-1} \vdash T_i$ for all $2 \leq i \leq l$
   - When domain knowledge described by ontologies is embedded in the design models, new properties $P_1, \ldots, P_t$ may arise. They should be proved. We write: $A_1, \ldots, A_m, A_{O_1}, \ldots, A_{O_m}, \vdash P_i$ for all $0 \leq i \leq t$

Remark. As mentioned in section 4.4, we have assumed that the same deduction logic (with $\vdash$ and $\models$) is associated to both the ontology and the design models. If this is not the case, alignment of the semantics of the ontologies and of the models should be performed. This is out of the scope of this paper.

5.6. Two cases

As stated previously, the chosen modeling language for ontologies, design models and annotations impacts both the description of the different involved model entities and the supported property checking. We consider that domain ontologies are described within set theory and first order logic. We show that such domain ontologies are useful to strengthen formal system development and verification in the engineering domain.

In this paper, the approach is deployed in two different situations with the same ontology model for both situations. It models concepts with attributes and inheritance. We show two applications cases of the methodology defined above on two formal methods representative of one of the two well-known families of formal methods

1. The first case is *model-checking* in section 6, it describes how state transition systems can be annotated by ontologies and checked within model checkers, and
2. The second case, in section 7, addresses the annotation of formal models expressed with the refinement and proof formal method Event-B.

Note that we study the case of two illustrative formal modeling languages and an ontology modeling language which are all expressed with set theory and first order logic. Thus, the semantic mismatch is no longer present in both cases.

6. Ontologies, behavioral models and model-checking

The first case related to design models annotation addressed by this paper concerns behavioral models and model checking. By behavior, we target process based models like web services, business processes, distributed systems, human computer interfaces, etc. In general, the behavior of such systems is captured by models expressed with labelled transition systems (*lts*) [9].

This section studies the case of labelled systems as underlying design models for the studied systems. Several properties can be checked on such systems. Here, we focus on the specific property weak bi-simulation [1,75]. This property is useful to compare two *lts*. It establishes behavioral equivalence of two labelled transition systems.

6.1. Design models: the case of lts as design models and bi-simulation

This section gives the basic definition of the bi-simulation relationship.

6.1.1. Labelled transition systems

A Labeled Transition System l is a structure $l = \langle S, s_0\ E,\ \longrightarrow\rangle$ where S is a set of states, $s_0 \in S$ denotes an initial state, E is a set of labels and $\longrightarrow \subseteq S \times E \times S$ is a transition relation between states.

When specifying systems by labelled transition systems, labels denote actions and the specific label $\tau \in E$ is used to denote internal actions i.e. non-observable actions.

6.1.2. The bi-simulation relationship

The notion of *bi-simulation* was initially defined by Milner [1] to compare processes (called observational equivalence). This relation defines equivalence on states of *lts*. Depending on the kind of considered actions (internal τ or observable), two kinds of observational equivalence exist: *strong bi-simulation* which considers observable actions and *weak bi-simulation* which considers both the observable and internal τ (stuttering) actions [1,75].

We consider the definition of bi-simulation for two different lts. Let $lts = \langle S, s_0\ E,\ \longrightarrow\rangle$ and $lts' = \langle S', s'_0\ E,\ \longrightarrow'\rangle$ be two transition systems *with the same set of labels* E and $S \cap S' = \varnothing$.

Definition 1. Strong bi-simulation. Strong bi-simulation relation $\sim \subseteq S \times S'$, is a *bi-simulation equivalence* defined on states. For a given action e, a transition $\xrightarrow{e}$ and two states p_i and q_i, we say that $(p_i, q_i) \in \sim$ if

$$\forall p_i \xrightarrow{e} p_j \in \longrightarrow : \exists q_i \xrightarrow{e} {}' q_j \in \longrightarrow' \wedge (p_j, q_j) \in \sim$$

$$\forall q_i \xrightarrow{e} {}' q_j \in \longrightarrow' : \exists p_i \xrightarrow{e} p_j \in \longrightarrow \wedge (q_j, p_j) \in \sim$$

By extension of this definition, two *lts* are strongly bi-similar (noted $\simeq_{lts}$) if their initial states are bi-similar i.e. $(s_0, s'_0) \in \sim$ (a strong bi-simulation including initial states can be built).

Definition 2. Weak bi-simulation. Weak bi-simulation relationship noted $\approx \subseteq S \times S'$, is a bi-simulation equivalence defined on states. For a given action e, a transition $\xrightarrow{e}$ and two states p_i and q_i, we say that $(p_i, q_i) \in \approx$ if

$$\forall p_i \xrightarrow{e} p_j \in \longrightarrow \exists q_i \xrightarrow{\tau^\star.e.\tau^{\star\prime}} q_j \in \longrightarrow' \wedge (p_j, q_j) \in \approx$$

$$\forall p_i \xrightarrow{\tau} p_j \in \longrightarrow \exists q_i \longrightarrow^{\tau^\star} q_j \in \longrightarrow' \wedge (p_j, q_j) \in \approx$$

$$\forall q_i \xrightarrow{e}{}' q_j \in \longrightarrow' \exists p_i \xrightarrow{\tau^\star.e.\tau^\star} p_j \in \longrightarrow \wedge (q_j, p_j) \in \approx$$

$$\forall q_i \xrightarrow{\tau}{}' q_j \in \longrightarrow' \exists p_i \xrightarrow{\tau^\star} p_j \in \longrightarrow \wedge (q_j, p_j) \in \approx$$

By extension two *lts* are weakly bi-similar (noted $\approx_{lts}$) if their initial states are weakly bi-similar i.e. $(s_0, s'_0) \in \approx$ (a weak bi-simulation including initial states can be built).

6.2. Some remarks

One of the main interests of bi-simulation is the capability to compare the behavior of two *lts*. From the engineering point of view, applications of such a relation can be the substitution of an *lts* by another one if they are bi-similar since they have the same observed behavior. Applications of this results in system engineering are numerous, we can cite adaptation, maintenance, substitution, redundancy, self-healing, etc.

Bi-simulation relationships are defined on a single set of labels and compare transitions with the same labels. So, in a specific design context where all the labels (actions) are shared by the design model, *lts* comparison can be performed using this definition. Several formal verification techniques offer the capability to check whether two *lts* are bi-similar (examples are proof and refinement with [10] and model checking with [11]).

6.3. Ontology: an ontology of labels

When considering different application domains, it appears that several behavioral systems may be defined in different contexts. As a consequence, a single system behavior may be described by different behavioral models associated to different labeled transition systems. In this case, comparing the behavior of such *lts* using bi-simulation as defined above may fail due to the appearance of different sets of labels in different *lts*.

One solution could be the definition relations between sets labels independently of any context of use. In particular, these relation may express equivalence or subsumptions on labels. An ontology of sets of labels is well suited to describe such relations.

We introduce a relation, Γ on pairs of labels, used to rewrite different labels. Let $lts = \langle S, s_0\ E,\ \longrightarrow\rangle$ and $lts' = \langle S', s'_0\ E',\ \longrightarrow'\rangle$ be two transition systems such that $S \cap S' = \varnothing$, $E \nsubseteq E'$ and $E' \nsubseteq E$. Let A be another set of labels different from the ones of $E \cup E'$, in other words $A \cap (E \cup E') = \varnothing$

Definition 3. Bi-directional relation on labels. $\Gamma \subseteq E - E' \times E' - E$ is a relation on labels of two labelled transition systems defined by

$$\forall\, \alpha \in E - E', \exists\, \beta \in E' - E \text{ such that } \quad (\alpha, \beta) \in \Gamma$$

$$\forall\, \beta \in E' - E, \exists\, \alpha \in E - E' \text{ such that } \quad (\alpha, \beta) \in \Gamma$$

The left and right projection functions $Proj_l$ and $Proj_r$ are associated to Γ.

Informally, Γ links labels belonging to the set of labels that are not in $E \cap E'$. Note that this relation belongs to the defined domain ontology.

6.4. Annotation: establishing correspondences between labels of lts

The annotation of *lts* as design models consists in exploiting the relationship on labels borrowed from the defined domain ontologies of labels. This ontology expresses equivalence or subsumption of labels. The annotation process consists first in rewriting the different labels in order to obtain *lts* with the same labels and second check bi-simulation of the transformed *lts*.

6.4.1. Rewriting labels

To rewrite labels, we define a rewriting function on labels belonging to sets of labels of the ontology.

Definition 4. Rewriting function on labels. Let $\Phi : E \times E' \longrightarrow (A \cup E \cup E' \cup \{\tau\})$ be a function on labels of two sets of labels in the ontology. Φ is defined by

$$\forall\, (\alpha, \beta) \in \Gamma\ \ \exists\, \gamma \in A \text{ such that } \ \ \Phi(\alpha, \beta) = \gamma$$

The definition of the rewriting function entails four different applications.

1. Substitution. If $\exists\, e \in A$ such that $\Phi(a, b) = e$ then labels a and b are replaced by a new label e in A.
2. Right replacement. If $\exists\, b \in E'$ such that $\Phi(a, b) = a$ then label $b \in E'$ is replaced by a label $a \in E$.
3. Left replacement. If $\exists\, a \in E$ such that $\Phi(a, b) = b$ then label $a \in E$ is replaced by a label $b \in E'$.
4. Hiding. $\Phi(a, b) = \tau$ denotes the case of a pair of labels that should be hidden on both labelled transition systems after rewriting.

Remark. When an ontology is used, substitution means that a subsuming concept is available and left (resp. right) replacement means that left (resp. right) label is subsumed by the right (resp. left) one. Hiding means that the label is not relevant for checked property. *If* Γ *is empty, then no comparison is possible.*

6.4.2. Relational bi-simulation

Once the rewriting function is applied and if succeeded, it becomes possible to describe the process to transform two *lts* into two new *lts* with the same set of labels.

Definition 5. Transforming labelled transition systems. $lts = \langle S, s_0\ E,\ \longrightarrow\rangle$ and $lts' = \langle S', s'_0\ E',\ \longrightarrow'\rangle$ are respectively rewritten to $lts^{\top} = \langle S^{\top}, s_0^{\top}\ E^{\top},\ \longrightarrow^{\top}\rangle$ and $lts^{\top'} = \langle S^{\top'}, s_0^{\top'}\ E^{\top'},\ \longrightarrow^{\top'}\rangle$ according to the label relation Γ and to the rewriting function on labels Φ as follows.

- $S^{\top} = S$ and $S^{\top'} = S'$ i.e. same sets of states.
- $s_0^{\top} = s_0$ and $s_0^{\top'} = s'_0$ i.e. same initial states.

- $E^{\top} = (E - Proj_l(\Gamma)) \cup A \cup \{\tau\}$ and $E^{\top'} = (E' - Proj_r(\Gamma)) \cup A \cup \{\tau\}$ sets of labels enriched with the rewritten labels thanks to the Φ rewriting function.
- transition relations are redefined on the new labels $\longrightarrow^{\top} \subseteq S^{\top} \times E^{\top} \times S^{\top}$ and $\longrightarrow^{\top'} \subseteq S^{\top'} \times E^{\top'} \times S^{\top'}$ where

$$\longrightarrow^{\top} = \{s \xrightarrow{e} t \in \longrightarrow \mid \forall e' \in E'.\ (e,e') \notin \Gamma\} - \{s \xrightarrow{e} t \in \longrightarrow \mid \forall e' \in E'.\ (e,e') \in \Gamma\}$$
$$\cup \{s \xrightarrow{a} t \mid \exists (e,e') \in \Gamma \wedge \Phi(e,e') = a\}$$
$$\longrightarrow^{\top'} = \{s' \xrightarrow{e'} t' \in \longrightarrow' \mid \forall e \in E.\ (e,e') \notin \Gamma\} - \{s' \xrightarrow{e'} t' \in \longrightarrow' \mid \forall e \in E.\ (e,e') \in \Gamma\}$$
$$\cup \{s' \xrightarrow{a} t \mid \exists (e,e') \in \Gamma \wedge \Phi(e,e') = a\}$$

6.5. Verification: property verification on annotated lts

The $lts^{\top}$ and $lts^{\top'}$ obtained after annotation are labelled transition systems with the same set of labels, since $E^{\top} = E^{\top'}$. It becomes possible to compare them with classical bi-simulation relationship.

This process leads to the definition of the relational bi-simulation that exploits a relation defined out of the context of use of the design models.

Definition 6. Relational weak bi-simulation relationship on lts. Let $\langle lts, lts', \Gamma, \Phi \rangle$ be a structure where

- lts and lts' are two labelled transition systems such that $S \cap S' = \varnothing$, $E \nsubseteq E'$ and $E' \nsubseteq E$,
- $\Gamma \subseteq E \times E'$ is a relationship on labels according to Definition 3,
- Φ is a label rewriting function according to Definition 4.

Then, $\approx_{lts}^{\Gamma,\Phi} \subseteq LTS \times LTS$ is relational weak bi-simulation relationship on labelled transition systems if there exists a weak bi-simulation relationship on labelled transition systems between the transformed lts. We write

$$(lts, lts') \in \approx_{lts}^{\Gamma,\Phi} \Longleftrightarrow (lts^{\top}, lts^{\top'}) \in \approx_{lts}$$

It becomes possible to compare labelled transition systems with different sets of labels.

6.6. Case studies

The principle of annotating labelled transition systems has been used in two case studies: formal verification of plastic interfaces and of web services composition.

6.6.1. Plastic user interfaces

Plastic user interfaces are human computer interfaces that may be adapted to the context of use. With the availability of different interaction modes (mouse, voice, touch screens, etc.) many user interfaces to interact with a given application can be envisioned. For two user interfaces UI_1 and UI_2 of a single application, two lts namely lts_1 and lts_2 can be modeled. Their labels correspond to the different interaction modes.

According to an ontology Ont_{UI} defining a relation $\Gamma_{Ont_{UI}}$ on interaction modes and a rewriting function $\Phi_{Ont_{UI}}$, we say that UI_1 and UI_2 satisfy plasticity property [76,77] if

$$lts_1 \approx_{lts}^{\Gamma_{Ont_{UI}}, \Phi_{Ont_{UI}}} lts_2$$

Note that if $\Gamma = \varnothing \wedge E1 \neq E2$ the plasticity property does not hold. We have experimented this approach with labelled transition systems expressed within the LOTOS process algebra and the bi-simulation checked with the CADP [11] toolbox.

6.6.2. Web services composition

Web services composition may also get benefits from our methodology. If we consider two composite services sw_1 and sw_2 expressed in a composition language like BPEL [78], it is possible to model sw_1 and sw_2 with two labelled transitions systems lts_1 and lts_2 with atomic services as labels [79–81]. Assuming an ontology of services Ont_{SW} [2–4] is available, then one can identify services that achieve the same goal. For example, the *sending by email* and the *sending by surface mail* atomic services or the *sending electronic invoice* and *sending paper invoice* atomic services of the ontology Ont_{SW} are considered as equivalent since they achieve similar functions.

Then, one can check if a composite service sw_1 can be substituted by another composite service sw_2 in case of failure or loss of quality of service etc. So, according to an ontology Ont_{SW} defining a relation $\Gamma_{Ont_{SW}}$ on atomic services and a rewriting function $\Phi_{Ont_{SW}}$, we say that sw_1 can be substituted by sw_2 [4] if

$$lts_1 \approx_{lts}^{\Gamma_{Ont_{SW}}, \Phi_{Ont_{SW}}} lts_2$$

Note that if $\Gamma = \varnothing \wedge E1 \neq E2$ the substitution property does not hold.

These two examples show how the benefits of making explicit domain knowledge in order to improve the quality of design models.

7. Ontologies, behavioral models and refinement and proof-based formal methods

This section presents the case of exploitation of ontologies by a proof and refinement based formal method.

7.1. The case of Event-B

7.1.1. Design models: the Event-B modeling language

Event-B is a formal modeling language for expressing state-based models of reactive systems. It is supported by two proof-assistant-based environments, namely Rodin [10] and Atelier-B [82]. The two environments can be used for developing the same models; both environments use the same kernel proof module and any Event-B model developed in one of these environments can be exported in the other environment. The Rodin platform is an open toolset platform, which is used for developing, analysing, validating and experimenting the Event-B methodology. Atelier-B is freely distributed but remains an industrial tool. Moreover, Atelier-B has first provided functionalities for building classical B models, which are models for developing only software. The construction of an Event-B model is based on concepts like sets, constants, axioms, theorems, variables, invariants, events; these syntactic constructions are organised in two kinds of structures, namely contexts and machines. Fig. 2 contains the general form of each possible component. Contexts express axiomatic static properties of the models and machines express dynamic behavioral properties (state-based features) of the models, which may contain variables, invariants, theorems and events. A machine $\mathcal{M}$ *sees* a context $\mathcal{D}$.

Contexts may contain carrier sets, constants, axioms, and theorems. Axioms describe properties of carrier sets and constants. Theorems derive properties that can be proved from the axioms. Proof obligations associated with contexts are straightforward: the stated theorems must be proved, which follow from the predefined axioms and theorems. Additionally, a context C can be seen by a machine M indirectly if the machine M explicitly sees a context D which is an extension of the context C. Variables v represents the state of the machine M. Variables, like constants, correspond to simple mathematical objects: sets, binary relations, functions, numbers, etcetera. They are constrained by invariants $I(v)$. Invariants are supposed to hold whenever variable values change.

A machine is organizing events modifying the state variables and it uses static information defined in a context. These basic structure mechanisms are extended by the refinement mechanism which provides a mechanism for relating an abstract model and a concrete model by adding new events or by adding new variables. This mechanism allows us to develop gradually Event-B models and to validate each decision step using the proof tools. The refinement relationship should be expressed as follows: a model M is refined by a model P, when P is simulating M. The final concrete model is close to the behavior of real system that is executing events using real source code. We give details now on the definition of events, refinement and guidelines for developing complex system models.

The consistency of a context or a machine in Event-B is achieved by proving *proof obligations* generated by tools [82,10] and sound with respect to the results of the previous section. If these proof obligations are discharged, then the structure (context or machine) is correct at least with respect to the typing.

An Event-B model organizes a set of *events* stating how state-variables may be modified, when observing the occurrence of one of them. Intuitively, an event is triggered when guard (the triggering condition of an event) evaluates to true. Due to non-determinism, if a given guard evaluates to true does not mean that the corresponding event is triggered. Indeed, several events may have guards evaluating to true and only one of them is triggered (interleaving of events).

Each event can be defined by a relationship before–after denoted as $BA(x, x')$. An event is characterized by its guard which is determined at the modeling phase and it can only be triggered if the guard is true. We will detail proof obligations generated for a given event e and explain the meaning of these proof obligations. For each event e, proof obligations are generated and discharged by the environment Rodin [10]. These two concepts allows us to illustrate the relationship between *ontologies* and *formal models* and what may be the gain of ontologies in the formal modeling process. The modeling process deals with various languages, as seen by considering the triptych of Bjørner [16–18,83]: $\mathcal{D}, \mathcal{S} \longrightarrow \mathcal{R}$. Here, the domain $\mathcal{D}$ deals with properties, axioms, sets, constants, functions, relations, and theories and it is written as a context enriched by ontological information. The system model $\mathcal{S}$ expresses a model or a refinement-based chain of models of the system. Finally, $\mathcal{R}$ expresses requirements for the system to be designed. Considering the Event-B modeling language, we notice that the language can express *safety* properties, which are either *invariants* or *theorems* in a machine corresponding to the system.

A context (see Fig. 2) provides the definition of the sets, constants, axioms for sets and constants, and theorems that can be derived from the axioms of the context $\mathcal{D}$. The context $\mathcal{AD}$ is a previous context that has already been defined, and it is extended to the current context. A context is validated when sets $S_1, \ldots, S_n$, constants $C_1, \ldots, C_m$, and axioms $ax_1, \ldots, ax_p$ are well formed and when all theorems $th_1, \ldots, th_q$ are proved.

A context clearly states the static properties of the (system) model under construction. The *extends* construct enables re-use by extending a previously defined context.

The proof process is based on the management of sequents, with an associated environment for proof called $\Gamma(\mathcal{D})$. The proof environment includes axioms, properties, and theorems already proved. An environment is initially provided, but the intention is to add new theorems. This means that we intend to prove the following properties in the sequent calculus style:

$$\text{for any } j \text{ in } \{1..q\}, \Gamma(\mathcal{D}) \vdash th_j : Q_j(S_1, \ldots S_n, C_1, \ldots, C_m).$$

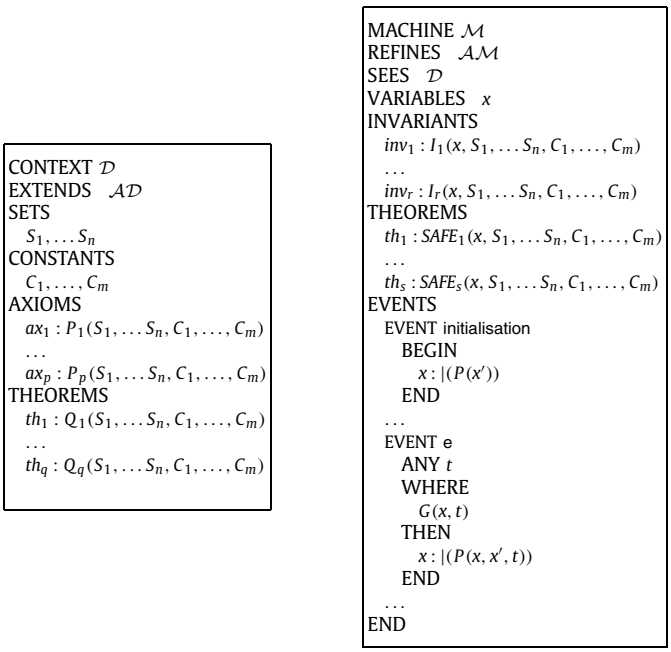

Fig. 2. Context and machine.

Theorems for the context are proved using the Rodin tool, but it is clear that the process for constructing the domain $\mathcal{D}$ is crucial to modeling the system, from consideration of the triptych of Bjørner [16–18,83] and variations of this methodology.

The possibility of re-using former definitions is crucial, but we do not consider this point in this paper. Instead, we *simulate* the re-use of theories by manipulating the contexts directly. Among the requirements, we can list the theorems of the context, and we can, in fact, interpret the triptych as follows: for any j in $\{1..q\}$, $\mathcal{D} \longrightarrow th_j : Q_j(S_1, \ldots S_n, C_1, \ldots, C_m)$. Here, it appears that the system is not mentioned, and this is the case for static properties. Therefore, we have an interpretation of the triptych for the static information, which can be re-used later for any system.

The dynamic part of a model is expressed using a *machine* (see Fig. 2). A machine is either a basic machine or a refinement of an abstract machine. A machine models a state via a list of variables x that are assumed to be modifiable by events listed in the machine. The view is assumed to be closed with respect to events. Each event maintains an assertion called an *invariant*, which is a conjunction of logical statements called inv_j. Each reached state satisfies properties of the theorem part called safety properties. Proof obligations are given in the last section, and they are generated and checkable by the RODIN framework. The validation of the machine M leads to the validation of the safety and invariance properties.

We can obtain a variation of the triptych ($\Gamma(\mathcal{D}, M)$ is an associated environment for proof) as follows.

- For any j in $\{1..r\}$,
  $\Gamma(\mathcal{D}, M) \vdash INITIALISATION(x') \Rightarrow I_j(x', S_1, \ldots S_n, C_1, \ldots, C_m)$
- For any j in $\{1..r\}$, for any event e of M,
  $\Gamma(\mathcal{D}, M) \vdash \Big(\bigwedge_{j\in\{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m)\Big) \wedge BA(e)(x, x') \Rightarrow I_j(x', S_1, \ldots S_n, C_1, \ldots, C_m)$
- For any k in $\{1..s\}$,
  $\Gamma(\mathcal{D}, M) \vdash \Big(\bigwedge_{j\in\{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m)\Big) \Rightarrow SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m)$

We use the temporal operator $\Box P$ to express the safety and invariant properties. This operator expresses that the property P is true in all the system states [84,85].

- For any j in $\{1..r\}$, $\mathcal{D}, M \longrightarrow \Box I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m)$.
- For any k in $\{1..s\}$, $\mathcal{D}, M \longrightarrow \Box SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m)$.

We summarize the requirements expressed by the machine M as follows.

$$\mathcal{D}, M \longrightarrow \Box \left(\begin{array}{l} \Big(\bigwedge_{j\in\{1..r\}} I_j(x, S_1, \ldots S_n, C_1, \ldots, C_m)\Big) \\ \Big(\bigwedge_{k\in\{1..s\}} SAFE_k(x, S_1, \ldots S_n, C_1, \ldots, C_m)\Big) \end{array} \right)$$

We will use the notation $\mathcal{I}(M)$ to stand for the invariants of the machine M and $\mathcal{SAFE}(M)$ to stand for the safety properties of the machine M. We have shown that requirements $\mathcal{R}$ are first expressed using the *always* temporal operator. In next subsections, we illustrate how ontological information can be used for enriching contexts as well as machines following the refinement-based process. We present two simple examples which are illustrating benefits for formal modeling when using ontologies.

7.2. An example from program verification in Event-B

A simple example is an annotated algorithm called *Example* and setting a value y to a value. The *explicit* knowledge of the designer of the algorithm is allowing him to state that the final value of y will be B. The *explicit* knowledge is required for inferring the correctness of the algorithm which is transformed into an Event-B machine listing the verification conditions to check derived from Floyd's annotations. Two knowledges are missing in the context in the first attempt to discharge the proof obligations generated by the environment: *an1* and *an2*:

- *an1* is added for inferring that there exists at least one value in S different of A.
- *an2* is added for stating that there is no other value different of A in S.

an1 and *an2* are two explicit knowledges of the designer of the algorithm. The prover is requiring new assumptions that are given by the designer, who is the expert of the problem.

ALGORITHM *Example*
CONSTANTS x
VARIABLES y
PRE $x \in S \wedge x = A \wedge y \in S$
POST $y \in S \wedge y = B$
BEGIN
$\ell_1 : \{x = A\}$
$y : |(y' \in S \wedge y' \neq x)$;
$\ell_2 : \{y = B\}$
END

CONTEXT $C1$
SETS S, L
CONSTANTS $l1, l2, A, B, x$
AXIOMS
$axm1 : partition(L, \{l1\}, \{l2\})$
$axm2 : A \in S$
$axm3 : B \in S$
$axm4 : x \in S$
$axm5 : x = A$
$an1 : A \neq B$ annotation1
$an2 : S \subseteq \{A, B\}$ annotation2
END

MACHINE $M1$
SEES $C1$
VARIABLES y, pc
INVARIANTS
$inv1 : pc \in L \wedge y \in S$
$inv2 : pc = l1 \Rightarrow y \in S$ *Floyd's annotation*
$inv3 : pc = l2 \Rightarrow y = B$ *Floyd's annotation*
$th1 : x \in S \wedge x = A \wedge y \in S \Rightarrow y \in S$ *checking precondition*
$th2 : y = B \Rightarrow y \in S \wedge y = B$ *checking postcondition*
EVENTS
EVENT INITIALISATION
BEGIN
$act1 : y :\in S$
$act2 : pc := l1$
END
EVENT e
WHEN
$grd1 : pc = l1$
THEN
$act1 : pc := l2$
$act2 : y : |(y' \in S \wedge y' \neq x)$
END
END

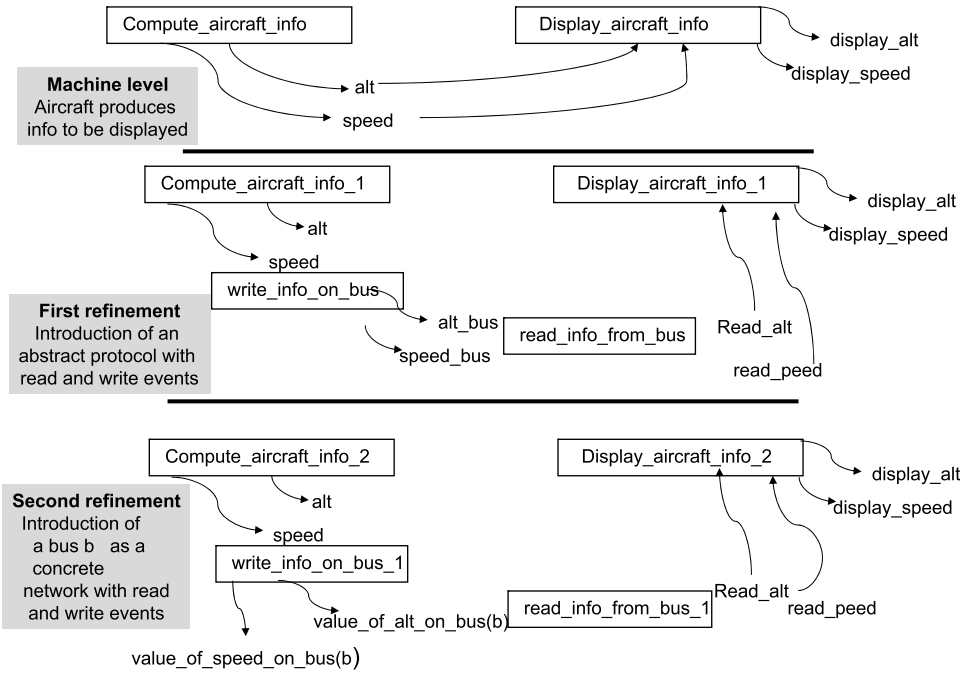

Fig. 3. A global view of the formal development.

When considering the statement of the triptych, we have the following relationship in the equation (1). It states that under the proof obligations generated and proved by the Rodin environment, the algorithm is partially correct. The explicit knowledges of the designer are encoded by the prover as annotations which are axioms. Why do we use axioms? In fact, the explicit knowledge is trusted by the prover and annotations are derived from expertise.

$$C1, M1 \longrightarrow (\text{PRE } x \in S \wedge x = A \wedge y \in S, \text{POST } y \in S \wedge x = B) \tag{1}$$

7.3. A simple avionic system

Let us consider a simple system issued from avionic system design. We identify two sub-systems: the first one is part of the flight management system acting in the closed world (heart of the avionic systems), it produces flight information, like *altitude* and *speed*; and the second is the display part of a passenger information system (open world). It displays, to the passengers, information issued from the closed world, here altitude and speed. The information is transmitted from the closed world to the open world within a communication bus. Communications are *unidirectional from the closed world to the open world only*.

The development of this system considers a formally expressed specification which is refined twice. Fig. 3 shows the structure of the development for this case study, and the whole Event-B developments are given in Appendix A.

The development, presented below, introduces the explicit knowledge carried out by ontologies, it is used for coding the ternary relationship called triptych. In Event-B, it is formalized within *contexts*. The ternary relationship is obtained by *annotation* i.e. linking the model elements, variables in our case, to the explicit knowledge.

7.3.1. Ontologies: contexts for defining explicit domain knowledge

The first step consists of introducing the explicit domain knowledge through a formal model for ontologies. In the simple case, this knowledge is defined by contexts. In this case, we are concerned by the description of the units that may be associated to the *altitude* and to the *speed* in the context DOMAIN_KNOWLEDGE_FOR_UNITS.

Meters, inches, kilometers per hour, and miles per hour are introduced to define distance speed measures. Conversion functions, that define equivalences in terms of ontology definitions, are described by the functions *inchtometer* and *mphtokph*. We do not detail these two functions but they can be made more precise by an implementation step at a later phase in the process.

7.3.2. Annotation: associating explicit knowledge to model variables

Once the explicit knowledge has been formalized, it becomes possible to annotate available concepts in the obtained formal models. In our case, the variables are annotated by explicitly referring to the ontology defined in the context of Fig. 4. Measurement units are introduced in an explicit way.

CONTEXT *DOMAIN_KNOWLEDGE_FOR_UNITS*
SETS
STATUS, *inch*, *meter*, *mph*, *kph*
CONSTANTS
OK, *KO*, *inchtometer*, *mphtokph*, *valt*, *vspeed*
AXIOMS
axm1 : $partition(STATUS, \{OK\}, \{KO\})$
onto-axm2 : $inchtometer \in inch \rightarrow meter$
onto-axm3 : $mphtokph \in mph \rightarrow kph$
axm4 : $valt \in inch$
axm5 : $vspeed \in mph$
END

Fig. 4. The ontological context for units.

The variables are then defined as follows:

$inv1 : speed \in mph$
$inv2 : alt \in inch$
$inv4 : display\_speed \in kph$
$inv5 : display\_alt \in meter$

When the annotations have been specified, the new invariant defines the ontological constraints that should be satisfied by the events. For example, one of the generated proof obligations for checking the preservation of $inv5 : display\_alt \in meter$ by the event Display_Aircraft_Info fails to prove that $alt \in meter$. Thus, we should modify the event Display_Aircraft_Info by adding the ontological information provided by the two functions *inchtometer* and *mphtokph*. The example is simple and gives an obvious way to solve the possible unproved proof obligation: without refinement it may be much more difficult to discover why similar proof obligations are not discharged. We think that the refinement is one of the key issues for *distilling* requirements and ontological information. This method highlights the interest of handling explicit knowledge for checking model correctness.

Consequently, the following events — Display_Air craft_Info and Compute_aircraft_info — require further description. In Particular, Display_aircraft_info has been modified in order to handle converted values issued from Compute_aircraft_info.

EVENT Compute_Aircraft_Info
WHEN
$grd1 : computing = KO$
THEN
$act1 : alt :\in inch$
$act2 : speed :\in mph$
$act3 : computing := OK$
END

EVENT Display_Aircraft_Info
WHEN
$grd1 : computing = OK$
THEN
$act1 : computing := KO$
$act2 : display\_speed := mphtokph(speed)$
$act3 : display\_alt := inchtometer(alt)$
END

7.3.3. First refinement: introducing an abstract communication protocol

As a next step, we can add new features in the current model COM1 by refining it into COM10.

The new model COM10 performs the same extension of the state as in the previous case using implicit knowledge. This is quite natural since none of these state variables (i.e. *step*) are annotated. Two new events model the reading to and the writing from the bus. The invariant is extended by sub-invariants *inv3* ... *inv9*. Notice the introduction of new kinds of invariants, labeled *onto-inv7*, *onto-inv8* and *onto-inv9*, borrowed from the context where the explicit knowledge is described. They define **ontological invariants**.

$inv3 : alt\_bus \in meter$
$inv4 : speed\_bus \in kph$
$inv5 : read\_speed \in kph$
$inv6 : read\_alt \in meter$
$onto\text{-}inv7 : step = written \Rightarrow alt\_bus = inchtometer(alt) \wedge speed\_bus = mphtokph(speed)$
$onto\text{-}inv8 : step = read \Rightarrow \left(\begin{array}{l} read\_alt = alt\_bus \wedge alt\_bus = inchtometer(alt) \\ \wedge read\_speed = speed\_bus \wedge speed\_bus = mphtokph(speed) \end{array} \right)$
$onto\text{-}inv9 : step = tocompute \Rightarrow \left(\begin{array}{l} display\_alt = read\_alt \wedge read\_alt = alt\_bus \\ \wedge alt\_bus = inchtometer(alt) \wedge display\_speed = read\_speed \\ \wedge read\_speed = speed\_bus \wedge speed\_bus = mphtokph(speed) \end{array} \right)$

CONTEXT *domain_knowledge_for_protocols*
EXTENDS *domain_knowledge_for_units*
SETS
bus, *tbus*
CONSTANTS
unidirectional, *bidirectional*, *tbus*
AXIOMS
axm1 : $partition(bus\_type, \{unidirectional\}, \{bidirectional\})$
axm2 : $tbus \in bus \rightarrow bus\_type \wedge abus \in bus \wedge tbus(abus) = unidirectional$
axm5 : $s \in bus \rightarrow kph \wedge s(abus) = mphtokph(vspeed)$
axm6 : $a \in bus \rightarrow meter \wedge a(abus) = inchtometer(valt)$
END

Fig. 5. The ontological context for bus protocols.

The two abstract events are refined by strengthening guards with respect to the new control variable (*step*). The new model introduces an abstract protocol for the bus.

EVENT Compute_Aircraft_Info_1
REFINES Compute_Aircraft_Info
WHEN
grd1 : $computing = KO$
grd2 : $step = tocompute$
THEN
act1 : $alt :\in inch$
act2 : $speed :\in mph$
act3 : $computing := OK$
act4 : $step := computed$
END

EVENT Display_Aircraft_Info_1
REFINES Display_Aircraft_Info
WHEN
grd1 : $computing = OK$
grd2 : $step = read$
THEN
act1 : $computing := KO$
act2 : $display\_speed := read\_speed$
act3 : $display\_alt := read\_alt$
act4 : $step := tocompute$
END

The two next events model the abstract protocol for exchanging the data. The abstract protocol manages the relationship between the measurement units. The ontological annotation appears in the invariant *inv*13: the protocol ensures the *correct* communication.

EVENT Write_Info_On_Bus
WHEN
grd1 : $step = computed$
THEN
act1 : $alt\_bus := inchtometer(alt)$
act2 : $speed\_bus := mphtokph(speed)$
act3 : $step := written$ END

EVENT Read_Infor_From_Bus
WHEN
grd1 : $step = written$
THEN
act1 : $step := read$
act2 : $read\_alt := alt\_bus$
act3 : $read\_speed := speed\_bus$
END

7.3.4. Context extension: need for explicit knowledge on the bus

The current system is still abstract and we have to add details concerning the bus. Following good engineering practice, the communication bus should be described independently of any usage in a given model. Here again, an ontology of communication media is needed. It is defined in a context that extends the one defined for measure units. The bus has specific properties that are expressed in a new context domain_knowledge_for_protocols (in Fig. 5).

Notice that the definition of explicit knowledge is modular. It uses contexts that import only those ontologies that are needed for a given development. Moreover, it is flexible since contexts can be changed, if the domain knowledge or the nature of the manipulated concepts evolves. The whole formal development of the system does not need to be rewritten.

7.3.5. Second refinement: concretizing the bus for communication

inv1 : $speedbus \in bus \rightarrow kph$
inv2 : $altbus \in bus \rightarrow meter$
onto-inv3 : $altbus(b) = alt\_bus$
onto-inv4 : $speedbus(b) = speed\_bus$
onto-inv6 : $tbus(b) = unidirectional$
onto-inv7 : $b \in bus$

The new invariant extends the previous one, whilst integrating the state of the bus. It also asserts that the bus is *unidirectional*, which is a very important issue for ensuring security over the communications. The invariant *onto-inv7* : $b \in bus$ is an ontological invariant and the context enriches the description of the domain. It explicitly expresses that the bus is *unidirectional*. Finally, the four events of the model COM10 are refined to concretize the actions over the bus b. The two first events are directly related to the computation and display components.

EVENT Compute_Aircraft_Info_2
REFINES Compute_Aircraft_Info_1
WHEN
  grd1 : $computing = KO$
  grd2 : $step = tocompute$
THEN
  act1 : $alt :\in inch$
  act2 : $speed :\in mph$
  act3 : $computing := OK$
  act4 : $step := computed$
END

EVENT Display_Aircraft_Info_2
REFINES Display_Aircraft_Info_1
WHEN
  grd1 : $computing = OK$
  grd2 : $step = read$
THEN
  act1 : $computing := KO$
  act2 : $display\_speed := read\_speed$
  act3 : $display\_alt := read\_alt$
  act4 : $step := tocompute$
END

The two next events — Write_Info_On_Bus and Read_Infor_From_Bus — model operations over the bus. They both deal with ontological annotations, where the more detailed characteristics of the bus are necessary for guaranteeing the safety of the global system.

EVENT Write_Info_On_Bus_1
REFINES Write_Info_On_Bus
WHEN
  grd1 : $step = computed$
THEN
  act1 : $altbus(b) := inchtometer(alt)$
  act2 : $speedbus(b) := mphtokph(speed)$
  act3 : $step := written$
END

EVENT Read_Infor_From_Bus_1
REFINES Read_Infor_From_Bus
WHEN
  grd1 : $step = written$
THEN
  act1 : $step := read$
  act2 : $read\_alt := altbus(b)$
  act3 : $read\_speed := speedbus(b)$
END

8. Conclusion and perspectives

Handling formally the domain knowledge in design models is a challenge in system and software engineering. When the capability to enrich design models with relevant knowledge and properties mined from the domain where a system evolves or a software runs is offered, the quality of the obtained design models is increased.

In this paper, we have proposed a generic methodology to make explicit the domain knowledge in formal system/software developments. This methodology advocates the use of formalized ontologies to model domain knowledge on the one hand, and a formally defined annotation relationship to link domain knowledge concepts with formal design models entities on the other hand. Ontologies are formalized as theories exploited by the formal modeling technique used to express design models. The annotation relationship enriches the proof contexts of the design models with axioms and theorems borrowed from the domain ontology.

Compared to the classical approaches of the semantic web or information retrieval, this approach is different because it addresses resources that are formal models with a formal semantics and which support formal validation and verification of properties. Such formal semantics definition and properties verification and validation are not required in case of web pages and more generally for documents. In general no semantics is associated to such documents. One may define weak XML models for these documents, but not a rigorously defined semantics. In our case, two formal semantics are defined, one for the ontologies and another for the design models. We make a clear separation between the semantics carried by the design models and the one carried by the domain ontology.

The defined methodology is not tied to a specific formal method nor to a particular ontology modeling language, but it is generic enough to be followed by different formal development approaches. It has been set up in two cases.

- Model checking of labelled transitions systems to check substitutability of design models of systems and/or software expressed with labelled transition systems. The annotation consisted in exploiting a relationship, explicitly defined in a domain ontology, between the labels of a labelled transition system. This relationship permitted to define a rewriting of labels. It has been applied to the case of plastic human computer interfaces and of semantic web services. More generally, it could be applied to study adaptive systems properties.

- Proof and refinement-based development methods illustrated by the Event-B formal method. Here, ontologies have been modeled by Event-B contexts and the annotation relationship has been formalized thanks to specific invariants, namely ontological invariants, expressing the annotation relationship. The proposed approach has been applied to the development of an avionic system involving hardware, network and software components characterized by a domain ontology formalized in an Event-B context. Ontological invariants were defined to define relevant safety properties and gluing invariants for refinements.

We have specifically addressed the case of system and/or software engineering requiring validation and verification of properties. To formalize the semantics of the manipulated models *for both ontologies and design models*, *a single* specific theoretical setting has been considered: first order logic and set theory to express both domain ontologies and design models and the underlying proof and verification procedures under a closed world assumption. This theoretical setting corresponds to the formal development methods we have experimented: model checking and proof and refinement with Event-B.

To the best of our knowledge, this work is the first attempt towards handling formally domain knowledge in formal development in an explicit manner. Indeed, the proposed approach makes a clear separation between the formalized domain ontology and the design model. The approach is completely modular. It clearly separates the design model from the domain ontology. This separation means that both ontologies and design models may evolve separately and asynchronously. Obviously, property checking shall be replayed each time an evolution of the domain ontology and/or of the design models occurs.

The work presented in this paper opens several other research paths we plan to investigate in the near future.

The asynchronous evolution of the domain ontologies and/or of the design models should be studied more precisely. Currently, any evolution requires a complete checking i.e. all the properties verification shall be performed after each evolution. The capability to identify and to localize, in the ontologies and/or in the design models, which parts are affected by an evolution would help in avoiding already performed properties verification and validation on the annotated design model. In other words, the proposed approach should be made more *compositional*. Our belief is that identification and localization of ontological invariants combined with refinement would help to achieve this objective.

Then, as mentioned above, we have chosen a fixed semantic setting with first order logic, state transitions systems and closed world assumption. Other semantic settings, for example with probabilities or an open world assumption may be set up. As a consequence, semantic heterogeneity appears with two specific satisfaction relationships ($\models_M$ and $\models_O$) and two specific entailment relationships ($\vdash_M$ and $\vdash_O$). In this case, an integrated semantics is required (semantic alignment) and the annotation relationship should be overloaded with relevant semantic mappings and correspondences.

Appendix A. Complete Event-B archive for the first case study SCP1

A.1. Context for SCP1

An Event-B Specification of C1
Creation Date: 12Feb2015 @ 11:27:51 PM

CONTEXT C1
SETS
S
L
CONSTANTS
$l1$
$l2$
A
B
x
AXIOMS
axm1 : $partition(L, \{l1\}, \{l2\})$
axm2 : $A \in S$
axm3 : $B \in S$
axm4 : $x \in S$
axm5 : $x = A$
axm6 : $A \neq B$
annotation 1
axm7 : $S \subseteq \{A, B\}$
END

A.2. Model for SCP1

An Event-B Specification of M1
Creation Date: 12Feb2015 @ 11:27:51 PM

MACHINE M1
SEES C1
VARIABLES
y
pc
INVARIANTS
inv1 : $pc \in L \land y \in S$
inv2 : $pc = l1 \Rightarrow y \in S$
Floyd's annotation
inv3 : $pc = l2 \Rightarrow y = B$
Floyd's annotation
inv4 : $x \in S \land x = A \land y \in S \Rightarrow y \in S$
checking precondition
inv5 : $y = B \Rightarrow y \in S \land y = B$
checking postcondition
EVENTS
Initialisation
begin
act1 : $y :\in S$
act2 : $pc := l1$
end
Event $e \mathrel{\widehat{=}}$
when
grd1 : $pc = l1$
then
act1 : $pc := l2$
act2 : $y :| (y' \in S \land y' \neq x)$
end
END

Appendix B. Complete Event-B Archive for the second case study

B.1. Context for SCP2

An Event-B Specification of DOMAIN_KNOWLEDGE_FOR_UNITS
Creation Date: 12Feb2015 @ 11:27:51 PM

CONTEXT DOMAIN_KNOWLEDGE_FOR_UNITS
SETS
$STATUS$
$inch$
$meter$
mph
kph
CONSTANTS
OK
KO
$inchtometer$
$mphtokph$
$valt$
$vspeed$
AXIOMS
axm1 : $partition(STATUS, \{OK\}, \{KO\})$
axm11 : $inchtometer \in inch \rightarrow meter$
axm12 : $mphtokph \in mph \rightarrow kph$
axm13 : $valt \in inch$
axm14 : $vspeed \in mph$
END

An Event-B Specification of DOMAIN_KNOWLEDGE_FOR_BUS
Creation Date: 12Feb2015 @ 11:27:51 PM

CONTEXT DOMAIN_KNOWLEDGE_FOR_BUS
EXTENDS DOMAIN_KNOWLEDGE_FOR_UNITS
SETS
bus
bus_type
CONSTANTS
unidirectional
bidirectional
tbus
s
a
abus
AXIOMS
axm1 : $partition(bus\_type, \{unidirectional\}, \{bidirectional\})$
axm2 : $tbus \in bus \rightarrow bus\_type \land abus \in bus \land tbus(abus) = unidirectional$
axm5 : $s \in bus \rightarrow kph \land s(abus) = mphtokph(vspeed)$
axm6 : $a \in bus \rightarrow meter \land a(abus) = inchtometer(valt)$
END

An Event-B Specification of CONTROL
Creation Date: 12Feb2015 @ 11:27:51 PM

CONTEXT CONTROL
SETS
STEPS
CONSTANTS
tocompute
computed
written
read
AXIOMS
axm1 : $partition(STEPS, \{tocompute\}, \{computed\}, \{written\}, \{read\})$
END

B.2. Machines and refinements for SCP2

An Event-B Specification of COM1
Creation Date: 12Feb2015 @ 11:27:51 PM

MACHINE COM1
SEES DOMAIN_KNOWLEDGE_FOR_UNITS
VARIABLES
alt
speed
display_speed
display_alt
computing
INVARIANTS
inv1 : $alt \in inch$
inv3 : $speed \in mph$
inv4 : $display\_speed \in kph$
inv5 : $display\_alt \in meter$
inv6 : $computing \in STATUS$
EVENTS
Initialisation
begin
act1 : $alt :\in inch$
act3 : $speed :\in mph$
act4 : $display\_speed :\in kph$
act5 : $display\_alt :\in meter$

act6 : $computing := KO$
end
Event *Computing_Aircraft_Info* $\widehat{=}$
when
grd1 : $computing = KO$
then
act1 : $alt :\in inch$
act2 : $speed :\in mph$
act3 : $computing := OK$
end
Event *Display_Aircraft_Info* $\widehat{=}$
when
grd1 : $computing = OK$
then
act1 : $computing := KO$
act2 : $display\_speed := mphtokph(speed)$
act3 : $display\_alt := inchtometer(alt)$
end
END

An Event-B Specification of COM10
Creation Date: 12Feb2015 @ 11:27:51 PM

MACHINE COM10
REFINES COM1
SEES DOMAIN_KNOWLEDGE_FOR_UNITS, CONTROL
VARIABLES
alt
speed
display_speed
display_alt
computing
step
alt_bus
speed_bus
read_alt
read_speed
INVARIANTS
inv1 : $step \in STEPS$
inv2 : $step = tocompute \Leftrightarrow computing = KO$
inv3 : $alt\_bus \in meter$
inv4 : $speed\_bus \in kph$
inv5 : $read\_speed \in kph$
inv6 : $read\_alt \in meter$
onto-inv7 : $step = written \Rightarrow alt\_bus = inchtometer(alt) \wedge speed\_bus = mphtokph(speed)$
onto-inv8 : $step = read \Rightarrow read\_alt = alt\_bus \wedge alt\_bus = inchtometer(alt) \wedge read\_speed = speed\_bus \wedge speed\_bus = mphtokph(speed)$
onto-inv9 : $step = tocompute \Rightarrow display\_alt = read\_alt \wedge read\_alt = alt\_bus \wedge alt\_bus = inchtometer(alt) \wedge display\_speed = read\_speed \wedge read\_speed = speed\_bus \wedge speed\_bus = mphtokph(speed)$
EVENTS
Initialisation
begin
act1 : $alt := valt$
act3 : $speed := vspeed$
act4 : $display\_speed := mphtokph(vspeed)$
act5 : $display\_alt := inchtometer(valt)$
act6 : $computing := KO$
act7 : $step := tocompute$
act8 : $alt\_bus := inchtometer(valt)$
act9 : $speed\_bus := mphtokph(vspeed)$
act10 : $read\_alt := inchtometer(valt)$
act11 : $read\_speed := mphtokph(vspeed)$
end
Event *Computing_Aircraft_Info* $\widehat{=}$

refines *Computing_Aircraft_Info*
when
grd1 : $computing = KO$
grd2 : $step = tocompute$
then
act1 : $alt :\in inch$
act2 : $speed :\in mph$
act3 : $computing := OK$
act4 : $step := computed$
end
Event *Display_Aircraft_Info* $\widehat{=}$
refines *Display_Aircraft_Info*
when
grd1 : $computing = OK$
grd2 : $step = read$
then
act1 : $computing := KO$
act2 : $display\_speed := read\_speed$
act3 : $display\_alt := read\_alt$
act4 : $step := tocompute$
end
Event *Write_Info_On_Bus* $\widehat{=}$
when
grd1 : $step = computed$
then
act1 : $alt\_bus := inchtometer(alt)$
act2 : $speed\_bus := mphtokph(speed)$
act3 : $step := written$
end
Event *Read_Infor_From_Bus* $\widehat{=}$
when
grd1 : $step = written$
then
act1 : $step := read$
act2 : $read\_alt := alt\_bus$
act3 : $read\_speed := speed\_bus$
end
END

An Event-B Specification of COM100
Creation Date: 12Feb2015 @ 11:27:51 PM

MACHINE COM100
REFINES COM10
SEES CONTROL, DOMAIN_KNOWLEDGE_FOR_BUS
VARIABLES
alt
speed
display_speed
display_alt
computing
step
read_alt
read_speed
speedbus
altbus
b
INVARIANTS
inv1 : $speedbus \in bus \rightarrow kph$
inv2 : $altbus \in bus \rightarrow meter$
inv3 : $altbus(b) = alt\_bus$
inv4 : $speedbus(b) = speed\_bus$
inv6 : $tbus(b) = unidirectional$
inv7 : $b \in bus$

EVENTS
Initialisation
  begin
    act1 : $alt := valt$
    act3 : $speed := vspeed$
    act4 : $display\_speed := mphtokph(vspeed)$
    act5 : $display\_alt := inchtometer(valt)$
    act6 : $computing := KO$
    act7 : $step := tocompute$
    act10 : $read\_alt := inchtometer(valt)$
    act11 : $read\_speed := mphtokph(vspeed)$
    act12 : $speedbus : |(speedbus' \in bus \rightarrow kph \wedge speedbus'(abus) = mphtokph(vspeed))$
    act13 : $altbus : |(altbus' \in bus \rightarrow meter \wedge altbus'(abus) = inchtometer(valt))$
    act14 : $b := abus$
  end
Event $Computing\_Aircraft\_Info \widehat{=}$
refines $Computing\_Aircraft\_Info$
  when
    grd1 : $computing = KO$
    grd2 : $step = tocompute$
  then
    act1 : $alt :\in inch$
    act2 : $speed :\in mph$
    act3 : $computing := OK$
    act4 : $step := computed$
  end
Event $Display\_Aircraft\_Info \widehat{=}$
refines $Display\_Aircraft\_Info$
  when
    grd1 : $computing = OK$
    grd2 : $step = read$
  then
    act1 : $computing := KO$
    act2 : $display\_speed := read\_speed$
    act3 : $display\_alt := read\_alt$
    act4 : $step := tocompute$
  end
Event $Write\_Info\_On\_Bus \widehat{=}$
refines $Write\_Info\_On\_Bus$
  when
    grd1 : $step = computed$
  then
    act1 : $altbus(b) := inchtometer(alt)$
    act2 : $speedbus(b) := mphtokph(speed)$
    act3 : $step := written$
  end
Event $Read\_Infor\_From\_Bus \widehat{=}$
refines $Read\_Infor\_From\_Bus$
  when
    grd1 : $step = written$
  then
    act1 : $step := read$
    act2 : $read\_alt := altbus(b)$
    act3 : $read\_speed := speedbus(b)$
  end
END

References

[1] R. Milner, A Calculus of Communicating Systems, Springer-Verlag, New York, Inc., Secaucus, NJ, USA, 1982.
[2] D. Martin, M. Paolucci, S. McIlraith, M. Burstein, D. McDermott, D. McGuinness, B. Parsia, T. Payne, M. Sabou, M. Solanki, et al., Bringing semantics to web services: the owl-s approach, in: Semantic Web Services and Web Process Composition, in: Lecture Notes in Computer Science, vol. 3387, 2005, pp. 26–42.
[3] D. Roman, U. Keller, H. Lausen, J. de Bruijn, R. Lara, M. Stollberg, A. Polleres, C. Feier, C. Bussler, D. Fensel, et al., Web service modeling ontology, Appl. Ontol. 1 (1) (2005) 77–106.
[4] Y. Aït-Ameur, A semantic repository for adaptive services, in: 2009 World Conference on Services — I, IEEE, 2009, pp. 211–218.

[5] D. Thevenin, J. Coutaz, Plasticity of user interfaces: framework and research agenda, in: Proceedings of INTERACT, vol. 99, 1999, pp. 110–117.
[6] J. Coutaz, G. Calvary, HCI and software engineering for user interface plasticity, in: J.A. Jacko (Ed.), Human–Computer Interaction Handbook: Fundamentals, Evolving Technologies, and Emerging Applications, third edition, CRC Press, 2012, pp. 1195–1220.
[7] J.-R. Abrial, L. Mussat, Introducing dynamic constraints in B, in: B98, 1998, pp. 83–128.
[8] J.-R. Abrial, Modeling in Event-B: System and Software Engineering, Cambridge University Press, 2010.
[9] J.E. Hopcroft, Introduction to Automata Theory, Languages, and Computation, Pearson Education, India, 1979.
[10] J.-R. Abrial, M.J. Butler, S. Hallerstede, T.S. Hoang, F. Mehta, L. Voisin, Rodin: an open toolset for modelling and reasoning in Event-B, Int. J. Softw. Tools Technol. Transf. 12 (6) (2010) 447–466.
[11] CADP-Team, Cadp official web site: http://cadp.inria.fr, 2013.
[12] H.J. Levesque, A logic of implicit and explicit belief, in: R.J. Brachman (Ed.), AAAI, AAAI Press, 1984, pp. 198–202.
[13] M. Uschold, Where are the semantics in the semantic web?, AI Mag. 24 (2003) 25–36, http://dl.acm.org/citation.cfm?id=958671.958674.
[14] E. Gabrilovich, S. Markovitch, Wikipedia-based semantic interpretation for natural language processing, J. Artif. Intell. Res. 34 (2009) 443–449.
[15] A. van Lamsweerde, L. Willemet, Inferring declarative requirements specifications from operational scenarios, IEEE Trans. Softw. Eng. 24 (1998) 1089–1114, http://dx.doi.org/10.1109/32.738341, http://dl.acm.org/citation.cfm?id=297569.297578.
[16] D. Bjorner, Software Engineering 1. Abstraction and Modelling, Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag, ISBN 978-3-540-21149-5, 2006.
[17] D. Bjorner, Software Engineering 2. Specification of Systems and Languages, Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag, ISBN 978-3-540-21150-1, 2006.
[18] D. Bjorner, Software Engineering 3. Domains, Requirements and Software Design, Texts in Theoretical Computer Science. An EATCS Series, Springer-Verlag, ISBN 978-3-540-21151-8, 2006.
[19] M. Jackson, P. Zave, Deriving specifications from requirements: an example, in: D.E. Perry, R. Jeffrey, D. Notkin (Eds.), Proceedings of the 17th International Conference on Software Engineering, Seattle, Washington, USA, April 23–30, 1995, ACM, 1995, pp. 15–24.
[20] P. Zave, M. Jackson, Four dark corners of requirements engineering, ACM Trans. Softw. Eng. Methodol. 6 (1) (1997) 1–30.
[21] P. Zave, Classification of research efforts in requirements engineering, ACM Comput. Surv. 29 (4) (1997) 315–321.
[22] D. Bjørner, A. Eir, Compositionality: ontology and mereology of domains, in: Concurrency, Compositionality, and Correctness. Essays in Honor of Willem-Paul de Roever, 2010, pp. 22–59.
[23] T.R. Gruber, Towards principles for the design of ontologies used for knowledge sharing, in: N. Guarino, R. Poli (Eds.), Formal Ontology in Conceptual Analysis and Knowledge Representation, Kluwer Academic Publishers, 1993.
[24] Y. Aït-Ameur, J.P. Gibson, D. Méry, On implicit and explicit semantics: integration issues in proof-based development of systems — version to read, in: Leveraging Applications of Formal Methods, Verification and Validation. Specialized Techniques and Applications — 6th International Symposium, Proceedings, Part II, ISoLA 2014, Imperial, Corfu, Greece, October 8–11, 2014, vol. 8803, Springer-Verlag, 2014, pp. 604–618.
[25] D. Garlan, B. Schmerl, Model-based adaptation for self-healing systems, in: Proceedings of the First Workshop on Self-healing Systems, WOSS '02, ACM, New York, NY, USA, 2002, pp. 27–32.
[26] D. Connolly, I. Horrocks, D. McGuinness, F. Patel-Schneider, A. Stein, Daml+oil reference description, World Wide Web Consortium.
[27] D. Brickley, R. V. Guha, RDF vocabulary description language 1.1: RDF schema, W3C Recommendation 10, 25 February 2014, available at: http://www.w3.org/TR/rdf-schema/.
[28] W. OWL Working Group, OWL 2 Web Ontology Language: Document Overview, W3C Recommendation, 27 October 2009, available at: http://www.w3.org/TR/owl2-overview/, 27 October 2009.
[29] P. Hitzler, M. Krötzsch, B. Parsia, P. F. Patel-Schneider, S. Rudolph (Eds.), OWL 2 Web Ontology Language: Primer, W3C Recommendation, 27 October 2009, available at: http://www.w3.org/TR/owl2-primer/, 27 October 2009.
[30] G. Pierra, H. Wiedmer, Industrial automation systems and integration parts library part 42: methodology for structuring part families, Tech. rep., Technical Report ISO DIS 13584-42, International Organization for Standardization, 30 May 1996, ISO/TC 184/SC4/WG2, 1996.
[31] G. Pierra, Context-explication in conceptual ontologies: the plib approach, in: Proceedings of the 10th ISPE International Conference on Concurrent Engineering, Enhanced Interoperable Systems, CE 2003, 2003.
[32] H. Knublauch, R.W. Fergerson, N.F. Noy, M.A. Musen, The protégé OWL plugin: an open development environment for semantic web applications, in: S.A. McIlraith, D. Plexousakis, F. van Harmelen (Eds.), The Semantic Web — ISWC 2004: Third International Semantic Web Conference, Proceedings, Hiroshima, Japan, November 7–11, 2004, in: Lecture Notes in Computer Science, vol. 3298, Springer, 2004, pp. 229–243.
[33] A free, open-source ontology editor and framework for building intelligent systems, http://protege.stanford.edu/.
[34] S.E.P. Bijan, Pellet: an owl dl reasoner, in: International Workshop on Description Logics, DL2004, 2004, pp. 6–8.
[35] V. Haarslev, R. Möller, Description of the RACER system and its applications, in: Working Notes of the 2001 International Description Logics Workshop, DL-2001, Stanford, CA, USA, August 1–3, 2001, in: CEUR Workshop Proceedings, vol. 49, 2001, CEUR-WS.org.
[36] V. Haarslev, R. Möller, RACER system description, in: Automated Reasoning, First International Joint Conference, Proceedings, IJCAR 2001, Siena, Italy, June 18–23, 2001, in: Lecture Notes in Computer Science, vol. 2083, Springer, 2001, pp. 701–706.
[37] B. Motik, KAON2 — scalable reasoning over ontologies with large data sets, ERCIM News 72 (2008), http://ercim-news.ercim.eu/kaon2-scalable-reasoning-over-ontologies-with-large-data-sets.
[38] S. Desprès, S. Szulman, Terminae method and integration process for legal ontology building, in: M. Ali, R. Dapoigny (Eds.), Advances in Applied Artificial Intelligence, 19th International Conference on Industrial, Engineering and Other Applications of Applied Intelligent Systems, Proceedings, IEA/AIE 2006, Annecy, France, June 27–30, 2006, in: Lecture Notes in Computer Science, vol. 4031, Springer, 2006, pp. 1014–1023.
[39] S. Handschuh, S. Staab, CREAM: creating metadata for the semantic web, Comput. Netw. 42 (5) (2003) 579–598.
[40] J. Broekstra, A. Kampman, SeRQL: an RDF query and transformation language, in: SWAD Europe Workshop on Semantic Web Storage and Retrieval, 2004.
[41] J. Trinkunas, Q. Vasilecas, A graph oriented model for ontology transformation into conceptual data model, Inf. Technol. Control 36 (1A) (2007) 126–132.
[42] S. Jean, G. Pierra, Y. Aït-Ameur, Domain ontologies: a database-oriented analysis, in: Web Information Systems and Technologies, International Conferences, WEBIST 2005 and WEBIST 2006. Revised Selected Papers, in: Lecture Notes in Business Information Processing, Springer, Berlin, Heidelberg, 2007, pp. 238–254.
[43] A. Farquhar, R. Fikes, J. Rice, The Ontolingua server: a tool for collaborative ontology construction, Int. J. Hum.-Comput. Stud. 46 (6) (1997) 707–727.
[44] C. Fankam, Y. Aït-Ameur, G. Pierra, Exploitation of ontology languages for both persistence and reasoning purposes — mapping plib, OWL and flight ontology models, in: WEBIST 2007 – Proceedings of the Third International Conference on Web Information Systems and Technologies, vol. WIA, Barcelona, Spain, March 3–6, 2007, INSTICC Press, 2007, pp. 254–262.
[45] J. Broekstra, A. Kampman, F.v. Harmelen, Sesame: a generic architecture for storing and querying rdf and rdf schema, in: Proceedings of the First International Semantic Web Conference on The Semantic Web, ISWC '02, Springer-Verlag, London, UK, 2002, pp. 54–68, http://dl.acm.org/citation.cfm?id=646996.711426.
[46] H. Dehainsala, G. Pierra, L. Bellatreche, Ontodb: an ontology-based database for data intensive applications, in: Proc. of the 12th Int. Conf. on Database Systems for Advanced Applications, DASFAA'07, in: LNCS, Springer, 2007.

[47] Z. Pan, J. Heflin, Dldb: extending relational databases to support semantic web queries, in: PSSS, 2003, pp. 109–113.
[48] S. Harris, N. Gibbins, 3store: efficient bulk RDF Storage, in: Proceedings of the 1st International Workshop on Practical and Scalable Semantic Systems, PPP'03, 2003, pp. 1–15.
[49] M.J. Park, J.H. Lee, C.H. Lee, J. Lin, O. Serres, C.W. Chung, An efficient and scalable management of ontology, in: Proceedings of the 12th International Conference on Database Systems for Advanced Applications, DASFAA'07, in: Lecture Notes in Computer Science, vol. 4443, Springer, 2007.
[50] M. Stocker, M. Smith, Owlgres: a scalable owl reasoner, in: The Sixth International Workshop on OWL: Experiences and Directions, 2008.
[51] G. Pierra, Context representation in domain ontologies and its use for semantic integration of data, J. Data Semant. 10 (2008) 174–211.
[52] P. Barlatier, R. Dapoigny, A type-theoretical approach for ontologies: the case of roles, Appl. Ontol. 7 (3) (2012) 311–356.
[53] R. Dapoigny, P. Barlatier, Modeling ontological structures with type classes in coq, in: Conceptual Structures for STEM Research and Education, 20th International Conference on Conceptual Structures, Proceedings, ICCS 2013, Mumbai, India, January 10–12, 2013, in: Lecture Notes in Computer Science, vol. 7735, Springer, 2013, pp. 135–152.
[54] Y. Aït-Ameur, H. Wiedmer, General resources, ISO-IS 13584-20, ISO, Genève, 1998, 88 pages.
[55] G. Pierra, Y. Aït-Ameur, E. Sardet, ISO, Genève, 2003 (660 pp.).
[56] G. Pierra, E. Sardet, ISO 13584-32 Industrial automation systems and integration Parts library Part 32: Implementation resources: OntoML: Product ontology markup language, ISO, 2010.
[57] ISO13584-42, Industrial automation systems and integration parts library part 42: Description methodology: methodology for structuring parts families, Technical report, International Standards Organization, 1998..
[58] S. Owre, N. Shankar, J.M. Rushby, The PVS specification language, Technical report, SRI International, June 14, 1993.
[59] Y. Bertot, P. Castéran, Interactive Theorem Proving and Program Development: Coq'Art: The Calculus of Inductive Constructions, Springer, 2004.
[60] A. Platzer, J.-D. Quesel, Keymaera: a hybrid theorem prover for hybrid systems (system description), in: Automated Reasoning, Springer, 2008, pp. 171–178.
[61] E.M. Clarke, O. Grunberg, D.A. Peled, Model Checking, MIT Press, 2000.
[62] G. Booch, J. Rumbaugh, I. Jacobson, The Unified Modeling Language User Guide, Addison-Wesley, 1999.
[63] R. France, A. Evans, K. Lano, B. Rumpe, The UML as a formal modeling notation, Comput. Stand. Interfaces 19 (7) (1998) 325–334.
[64] K. Bontcheva, V. Tablan, D. Maynard, H. Cunningham, Evolving GATE to meet new challenges in language engineering, Nat. Lang. Eng. 10 (3/4) (2004) 349–373.
[65] H. Cunningham, D. Maynard, K. Bontcheva, V. Tablan, N. Aswani, I. Roberts, G. Gorrell, A. Funk, A. Roberts, D. Damljanovic, T. Heitz, M. A. Greenwood, H. Saggion, J. Petrak, Y. Li, W. Peters, Text Processing with GATE (Version 6), 2011, http://tinyurl.com/gatebook.
[66] S. Handschuh, R. Volz, S. Staab, Annotation for the deep web, IEEE Intell. Syst. 18 (5) (2003) 42–48.
[67] A. Chebotko, Y. Deng, S. Lu, F. Fotouhi, A. Aristar, An ontology-based multimedia annotator for the semantic web of language engineering, Int. J. Semantic Web Inf. Syst. 1 (1) (2005) 50–67.
[68] Y. Lu, H. Panetto, Y. Ni, X. Gu, Ontology alignment for networked enterprise information system interoperability in supply chain environment, Int. J. Comput. Integr. Manuf. 26 (1–2) (2013) 140–151.
[69] L.S. Mastella, Y. Aït-Ameur, S. Jean, M. Perrin, J. Rainaud, Semantic exploitation of engineering models: an application to oilfield models, in: A.P. Sexton (Ed.), Dataspace: The Final Frontier, 26th British National Conference on Databases, Proceedings, BNCOD 26, Birmingham, UK, July 7–9, 2009, in: Lecture Notes in Computer Science, vol. 5588, Springer, 2009, pp. 203–207.
[70] N. Belaid, S. Jean, Y. Aït-Ameur, J. Rainaud, An ontology and indexation based management of services and workflows application to geological modeling, Int. J. Electron. Bus. Manag. 9 (4) (2011) 296–309.
[71] D.S. Zayas, A. Monceaux, Y. Aït-Ameur, Knowledge models to reduce the gap between heterogeneous models: application to aircraft systems engineering, in: R. Calinescu, R.F. Paige, M.Z. Kwiatkowska (Eds.), 15th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2010, Oxford, United Kingdom, 22–26 March, 2010, IEEE Computer Society, 2010, pp. 355–360.
[72] J. Goguen, R. Burstall, Introducing institutions, in: E. Clarke, D. Kozen (Eds.), Logics of Programs, in: Lecture Notes in Computer Science, vol. 164, Springer, Berlin, Heidelberg, 1984, pp. 221–256.
[73] M. Codescu, T. Mossakowski, O. Kutz, A categorical approach to ontology alignment, in: P. Shvaiko, J. Euzenat, M. Mao, E. Jiménez-Ruiz, J. Li, A. Ngonga (Eds.), Proceedings of the 9th International Workshop on Ontology Matching Collocated with the 13th International Semantic Web Conference, ISWC 2014, Riva del Garda, Trentino, Italy, October 20, 2014, in: CEUR Workshop Proceedings, CEUR-WS.org, vol. 1317, 2014, pp. 1–12, http://ceur-ws.org/Vol-1317.
[74] C. Lange, O. Kutz, T. Mossakowski, M. Grüninger, The distributed ontology language (DOL): ontology integration and interoperability applied to mathematical formalization, in: J. Jeuring, J.A. Campbell, J. Carette, G.D. Reis, P. Sojka, M. Wenzel, V. Sorge (Eds.), Intelligent Computer Mathematics — 11th International Conference, AISC 2012, 19th Symposium, Calculemus 2012, 5th International Workshop, DML 2012, 11th International Conference, MKM 2012, Systems and Projects, Held as Part of CICM 2012, Proceedings, Bremen, Germany, July 8–13, 2012, in: Lecture Notes in Computer Science, vol. 7362, Springer, 2012, pp. 463–467, http://dx.doi.org/10.1007/978-3-642-31374-5.
[75] R. Milner, Communication and Concurrency, Prentice Hall, Inc., Upper Saddle River, NJ, USA, 1989.
[76] Y. Aït-Ameur, A. Chebieb, Invited talk: Checking system substitutability: an application to interactive systems, in: Modeling Approaches and Algorithms for Advanced Computer Applications, in: Studies in Computational Intelligence, vol. 488, Springer, 2013, p. 3.
[77] A. Chebieb, Y. Aït-Ameur, Formal verification of plastic user interface exploiting domain ontologies, in: Proceedings of the 9th International Symposium on Theoretical Aspects of Software Engineering, IEEE Computer Society Press, 2015.
[78] OASIS, Web Services Business Process Execution Language Version 2.0, http://bpel.xml.org/, April 2007.
[79] I. Aït-Sadoune, Y. Aït-Ameur, A proof based approach for modelling and verifying web services compositions, in: 14th IEEE International Conference on Engineering of Complex Computer Systems, ICECCS 2009, Potsdam, Germany, 2–4 June 2009, IEEE Computer Society, 2009, pp. 1–10.
[80] I. Aït-Sadoune, Y. Aït-Ameur, Stepwise design of BPEL web services compositions: an event_b refinement based approach, in: Software Engineering Research, Management and Applications 2010 [selected papers from the 8th ACIS International Conference on Software Engineering Research, Management and Applications], SERA 2010, Montreal, Canada, May 24–26, 2010, in: Studies in Computational Intelligence, vol. 296, Springer, 2010, pp. 51–68.
[81] I. Aït-Sadoune, Y. Aït-Ameur, Stepwise development of formal models for web services compositions: modelling and property verification, in: Transactions on Large-Scale Data- and Knowledge-Centered Systems, vol. 10, 2013, pp. 1–33.
[82] ClearSy, Aix-en-Provence (F), Atelier B, http://www.atelierb.eu, 2002.
[83] D. Bjørner, M.C. Henson (Eds.), Logics of Specification Languages, EATCS Textbook in Computer Science, Springer, 2007.
[84] Z. Manna, A. Pnueli, The Temporal Logics of Reactive and Concurrent Systems — Specification, Springer-Verlag, 1992.
[85] Z. Manna, A. Pnueli, The Temporal Logics of Reactive and Concurrent Systems — Safety, Springer-Verlag, 1995.