Using Meta-Knowledge at the Application Level

Yannis Kalfoglou YANNISK@DAI.ED.AC.UK
David Robertson DR@DAI.ED.AC.UK
School of Artificial Intelligence, Division of Informatics,
University of Edinburgh, 80 South Bridge,
Edinburgh EH1 1HN, Scotland

Austin Tate A.TATE@ED.AC.UK
Artificial Intelligence Applications Institute,Division of Informatics,
University of Edinburgh,80 South Bridge,
Edinburgh EH1 1HN, Scotland

Abstract

Ontologies have become popular in the Artificial Intelligence community as a way to standardise representation of domain knowledge. Despite their advocated use in areas such as knowledge sharing and reuse there is little discussion in the community regarding their application in other areas. In this paper we explore the use of ontologies in improving systems engineering reliability by consistency checking with respect to ontological axioms - making it possible to reason about the correctness of an application with respect to ontological constraints. We applied this approach to diverse areas, each of which demonstrates a different use: deploying ontological axioms in business process modelling and enriching the axiomatisation of an Air Campaign Planning(ACP)-based application; identification of ontological constraints in ecological modelling; and evaluation of ontologies on system dynamics theory at the application level.

1. Introduction

The role of an ontology is to represent explicitly a shared understanding of the important concepts in some domain of interest. They describe a conceptualisation of a domain in a knowledge representation formalism that can be used and shared among different agents. Their main objective is to promote knowledge sharing and reuse and the majority of applications reported in the literature is focussed on how to deliver them(see, for example, (Uschold & Gruninger, 1996), (Valente, Russ, MacGrecor, & Swartout, 1999), (Gangemi, Pisanelli, & Steve, 1998)).

However, there are few applications in a key area identified by Uschold and Gruninger in (Uschold & Gruninger, 1996), that of systems engineering reliability, where the formal representation of domain knowledge enables the use of consistency checking resulting in more reliable systems. Some of the reasons for this shortage are the level of formality required and the efficiency of the way in which this consistency checking can be implemented. Most applications reported in the literature use informal or semi-formal ontologies (Benjamins & Fensel, 1998) and in situations where formal ontologies were used(Polyak, Lee, Gruninger, & Menzel, 1998) consistency checking was not their objective. The role anticipated(but seldom delivered) for the ontological axioms in a formal ontology is to restrict the possible interpretations ontological constructs could have. The forms of description allowed by those using the ontology are required to be consistent with these formal constraints but seldom is checked when building systems using ontologies.

Moreover evaluation of ontologies is rarely discussed in the literature. Some guidelines are available((Gomez-Perez, 1996), (Gruninger, 1996)) and broad comparisons have been made with knowledge-based systems evaluation (Gomez-Perez, Juristo, & Pazos, 1995), but there are no detailed reports or applications on how these can be applied.

On the other hand, there is a need to verify the early phases of systems development with respect to domain knowledge. Most error checking techniques concentrate on how to detect errors with respect to the mathematical language underpinning the formal model, see for example (Friedman & Voas, 1995), for instance the type checking. However, during, the early phases of software development the most difficult errors to detect are those related to domain knowledge. Specifications and programs can be mathematically elegant but reflect a misunderstanding of domain knowledge. We call these sorts of errors conceptual errors. In order to detect them we need more information(which we show that formal ontologies can partially provide) and a way of harnessing this information to check for errors.

In this paper we present a novel way of deploying ontological axioms, now being used in ontology building, to detect conceptual errors in the early phases of software development, such as specifications. This results in a conformance check of the application to the ontology that is based on. In doing this, domain knowledge which previously was tacit in the application becomes explicit. We illustrate the approach in figure 1.

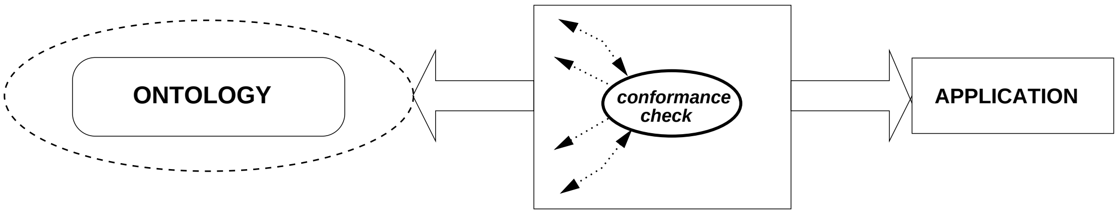

Figure 1: Conformance check of application to ontology.

The dashed line surrounding the ontology denotes the current ontologies' evaluation scope and the double arrow that connects the application with the ontology symbolises the use of ontologies at the application level. As we can see from figure 1 our mechanism is placed in between these two to enforce the conformance check of application to ontology. There can be variations of this diagram with respect to possible instantiations of the ontology/application pair. In section 6 we elaborate on these instantiations along with examples to demonstrate their use. We also investigated an elaboration of this check in the area of evaluation of ontologies. That is, we evaluated the ontology for specific kinds of inconsistencies at the application level rather than the knowledge level where the available evaluation guidelines are applied. An example of this is presented in section 6.

In section 2 we discuss related work classified in three broad areas: use of ontological commitments in knowledge bases(section 2.1), evaluate ontologies(section 2.2), and deploy formal specifications to verify and validate(section 2.3) knowledge-based systems(hereafter, KBS). In section 2.4 we present how we bring together the most promising ideas of these research efforts to materialise our approach in formally verifying specifications of KBS by deploying ontologies. To do that, we present a formal theory for representing ontological axioms in section 3 in a way that can be used in combination with an inference engine described in section 4 as input to the error checking mechanism of section 5. The multi-layer architecture is presented in the same section(5) whereas the areas to which we have apply are described in section 6: evaluation of ontologies at the application level in section 6.1, use of ontological axioms in section 6.2, identification of ontological constraints from an application in section 6.3 and use of domain-specific constraints in an ontology-tailored application in section 6.4. We discuss limitations of this work and elaborate on further uses in section 7. We conclude our work in section 8.

2. Related work

We use ontologies to detect conceptual errors in specifications. There are many areas to which our research could be thought as relevant. The closest ones though, belong to two broad fields: ontologies and error checking in specifications. The former is of particular interest to us since we use ontological products, like axioms, to implement our checks. Hence, we further scrutinised and classified related work to two loosely defined areas: use of ontological commitments(section 2.1) and evaluation of ontologies(section 2.2). The latter is a broad field and attracts the interest of researchers from various disciplines: software engineering, knowledge engineering, hardware design, etc. To keep the focus on the most relevant issues, we identified an area of error checking in specifications that shares common ground with our research: the use of formal specifications in the verification and validation(hereafter, V&V) of KBS. We present the promising ideas from this area in section 2.3 whereas in 2.4 we summarise the whole section by presenting our approach in accordance with the areas presented here.

2.1 Using ontological commitments

One of the early contributions and among the few originating from systems engineering was that of the `Comet`(Mark, Tyler, McGuire, & Schossberg, 1992) and `Cosmos`(Mark, Schlossberg, Tyler, & McGuire, 1994) systems. Both systems aim at developing knowledge bases by capturing the set of ontological commitments that define the interdependencies among key terms in the ontology. Their role is to assess the impact of changes in their world and provide context-specific guidance to their users on what modules may be relevant to include in the design, and what design modifications will be required in order to include them(Mark, Dukes-Schlossberg, & Kerber, 1995). The key idea behind this work was to make use of the ontological commitment expressed by the underlying ontology in the system's development process. The importance of ontological commitment is stressed by Guarino in (Guarino, Carrara, & Giaretta, 1994) where it is pointed out that ontologies must clearly express ontological commitments if the terms they specify are to be used without misinterpretation. However, the issue of where these commitments come from and, most importantly, how are they verified was not addressed.

In the context of the TOVE project(EIL, 1995), ontological commitments played a key role in the definition of an ontology's competence: a set of queries that the ontology can answer. These questions, called competency questions, were used to evaluate the expressiveness of the ontology that is required to represent them and characterise their solutions(Gruninger & Fox, 1995). They do not generate ontological commitments but are used to evaluate them. The TOVE ontologies are built on top of foundational theories such as situation calculus. Foundational theories provide the semantics for the ontology and their axioms serve as a basis for the implementation of competency questions. Despite the fact that foundational theories were used to reason about the terminology of the ontologies using their axiomatisation they are generic and were only applied within the scope of evaluation of ontologies depicted in figure 1. This does not contribute to the problem of how to evaluate ontologies when applying them as Fox and Gruninger pointed out in (Fox & Gruninger, 1997): "[...] by providing formal definitions of objects and their relations and attributes, we make it possible for users to understand their intended meaning. Though this does not guarantee that programs that access an ontology will interpret the results correctly".

In the DISCOVER project((Waterson & Preece, 1999)), the role of ontological commitment was further analysed and operationalised. The authors state that ontological commitment is a key issue for knowledge sharing and reuse and they applied existing verification techniques from the KBS literature to check the commitment of a knowledge base to an ontology. In that project the role of the ontology was to act as a background body of knowledge against which a knowledge base can be validated.

2.2 Evaluating ontologies

In the area of evaluation of ontologies Gomez-Perez discuss in (Gomez-Perez, 1996) criteria to verify knowledge sharing technology. The importance of evaluation is stressed by the same author in (Gomez-Perez, 1995): "[...] it is unwise to implement a software application that relies on ontologies written by others without first evaluating and assessing their definitions and axioms". She provides guidelines on how to prove consistency of an ontological definition by proving that the definition is internally and metaphysically consistent. Internal consistency requires that the informal and formal definitions of an ontology to have the same meaning. The most interesting case of consistency is the metaphysical one which is defined by Gomez-Perez as proving that there is no contradiction in the interpretation of the formal definition with respect to the real world. That is, to prove compliance of the world model with the world modelled formally. This sort of consistency is difficult to enforce in a mechanised manner and the manual check is based on ontological commitments explicitly made by the ontology.

Recently, Gomez-Perez has proposed an error typology which classifies potential errors made during structuring knowledge in taxonomies(Gomez-Perez, 1999). Although the categories proposed are 'classes-biased' in the sense that they talk about a particular ontology design formalism, that is, frames, there are two types identified that are similar to what we call a conceptual errors In particular, the author defines *semantic errors* as "incorrect semantic classification[...]classes a concept as a subclass of a concept to which it does not really belong". Further, she identifies *incompleteness errors* as errors that occur whenever "concepts are classed without accounting for them all[...]concepts existing in the domain are overlooked." These efforts in identifying potential errors which might occur during ontology construction are particularly useful when implementing the ontology. In our work, however, we are mostly concerned with the deployment of ontology in various applications and detection of errors which could occur when misusing that ontology. Since there can be applications that are not using classes, at least at their specification level, we implement our checks using a neutral formalism presented in section 3. This sort of work falls in the category identified by Gomez-Perez as 'future work': "(a)develop independent tools for evaluating ontologies built under a given knowledge representation paradigm, (b)create application-dependent and end-user methods to judge the adequacy of an ontology for a given application."

2.3 Formal specifications of KBS

We now shift to an application area related to the end-product of our research: V&V of KBS. This is a long-standing theme in the AI research agenda. *Verification* is the process aimed at demonstrating whether a system meets its specified requirements. This is often referenced as "building the system right" (O'Keefe, Balci, & Smith, 1987). *Validation* is a process aimed at demonstrating whether a system meets its user's true requirements, also referred to as "building the right system". V&V can be viewed as a set of requirements and an associated process in which the techniques are applied, as part of the whole development process(Meseguer & Preece, 1995).

Much of the work done in V&V KBS is based on the assumption that, because KBS are difficult to specify, practical V&V techniques should not depend upon the existence of detailed specification documents(Meseguer & Preece, 1995). This is seen in the work of verification by anomaly checking(Preece, Shinghal, & Batarekh, 1992). The authors provide a set of verification properties and compare a set of tools against them. In particular, they check KBS for domain-independent anomalies such as inconsistency and incompleteness. More verification properties are presented in (Preece & Shinghal, 1994) along with the detection of errors in actual KBS. However, most of the validation approaches developed for KBS assume the existence of an implemented system as in prototyping. This late validation process has some drawbacks the most important being that when an acceptable system is deemed to have been produced, it is not clear what the system actually does or it may fail to do(Meseguer & Preece, 1995). In addition, verification of KBS against some domain-independent properties does not solve the problem of demonstrating its correctness. As Meseguer and Preece report:

> "An example of a domain-independent property is consistency, which means that from a consistent input the KBS cannot produce a contradictory output. Domain-independent properties appear as prerequisites for adequate functioning of a KBS, and they should be tested. However, although they are necessary they are not sufficient because they say little about the actual KBS correctness"(Meseguer & Preece, 1995)

To overcome these problems with the early validation and cope with the verification of KBS correctness, the use of formal specifications(Meseguer & Preece, 1995) has been investigated. Formal specification techniques provide levels of description which support both V&V while V&V techniques feed back to assist the development of the specifications. Their contribution was studied from the viewpoint of the dominant techniques for V&V of KBS. We summarise them below as reported in the review of Meseguer and Preece: *inspection*, *static verification*, *empirical testing*, and *empirical evaluation*. *Inspection* refers to the detection of semantically incorrect knowledge in the KB and is usually performed manually by human experts. *Static verification* checks the KB for anomalies. These are static patterns that suggest the presence of an error in the encoded knowledge. *Empirical testing* involves execution of the system on sample data sets to check its correctness. For complete correctness the testing has to be exhaustive but this is not computationally feasible in real systems. Two kinds of testing are identified: *structural* testing which aims at executing as many of the KBS components as possible, and *functional* testing which checks the function of a KBS by comparing its observed input/output relationship with that specified in the requirements. *Empirical evaluation* deals with issues such as technical performance, acceptability, inclusion in the organisation, responsibility issues. It is a human activity which is highly application-dependent.

In the inspection process, formal specifications of a KBS can alleviate the problem of ambiguity regarding the interpretation of informal specifications, and used for direct validation from domain experts. However, it is argued that the formal notations employed in the specifications impose a comprehension problem for domain experts who are unfamiliar with them. As a solution the inspection of the semi-formal conceptual model which underpins the KBS development has been proposed.

Static verification is an area where formal specification techniques have been applied most. They allow verification to be performed on the same specification and to check either the specification or domain-dependent properties. In addition, some formal specification languages provide additional opportunities for verification. For example, the modular architecture and declaration of hierarchies of types provided by languages such as DESIRE(Treuer & Wetter, 1993), KARL and $(ML)^2$(Fensel & van Harmelen, 1994) allow to check for additional properties, like violation of modularity, and type mis-matches.

Formal specifications in empirical testing allow developers to test the specification itself. This helps them to assess whether they are specifying the intended system. It might require symbolic execution of the specification but specially designed specification languages(Fensel & van Harmelen, 1994) provide this feature. However, this does not waive the need to test the implemented system because the transformation from specification to implementation is usually done manually and new errors can be introduced.

The area of empirical evaluation is one which seems to have benefited little from the use of formal specifications. The is no experience of applications of formal specification in this V&V aspect, but given the benefits that formal specifications may offer, it is foreseeable that they may be applied in the evaluation process of KBS.

Although the role of formal specifications in the KBS development has been praised by many and more uses are envisaged in the foreseeable future there is an area that more work is required. As Meseguer and Preece report:

> "[...] the question of verifying the specifications themselves is still open, as few of the existing special purpose KBS specification languages have well developed proof techniques at present. If general purpose specification languages are used, then their existing proof techniques apply, but these have been found to entail a great deal of labour when KBS are specified"(Meseguer & Preece, 1995)

We will revisit this point in the next section where we connect the areas investigated in the last three section with our approach in deploying ontologies to check for conceptual errors in formal specifications of KBS.

2.4 Deploying at the application level: a unifying approach

In the next section we present a generic representation formalism for ontological axioms to enable their use in the multi-layered architecture discussed in section 5. This can solve some of the problems discussed in sections 2.1 and 2.2. Verification of ontological commitments(recognised in (Mark et al., 1995)) can be done by regarding them as a layer in the structure we propose which is then checked exhaustively against constraints on the layer above; the problem of guaranteeing the correct interpretations of formal definitions by programs that access an ontology(faced by Fox and Gruninger) can be contained by transforming the ontological axioms to the constraints format we adopt and placing them along with the programs in the layered structure to enable automatic checking of a program's conformance to the axiomatisation during its execution.

Full metaphysical consistency checking(as described by Gomez-Perez) is impossible to perform formally because it requires an accurate model of the real world. However, the layering approach we propose can help engineers to judge the correctness of their models with respect to ontological constraints in an integrated environment. These are actually the "application-dependent and end-user methods to judge the usability and utility of an ontology" that Gomez-Perez call for.

Figure 2 is an aid to visualising the scope of our research with respect to previous work. The left and middle parts are taken directly from the literature, on ontologies and V&V of KBS respectively. The right part is a view of our research as a result of combining the other two parts.

The left part is taken from the work of Valente and Breuker(Valente & Breuker, 1996) on core ontologies. It shows the interrelations between different elements in the ontology domain. As the authors describe: "Conceptualisations and domains are abstract things (represented by irregular polygons) while knowledge representations and ontologies are concrete things (represented by rectangles), for example, sentences in some symbolic language. Their interrelations are represented as ovals, and the direction of reading is indicated by the direction of the arrows. These interrelations are as follows: (i) an ontology describes a conceptualisation using definitions of the elements of the conceptualisation (objects, concepts, relations); (ii) a conceptualisation provides ontological commitments which are used in elaborating knowledge representations, and these commitments are embedded either in the knowledge representation language, or in the knowledge base; (iii) a knowledge representation represents a domain (a part of the world).".

The middle part is taken from the work of Meseguer and Preece on the use of formal specifications in the KBS development(Meseguer & Preece, 1995). The description is as follows: "[...] the informal description - typically called the 'conceptual model' in various methodologies - is iteratively refined to create a more precise formal specification, with both descriptions undergoing gradual modification during the process. The formal specification keeps the structure and vocabulary of the conceptual model. It is kept to make easier the

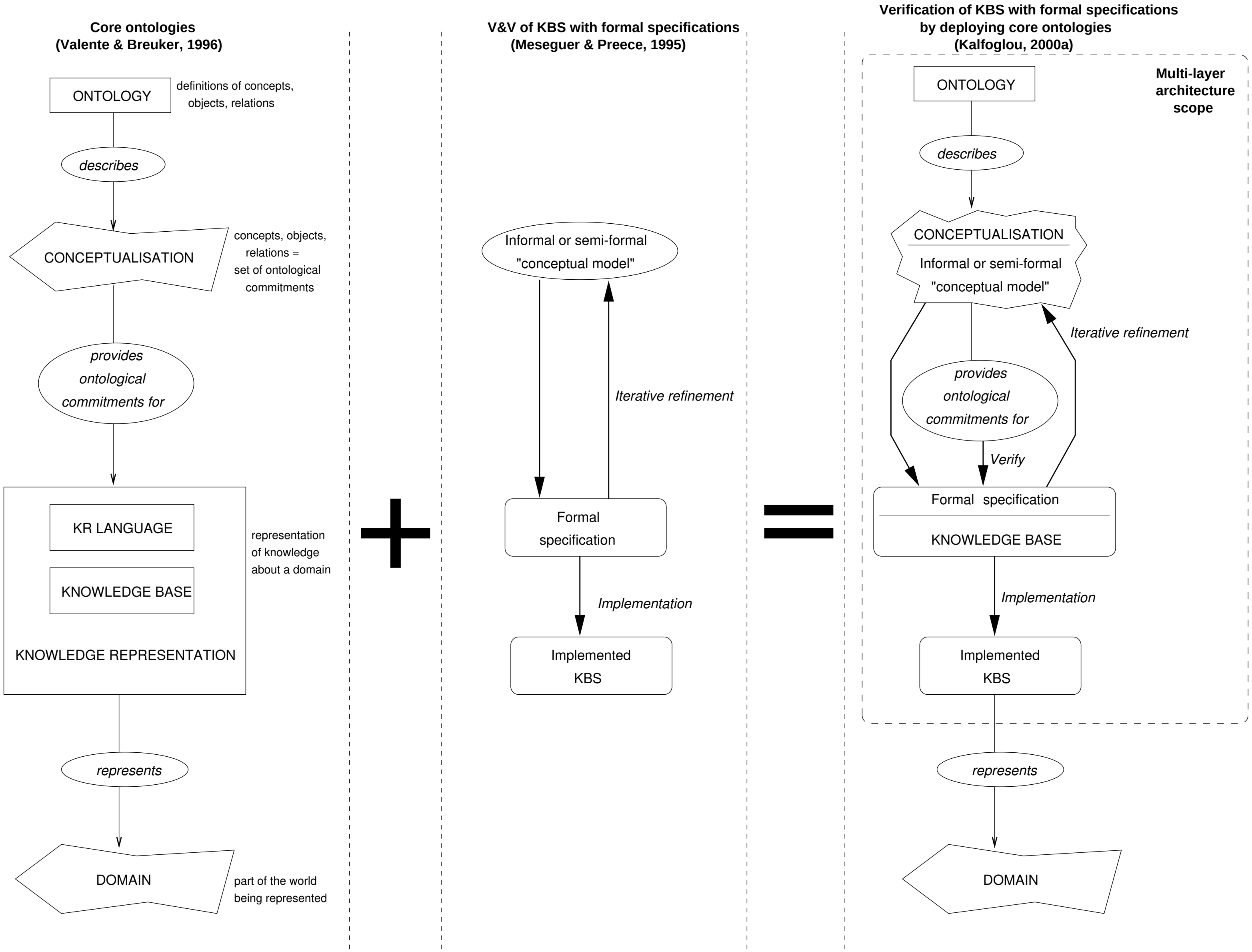

Figure 2: Verification of KBS with formal specifications by deploying core ontologies.

communication with domain experts in the development and validation process. The formal description will form the basis for the implementation.".

We combine these two parts to produce a single diagrammatic view of our research. As we can see from the right part, the role of the ontology has not changed: it describes a conceptualisation using definitions of the elements of that conceptualisation. This corresponds to the "conceptual model" of Meseguer and Preece's diagram. It provides ontological commitments which are embedded in the knowledge base. It can also be iteratively refined to a formal specification as Meseguer and Preece describe in their diagram. In addition, our research allows for verification of these commitments in the formal specification. As in Meseguer and Preece's diagram it forms the basis for the implementation of a KBS which represents a domain according to Valente and Breuker's diagram. That is because the whole process of constructing the initial "conceptual model" started from a given ontology, which by definition captures domain knowledge. The dashed line surrounding the elements(ontology, conceptualisation, ontological commitments, formal specification and KBS) denotes the scope of our multi-layer architecture(section 5). By using this mechanism we are able to verify the ontological commitment in various phases of ontology deployment: in the ontology itself by checking the correctness of its definitions against the ontological axioms, down to the formal specification of the system to be implemented by verifying its conformance to the underlying ontology.

3. Error detection theory

We introduce the representation of ontological axioms in the context of predicate calculus in 3.1 and then we elaborate on the reasoning theory we developed in 3.2 which is used as a basis for the inference engine of section 4.

Predicate calculus is not the language used in most KBS projects but it is part of the lineage of most formal KBS languages and has a direct correspondence to standard interchange formats such as KIF(Genesereth & Fikes, 1992). There are also well known routes for translation to Horn clauses which, via Prolog, we use as an executable specification language. Although not so closely targeted on KBS applications as other formalisms it turned out to be appropriate for the ontological examples we encountered, so a shift to a more specialised language was unnecessary for us. Readers interested in more specialised languages should refer to (Gomez-Perez & Benjamins, 1999) and (Fensel, Horrocks, van Harmelen, Decker, Erdmann, & Klein, 2000). This said, it is important to remember that a similar method could be used with other formal languages provided they supply the same form of close connection between ontological constraints and executable specifications.

3.1 Representing ontological axioms

The importance of having defined axioms in an ontology has been stressed by Gruber since the early days of ontology research:

> "Ontologies are often equated with taxonomic hierarchies of classes, class definitions, and the subsumption relation, but ontologies need not be limited to these forms. Ontologies are also not limited to conservative definitions, that is, definitions in the traditional logic sense that only introduce terminology and do not

add any knowledge about the world. To specify a conceptualisation one needs to state axioms that do constrain the possible interpretations for the defined terms."(Gruber, 1993)

These ontological axioms adopt the following formal representation:

$$(\alpha) \quad \forall X1, ..., Xn.G \rightarrow C$$

where G is a unit goal and $X1, ...Xn$ are all variables in G, and C is a condition composed of logical connectives($\wedge, \vee, \neg$) and/or unit goals. The condition C must be composed of valid ontological constructs and it must be true when the unit goal G is true.

So, for example, an axiom of a formal ontology, the Process Interchange Format(PIF) ontology((Lee, Gruninger, Jin, Malone, Tate, Yost, & other members of the PIF working group, 1998)) given in textual format:

> "An object can *participate in* an activity only at those timepoints at which both the object exists and the activity is occurring"

can be written according to formula (α) in predicate calculus as follows:

$$\forall O, A, T.participates\_in(O, A, T) \rightarrow exists\_at(O, T) \wedge is\_occurring\_at(A, T).$$

The role of this axiom is to restrict possible interpretations of the ontologically defined relation *participates_in*. So, whenever someone using the PIF ontology describes the relation in a way that does not conform to the axiomatised definition this will reveal a potential discrepancy.

In cases where the unit goal G is a composite one we transform it to a monadic literal since we are interested in having monadic predicates at the preconditions of axioms. For instance, in the CHEMICALS ontology((Aguado, Banon, Bateman, Bernandos, Fernandez, Gomez-Perez, Nieto, Olalla, Plaza, & Sanchez, 1998)) the axiomatised definition of the melting point for halogens given in textual format:

> "The highest melting point for halogens is 302C"

is written in predicate calculus, taken directly from (Aguado et al., 1998), as follows:

$$\forall H, M.halogen(H) \wedge melting\_point(H, M) \rightarrow M \leq 302 * degree\_celsius.$$

We apply the following rewrite rules:

$$A \wedge B \rightarrow C \Rightarrow A \rightarrow (B \rightarrow C) \Rightarrow A \rightarrow \neg(B \wedge \neg C)$$

to break the conjunction in the precondition. The '$\Rightarrow$' operator used to denote the rewrite of the original part to the transformed part. The axiom is now written, according to formula (α), as follows:

$$\forall H, M.halogen(H) \rightarrow \neg(melting\_point(H, M) \wedge \neg(M \leq 302 * degree\_celsius)).$$

We are interested in proofs over existentially quantified goals, so the formula (α) is transformed into a normal form:

$$(\beta)\ \ \forall X1,...,Xn.G \rightarrow C \Rightarrow \neg\exists X1,...Xn.G \wedge\neg C$$

We then identify the predicate G derived from the left hand side of the original implication of formula (β) and lose the existential quantifier and outer negation. These expressions will be used to test for errors on goals in the proof as we describe in section 5 and we define the clause `error/2` to represent them. Hence, the right part of formula (β) will be:

$$(\gamma)\ \ \neg\exists X1,...,Xn.G \wedge \neg C \Rightarrow error(G, \neg C).$$

So, for the ontological axiom on the melting point of halogens given above a translation to normal form will be:

$$(\delta)\ \forall H, M.halogen(H) \rightarrow \neg(melting\_point(H,M)\wedge\neg(M \leq 302{*}degree\_celsius))$$
$$\Rightarrow \neg\exists H, M.halogen(H)\wedge(melting\_point(H,M)\wedge\neg(M \leq 302{*}degree\_celsius)).$$

As in formula (γ), we transform formula (δ) to the specific error format:

$$(\epsilon)\ error(halogen(H),(melting\_point(H,M)\wedge \neg(M \leq 302 * degree\_celsius))).$$

The use of this error condition, which we call an ontological constraint, is the following: whenever this condition is true with respect to the program that uses the CHEMICALS ontology we have a conceptual error occurrence. That is, for a halogen H, we can prove that its melting point M, is not less than or equal to 302 degrees as it must be.

In the same manner, for the PIF axiom on the definition of the *participates_in* relation given above, we derive its ontological constraint in the form of an error condition:

$$(\zeta)\ error(participates\_in(O,A,T), \neg(exists\_at(O,A)\wedge is\_occurring\_at(A,T))).$$

This condition can be given a declarative reading: we have an error occurrence with respect to the *participates_in* relation used in a program that adopts the PIF ontology, whenever we can prove that an object O participates in an activity A at a timepoint T, and it is not provable that object O exists at activity A, occurring at timepoint T.

3.2 Reasoning theory

In order to reason about the conceptual errors found in programs that adopt ontologies and the correctness of ontological constraints(in the error condition format described above) we deployed to detect them, we developed a logical theory. In this theory we have the following standard constructs which we assume to be constants in the expressions which follow:

Let Gs be the set of all goals in the program(i.e., a specification) defined as follows:

$Gs = \{G1, ..., Gn\}$

Let Es be the abstract set of all possible errors on goals in the specification. This set refers to all possible violations of the ontological constructs used in the specification. It is impossible to know beforehand what these might be, but we represent them as: $e(G, E)$ where E is an error for the goal G. Every goal in the specification can have an arbitrary number of subgoals and their associated errors.

We define the clause: $error(G, E)$ to denote that E is an error condition defined on goal, G. This is the error condition format introduced in the previous section, and it represents an ontological constraint defined over the goal G.

We then identify three cases in the reasoning process: the first one refers to situations where no conceptual errors are found in the specification; the second cover cases where we found conceptual errors; and the third one is an elaboration of the two previous cases: we found conceptual errors but they might be erroneously reported depending on the correctness of error conditions(i.e., ontological constraints) and/or specification. Each case is described below:

3.2.1 Case A: no conceptual errors found

Textual description:
We found no conceptual errors. We cannot prove - via the inference engine described in section 5 - the error condition(s)(even the negative ones) defined on goal(s) given for proof.
Definition A:
There does not exist an error condition, E, defined on the given goal, G, such that E is provable:
$\forall G, E.\ G \in Gs \rightarrow (\neg(error(G, E) \wedge provable(E)) \rightarrow e(G, E) \notin Es)$

3.2.2 Case B: conceptual errors found

Textual description:
We found conceptual errors. We can prove - via the inference engine described in section 5 - the error condition(s) defined on goal(s) given for proof.
Definition B:
There exists an error condition, E, defined on the given goal, G, such that E is provable:
$\forall G, E.\ G \in Gs \rightarrow ((error(G, E) \wedge provable(E)) \rightarrow e(G, E) \in Es)$

3.2.3 Case C: conceptual errors found but maybe erroneously reported

Textual description:
We found conceptual errors. We can prove - via the inference engine described in section 5 - the error condition(s) defined on the given goal(s). However, these condition(s) might be erroneously defined or their dependent goals might be erroneous. Therefore we identify two cases: (a)we found conceptual errors that are erroneously reported because of an erroneous error condition and, (b)we found conceptual errors and erroneous error conditions.

From cases A and B we infer the consequences given below. In appendix A we describe how we obtain these logical consequences.
Consequence A: (erroneous error condition)

1. There exists an error condition, $E1$, defined on the given goal, G, which is not provable, but there exists an error condition, $E2$, defined on that condition, $E1$, such that $E2$ is provable:

$$\forall G, E1, E2.\ G, E1 \in Gs \rightarrow$$
$$(\neg\ (error(G, E1) \wedge provable(E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$$
$$e(G, E1) \notin Es \wedge e(E1, E2) \in Es)$$

2. There exists a negative error condition, $\neg E1$, defined on the given goal, G, which is not provable, but there exists an error condition, $E2$, defined on condition, $E1$, such that $E2$ is provable:

$$\forall G, E1, E2.\ \ G, E1 \in Gs \rightarrow \\ (\neg\ (error(G, \neg E1) \wedge provable(\neg E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow \\ e(G, \neg E1) \notin Es \wedge e(E1, E2) \in Es)$$

Consequence B: (erroneous error condition and conceptual error reported)

1. There exists an error condition, $E1$, defined on the given goal, G, which is provable, and there does not exist an error condition, $E2$, defined on that condition, $E1$, such that $E2$ is provable:

$$\forall G, E1, E2.\ G, E1 \in Gs \rightarrow \\ ((error(G, E1) \wedge provable(E1)) \wedge \neg\ (error(E1, E2) \wedge provable(E2)) \rightarrow \\ e(G, E1) \in Es \wedge e(E1, E2) \notin Es)$$

2. There exists an error condition, $E1$, defined on the given goal, G, which is provable, but there also exists an error condition, $E2$, defined on that condition, $E1$, such that $E2$ is also provable:

$$\forall G, E1, E2.\ \ G, E1 \in Gs \rightarrow \\ ((error(G, E1) \wedge provable(E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow \\ e(G, E1) \in Es \wedge e(E1, E2) \in Es)$$

In this case, consequences A1 and A2 refer to erroneous error conditions. As a consequence, the conceptual errors found based on these conditions are erroneously reported. These conditions are erroneously defined ontological constraints - derived from ontological axioms as we described in the previous section - and their detection relies on the provision of meta-axioms, expressed as additional error conditions in the consequences. Similarly, consequences B1 and B2 refer to erroneous error conditions and reported conceptual errors. Their detection also relies on the availability of meta-level axioms. In section 5 we present the multi-layer architecture which makes it possible to integrate layers and meta-layers.

Now that we have defined a representation formalism for ontological axioms and a theory to reason about conceptual errors based on those axioms, we need an inference engine that allows us to automate those checks. In the next section we describe such an engine.

4. Inference engine

The basic idea of our form of error checking is to augment the mechanism used to execute a program(or executable specification) with a system which checks for ontological errors without altering the path of program execution. Our choice for Prolog for our executable language allows us to do this easily, using well known methods of meta-interpretation. The principle of meta-interpretation is to describe in the language of a computational logic a strategy used for problem solving in that logic. Once made explicit in this way, it is easier to augment the strategy, for example to include error checking.

A common inference strategy in trying to establish truth when proving goals is goal reduction. This is made explicit through meta-interpretation based on the standard 'vanilla' model[1]. The predicate $provable/1$ introduced in section 3.2 is used as follows: $provable(Goal)$ is true if $Goal$ is true with respect to the program being interpreted. The inference engine is given in Horn Clause notation:

(1) $provable((A \wedge B)) \leftarrow provable(A) \wedge provable(B).$
(2) $provable((A \vee B)) \leftarrow provable(A).$
(3) $provable((A \vee B)) \leftarrow provable(B).$
(4) $provable(\neg X) \leftarrow \neg provable(X).$
(5) $provable(X) \leftarrow meta(X, M) \wedge provable(M).$
(6) $provable(X) \leftarrow builtin(X) \wedge X.$
(7) $provable(X) \leftarrow clause(X, B) \wedge provable(B).$

The interpreter has the following declarative reading: line (1) states that a conjunction of goals $A \wedge B$ is true when both A and B is true. Lines (2) and (3) state that a disjunction $A \vee B$ is true when either A or B is true. Line (4) deals with negative goals; it states that $\neg X$ is assumed true if we cannot prove X. In lines (5) and (6) specific idioms of the implementation language are treated; if X is a non logical expression like a meta-predicate then X is true if after applying successfully the relation $meta(X, M)$ to obtain a new goal M, M is provable. If X is a builtin expression then it is always true. In line (7) the strategy of goal reduction is realised: goal X is true if there is a clause $X \leftarrow B$ in the interpreted program such that B is true.

The interpreter defined above describes the way of executing a Horn Clause specification. In the next section we show how this can be overlaid with an error checking mechanism.

5. Error checking mechanism

We define an error checking mechanism tailored to detect conceptual errors. It interprets the inference mechanism described above(via clause (D) below) and uses ontological constraints of the form given in section 3.1. The mechanism is given in Horn Clause notation:

(A) $onto\_prove((X1 \wedge X2), E) \leftarrow onto\_prove(X1, E1) \wedge onto\_prove(X2, E2) \wedge$
$E = E1 \cup E2.$
(B) $onto\_prove(\neg X, []) \leftarrow \neg provable(X).$
(C) $onto\_prove(X, E) \leftarrow clause(provable(X), B) \wedge onto\_prove(B, Eb) \wedge$
$error\_check(X, Ex) \wedge E = Ex \cup Eb.$
(D) $onto\_prove(X, []) \leftarrow system\_call(X) \wedge X.$
(E) $error\_check(X, S) \leftarrow setof(errors\_found(X, E, Es), conceptual\_error(X, E, Es), S).$
(F) $error\_check(X, []) \leftarrow \neg conceptual\_error(X, \_, \_).$
(G) $conceptual\_error(X, E, Es) \leftarrow error(X, E) \wedge copy\_term(E, E1) \wedge$
$onto\_prove(E1, Es).$

We use the following predicates in the mechanism:

1. Described in (Sterling & Shapiro, 1994).

- $onto\_prove(Goal, Errors)$ to denote that in trying to prove a goal, $Goal$, we discovered conceptual errors, $Errors$;
- $error\_check(Goal, Set)$ to denote that the error check we performed on goal, $Goal$, yield the set of conceptual errors, Set;
- $conceptual\_error(Goal, Error, Errors)$ to denote that we found a conceptual error, $Error$, and its dependent errors, $Errors$, in trying to prove goal, $Goal$.

The error checking mechanism can be given a declarative reading: line (A) states that a conjunction of goals $X1 \wedge X2$ will yield error, E, if we can find error $E1$ in trying to prove $X1$, and error $E2$ in trying to prove $X2$, and E is the union of $E1$ and $E2$. Line (B) states that we get no errors([]) on a negated goal X when we cannot prove it via the inference mechanism. This means that any errors in the exploration of the failed proof are ignored. Line (C) realises the error checking mechanism: a goal X will yield error E if in applying the inference mechanism to prove goal X we can get its subgoal B; and we can find new error Eb on that subgoal, B; and we can perform an error check on goal X to yield error Ex; and E is the union of Ex and Eb. In line (D) we define the predicate $system\_call(X)$ to denote that we get no errors on goal X when it is a solution given directly by Prolog.

Lines (E) and (F) implement the error checking: in (E) we declare that an error check on goal X will yield a set of errors S if we can find conceptual errors E and its dependent errors Es on X and set S is composed of all occurrences of E and Es represented as the template $errors\_found(X, E, Es)$. This is an implementation of the standard builtin predicate $setof(Template, Goal, Set)$ which has the following meaning(quoted from (SICStus, 1995)): "Set is the set of all instances of $Template$ such that $Goal$ is satisfied, where that set is non-empty". In (F) we declare that we get no errors on goal X if there is no conceptual error on that goal.

Line (G) implements the interface to ontological constraints. It states that there exist a conceptual error E and its dependent errors Es on goal X, if an ontological constraint on goal X is satisfied identifying the error E, and we can find its dependent errors Es regarding $E1$ as a new goal to check which is a replica of E except that all its variables are new. This is given by the standard builtin predicate $copy\_term(Term, CopyOfTerm)$ the meaning of which is(quoted from (SICStus, 1995)): "$CopyOfTerm$ is a renaming of $Term$, such that new variables have been substituted for all variables in $Term$".

The advantage is that we separate the inference strategy from error checking strategy. That means we can plug-in another inference strategy(expressed as $provable/1$) without changing the error checking strategy(expressed as $onto\_prove/2$).

Existing ontological constraints often need enhancements with extra constructs to capture application specific errors. Ideally, those constructs should be part of the ontology and we should be able to check them in the same way as for existing constructs. In an approach where all ontological constructs and constraints belong to the same level, addition of extra constructs might cause problems with the explanation of errors with respect to their triggering conditions. If those conditions use extra constructs which are not part of the original ontology this should be made explicit through the error checking mechanism. To tackle these issues we invented an indexing format to separate the program that adopts the ontology constructs from the ontological constraints. The introduction of this indexing

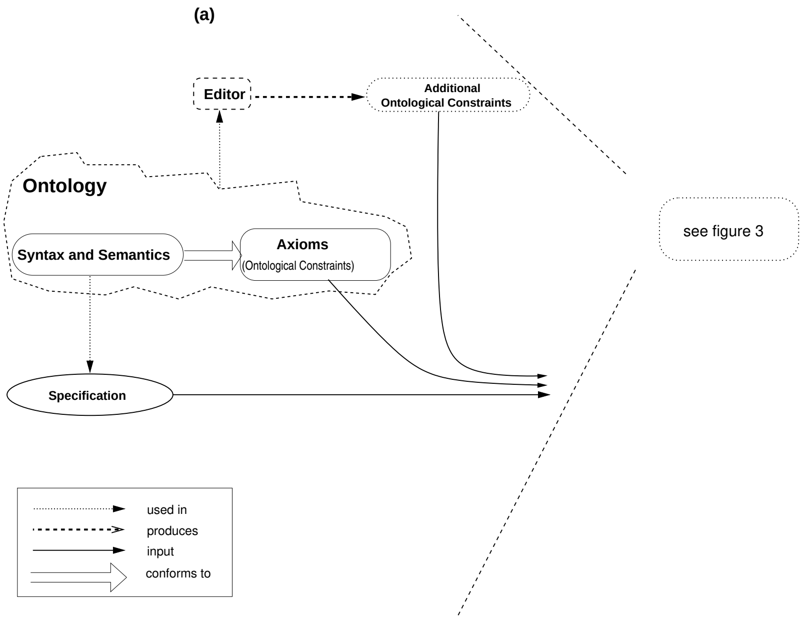

Figure 3: The Multi-layer architecture(part a): a single layer.

format to the meta-interpreter resulted in a multi-layer architecture which allows us to be selective regarding the layer from which we want to start checking. The whole indexed meta-interpreter is documented in (Kalfoglou & Robertson, 1999c) and in ((Kalfoglou, 2000a), ch.4,pages:62-67) along with the code that implements it. Here we include a diagrammatic version of the multi-layer architecture at figures 3 and 4 for a single layer and multiple layers, respectively. A description of a layer follows:

Specification construction starts by adopting the syntax and semantics of the ontology. We use Horn Clauses as a specification construction formalism with the normal Prolog execution model. The ontological constructs will not be the only parts of the specification. In fact, it is normally impractical to construct an executable specification by using only the ontology's constructs. However, our meta-interpreter is tolerant to this: we can check all the ontological parts of the specification regardless of how they interact with other parts of the specification. Moreover, the meta-interpreter can parse any specification regardless of the presence or absence of ontological parts.

Ontological constraints are used to verify the correct use of ontological constructs in the specification. This can be done automatically with the meta-interpreter and as we describe in section 6.4 there can be extra, application specific, ontological constraints that are constructed with the use of supporting tools, like the editor we describe in (Kalfoglou & Robertson, 1999b).

Figure 4 shows our approach in a multi-layer perspective. This allows us to check the correctness of ontological constraints themselves. Whether they are provided by ontological engineers in the form of ontological axioms or are application specific error conditions they may be erroneously defined. This could lead to an erroneous error diagnosis with pernicious

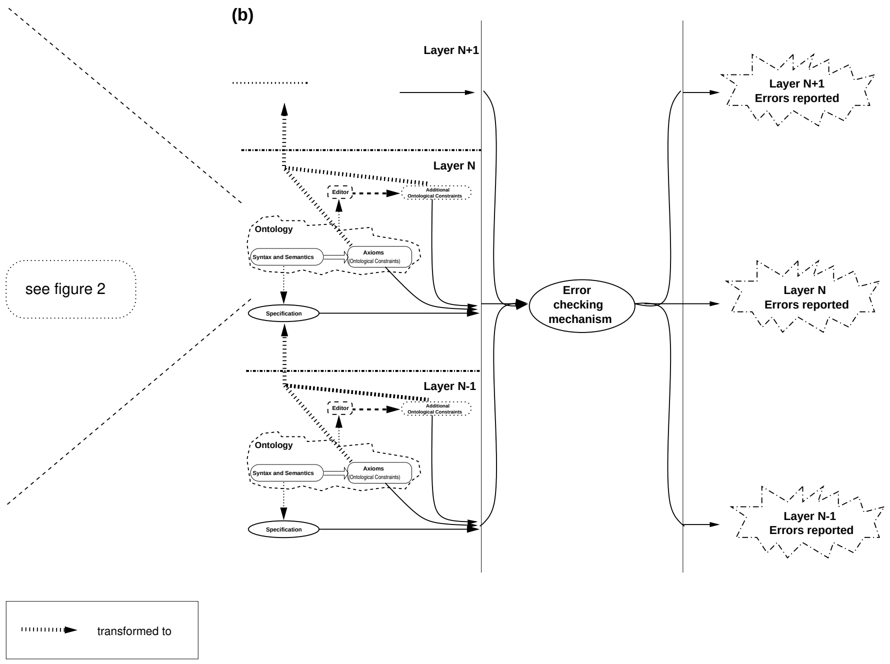

Figure 4: The Multi-layer architecture(part b): the approach as a whole in a multi-layer perspective.

side effects. However, our proofs that errors exist are done using the same mechanism as for specifications, making it possible to define constraints on error ontologies. The advantage of this approach is that we can use the same core mechanism, the meta-interpreter program, to check specifications and their ontological constraints simultaneously. Ultimately, this layer checking can be extended to an arbitrary number of layers upwards, until no more layers can be defined. Examples of the use of the multi-layer approach are included in the next section, while here we draw the attention of the reader to the format we adopt to represent specification statements in analogy with the *clause/2* builtin predicate described previously. The only difference is that we add one more argument to the clause to indicate the index of a layer. The format is as follows:
specification(Index,$(A \leftarrow B)$*)*
The same addition has been made to the error condition format introduced in section 3.1.

6. Example cases

In this section we describe four cases to which we applied the approach. Each of these corresponds to a different mode of use of our method as described below.

As we mentioned earlier there can be variations regarding the instantiations of the ontology/application pair described in figure 1. These are: (i)applying the mechanism in a situation where the ontology exists but not the application, (ii)the application exists but not the ontology and, (iii)both the ontology and the application exist. In the first case, an application was designed according to a scenario described in the literature and supposedly conformed to the existing ontology. We then applied the mechanism to verify this conformance. In the second case, an existing application was used to derive potential ontological constraints applicable to a particular domain. We then used the mechanism to verify conformance of the application to these constraints. In the last case, the mechanism was deployed to verify the conformance of an existing application to an existing ontology. In all of these cases we devised and introduced artificial errors in the application in order to test the mechanism.

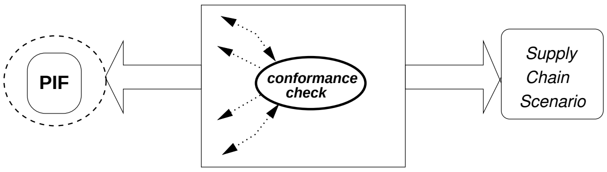

Figure 5: Conformance check of scenario-drawn application to the PIF ontology.

In the first case described above, the ontology used was the Process Interchange Format (PIF) ontology(Lee et al., 1998). We took advantage of the rich axiomatisation of PIF to formulate our error conditions and we developed a specification, drawn from a publicly available scenario, that adopts the syntax and semantics of PIF. We then devised an erroneous specification statement that violates the PIF axiomatisation and detected the error with the mechanism. In figure 5 we illustrate the instantiation of figure 1 with respect to this case while in section 6.2 we present the case in detail.

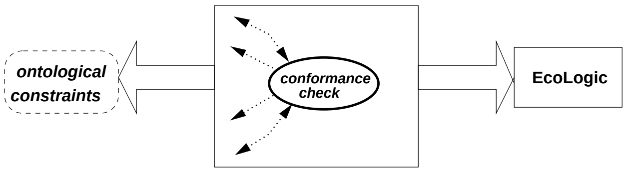

Figure 6: Conformance check of `EcoLogic` specifications to designated ontological constraints.

The *EcoLogic* case gave us the ability to explore the second use described above. We used the specifications found in the *EcoLogic* book(Robertson, Bundy, Muetzefeldt, Haggith, & Uschold, 1991) to derive ontological constraints with respect to those specifications. These constraints represent broad ecological knowledge applicable to other models. Similar approaches are discussed in (van der Vet & Mars, 1998) where the authors elaborate on the bottom-up approach in ontology construction. An error case was devised based on the derived axioms and it is described in section 6.3. The approach is illustrated in figure 6.

In the final case, both the ontology and the specification to which it was applied were available beforehand, taken directly from publicly available resources. Our aim in applying the mechanism in such a case was to show that existing ontologies can be applied to existing specifications. However, we had to make augmentations with respect to the ontology's axiomatisation. We enriched the AIRCRAFT(Swartout, Patil, Knight, & Russ, 1996) ontology's axiomatisation with the help of the editing system described in (Kalfoglou & Robertson, 1999b). As previously, we introduced an error which was detected later by the mechanism; the case is described in section 6.4 and illustrated in figure 7.

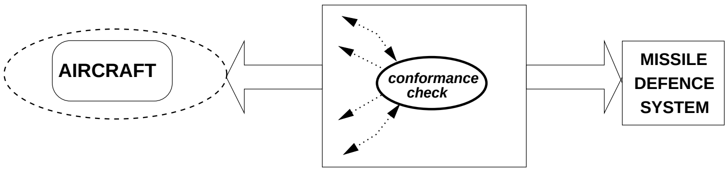

Figure 7: Conformance check of the `Missile Defence System` application to the AIRCRAFT ontology.

The cases are described in the next sections whereas here we draw the attention of the reader to a fourth case which is related to the ontology evaluation scope. An elaboration of the third case is the extension of the evaluation scope we achieved after applying the mechanism to verify the conformance of exemplar applications to a given ontology. In doing this, we check both the ontology and applications while we consider application-specific constructs in our evaluation. The case employs the multi-layer architecture and deserves special attention: we discovered real errors in the original ontology. These are explained thoroughly in ((Kalfoglou, 2000a), ch.6,pages:102-130) whereas here we present a small portion of that case.

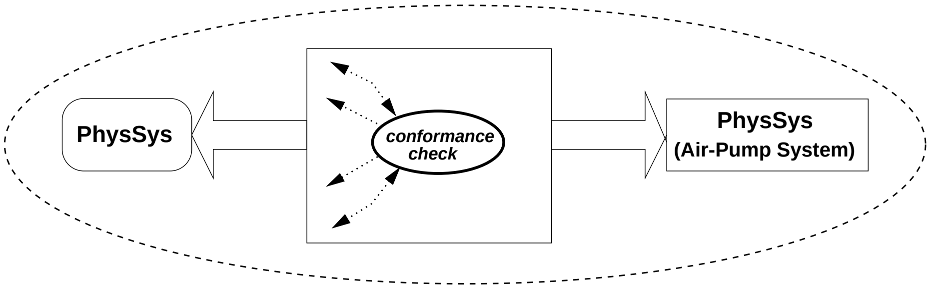

Figure 8: Extending the current evaluation scope and check the consistency of PhysSys ontology at the application level.

The ontology used is the PhysSys which was the most suitable to apply the multi-layer approach: the ontology itself is made up of different layers. The advantage of having the ontology as a specification enabled us to evaluate it at the application level. In doing this, we extend the current evaluation scope to include the application level where an example application demonstrates the usage of the PhysSys ontology. This is illustrated in figure 8 where the dashed line surrounding the PhysSys, which denotes the evaluation scope, has been extended. An error which we introduced in the ontology's upper layer was captured at the application level.

We close the section by discussing, briefly, implementation issues in section 6.5. Lastly, we should mention that there is also the case when neither the ontology nor the application exist. However, in this case the question should not be how to deploy the mechanism but what to build first - ontology or application - which is a matter of debate.

6.1 PhysSys

The case is described in detail in (Kalfoglou & Robertson, 1999b) and the whole ontology is been evaluated in ((Kalfoglou, 2000a), ch.6,pages:102-130). Here, we will elaborate on the layered design of the ontology, the system to which it was applied and the detection of an error at the application level.

The PhysSys is a formal ontology documented in (Borst, Akkermans, & Top, 1997) which captures the domain of system dynamics theory as practised in engineering modelling, simulation and design. It expresses different conceptual viewpoints on a physical system and consists of three engineering ontologies formalising these viewpoints. In our example case we deal with one of these ontologies, the *component* ontology.

The *component* ontology is constructed from mereology, topology and systems theory. To demonstrate a component viewpoint based on the *component* ontology, we illustrate in figure 9 (from (Borst et al., 1997)) a structural-topological diagram for a physical system, like an air pump.

All the ontologies were implemented, originally, in Ontolingua using the Ontology server (Farquhar, Fikes, Pratt, & Rice, 1996). We translated them to the target language we use: Prolog. Although we could use the automatic translation provided by the server we chose to do this manually. This made it easier to translate selective parts of the ontologies while preserving the syntactic elegance of the resulting Horn clauses.

6.1.1 Mereology - Layer 2

The top layer of the architecture, layer 2, consists of the mereology ontology. It provides definitions for mereological relations to specify decomposition and the properties that any decomposition should have. We rewrite Ontolingua statements of the form: $A \leftrightarrow B$ as: $A \leftarrow B$ for specification statements and $A \rightarrow B$ as ontological axioms which must not be violated. These can be rewritten as error conditions, as we described in an earlier section(3.1), of the form: $error(A, \neg B)$. An excerpt of the mereology ontology is given below in the specification format we adopt:

```
specification(2,(individual(X) ← equal(X,X))).
specification(2,(proper_part_of(X,Z) ← part_of(X,Z))).
specification(2,(proper_part_of(X,Z) ← part_of(X,Y) ∧ proper_part_of(Y,Z))).
```

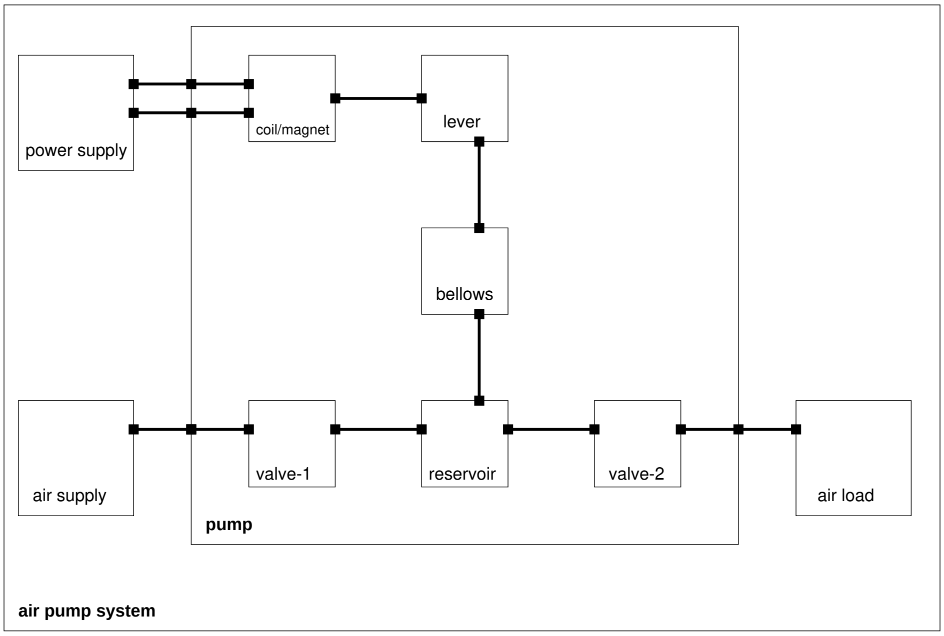

Figure 9: The component view of a physical system: It shows the topology for an air pump. Sub-components are drawn inside the area defined by their super-component. The small solid blocks are the interfaces through which components are connected.

```
specification(2,(direct_part_of(X,Y) ←proper_part_of(X,Y) ∧
                                        ¬ (proper_part_of(Z,Y)
                                        ∧ proper_part_of(X,Z)))).
specification(2,(disjoint(X,Y) ← ¬ (equal(X,Y) ∨ (proper_part_of(Z,X) ∧
                                        proper_part_of(Z,Y))))).
specification(2,(simple_individual(X) ←individual(X) ∧
                                          ¬ proper_part_of(_,X))).
```

The declarative reading of these clauses is the following: the first clause realises the notion of mereologically individual. An individual `X` is a mereological individual when `equal(X,X)` holds. The relation `equal(X,Y)` defines which individuals are considered to be mereologically equal and it usually holds for `equal(X,X)`. The second and third clause represent the `proper_part_of` relation. When an individual `X`, is part of individual `Z` then the `proper_part_of` relation holds. In the third clause the recursive definition of `proper_part_of` realises the transitivity property. We use the `part_of/2` predicate to express static relations that hold with respect to the air pump system illustrated in figure 9. The `direct_part_of/2` predicate realises the direct relation of individual `X` to individual `Z` without the transitivity property taken into account. Clause `disjoint/2` holds for individuals that are not mereologically equal or do not share a part. Finally, the `simple_individual/1` predicate states that an individual `X` is regarded as a simple individual when it has no decomposition.

Apart from these specification statements we can also write down, directly from the Ontolingua syntax, error conditions with respect to the specification given above:

```
error(3,individual(X), ¬ equal(X,X)).
error(3,proper_part_of(X,Y),proper_part_of(Y,X)).
error(3,direct_part_of(X,Y), ¬ (proper_part_of(X,Y) ∧ ¬(proper_part_of(Z,Y)∧
                                 proper_part_of(X,Z)))).
error(3,disjoint(X,Y), (equal(X,Y) ∨ (proper_part_of(Z,X) ∧
                        proper_part_of(Z,Y)))).
error(3,simple_individual(X), ¬ (individual(X) ∧ ¬ proper_part_of(_,X))).
```

Notice that the `individual`, `direct_part_of`, `disjoint` and `simple_individual` error conditions are identical to the preconditions of corresponding axioms in the specification. This is because each error/precondition pair was obtained by "splitting" a double implication, as described earlier. In such circumstances the specification clauses are guaranteed to be consistent with the error conditions but this can change if we add new clauses to the specification or adapt it.

According to the multi-layered architecture presented in section 5 the specification statements are placed in layer 2 while the error conditions that monitor them belong to a layer above, layer 3. In layer 2 we found also definitions of monadic predicates `equal/2` and `part_of/2` with respect to instances of the air pump system. These are application-specific constraints and will be used to reason about the specification in section 6.1.4.

6.1.2 Topology - Layer 1

At layer 1 of the architecture we place the topology ontology. This ontology provides the means to express that individuals are connected. Axioms ensure that only sound connections can be made. We apply the same principles to transform the Ontolingua syntax in the specification format:

```
specification(1,(connection(C) ← connects(C,X,Y))).
specification(1,(connects(C,X,Y) ← (connect(C,X,Y) ∨ connect(C,Y,X)))).
```

The first clause states that a connection `C` connects two individuals `X` and `Y`. The `connects/3` predicate realises the symmetrical property that holds for the *connects* relation. It uses the application-specific predicate `connect/3` to express instances with respect to the air pump system in figure 9; these are described in (Kalfoglou & Robertson, 1999b).

The ontological constraints of this topological layer are the following:

```
error(2,connection(C), ¬ connects(C,X,Y)).
error(2,connects(C,X,Y), ¬ connects(C,Y,X)).
error(2,connects(C,X,Y), (part_of(X,Y) ∨ part_of(Y,X))).
error(2,connects(C,X1,Y1),(connects(C,X2,Y2) ∧
```

```
                              ((disjoint(X1,X2)∧disjoint(X1,Y2))
                              ∧ (disjoint(Y1,X2) ∧ disjoint(Y1,Y2))))).
```

The first two conditions are used to trap side-effects of the symmetrical property that holds for the air pump system connections as well as invalid definitions of connections. The third condition prohibits that a part is connected to itself or its whole. The last error condition is used to detect errors when a connection connects an entirely separated pair of individuals. It uses the mereological relation *disjoint* that has already been defined in layer 2. These conditions are placed in layer 2 of the architecture to monitor the topological statements of layer 1.

6.1.3 Systems theory - Layer 0

At the lowest layer of the architecture, layer 0, we place the systems theory ontology. It defines standard system-theoretic notions such as system, sub-system, system boundary, environment, openness/closeness, etc. An excerpt of this ontology is given below in the specification format we adopt:

```
specification(0,(in_system(X,S) ← proper_part_of(X,S)∧system(S)∧¬system(X))).
specification(0,(in_boundary(C,S) ←connection(C)∧system(S)∧connects(C,X,Y)∧
                                  in_system(X,S) ∧ ¬ in_system(Y,S))).
specification(0,(subsystem(SUB,SUP) ← system(SUB) ∧ system(SUP) ∧
                                  proper_part_of(SUB,SUP))).
specification(0,(open_system(S) ← system(S) ∧ in_boundary(C,S))).
specification(0,(closed_system(S) ← system(S) ∧ ¬ open_system(S))).
```

The declarative reading for the above systems theory specification is as follows: the `in_system/2` predicate holds for individuals that are in the system and are not sub-systems of it; the `in_boundary/2` predicate defines a connection to be in the boundary of a system when it connects an individual in the system to an individual outside the system; the `subsystem/2` predicate holds for individuals that are part of a system and must be a system themselves; the `open_system/1` predicate declares a system to be open when a connection of that system belongs to its boundaries; and finally the `closed_system/1` predicate states that a system is a closed system when it is not an open system. Apart from these definitions we found also constructs of the application level, like definitions of system instances with respect to the diagram of figure 9: *pump*, *powerSupply*, *airSupply* and *airLoad* are all systems.
The ontological constraints of systems theory are:

```
error(1,in_system(X,S), ¬ (proper_part_of(X,S) ∧ system(S) ∧ ¬ system(X))).
error(1,in_boundary(C,S), ¬(connection(C) ∧ system(S) ∧ connects(C,X,Y) ∧
                          in_system(X,S) ∧ ¬ in_system(Y,S))).
error(1,subsystem(SUB,SUP), ¬ (system(SUB) ∧ system(SUP) ∧
                          proper_part_of(SUB,SUP))).
error(1,open_system(S), ¬ in_boundary(airLoad_X_valve2,S)).
```

The first three error conditions are direct transformations from the given Ontolingua code used to detect possible misuse of the specification given above. However the last error condition is a customised condition tailored to the air pump system and constructed with the help of editing tools we described elsewhere(Kalfoglou & Robertson, 1999b). As we can see from figure 9 it is an error whenever a system for which the `airLoad_X_valve2` is not in its boundaries is regarded as an open system. These conditions belong to layer 1 in the architecture to monitor the systems theory layer 0.

6.1.4 Reasoning with meta-knowledge

This layered specification can be executed to check whether various properties of the air pump system(figure 9) hold with respect to the ontological definitions of each of the three layers: mereological, topological and systems theory. In (Kalfoglou & Robertson, 1999b) we checked the specification with respect to specific mereological properties regarding the *disjoint* relation; topological properties regarding the *connects* relation; and from the systems theory point of view checking of the closeness/openness property regarding the air-pump system.

Let us assume that at the mereological layer, layer 2 of our architecture, the ontologist makes the following erroneous definition:

```
specification(2,(disjoint(A, B) ← ¬ proper_part_of(A, B))).
```

This definition states, erroneously, that for two individuals A and B, if A is not a mereological part of B the *disjoint* relation holds.

Although this sort of error is detectable at the mereological layer where it occurs with the use of the multi-layer by defining the layer from which to start checking we postponed the error detection for the next layer.

In the systems theory layer we check the specification to see whether the *pump* is an open system. We will get a correct, positive answer. However, there exists hidden errors which are trapped and reported as follows:

```
error_condition_satisfied(3,disjoint(valve2,valve2),(equal(valve2,valve2) ∨
                                      proper_part_of(Z,valve2) ∧
                                      proper_part_of(Z,valve2)))
error_condition_satisfied(3,disjoint(airLoad,airLoad),(equal(airLoad,airLoad)∨
                                      proper_part_of(Z,airLoad) ∧
                                      proper_part_of(Z,airLoad)))
error_condition_satisfied(2,connects(airLoad_X_valve2,valve2,airLoad),
          (connects(airLoad_X_valve2,airLoad,valve2) ∧
          (disjoint(valve2,airLoad) ∧ disjoint(valve2,valve2)) ∧
          disjoint(airLoad,airLoad) ∧ disjoint(airLoad,valve2)))
path:  [in_boundary(airLoad_X_valve2,pump)|
          (connection(airLoad_X_valve2) ∧ system(pump) ∧
          connects(airLoad_X_valve2,valve2,airLoad) ∧ in_system(valve2,pump)∧
          ¬ in_system(airLoad,pump))]
```

Three errors have been detected: two with respect to the erroneous definition of *disjoint* relation given above and one for the definition of *connects* relation over `airLoad_X_valve2` connection. Recall from the error conditions of layer 1, the definition `in_boundary(airLoad_X_valve2,S)` is used as a condition over the `open_system/1` predicate. The multi-layer architecture allows for check in the error conditions themselves and this enabled the detection of *disjoint* relation misuse. The error at the topological layer with respect to the *connects* relation was detected because the condition defined over that relation was proved by the meta-interpreter. In particular, the `disjoint(reservoir, reservoir)` and `disjoint(valve2,valve2)` that belong in the condition of *connects* relation are erroneous and reported at the topological layer. We also include the execution path we accumulate that helps to locate errors.

The detection of errors at the topological and mereological layers, 1 and 2 respectively, is important given that the behaviour of the system at the systems theory layer was the correct one. This will lead, probably, to the propagation of the error in subsequent phases of the system's development which will make its detection even more difficult as the level of complexity increases and the top layer, that of mereology, becomes more and more hidden in the system.

6.2 PIF

In this section we will demonstrate the use of the mechanism by adopting an ontology from the process interoperability domain, the Process Interchange Format(PIF) ontology(Lee et al., 1998). The aim of PIF is to develop an interchange format to help automatically exchange process descriptions among a variety of business modelling and support systems such as workflow software, flow charting tools, planners, process simulation systems and process repositories. The core of PIF consists of the minimal sets of constructs necessary to translate simple but non-trivial process descriptions. In addition, PIF can be extended to represent local needs of individual groups with the use of Partially Shared Views(PSV) described in (Lee & Malone, 1990). The PIF ontology's focal point is a process, which is a set of activities that stand in certain relations to one another and to objects over timepoints.

We have chosen a scenario proposed by others for the PIF evaluation: the *"supply chain scenario"* [2]. We will present a detection of an error concerning a small part of the scenario, due to space limitations. For a complete reference to the scenario see (Polyak, 1998) and (Polyak et al., 1998).

Our model represents the "process document request" in the transportation company. We quote the description of this process as given in (Polyak, 1998):

> "...Shipping orders are received at the documentation department by a manager. The manager delegates the task to an employee. The delegated employee completes the shipping forms and the customs documents and returns them to the manager for approval. The manager then approves the forms and sends a notice of completion".

A fragment of the specification is given below in the specification format we adopt:

```
specification(0,(object(X) ← agent(X))).
specification(0,(agent(manager) ← true)).
specification(0,(agent(employee) ← true)).
specification(0,(activity(approve_forms) ← true)).
specification(0,(activity(complete_forms) ← true)).
```

2. This scenario was adopted from the Workflow Management Coalition's(WfMC) workflow interoperability demonstration presented at the 1996 Business Process and Workflow Conference in Amsterdam (WfMC, 1996).

specification(0,(performs(Agent,Act) ← object(Agent) ∧ activity(Act))).

The first line declares that an *agent* is a specialisation of an *object* (Lee, Gruninger, Jin, Malone, Tate, & Yost, 1996). In subsequent lines we have the declarations of agents and activities that these agents perform with respect to the process being modelled. The last line describes that the performs/2 relation must be defined over objects and activities by following the PIF documentation((Lee et al., 1996),(Lee et al., 1998)).

However, this sort of representation will allow an erroneous assignment of activity to agent. To prevent such an error we take into account the existing PIF axiomatisation. We reinforce the definition of performs/2 relation by introducing the notion of *capability* as expressed by the PIF ontology. According to PIF documentation "an agent is distinguished from other agents by what it is capable of doing or its skills"(Lee et al., 1996). This could be expressed as a binary relation that maps agents with their capabilities in the form of activities:

specification(0,(capability(manager,approve_forms) ← true)).
specification(0,(capability(employee,complete_forms) ← true)).

and the enhanced *performs* relation is now axiomatised as follows:

$$performs(Agent, Activity) \rightarrow capability(Agent, Capability) \wedge Activity = Capability.$$

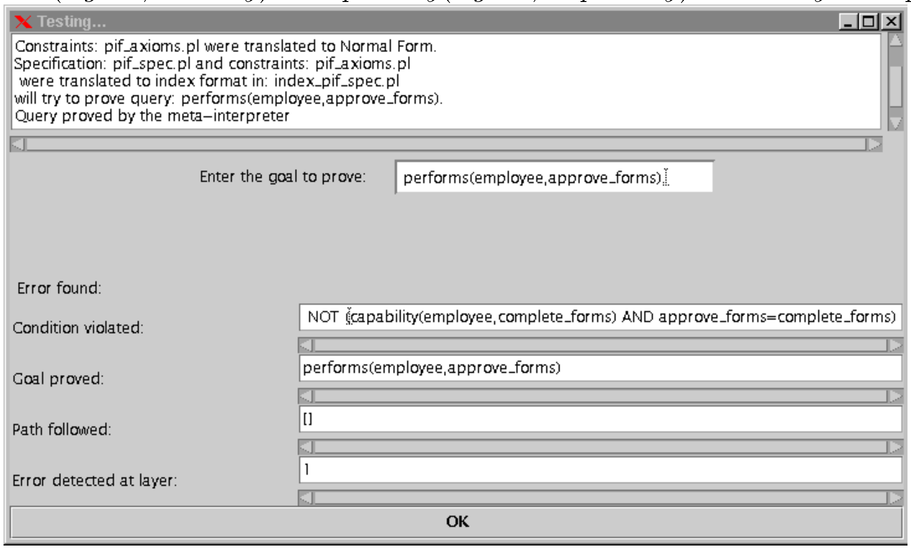

Figure 10: A screenshot of the Ontological Constraints Manager(OCM) tool: An error detection dialog box.

This ontological constraint is then transformed automatically to normal form and is written in the error condition format we adopt:

error(1,performs(Agent,Activity), ¬ (capability(Agent,Capability) ∧

```
                                        Activity = Capability)).
```

to express that it is an error when an activity performed by an agent is not one of which it is capable.

Without the presence of this error condition, we can use the specification by asking: 'Will the agent *employee* perform the *approve_forms* activity?' the result of which will be a positive answer since there is nothing in the specification to prevent the occurrence of this error. Assuming the ontological constraint given above, our meta-interpreter can detect this error.

The results of error detection are shown in figure 10 where we include a screenshot of the Ontological Constraints Manager(OCM), a software tool we developed to perform such checks operating from a Java front-end.

As we can see from figure 10 the given goal for proof conforms to the erroneous ontological definition of *performs* relation but contradicts to the ontological constraint given above. This is highlighted by the inequality of *employee's* capability(*complete_forms*) and the erroneous activity that intend to perform(*approve_forms*).

To facilitate the definition of ontological constraints such as the one described above we have built two editing tools which provide built-in checks for conflict and subsumption occurrence. Here we will describe the first tool which we call the 'relations editor' whereas the second tool, the 'constraints editor' is described in section 6.4 along with the example case of AIRCRAFT ontology.

With the relations editor, one can define unary, binary and ternary relations that hold over ontological concepts and choose logical connectives to link them. The collection of concepts from the ontology as well as the distribution of variables that will be shared among the literals is done automatically, the user only has to select the concepts he wants to use. For example, editing an axiom of the PIF core, which is described textually as: 'The *before* relation holds only between timepoints.", will result in the following Horn clause:
$before(A, B) \leftarrow point(A) \wedge point(B)$.
In figure 11 we include a screenshot of the relations editor at the point where the user will select the logical connective for the above relation.

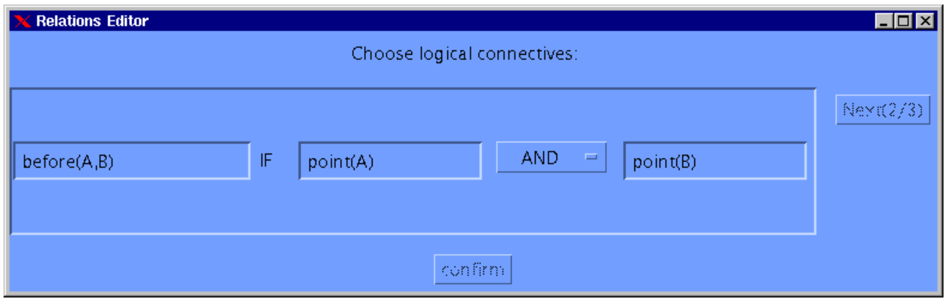

Figure 11: A screenshot of the OCM tool: the relations editor.

If we are interested in using the above relation as an axiom then at this stage we can add it to the existing ontology axiomatisation after conflict and subsumption checking is done. The sort of conflict check we apply declares two axioms as being contradictory to each other if they are variants but still not unifiable after having their variables temporarily bound. Two axioms are variants if they are the same although they might have different variable names. In our checks for variance we ignore the 'not' operator, if exist. This will be checked in the unification test in order to reveal potential conflict. For example, assume that the ontology already contains the axiom: $participates\_in(O, A, T) \leftarrow exists\_at(O, T) \wedge is\_occurring\_at(A, T)$, and we try to add the axiom: $participates\_in(Obj, Act, T1) \leftarrow exists\_at(Obj, Act) \wedge \neg is\_occurring\_at(Act, T1)$, this will cause a conflict alert because these two axioms are contradictory to each other. The subsumption check will ensure that for two axioms that their heads match, we won't let a more generic one subsume an existing detailed one. This is a limited form of subsumption check that will prevent specific information loss caused by a generic axiom. For example, assume the axiom above and that the ontology already contains the: $before(P, Q) \leftarrow point(foo(P)) \wedge point(foo(Q))$, this will cause a subsumption warning to the user since variables A and B from the new axiom will subsume predicates $foo(P)$ and $foo(Q)$, respectively. In both the conflict and subsumption check no action is taken by the system apart from warning the user because there are cases where we might want to include both axioms in the ontology.[3]

However, if the relation is to be used as an ontological constraint in the form of an error condition not to be satisfied by the specification then we apply the procedure described in section 3.2 to produce the following ontological constraint:
$error(1, before(A, B), \neg(point(A) \wedge point(B)))$.

6.3 EcoLogic

In this case we deal with ecological modelling an area which is concerned with complex biological systems that are difficult to decide how to represent them in a simplified form as simulation models. In the *EcoLogic* book(Robertson et al., 1991), Robertson and colleagues elaborate on the use of logic to represent those simulation models. We borrowed one of these models to apply our approach: the "rabbit-grass energy flow" based in a system dynamics model represented in (Forrester, 1961). The whole model is given in logic in (Robertson et al., 1991); here we give a textual description and a diagrammatic illustration in figure 12(from (Robertson et al., 1991)):

As stated in (Robertson et al., 1991): "there exist two state variables, *grass* and *rabbit*. The changes to these state variables over time are affected by the flows of *photosynthesis*, *respiration*, *grazing* and *defecation*. The *photosynthesis* flow is from the environmental "source/sink" into grass. The *grazing* flow transfers material from *grass* to *rabbit*. Flows of *respiration* empty out of *grass* and *rabbit*. An additional flow of *defecation* also exists from *rabbit*. The rates of transfer for each of these flows are controlled by an equation which takes as inputs a parameter and the values of the state variables involved in that flow". The recursive definition of state variable is given in (Robertson et al., 1991) while here we discuss the ontological constraint drawn from the model:

According to domain knowledge, for any given state variable S, that has value N, at time T, there is not exist state variable S1, which is different than the previous, has value

3. This is a debatable issue and Visser and colleagues elaborate on this in (Visser, Jones, Bench-Capon, & Shave, 1998).

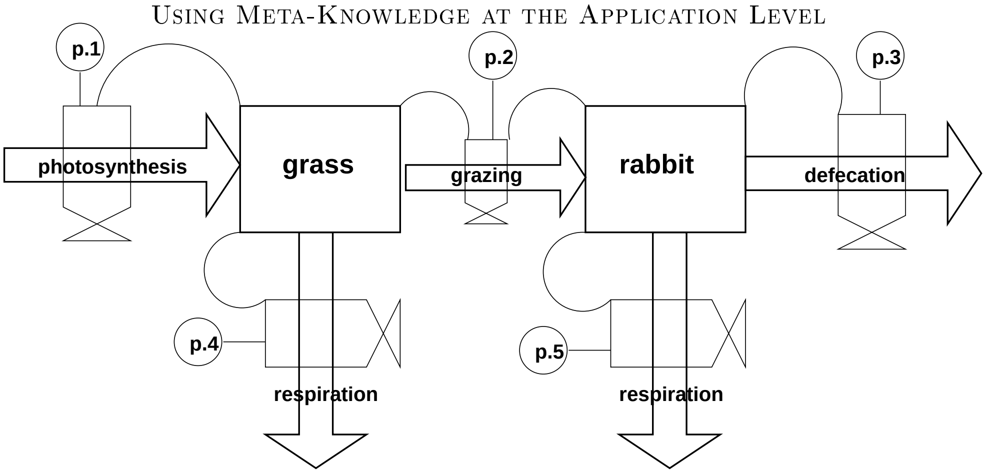

Figure 12: Rabbit-grass energy flow

N1 at the same time(T), N1 is greater than N, and S1 is lower in the food chain than S at that time(T). Note that this condition is tailored to the specific application and holds for the existing values of the regulating parameter. The condition is axiomatised as follows:

$$state\_variable(S,T,N) \rightarrow \neg(state\_variable(S1,T,N1) \wedge \neg\ S1 = S \wedge N1 < N \\ \wedge\ lower\_in\_food\_chain(S1,S,T)).$$

and transformed to normal form and subsequently to the error condition format:

```
error(1,state_variable(S,T,N), (state_variable(S1,T,N1) ∧ ¬ S1=S ∧ N1<N ∧
                                 lower_in_food_chain(S1,S,T))).
```

In addition, this axiom introduces the concept of *lower_in_food_chain.* This concept is drawn from domain knowledge and regarded as a meta-level specification in the context of our model. It is formalised as follows:

```
specification(1,(lower_in_food_chain(S1,S,T) ← flow(Flow,S1,S,T,_) ∧
                                                    eating_flow(Flow))).
specification(1,(lower_in_food_chain(S1,S,T) ←flow(Flow,S2,S,T,_) ∧
                                              eating_flow(Flow) ∧
                                              lower_in_food_chain(S1,S2,T))).
```

The first clause states that whenever there is an *eating_flow* from a state variable S1, to another state variable S, at time T, then S1 is lower in the food chain than S. The second clause is a recursive definition of the above to enable the model backtrack on all possible state variables. The definitions of *lower_in_food_chain* are placed in layer 1 of the architecture to be used in the ontological constraint introduced at that layer. However they include a new concept, that of *eating_flow.* This is then included in that layer and instantiated to the flows of the particular model:

```
specification(1,(eating_flow(grazing) ← true)).
```

To ensure proper use of the *lower_in_food_chain* concept we introduce a constraint at a layer above, layer 2:

```
error(2,eating_flow(F), ¬ F=grazing).
```

This constraint states that it is an error whenever an *eating flow* is not of type *grazing.*

If we execute the simulation model with the correct definitions given above, and check how variables in the model are ordered in the food chain we will get the answer that state variable *grass* is lower in the food chain than state variable *rabbit*.

However, assume that the specifier defines, erroneously, the eating flow instances in the model: `specification(1,(eating_flow(photosynthesis) ← true)).`
`specification(1,(eating_flow(defecation) ← true)).`

If we query the model again then we will get the following erroneous order: that *rabbit* is lower in the food chain than *grass*. With the ontological constraints given above the meta-interpreter traps the error and reports it appropriately:

```
error_condition_satisfied(2,eating_flow(photosynthesis),¬photosynthesis=grazing)
path:  [lower_in_food_chain(rabbit,grass,2)|
        (flow(photosynthesis,source_sink,grass,2,399.6),
        eating_flow(photosynthesis),
        lower_in_food_chain(rabbit,source_sink,2))]
error_condition_satisfied(2,eating_flow(defecation), ¬ defecation=grazing)
path:  [lower_in_food_chain(rabbit,source_sink,2)|
        (flow(defecation,rabbit,source_sink,2,0.9),
        eating_flow(defecation))]
```

Two errors have been detected. The first one has instantiate the variable `F` of the `eating_flow/1` clause to *photosynthesis* and the second one to *defecation*. However, both of them satisfy the error condition on *eating flow*, that is the inequality with the *grazing* flow. As previously, extra information concerning the execution path is provided.

The ontological constraint for the *state_variable* clause is tailored to the particular model. However, it uses the *lower_in_food_chain* concept which represent broad ecological knowledge and can be applied to similar applications. The same can be said for the *eating_flow* concept and the condition that monitor its use. We abstracted those concepts from the model, by placing them in another layer in the architecture which enabled us to check the domain knowledge without sacrificing the execution of the model that uses it. In ((Kalfoglou, 2000a), ch.5,pages:79-88) Kalfoglou elaborates on further conceptual error checks regarding the ecological modelling domain.

6.4 AIRCRAFT

This case differs from the previous ones in that it uses a pre-existing ontology and a pre-existing specification. The specification we used is an implementation of the Prototype System Description Language(PSDL) simulator applied to the missile defence system(MDS), a prototype system designed to protect an allied vessel from missiles attack. The specification is documented in (Luqi & Cooke, 1995) along with a description of the MDS, originally suggested by Lehman in (Lehman, 1990), to which it was applied. The ontology which we used was the AIRCRAFT ontology[4], part of the Ontosaurus project implemented at the Information Sciences Institute(ISI), University of Southern California(USC). We discuss the case in detail in (Kalfoglou & Robertson, 1999a); here we elaborate on the augmenta-

4. Electronically accessible from http://www.isi.edu/isd/ontosaurus.html

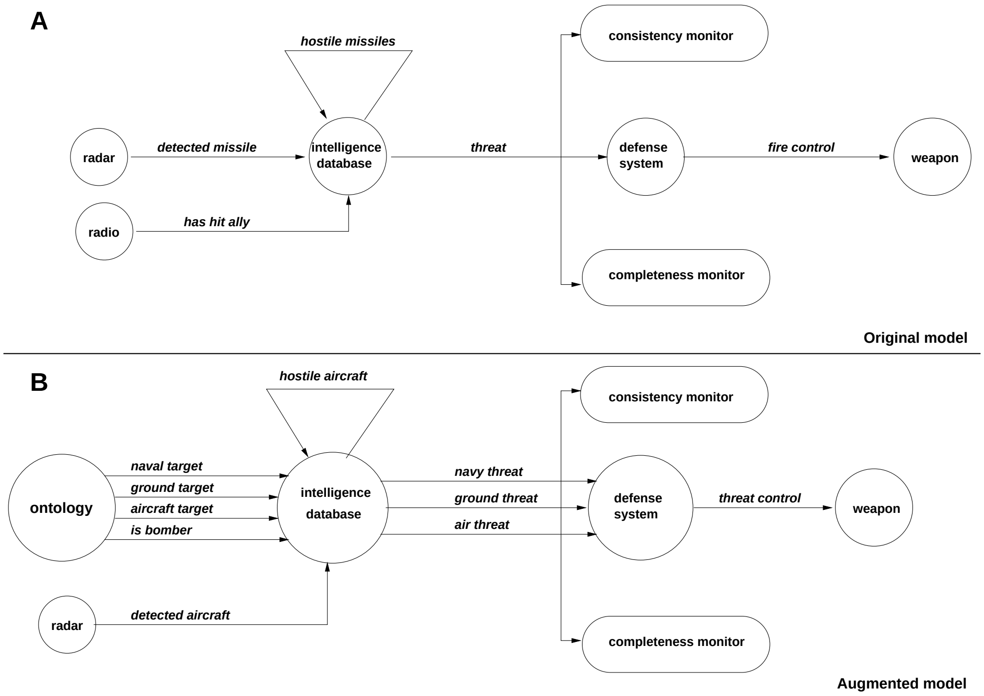

Figure 13: The missile defence prototype: original and augmented models: Standard operators of the system are represented in circles(radar,radio/ontology,intelligent database, defense system, weapon), execution monitoring operators are represented in rounded boxes (consistency, completeness), data streams are represented as arrows that link the operators(detected missile,has hit ally, hostile missiles, threat, fire control).

tion we made to the specification with respect to the AIRCRAFT ontology and the use of application-specific constraints to detect conceptual errors.

In figure 13 we illustrate the MDS in two panes. Pane A shows the original model as described in (Luqi & Cooke, 1995): two operators, *radio* and *radar* used to detect information concerning an incoming missile to the vessel's airspace which is then use the data streams *detected_missile* and *has_hit_ally* to allow the *intelligence_database* operator determine whether the missile should be regarded as a threat updating the data stream *hostile_missiles*. *Defense system* and *Defensive weapon* operators then treat the missile appropriately reading values from *threat* and *fire_control* data streams. In addition to the MDS, Luqi and Cooke in (Luqi & Cooke, 1995) had define two monitoring operators, *consistency* and *completeness* to detect context shifts in the specification in support of software evolution.

In Pane B of figure 13 we illustrate the augmentations we made: we adopted the original prototype system to treat hostile aircraft instead of missiles that enter an ally's airspace. The *radio* operator which categorised missiles in the original model, was replaced by an *ontology* operator which classifies aircraft according to the AIRCRAFT ontology. This allows the *intelligence database* operator to classify the form of threat posed by the aircraft.

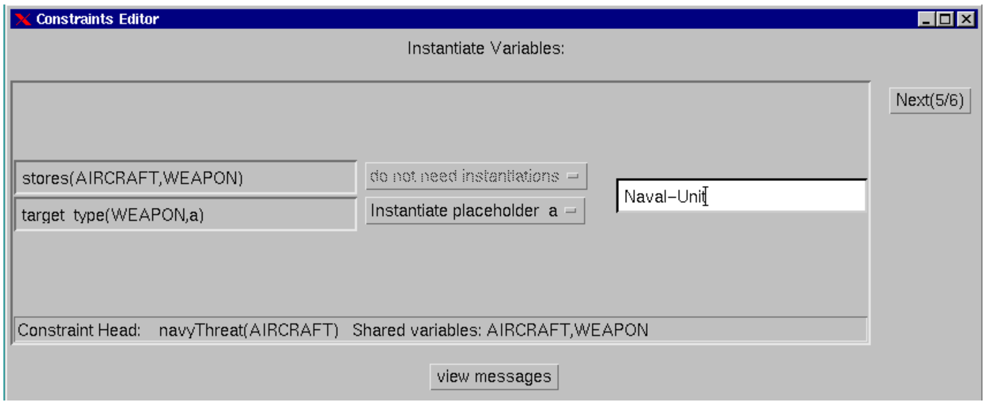

Figure 14: A screenshot of the Constraints Editor that shows the step of instantiating variables in the constraint's body relations. AIRCRAFT and WEAPON are variables that will be shared among the relations in the body and head of the constraint. The user instantiates the second argument of the `target_type/2` relation to 'Naval-Unit'.

The use of the AIRCRAFT ontology allowed us to enrich the treatment of a threat. We modelled three different kinds of threats(navy,ground,aircraft) in accordance with the formally defined in the ontology target types of the weapons that an aircraft stores, as they appear. Moreover, we used the ontology's construct `bomber` to determine dynamically whether a detected aircraft, for which there is no information available about its targets, should be regarded as hostile and treated appropriately. This information belongs to the context model of the MDS and used to accumulate new contextual information. Although we could define additional predicates to construct rules that govern those determinations we used constructs of the underpinning ontology for this purpose. For example, a detected aircraft is considered a navy threat when it stores weapons that target on naval units. This can be axiomatised in the following ontological definition:

$$navyThreat(A) \rightarrow aircraft(A) \wedge stores(A, W) \wedge target\_type(W, naval\_unit).$$

Note that the interpretation of the concepts `aircraft/1, stores/2` and `target_type/2` is formally given in the ontology. This lightens the workload of the specifier since it has only to define a rule that will hold over these concepts, that of `navyThreat/1`. In support of this, we developed an editing system that provide assistance in constructing those rules.

The system is described in (Kalfoglou & Robertson, 1999b) along with explanatory examples of its use; here we include, in figure 14, a screenshot of the editor at the stage of instantiating the variables to be used in the conditional part of the constraint. The constraint produced by the editor is given in the error condition format:

```
error(1,navyThreat(AIRCRAFT),(¬ (aircraft(AIRCRAFT) ∧ stores(AIRCRAFT,WEAPON)∧
                                  target_type(WEAPON,'Naval-Unit')))).
```

Assume that the specifier of the context model defines erroneously a rule that determines whether or not an aircraft should be regarded as a navy threat:

$$navyThreat(A) \leftarrow aircraft(A) \wedge mission(A, M) \wedge combat(M).$$

This rule classifies an aircraft as a navy threat according to the nature of the mission it performs. Whenever this is a combat mission the detected aircraft will be regarded as a navy threat. Although the above rule is correct it is under-defined: an aircraft that carries anti-aircraft and anti-ground weapons will be treated as naval threat. That is because it performs combat missions which do not threaten a naval unit(i.e.: interception, interdiction, SEAD(Suppress of Enemy Air Defences), etc.)

According to the ontology, an F-117 aircraft carries weapons that target on ground and aircraft units but not on naval units. The following fragment of execution demonstrates the erroneous behaviour of the system, in the form of values written in the simulator's data streams:

```
..
read_from(intelligence_database,F-117,detected_aircraft)
wrote_into(intelligence_database,true,navy_threat)
wrote_into(defense_system,shootToProtectShip,threat_control)
..
```

As we see, F-117 is treated by the system as a navy threat and causes the *defense_system* operator erroneously to fire and shoot it down to protect a ship in the ally airspace.

However, with the ontological constraint given above, we capture this sort of error and report it as follows:

```
error_condition_satisfied(1,navyThreat(F-117),(¬ (aircraft(F-117) ∧
                                                stores(F-117,'AIM-9M') ∧
                                                target_type('AIM-9M','Naval-Unit')))).
```

The error is detected because the weapon `AIM-9M` that is carried by the F-117 does not target naval units as it should in order for the F-117 to be a navy threat.

6.5 Implementation

The Ontological Constraints Manager(OCM), mentioned in section 6.2, was written in Java and serves as a front-end of several Prolog applications. The editing tools of sections 6.4 and 6.2 were written in Java as well and run through the OCM. We used the Java package `jasper` to link Prolog with the Java front-end, which is a bi-directional Java to Prolog interface developed from SICStus[5]

7. Limitations and further application areas

Our approach has some limitations. We summarise these in three main areas: the error checker, ontological constraints editor, and the meta-interpreter technique. In particular,

5. The package is documented online under the heading "Mixing Java and Prolog"' in the following URL(as of Feburary of 2000): http://www.sics.se/isl/sicstus/docs/latest/html/sicstus.html

the error checking approach we adopt cannot guarantee that it will reveal all conceptual error occurrences in the specification and/or ontology. This stems from basic limitations of knowledge engineering and software testing: on one hand, we rely on the provision of the right ontological constraint to detect the error and, on the other, we hope that we chose the right goal to test for errors. As far as the provision of the ontological constraints is concerned, to our surprise in most of the ontologies we have encountered no rich axiomatisation was provided. Even in some of the most carefully designed ontologies, we had to improvise a constraint in order to enhance the existing axiomatisation and enable trapping of subtle error occurrences. This is additional work for the engineer. Nevertheless, the gain can be worthwhile as we demonstrate in section 6 and in ((Kalfoglou, 2000a), ch.6,pages:102-130), where we applied the approach successfully and reveal subtle errors not only in the specifications but also in existing, published ontologies. The problem of choosing the right goal to test is a concern in our method. That is because we only check goals on an execution path so executing the 'wrong' goal will fail to reveal the error. However, the way that our error checking is deployed will guarantee that when we check the a goal any errors detected are relevant to the execution. The goals which we check for errors are always those which have succeeded in the proof. This means that they are fully instantiated before error checking is done, making those checks more directed. This, however, raises the problem of what to do with goals which fail. One could argue that these should be checked as well, since they were explored as part of the search for a solution. We take the view that they should not be error checked because they do not, in the end, form part of the correct proof. Conformance of a specification to an ontology is checked by exhaustively checking the goal against the ontological constraints. As a consequence we raise an issue of computational cost here because although the goal requires one proof, the constraints are checked exhaustively. Another potential limitation might be the lack of automatic error correction. This is an area that we have not explore in depth. This decision stems from the nature of the errors we are dealing with. We argue that there is no foreseeable way to automate the process of correcting conceptual error occurrences. For example, if we apply the mechanism to check an ontology(as we did in (Kalfoglou, 2000a), ch.6,pages:102-130) and we manage to reveal an erroneous ontological definition, then what should the correct one be like? The answer is not at all obvious especially when the revealed error is concerned with an erroneous ontological constraint. It is hard to imagine an automatic way of building correct ontological axioms to replace erroneous ones.

The third area of limitations is that of the meta-interpreter. Although we provide two different approaches in building these designated meta-interpreters, one of which was presented in section 5, there are some limitations in their operation. For instance, the treatment of non-logical features like cut(!), built-in predicates other than those we treat($findall$ and $setof$), and system calls(i.e., invocation of external procedures/systems). There are some solutions described in the literature with respect to treatment of built-in predicates(see (Robertson et al., 1991)), but these are routine extensions to the pure Prolog interpreter and distract from our main concern. Rather, we concentrated on various ways of deploying meta-interpreters and integrating them with the conceptual errors theory provided in section 5.

We cope with some of the limitations by using the ontological constraints editor we built. Take for example, the problem of building extra axioms. We used this editor to build some of the axioms described in section 6. However, the editor is not meant to be a full-fledged ontological axioms building tool that will support free-form editing. It actually uses a heuristic mechanism which forces the user to employ as many existing ontological constructs in the axiom as possible. The rationale behind this is that this way of building an axiom will make it easier to reuse it in other applications. That is because the axiom is composed of existing ontological constructs, whereas in free axiom editing freedom to use and define new constructs will inject update issues with respect to the ontology.

The system we have described is one of many potential approaches to ontology management. Despite the fact that ontology development tools cover a wide spectrum of issues and variety of them is available, little work has been done in the areas of deployment, evaluation, and maintenance. To deploy ontologies correctly we need not only self-contained, reuse-oriented tools and technologies as for example in the HPKB(Cohen, Schrag, Jones, Pease, Lin, Starr, Gunning, & Burke, 1998) and IBROW(Fensel, Benjamins, Motta, & Wielinga, 1999) projects. It is also necessary to record and organise our experiences in having applied them in order to improve future ontology deployment. It is hard to gain this sort of experience from the literature because few cases of comprehensive ontology reuse and deployment on a large scale are reported(see, for example, (Uschold, Clark, Healy, Williamson, & Woods, 1998), (Borst et al., 1997), (Valente et al., 1999)). Even those which are reported do not normally discuss the hidden assumptions and tradeoffs identified during testing.

In this paper we presented a novel way of deploying ontologies which assists to evaluating them in applications. In (Kalfoglou & Robertson, 2000), we show how we can manage the experiences collected during this sort of deployment. In particular, we discuss how experience management as practised in the SE community through the use of *Experience Factories*(Basili, Caldiera, & Rombach, 1994) and their constituents *Experience Bases* can facilitate ontology deployment. We will not elaborate on this matter here due to space limitations but we point the interested reader to the aforementioned publication and to ((Kalfoglou, 2000a), ch.7,pages:131-148) for a thorough analysis. However, we would like to close the section by mentioning the linkage of this experience management effort with the ontology characterisations work. The latter were originally introduced in (Uschold, 1998) and further analysed in (Uschold & Jasper, 1999). In (Kalfoglou, 2000b) we found it useful to integrate these characterisations - which provide ontology-specific information - to our experience management framework as a means to organise and retrieve experiences on testing in similar ontology deployment efforts.

8. Conclusions

The use of ontologies we described in this paper diverge from the mainstream uses encountered in this field. We aimed to support the application of ontologies in an area where little work has been reported by the community: systems engineering and in particular to enforce systems reliability. To achieve this we employ ontological constraints to reason about the correctness of executable, declarative specifications. We described a detailed methodology of transforming the axioms founded in ontologies to a designated constraints format which is used by an inference engine tailored to check for ontological errors in specifications. We operate those checks via a multi-layer architecture we developed which also makes it possible to check ontologies themselves.

We applied the approach in a variety of application domains including business process modelling, air-campaign planning, ecological modelling and system dynamics. This made it possible to elaborate on a generic method for checking the conformance of applications with respect to the underlying ontology by deploying its ontological constraints. The significance of our method is that we perform the evaluation at the application level taking into account specific characteristics of that level. This facilitates the integration of ontologies in applications while preserving their identity as being a separate layer in the multi-layer architecture.

Acknowledgments

The research described in this paper is supported by a European Union Marie Curie Fellowship(programme: Training and Mobility of Researchers).

Appendix A. Transformations of cases A and B in section 3.2

We are using the following rule:

$$(\mathcal{A})\ A \rightarrow C \bigwedge B \rightarrow D \Rightarrow A \wedge B \rightarrow C \wedge D$$

For clarity and brevity we will not deal with quantifiers. Cases A and B are presented as follows:

(1) $\neg(error(G, E) \wedge provable(E)) \rightarrow e(G, E) \notin Es$
(2) $(error(G, E) \wedge provable(E)) \rightarrow e(G, E) \in Es$

we make the following substitutions: E with $E1$ in formula 1 which is now written as:

(3) $\neg(error(G, E1) \wedge provable(E1)) \rightarrow e(G, E1) \notin Es$

and we substitute G with $E1$ and E with $E2$ in formula 2 which is now written as:

(4) $(error(E1, E2) \wedge provable(E2)) \rightarrow e(E1, E2) \in Es$

we apply the rule $(\mathcal{A})$ in formulae 3 and 4 to obtain the **consequence A1**:

(5) $\neg(error(G, E1) \wedge provable(E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, E1) \notin Es \wedge e(E1, E2) \in Es$

we also make the following substitutions: G with $E1$ and E with $E2$ in formula 1 which is now written as:

(6) $\neg(error(E1, E2) \wedge provable(E2)) \rightarrow e(E1, E2) \notin Es$

and we substitute E with $E1$ in formula 2 which is now written as:

(7) $(error(G, E1) \wedge provable(E1)) \rightarrow e(G, E1) \in Es$

we apply the rule $(\mathcal{A})$ in formulae 6 and 7 to obtain the **consequence B1**:

(8) $(error(G, E1) \wedge provable(E1)) \wedge \neg\ (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, E1) \in Es \wedge e(E1, E2) \notin Es$

we also perform the following substitutions: E with $\neg E1$ in formula 1 which is now written as:

(9) $\neg(error(G, \neg E1) \wedge provable(\neg E1)) \rightarrow e(G, \neg E1) \notin Es$

we apply the rule $(\mathcal{A})$ in formulae 9 and 4 to obtain **consequence A2**:

(10) $\neg(error(G, \neg E1) \wedge provable(\neg E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, \neg E1) \notin Es \wedge e(E1, E2) \in Es$

and we apply rule $(\mathcal{A})$ in formulae 7 and 4 to obtain **consequence B2**:

(11) $(error(G, E1) \wedge provable(E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, E1) \in Es \wedge e(E1, E2) \in Es$

References

Aguado, G., Banon, A., Bateman, J., Bernandos, S., Fernandez, M., Gomez-Perez, A., Nieto, E., Olalla, A., Plaza, R., & Sanchez, A. (1998). ONTOGENERATION: Reusing domain and linguistic ontologies for Spanish text generation. In Gomez-Perez, A., & Benjamins, R. (Eds.), *Proceedings of Workshop on Applications of Ontologies and Problem-Solving Methods, ECAI'98, Brighton, England.*

Basili, V., Caldiera, G., & Rombach, D. (1994). Experience Factory. In Marciniak, J. (Ed.), *Encyclopedia of Software Engineering*, Vol. 1, pp. 469–476. John Wiley & Sons.

Benjamins, R., & Fensel, D. (1998). The Ontological Engineering Initiative-KA2. In Guarino, N. (Ed.), *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy*, pp. 287–301. IOS Press.

Borst, P., Akkermans, H., & Top, J. (1997). Engineering Ontologies. *International Journal of Human-Computer Studies*, *46*, 365–406.

Cohen, P., Schrag, R., Jones, E., Pease, A., Lin, A., Starr, B., Gunning, D., & Burke, M. (1998). The DARPA High Performance Knowledge Bases project. *AI Magazine*, *19*(4), 25–49.

EIL, E. I. L. (1995). TOVE Project, University of Toronto, Canada. available from http://www.ie.utoronto.ca/EIL/tove/ontoTOC.html.

Farquhar, A., Fikes, R., Pratt, W., & Rice, J. (1996). The Ontolingua Server: a Tool for Collaborative Ontology Construction. In *proceedings of the 10th Knowledge Acquisition Workshop, KAW'96,Banff,Canada.* Also available as KSL-TR-96-26.

Fensel, D., Benjamins, V., Motta, E., & Wielinga, B. (1999). UPML: A Framework for knowledge system reuse. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI'99, Stockholm, Sweden*, pp. 16–21.

Fensel, D., Horrocks, I., van Harmelen, F., Decker, S., Erdmann, M., & Klein, M. (2000). OIL in a Nutshell. In *Proceedings of the 12th International Conference on Knowledge Engineering and Knowledge Management(EKAW'00), Juan-Les-Pins, France.* (to appear).

Fensel, D., & van Harmelen, F. (1994). A comparison of languages which operationalise and formalise KADS models of expertise. *The Knowledge Engineering Review*, *9*(2), 105–146.

Forrester, J. (1961). *Industrial Dynamics.* MIT Press. ISBN: 0-262-060035.

Fox, M., & Gruninger, M. (1997). On Ontologies and Enterprise Modelling. In *Proceedings of International Conference on Enterprise Integration Modelling Technology 97.* Springer-Verlag.

Friedman, M., & Voas, J. (1995). *Software Assessment: Reliability, Safety, Testability.* Willey & Sons. ISBN: 0-471-01009-X.

Gangemi, A., Pisanelli, D., & Steve, G. (1998). Ontology Integration: Experiences with Medical Terminologies. In Guarino, N. (Ed.), *Proceedings of the 1st International Conference on Formal Ontology in Information Systems, FOIS'98, Trento, Italy*, pp. 163–178.

Genesereth, R., & Fikes, R. (1992). *Knowledge Interchange Format* (3.0 edition). Computer Science Dept., Stanford University. Technical Report, Logic-92-1.

Gomez-Perez, A. (1995). Guidelines to Verify Completeness and Consistency in Ontologies. In *Proceedings of the 3rd World Congress on Expert Systems*, pp. 901–908.

Gomez-Perez, A. (1996). Towards a Framework to Verify Knowledge Sharing Technology. *Expert System with Applications*, *11*(4), 519–529.

Gomez-Perez, A. (1999). Evaluation of Taxonomic Knowledge in Ontologies and Knowledge Bases. In *Proceedings of the 12th Knowledge Acquisition, Modelling and Management Workshop(KAW'99), Banff, Canada.*

Gomez-Perez, A., & Benjamins, R. (1999). Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden.* http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

Gomez-Perez, A., Juristo, N., & Pazos, J. (1995). Evaluation and Assessment of the Knowledge Sharing Technology. In N.J.I.Mars (Ed.), *Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB & KS'95),Twente, The Netherlands*, pp. 289–296.

Gruber, T. (1993). A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, *5*(2), 199–220.

Gruninger, M. (1996). Designing and Evaluating Generic Ontologies. In *Proceedings of the workshop on ontological engineering, ECAI'96, Budapest, Hungary.*

Gruninger, M., & Fox, M. (1995). Methodology for the Design and Evaluation of Ontologies. In *Proceedings of the IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Quebec,Canada.*

Guarino, N., Carrara, M., & Giaretta, P. (1994). Formalizing Ontological Commitments. In *In Proceedings of the 12th National Conference on Artificial Intelligence(AAAI'94), Seattle, Washington, USA.*

Kalfoglou, Y. (2000a). *Deploying Ontologies in Software Design.* Ph.D. thesis, Department of Artificial Intelligence, University of Edinburgh.

Kalfoglou, Y. (2000b). On the convergence of core technologies for knowledge management and organisational memories: ontologies and experience factories. In *proceedings of the ECAI 2000 Workshop on Knowledge Management and Organisational Memories(W11-KMOM'00), Berlin, Germany.*

Kalfoglou, Y., & Robertson, D. (1999a). A Case Study in Applying Ontologies to Augment and Reason about the Correctness of Specifications. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pp. 64–71. Also as: Research Paper No.927, Dept. of AI, University of Edinburgh.

Kalfoglou, Y., & Robertson, D. (1999b). Managing Ontological Constraints. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden* http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/. Also as: Research Paper No.948, Dept. of AI, University of Edinburgh.

Kalfoglou, Y., & Robertson, D. (1999c). Use of Formal Ontologies to Support Error Checking in Specifications. In Fensel, D., & Studer, R. (Eds.), *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management(EKAW99), Dagstuhl, Germany*, pp. 207–220. Also as: Research Paper No.935, Dept. of AI, University of Edinburgh.

Kalfoglou, Y., & Robertson, D. (2000). Applying Experienceware to support ontology deployment. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, SEKE2000, Chicago, IL, USA*, pp. 266–275.

Lee, J., Gruninger, M., Jin, Y., Malone, T., Tate, A., & Yost, G. (1996). The PIF Process Interchange Format and framework - version 1.1. Working paper No.194, MIT Center for Coordination Science.

Lee, J., Gruninger, M., Jin, Y., Malone, T., Tate, A., Yost, G., & other members of the PIF working group (1998). The PIF Process Interchange Format and framework. *Knowledge Engineering Review*, *13*(1), 91–120.

Lee, J., & Malone, T. (1990). Partially Shared Views: A Scheme for Communicating between Groups Using Different Type Hierarchies. *ACM Transactions on Information Systems*, *8*(1), 1–26.

Lehman, M. (1990). Keynote address. In *Proceedings of the 4th International Workshop on Computer-Aided Software Engineering, IEEE CASE'90.* Irvine, California, USA.

Luqi, & Cooke, D. (1995). How to combine nonmonotonic logic and rapid prototyping to help maintain software. *International Journal of Software Engineering and Knowledge Engineering*, *5*(1), 89–118.

Mark, W., Dukes-Schlossberg, J., & Kerber, R. (1995). Ontological Commitment and Domain-Specific Architectures: Experience with Comet and Cosmos. In N.J.I.Mars (Ed.), *Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB & KS'95), Twente, The Netherlands*, pp. 33–45.

Mark, W., Schlossberg, J., Tyler, S., & McGuire, J. (1994). Cosmos: A System for Supporting Engineering Negotiation. *Concurrent Engineering: Research and Applications*, *2*, 173–182.

Mark, W., Tyler, S., McGuire, J., & Schossberg, J. (1992). Commitment-Based Software Development. *IEEE Transactions on Software Engineering*, *18*(10), 870–884.

Meseguer, P., & Preece, A. (1995). Verification and validation of knowledge-based systems with formal specifications. *The Knowledge Engineering Review*, *10*(4), 331–343.

O'Keefe, M., Balci, O., & Smith, E. (1987). Validating expert system performance. *IEEE Expert*, *2*(4), 81–90.

Polyak, S., Lee, J., Gruninger, M., & Menzel, C. (1998). Applying the Process Interchange Format(PIF) to a Supply Chain Process Interoperability Scenario. In Gomez-Perez, A., & Benjamins, R. (Eds.), *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England.*

Polyak, S. (1998). A Supply Chain Process Interoperability Demonstration using the Process Interchange Format(PIF). Research Paper No.917, Department of Artificial Intelligence, University of Edinburgh.

Preece, A., & Shinghal, R. (1994). Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, *9*(8), 683–702.

Preece, A., Shinghal, R., & Batarekh, A. (1992). Principles and practice in verifying rule-based systems. *The Knowledge Engineering Review*, *7*(2), 115–141.

Robertson, D., Bundy, A., Muetzefeldt, R., Haggith, M., & Uschold, M. (1991). *ECO-LOGIC Logic-Based Approaches to Ecological Modelling.* MIT Press. ISBN 0-262-18143-6.

SICStus (1995). SICStus Prolog User's Manual. ISBN 91-630-3648-7, Intelligent Systems Laboratory - Swedish Institute of Computer Science.

Sterling, L., & Shapiro, E. (1994). *The Art of Prolog* (4th edition). MIT Press. ISBN 0-262-69163-9.

Swartout, B., Patil, R., Knight, K., & Russ, T. (1996). Toward Distributed Use of Large-Scale Ontologies. In *Proceedings of the 10th Knowledge Acquisition, Modeling and Management Workshop(KAW'96),Banff,Canada.*

Treuer, J., & Wetter, T. (1993). *Formal specification of Complex Reasoning systems.* Ellis-Horwood.

Uschold, M. (1998). Where are the Killer Apps?. In Gomez-Perez,A. and Benjamins,R. (Ed.), *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England.*

Uschold, M., Clark, P., Healy, M., Williamson, K., & Woods, S. (1998). An Experiment in Ontology Reuse. In *Proceedings of the 11th Knowledge Acquisition Workshop, KAW98, Banff, Canada.*

Uschold, M., & Gruninger, M. (1996). Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, *11*(2), 93–136.

Uschold, M., & Jasper, R. (1999). A Framework for Understanding and Classifying Ontology Applications. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden.* http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

Valente, A., & Breuker, J. (1996). Towards Principled Core Ontologies. In *Proceedings of the 10th Knowledge Acquisition of Knowledge-Based Systems Workshop(KAW'96),Banff,Canada.*

Valente, A., Russ, T., MacGrecor, R., & Swartout, W. (1999). Building and (Re)Using an Ontology for Air Campaign Planning. *IEEE Intelligent Systems*, *14*(1), 27–36.

van der Vet, P., & Mars, N. (1998). Bottom-Up Construction of Ontologies. *IEEE Transactions on Knowledge and Data Engineering*, *10*(4), 513–526.

Visser, P., Jones, D., Bench-Capon, T., & Shave, M. (1998). Assessing Heterogeneity by Classifying Ontology Mismatches. In Guarino, N. (Ed.), *Proceedings of 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy*, pp. 148–162. IOS Press.

Waterson, A., & Preece, A. (1999). Verifying Ontological Commitment in Knowledge-based Systems. *Knowledge-Based Systems*, *12*, 45–54.

WfMC (1996). Workflow Management Coalition: Abstract Specification. WFMC-TC 1012, WfMC. Interoperability demonstration presented at the 1996 Business Process and Workflow Conference in Amsterdam.