Managing Ontological Constraints

Yannis Kalfoglou & David Robertson
School of Artificial Intelligence,
Institute for Representation and Reasoning,
Division of Informatics,
University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN, Scotland
Email: {yannisk,dr}@dai.ed.ac.uk

Abstract

We explore the use of ontological constraints in a new way: how they can be deployed to facilitate a software system's formal evaluation. We present a formalism for ontological constraints and elaborate on a meta interpretation technique in the field of ontologies. Ontological constraints often need enhancements to capture application-specific discrepancies. We propose an editing system that provides guidance in building those constraints and we explain how this helps us to detect conceptual errors that reflect a misuse of ontology's constructs. We describe a multilayer architecture for performing such checks and we demonstrate its usage via an example case. We speculate on the potential impact of the approach for the system's design process.

1 Introduction

In the AI ontological community most work is focused on the two issues that ontologies claim to deliver: knowledge sharing and reuse. In the recent years, developments in the field have been fast and new approaches on developing, browsing and editing ontologies have emerged. However, the observed dearth of applications that is reported in [Uschold,M., 1998] is a hard reality and ontological engineers are working hard to alleviate the situation. One problem in doing this is in finding practical ways to make use of the collection of axioms(often in first-order predicate logic or similar notation) which are intended by developers of formal ontologies to constrain their use. This paper gives a novel way of making use of these.

1.1 Ontological constraints

These are usually have the form of ontological axioms. We describe in textual form an axiom of a formal ontology, the Process Interchange Format(PIF) ontology([Lee *et al.*, 1998]):

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

This axiom can be formalised in first order logic as follows:

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

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. For example, the following definition, which might be part of a logic program using the ontology

$$participates\_in(O, A, T) \leftarrow exists\_at(O, T) \wedge performs(O, A).$$

could be erroneous with respect to the axiom given above, depending on how *is_occurring_at* is defined for activities. However, it uses valid ontological constructs which makes it difficult to detect. It reflects a misunderstanding of ontology's semantics and can only be detected by checking its conformity with the existing axiomatisation.

As we describe later in the paper existing ontological axioms need to be enhanced to capture domain-specific discrepancies. This practice is often encountered when we move from top level ontologies down to domain specific ontologies where the order of specificity increases[1].

1.2 Organisation of this paper

This paper is organised as follows: in section 2 we present a formalisation for ontological constraints along with a meta interpretation technique that makes it possible to check whether goals that succeed in the proofs violate those constraints. We present a flexible multilayer approach to facilitate this sort of checks and in section 3 we elaborate on tools that help us to construct ontological constraints. We discuss benefits of the approach and demonstrate a brief use of the multilayer architecture in sections 4 and 5 respectively. In section 6 we give pointers to related work.

2 Formalising ontological constraints

In this section we present a formalisation for ontological constraints and how we chose to represent them in a specific error format. In section 2.1 we elaborate on a generic inference mechanism which is made explicit through meta-interpretation and we present an error checking mechanism tailored to the particular ontological constraints. In section 2.2 we show how we realise this theoretical model in a multilayer approach that combines the inference mechanism along with the error checking and gives us additional advantages which we discuss in the sequel. The ontological constraints adopt the following notation:

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

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.

We are interested in proofs over existentially quantified goals, so the formula (α) is transformed into normal form where the '$\Rightarrow$' operator below connects the original(left) to the transformed version(right):

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

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 since these expressions will be used to test for errors on goals in the proof.
Hence, the right part of formula (β) will be:

[1] In [Chandrasekaran *et al.*, 1999] the authors provide an account for various kinds of ontologies reported in the literature and elaborate on the role of domain-specific and method ontologies for KBSs development.

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

So, for example, given the ontological constraint: $P(X) \rightarrow \neg(Q(X) \wedge R(X))$, a translation to normal form will be:

(δ) $\forall X.P(X) \rightarrow \neg(Q(X) \wedge R(X)) \Rightarrow \neg\exists X.P(X) \wedge Q(X) \wedge R(X)$.

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

(ϵ) $\neg\exists X.P(X) \wedge Q(X) \wedge R(X) \Rightarrow error(P(X),(Q(X) \wedge R(X)))$.

2.1 Meta interpretation in Ontologies

A common inference strategy in trying to establish truth when proving goals is the goal reduction. This is made explicit through meta interpretation based on the standard 'vanilla' model[2]. We use by convention the predicate $solve/1$ as follows: $solve(Goal)$ is true if $Goal$ is true with respect to the program being interpreted. The inference mechanism is given in FOPC notation:

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

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.

We define an error checking mechanism tailored to detect ontological errors. It works with the inference mechanism described above and uses ontological constraints of the form given in section 2. The mechanism is given in FOPC notation:

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

We use the following predicates in the mechanism:

- $onto\_solve(Goal,Errors)$ to denote that in trying to prove a goal, $Goal$, we discovered ontological errors, $Errors$;

[2]described in [Sterling,L. and Shapiro,E., 1994].

- *error_check*$(Goal, Set)$ to denote that the error check we performed to goal, $Goal$, yield the set of ontological errors, Set;

- *ontological_error*$(Goal, Error, Errors)$ to denote that we found an ontological 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. Line (D) states that we get no errors on goal X when X cannot be proved through the inference mechanism or X is a conjunctive goal or X is a negative goal.

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 ontological 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 ontological error on that goal.

Line (G) implements the interface to ontological constraints. It states that there exist an ontological 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 $solve/1$) without changing the error checking strategy(expressed as *onto_solve*$/2$).

However, when we implement this inference mechanism separately from the error checking one we have to face some interesting issues. As we describe in the next section, 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 specification that adopts the ontology constructs from the ontological constraints. In doing this, ontological constraints are viewed as a meta-level specification which makes it possible not only to perform the same sort of checks on them with respect to meta-level constraints but to apply the checks selectively on the layers you want.

To realise this we implemented a multilayer architecture, which we describe in the next section, and we combined the inference mechanism with the error checking one. The whole combined meta-interpreter is explained in [Kalfoglou,Y. and Robertson,D., 1999b]. To summarise the section we list below issues that are tackled with the multilayer approach along with pointers to the rest of the paper where we discuss their implementation:

- how to layer the error checks? see, section 2.2, realisation of multilayer approach;
- where do ontological constraints come from? see, section 3, where an editing system is discussed;
- how are multilayer errors explained? - see, section 5 where an error check is realised at a meta layer

2.2 Multilayer architecture

In this section we describe the multilayer architecture we adopt. We include a diagrammatic version of the multilayer architecture at figure 1 where we include at the left part a magnification of a layer. A description of a layer follows:

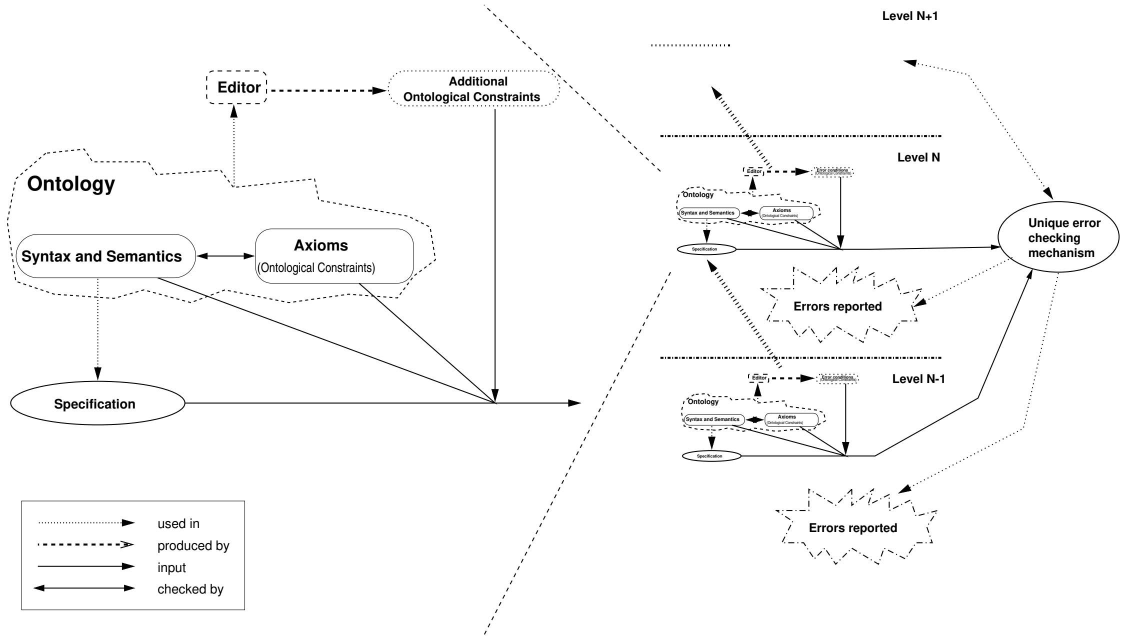

Figure 1: **The Mutlilayer architecture.** The right part shows the approach as a whole, whereas the left part is a magnification of a layer.

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. This allows us to interpret the specification declaratively based on the underpinning computational logic while the procedural interpretation makes it possible to check the correctness of the specification automatically by using the meta-interpreter mechanism.

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. Other constructs are normally added to customise the specification for the particular domain of application. These will not benefit from the meta-interpreter error checking mechanism but can be checked in the normal way.

Ontological axioms are used to verify the correct use of ontological constructs in the specification. Their role is to ensure that the correct interpretations of ontological constructs will be given. This can be done automatically with the meta-interpreter and as we described in section 1.1 there can be extra, application specific ontological constraints that are constructed with the use of supporting tools, like the editor we describe in section 3.

The specification along with the ontological constraints is interpreted by the meta-interpreter. Whenever a statement in the specification does not satisfy the ontological constraints an error is reported.

The right part of figure 1 shows our approach in a mutlilayer 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 side effects. However, our proofs that error 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. A brief example of the multilayer approach is given in section 5, 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 in section 2.1. 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 description format given in section 1.1.

3 Building ontological constraints

The existing set of ontological constraints can be augmented by adding extra constraints. A similar approach was introduced in [Uschold *et al.*, 1998], where the authors report that they had to add extra ontological axioms in their specification formulation in order to prove some concepts that were treated as primitives in the underpinning ontology. We have elaborate further on the notion of defining extra, application specific ontological constraints which results in a customised axiomatisation. However, such an axiomatisation it might be difficult to reuse when it utilises constructs that are not part of the original ontology.

We believe that the user should be provided with support in constructing both kinds of constraints, generic and application specific ones. Whatever the choice, the constructs used in the constraints should conform to the ontology vocabulary and be consistent with the existing constraints. In the case of application specific constraints these checks can only be done with respect to those constructs that are drawn from the ontology.

With this aim in mind we have build two editing tools that facilitate the construction of ontological constraints and provide builtin checks for conflicts and subsumption occurrence. We will elaborate on design choices and use of these tools via two short example cases: a construction of a generic constraint borrowed from the Process Interchange Format(PIF)[3] ontology and a construction of an application specific constraint borrowed from

[3]documentation for the PIF ontology can be found in [Lee *et al.*, 1998].

our work([Kalfoglou,Y. and Robertson,D., 1999a]) in the AIRCRAFT ontology[4,5] application.

In the case of building a generic constraint the user 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. The result of editing an axiom of the core PIF, which is given below in textual form:

"The *before* relation holds only between timepoints"

is as follows in FOPC notation:

$$before(P, Q) \rightarrow point(P) \wedge point(Q)$$

If we are interested in using the constraint as an axiom then at this stage we are ready to add it to the existing ontology axiomatisation after a conflict and subsumption checking is done. The sort of conflict check we apply declares two axioms as being contradictory to each other if after a matching of their heads have been successful, their subgoals have the same symbols but still are not unifiable after having their variables temporarily bounded. The subsumption check will ensure that for two axioms that their heads match, we won't let a more generic one to 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: $before(A, B) \rightarrow point(foo(A)) \wedge point(foo(B))$, this will cause a subsumption warning to the user since variables P and Q from the new axiom will subsume predicates $foo(A)$ and $foo(B)$, 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[6].

However, if the constraint is to be used as an error condition not to be satisfied by the specification then we are interested in the normal form of the above logical assertion. We apply a standard set of rewrite rules to produce the normal form as a conjunction of literals. We apply these rules exhaustively, and the axiom given above will automatically translated to:

$$\neg(before(P, Q) \wedge \neg(point(P) \wedge point(Q)))$$

This will be transformed automatically to the error condition format we adopt:

$\texttt{error(}\mathit{Index}\texttt{, before(P,Q), }\neg\texttt{(point(P) }\wedge\texttt{ point(Q)))}$

where *Index* denotes the layer that this condition belongs to which is the layer above the specification to be checked in terms of the multi-layer architecture we presented in section 2.2.

The second editing tool facilitates the construction of application specific ontological constraints. It uses an heuristic for retrieving ontological relations as candidate parts of the constraint to be build. The taxonomy of concepts is taken into account to constrain the choices of the user in selecting candidate relations and to verify that the maximum possible set of relations is retrieved. Apart from these relations there is a choice of augmenting the constraint with extra predicates or new relations to express complicated constraints whenever this is not possible with the available relations set. As in the previous editing tool the distribution of variables that will be shared among the constraint's literals is done automatically.

[4]the AIRCRAFT ontology is documented online, in February of 1999 the URL was: http://www.isi.edu/isd/ontosaurus.html.

[5]In [Valente *et al.*, 1999] the authors describe a use of the AIRCRAFT ontology.

[6]this is a debatable matter and [Visser *et al.*, 1998] elaborate on that issue.

Let us look in detail the construction of the constraint we used in the AIRCRAFT ontology experiment([Kalfoglou,Y. and Robertson,D., 1999a]). Figure 2 illustrates a selection of ontological relations drawn from the AIRCRAFT ontology which will be processed by the extraction mechanism.

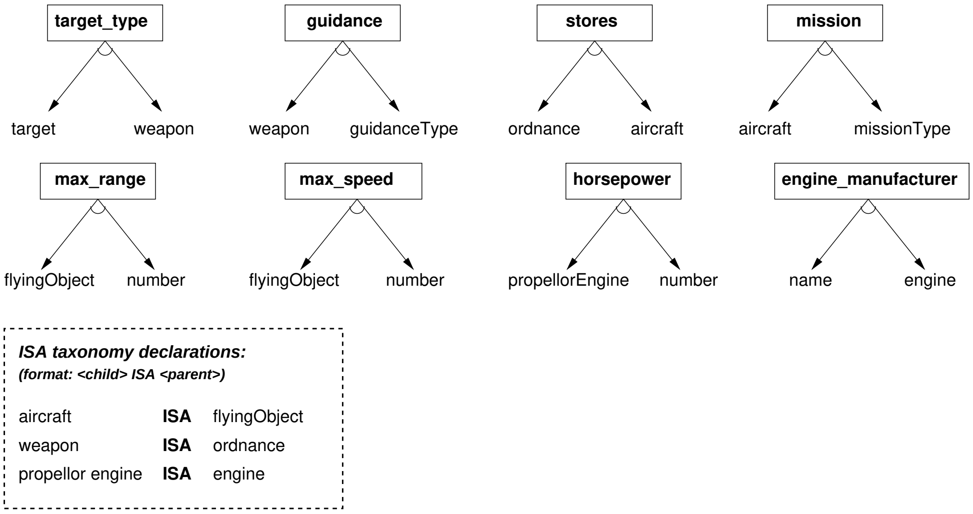

Figure 2: **A selection of relations from the AIRCRAFT ontology.**The relations are represented as rectangular boxes with the concepts that hold over stemming from them. We also include a subset of the `ISA` taxonomy declarations

The relations retrieval is initiated by typing a keyword which is an ontology concept chosen by the user. In our example this keyword was the concept *weapon*. As we can see from figure 2 the relations that hold directly over this concept are `target_type` and `guidance`. These will be the first to accumulate as candidate ones. However, by using the taxonomy of the ontology which declares that *weapon* is a type of *ordnance*, we will retrieve the relation `stores` as well. There are no more relations to be retrieved based on the particular concept, so the mechanism will proceed to retrieve potential relations based on associated keywords. Those will be the remaining concepts from the relations that already have been retrieved: *target*, *guidanceType* and *aircraft*. The same process is applied for each of the new concepts which will result in the retrieval of three more relations: `mission`, `max_range` and `max_speed`. Any new concepts that will accompany the new relations(i.e. *number*) will not be regarded as new keywords to try since this will result in retrieval of relations dissociated with the original keyword. Therefore, at this point the algorithm terminates since there are no more relations to retrieve nor there are concepts from the original retrieved relations set that haven't been checked yet.

Once the candidate relations have been retrieved the user selects the ones he wants to include in the constraint. In our case, those were: `stores` and `target_type`. The next step is to bind the variables that will be shared among the relations. Again the taxonomy of concepts is taken into account to automatically bind variables that are of the same type or connected with an `ISA` relation. This is the case for the concept *weapon* which is child of *ordnance*. Their places in the relations will be occupied by the same variable. In the sequel, the user defines a name for the constraint to be built, in our case `navyThreat` and the type of variables that will be used in the constraint's head. Those are selected from the ones that used in the constraint body and optionally the user can add the corresponding concepts in the constraint's body for type checking. Figure 3 shows a screenshot of the editor at this stage of constraint building.

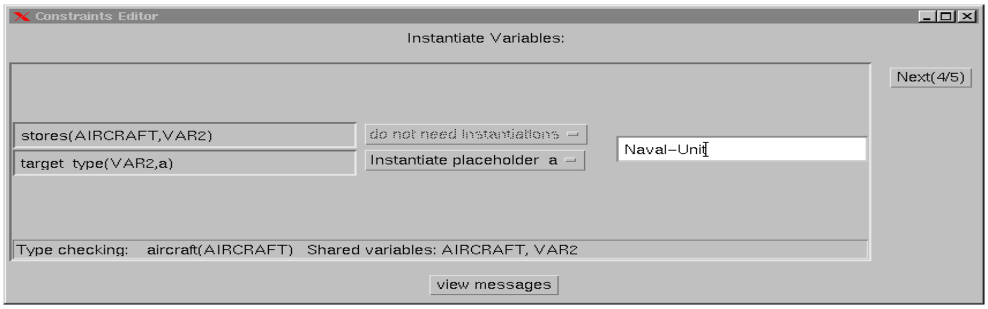

Figure 3: A screenshot that shows the step of instantiating variables in the constraint's relations.

As we can see the user has choose to add the concept *aircraft/1* to the constraint's body for type checking following his declaration that the constraint head, *navyThreat*, will be a unary predicate and defined over *aircraft*. Only one variable has to be instantiated, the second variable of relation `target_type`. There can be an augmentation of the constraint with extra predicate or relations with respect to this variable, but in our case, this was bounded to the constant 'Naval-Unit'.

The final step is to link the constraint literals with logical connectives($\neg, \vee, \wedge$). After a conflict and subsumption checking is done the constraint is ready to add in the constraints base and transformed automatically in the error condition specific format. We give them below in FOPC notation and in error condition format:

in FOPC:

$$\neg(navyThreat(AIRCRAFT) \wedge \neg(aircraft(AIRCRAFT) \wedge stores(AIRCRAFT, VAR2) \wedge target\_type(VAR2, \text{'Naval-Unit'}))).$$

in error condition format:

error(*Index*,navyThreat(AIRCRAFT),(¬ (aircraft(AIRCRAFT)∧ stores(AIRCRAFT,VAR2)∧ target_type(VAR2,'Naval-Unit')))).

3.1 Implementation

The editing tools were written in Java and are executable as applets in Web browsers with Java support. Alternatively they can run as Java applications. They serve as the front-end of the ontological constraints manager implemented in Prolog. 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[7].

4 Benefits of the approach

The use of ontological constraints we propose has several benefits for the system's design process. We summarise below the most important of them along with pointers to documented work:

[7] the package is documented online; the URL in February of 1999 was: http://www.sics.se/isl/sicstus/sicstus_12.html.

- augmentation of specifications with formally defined constructs drawn from the underpinning ontology. We argue that these constructs might be reused in other similar applications which may result in a cost-effective solution for the design process. (see, for example, [Kalfoglou,Y. and Robertson,D., 1999a]);
- use of ontological constraints to detect conceptual errors in specifications that use ontological constructs. This has a potential impact to the early phases of software design since it makes it possible to detect errors that were previously uncaught. Moreover, these ontological constraints might be reused to detect similar kind of errors in other applications. (see, for example, [Kalfoglou,Y. and Robertson,D., 1999b]);
- the multilayer architecture might be used to ease the mapping of ontologies and spot mismatches of concepts and relations in an arbitrary number of layers. This may be useful when applied to methodologies that impose a layered ontology design approach(see, for example, [van der Vet,P. and Mars,N., 1998]).

5 Use of multilayer architecture

We will demonstrate briefly the use of the multilayer architecture via an example case borrowed from [Kalfoglou,Y., 1998] which is based on the formal specifications described in [Robertson *et al.*, 1991]. Our example case is a fragment of the *Rabbit-Grass energy flow* model described in that book. For brevity we do not include details of the model here, but we list below the representation of state variables and how their values obtained in the specification format we adopt:

```
specification(0,(state_variable(S,T,N) ← initial_time(T) ∧
                                         initial_value(S,N))).
specification(0,(state_variable(S,T,N) ← ¬ initial_time(T) ∧
                                         previous_time(T,Tp) ∧
                                         state_variable(S,Tp,Np) ∧
                                         input_and_output_flows(S,Tp,Fi,Fo) ∧
                                         N is Np + Fi - Fo)).
```

The declarative meaning of this top level goal of the specification is as follows: at the initial time point of the simulation, a state variable will have whatever value was set as its initial value. At times other than initials, a state variable will have the value, N, if: its value, Np, at the previous time point, Tp, is known; the sum, Fi, of all input flows at the previous time are known; the sum, Fo, of all output flows at the previous time are also known; and N equals Np+Fi-Fo.

The definition of a state variable is recursive; obtaining a value for the state variable at some time, $\hat{T}$, depends on obtaining its value at some previous time, Tp. The model also includes definition of the previous_time/2 clause, calculation of the input and output flows and definitions of initial values for initial time, state variables and for the parameters that regulate the flows.

This fragment belongs to the lowest layer of the architecture, layer 0, and conforms to the syntax and semantics of the underpinning ontology. At a layer above, layer 1, we have the following ontological constraint expressed in the form of an erroneous statement:

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

The declarative interpretation of this condition is as follows: for any given state variable, S, that has the value, N, at time, T, there exists a state variable, S1, which is different than the previous, and has the value, N1, at the same time(T), and N1 is greater than N, then it is an error whenever S1 is lower in food chain than S at time T. This error condition, if satisfied, will demonstrate the presence of a potential conceptual error. Note that this error condition is tailored to the specific application and holds for the existing values of the regulating parameters and only for the parameters defined in the model.

At this point a new relation is introduced: the *lower_in_food_chain*. This, along with any accompanied constructs are added in the architecture at a layer 1 which is above the existing specification layer, 0. The relative fragment is given:

```
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))).
specification(1,(eating_flow(photosynthesis) ← true)).
specification(1,(eating_flow(defecation) ← true)).
```

This recursive definition of the *lower_in_food_chain* relation is used to check whether a state variable, S1, is lower in the food chain from state variable, S, at time, T, based on the values of the flows for the particular state variables and their status as eating flows. This is expressed via the predicate *eating_flow* which is an extra concept and therefore added in layer 1 of the specification.

With the flexible architecture we can define ontological constraints to check this layer as we did for the layer 0. The same meta-interpreter program will be used to check the conformity with the ontological constraint given below:

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

This constraint, denotes that it is an error whenever an eating flow is not *grazing*. It is placed in layer 2 to monitor the specification at the layer beneath, layer 1.

We can execute the meta-interpreter program to check the specification with respect to ontological constraints defined above starting from the lowest layer. This is the default way of using the meta-interpreter as we described in section 2. However, there is an option to start checking from the layer we want. In our example we followed this and started checking from layer 1 that includes the relation we want to check: *lower_in_food_chain*. The answer to the Prolog query: 'Is *rabbit* lower in the food chain than *grass* at time 2?' will be, erroneously, a positive one but our meta interpreter will report the following errors:

```
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 constraint of layer 2, that is the inequality with the *grazing* flow. Extra information concerning the execution path is provided which is useful for locating errors. The discrepancy was found because the error condition was satisfiable by the interpreter, since there is not exist a definition of *eating flow* in the specification that matches to *photosynthesis* or *defecation*.

6 Related work

In this section we provide pointers to relevant work in the area of ontological constraints management. Although there is not exist a deep core of background work that deploys the constraints in the way we do we list below indicative contributions that tackle similar problems. The terminology used is often different and the use of constraints differs from the one we explore though there is an overlap in the intention of the systems.

In [Gruninger & Fox, 1995], the authors report that as part of the TOVE project[8] they introduced the notion of an ontology's competence: a set of queries that the ontology can answer. These queries also evaluate the expressiveness of the ontology that is required to represent them and characterise their solutions. These competency questions do not generate ontological commitments, they are used to evaluate the ontological commitments have been made. The value of ontological commitment and its role to ontology development and application is discussed in [Guarino,N., 1998]. The author states that the ontological commitment should be made explicit when applying the ontology in order to facilitate its accessibility, maintainability, and integrity. This will lead to an increase of transparency for the application software which based on that ontology.

In the area of characterisations of the discrepancies that we described a way of detecting them, [Visser *et al.*, 1998] elaborate on the notion of ontology mismatches and provide a classification for such mismatches. Those mismatches occur when we try to map heterogeneous systems and arose from the differences of their underpinning ontologies. Their contribution is a set of guidelines that helps to identify the type of mismatch and assess the level of difficulty in resolving them. In [Gomez-Perez,A., 1996], the author discuss criteria that should be used to verify consistency and completeness of an ontology.

In the area of software development an indicative work that points to the use of commitments which must be met by the system's modules is described in [Mark *et al.*, 1992]. The authors elaborate on the idea of constraints that each of the system's modules has to conform to. These constraints are drawn from the underpinning ontology and the system that manages them, the *Comet*, aims at providing context-specific guidance to the system's architect on what modules may be relevant to include in the design, and what design modifications will be required in order to include them.

Acknowledgements

The research described in this paper is supported by a European Union Marie Curie Fellowship(programme: Training and Mobility of Researchers) for the first author and a EPSRC IT Advanced Fellowship for the second author.

[8] the TOVE ontology is electronically accessible via the URL(on February of 1999): http://www.ie.utoronto.ca/EIL/tove/toveont.html.

References

[Chandrasekaran *et al.*, 1999] Chandrasekaran, B., Josephson, R. and Benjamins, R. January 1999: What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14(1):20–26.

[Gomez-Perez,A., 1996] Gomez-Perez,A. 1996: A framework to Verify Knowledge Sharing Technology. *Expert Systems with Application*, 11(4):519–529. Also as Stanford's University, Knowledge Systems Laboratory, Technical Report, KSL-95-10.

[Gruninger & Fox, 1995] Gruninger, M. and Fox, M.S. August 1995: Methodology for the Design and Evaluation of Ontologies. In *Proceedings of Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Quebec,Canada.*

[Guarino,N., 1998] Guarino,N. June 1998: Formal Ontology and Information Systems. In *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy.* IOS Press, pages 3–15.

[Kalfoglou,Y., 1998] Kalfoglou,Y. December 1998: Applying Ontological Engineering to Ecological Modelling: Detection of Conceptual Errors. Technical Paper No.47, Department of Artificial Intelligence, University of Edinburgh.

[Kalfoglou,Y. and Robertson,D., 1999a] Kalfoglou,Y. and Robertson,D. January 1999: An Experiment in Applying Ontologies to Augment and Reason about the Correctness of Specifications. Research Paper No.927, Department of Artificial Intelligence, University of Edinburgh.

[Kalfoglou,Y. and Robertson,D., 1999b] Kalfoglou,Y. and Robertson,D. May 1999: Use of Formal Ontologies to Support Error Checking in Specifications. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management(EKAW99), Dagstuhl, Germany.* Also as: University of Edinburgh, Dept. of AI, Research Paper No.935.

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

[Mark *et al.*, 1992] Mark, W., Tyler, S., McGuire, J. and Schossberg, J. October 1992: Commitment-Based Software Development. *IEEE Transactions on Software Engineering*, 18(10):870–884.

[Robertson *et al.*, 1991] Robertson, D., Bundy, A., Muetzefeldt, R., Haggith, M. and Uschold, M. 1991: *ECO-LOGIC Logic-Based Approaches to Ecological Modelling*: MIT Press, ISBN 0-262-18143-6.

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

[Sterling,L. and Shapiro,E., 1994] Sterling,L. and Shapiro,E. 1994: *The Art of Prolog*: MIT Press, 4th edition, ISBN 0-262-69163-9.

[Uschold *et al.*, 1998] Uschold, M., Clark, P., Healy, M., Williamson, K. and Woods, S. April 1998: An Experiment in Ontology Reuse. In *Proceedings of the 11th Knowledge Acquisition Workshop, KAW98, Banff, Canada.*

[Uschold,M., 1998] Uschold,M. August 1998: Where are the Killer Apps? In *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England.*

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

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

[Visser *et al.*, 1998] Visser, P.R.S., Jones, D.M., Bench-Capon, T.J.M. and Shave, M.J.R. June 1998: Assessing Heterogeneity by Classifying Ontology Mismatches. In *Proceedings of 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy.* IOS Press, pages 148–162.