Managing Ontological Constraints

Yannis Kalfoglou
yannisk@dai.ed.ac.uk

David Robertson
dr@dai.ed.ac.uk

School of Artificial Intelligence,
Institute for Representation and Reasoning,
Division of Informatics,
University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN, Scotland

Abstract

We explore the use of ontological constraints in a new way: deploying them in 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 ontological 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 ways of developing, browsing and editing ontologies have emerged. However, the observed dearth of applications reported in [Usc98] 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.

Proceedings of the IJCAI-99 workshop on Ontologies and Problem-Solving Methods (KRR5) Stockholm, Sweden, August 2, 1999

(V.R. Benjamins, B. Chandrasekaran, A. Gomez-Perez, N. Guarino, M. Uschold, eds.)

http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/

1.1 Ontological constraints

These are usually given as axioms in predicate calculus(or similar). We describe in textual form an axiom of a formal ontology, the Process Interchange Format(PIF) ontology([LGJ+98]):

> "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 predicate calculus as follows:

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

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. 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, despite the fact that it uses syntactically valid ontological constructs. It reflects a misunderstanding of ontology's semantics and can only be detected by checking its conformity with the ontological constraints.

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].

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 a 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:

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

[1] In [CJB99] 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.

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

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

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

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

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) $onto\_solve((X1 \wedge X2), E) \leftarrow$
$\quad onto\_solve(X1, E1) \wedge onto\_solve(X2, E2) \wedge$
$\quad E = E1 \cup E2.$
(B) $onto\_solve(\neg X, []) \leftarrow \neg solve(X).$
(C) $onto\_solve(X, E) \leftarrow$
$\quad clause(solve(X), B) \wedge onto\_solve(B, Eb) \wedge$

[2] described in [Ste94].

$error\_check(X, Ex) \wedge E = Ex \cup Eb.$

(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, $Errors$;
- $error\_check(Goal, Set)$ 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 [SIC95]): "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 [SIC95]): "$CopyOfTerm$ 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 [Kal99b]. 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

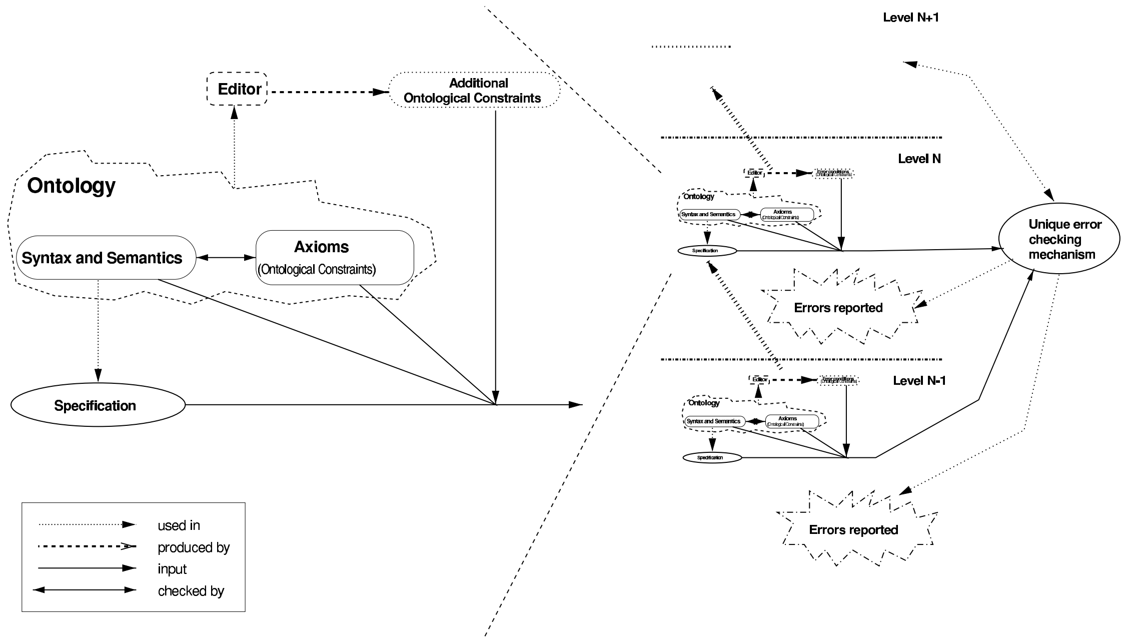


Figure 1: **The Multilayer architecture.** The right part shows the approach as a whole, whereas the left part is a magnification of a layer.

architecture at figure 1 where we include at the left part a magnification of one layer. 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. 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 multilayer 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 [UCH+98], 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. We believe that the user should be provided with support in utilising both kinds of constraints, existing and new ones tailored to the particular application. Whatever the choice, the constructs used in the constraints should conform to the ontology vocabulary and be consistent with the existing constraints. However the application specific constraints can use extra constructs which are not part of the underpinning 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 our work([Kal99a]) 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:

$$\forall P, Q. before(P,Q) \rightarrow point(P) \land point(Q)$$

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: $before(A,B) \rightarrow point(foo(A)) \land point(foo(B))$, 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:

$$\neg\exists P, Q.(before(P,Q) \land \neg(point(P) \land point(Q)))$$

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

error(*Index*, before(P,Q), ¬(point(P) ∧ 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.

Let us look in detail the construction of the constraint we used in the AIRCRAFT ontology experiment([Kal99a]). Figure 2 illustrates a selection of ontological relations drawn from the AIRCRAFT ontology which will be processed by the extraction mechanism.

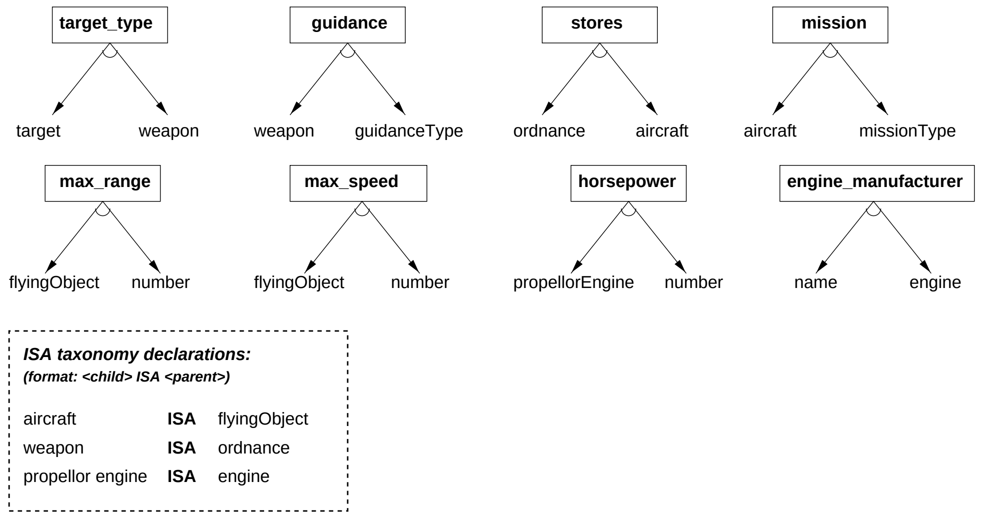
The relations retrieval is initiated by typing a keyword which is an ontology concept chosen by the user. In our example this keyword was the concept *weapon*. As we can see from figure 2 the relations that hold directly over this concept are `target_type` and `guidance`. These will be the first to accumulate as candidate ones. However, by using the taxonomy of the ontology which declares that *weapon* is a type of *ordnance*, we will retrieve the rela-

[3]documentation for the PIF ontology can be found in [LGJ+98].

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

[5]In [VRMS99] the authors describe a use of the AIRCRAFT ontology.

[6]this is a debatable matter and [VJBCS98] elaborate on that issue.



ISA taxonomy declarations:
(format: <child> ISA <parent>)

aircraft	**ISA**	flyingObject
weapon	**ISA**	ordnance
propellor engine	**ISA**	engine

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

tion `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 define 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. This step is illustrated in Figure 3.

As we can see from the set of available variable types: *aircraft*, *target* and *weapon*, the user has choose to define one variable of type *aircraft* in the constraint's head. In the sequel, we 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. Figure 4 shows a screenshot of the editor at this stage of constraint building.

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:

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

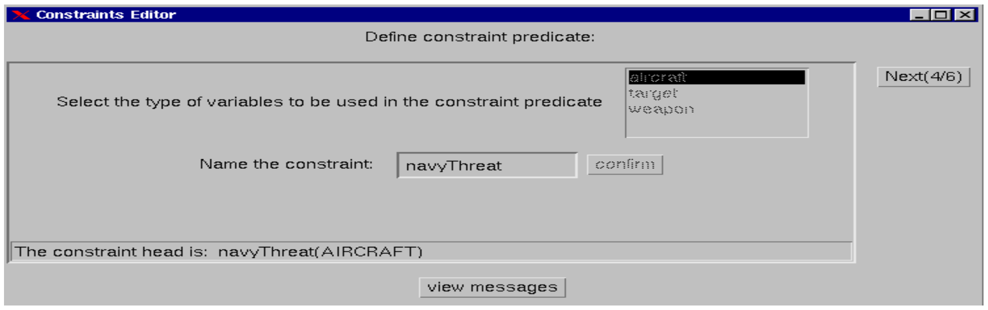
in error condition format:

error(*Index*,navyThreat(AIRCRAFT),
(¬ (aircraft(AIRCRAFT)∧
stores(AIRCRAFT,WEAPON)∧
target_type(WEAPON,'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].

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

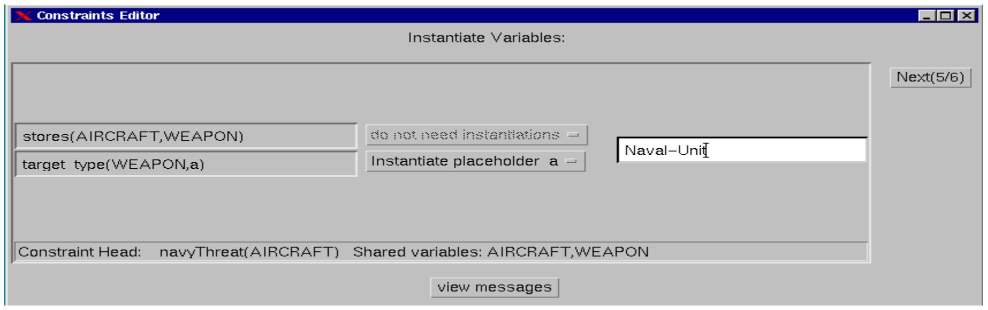


Select the type of variables to be used in the constraint predicate

Name the constraint: navyThreat

The constraint head is: navyThreat(AIRCRAFT)

Figure 3: **A screenshot** that shows the step of defining the head of the constraint to be built along with the type of variables that will be used in it.



stores(AIRCRAFT,WEAPON)

target type(WEAPON,a)

Naval–Unit

Constraint Head: navyThreat(AIRCRAFT) Shared variables: AIRCRAFT,WEAPON

Figure 4: **A screenshot** that shows the step of instantiating variables in the constraint's relations.

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:

- 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. In [Kal99a] we explore the feasibility of this approach via an example case in the domain of ACP(Air Campaign Planning) realised through the AIRCRAFT ontology([VRMS99]);
- 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. In [Kal99b] we present an application of this approach in the domain of ecological modelling;
- 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, [van98]).

5 Use of multi-layer architecture

We will demonstrate briefly a motivating example on the use of the multi-layer architecture borrowed from the PHYSSYS ontology. The ontology, which is is documented in [BAT97], is a formal ontology based upon system dynamics theory as practiced 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 quote [BAT97]:

> "In a separate ontology of mereology a *part-of-relation* is defined that formally specifies the intuitive engineering notion of system or device decomposition. This mereological ontology is then imported into a second separate ontology which introduces *topological connections* that connect mereological individuals. This topological ontology provides a formal specification of what

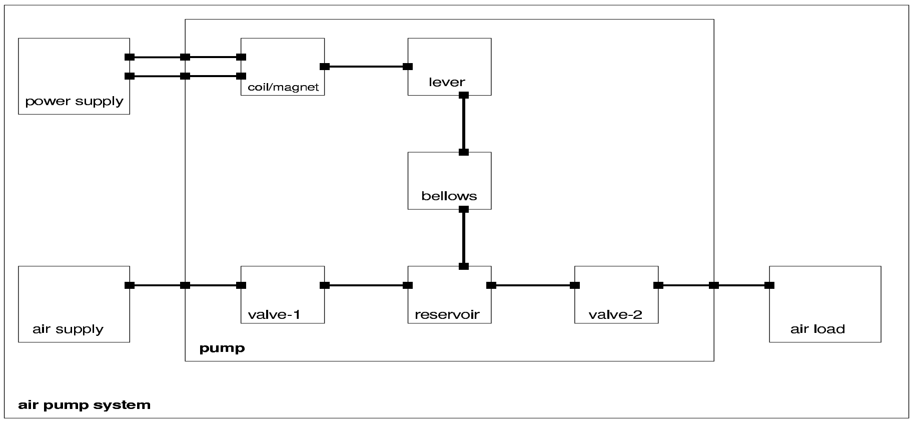


Figure 5: **The component view of a physical system:** It shows the topology for an air pump. Sub-components are drawn inside the area defined by their super-component. The small solid blocks are the interfaces through which components are connected.

> the intuitive notion of a network layout actually means and what its properties are. The ontology of systems theory includes the topological ontology and defines concepts like(open or closed) systems, system boundary, etc., on top of it."

To demonstrate a component viewpoint based on the *component* ontology, we illustrate in figure 5 (borrowed from [BAT97]) a structural-topological diagram for a physical system, like an air pump.

The principles underlying the construction and usage of the PHYSSYS ontology are beyond the scope of this paper; we point the interested reader to [BBWA96] for further details. In the sequel, we elaborate on the application of the multi-layered architecture to each of the three ontologies included in the *component* ontology: mereology, topology and systems theory.

All the ontologies were implemented, originally, in Ontolingua using the Ontology server([FFPR96]). We translated them to the target language we use: in 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. In [Bri99] the author elaborates on the Ontolingua syntax to Prolog translation issue.

5.1 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(2), 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))).
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 5. 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 2.2 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.

5.2 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 predicate `connect/3` to express instances with respect to the air pump system in figure 5. These are:

```
specification(1,(connect(valve1_X_reservoir,
                         valve1,reservoir) ← true)).
specification(1,(connect(reservoir_X_valve2,
                         reservoir,valve2) ← true)).
specification(1,(connect(bellows_X_reservoir,
                         bellows,reservoir) ← true)).
specification(1,(connect(lever_X_bellows,
                         lever,bellows) ← true)).
specification(1,(connect(coilMagnet_X_lever,
                         coilMagnet,lever) ← true)).
specification(1,(connect(airSupply_X_valve1,
                         airSupply,valve1) ← true)).
specification(1,(connect(airSupply_X_valve1,
                         airSupply,pump) ← true)).
specification(1,(connect(powerSupply_X_coilMagnet,
                         powerSupply,coilMagnet) ← true)).
specification(1,(connect(powerSupply_X_coilMagnet,
                         powerSupply,pump) ← true)).
specification(1,(connect(airLoad_X_valve2,
                         airLoad,valve2) ← true)).
specification(1,(connect(airLoad_X_valve2,
                         airLoad,pump) ← true)).
```

Note that the connections `airSupply_X_valve1`, `powerSupply_X_coilMagnet` and `airLoad_X_valve2` are the external connections of the system regarding the pump system whose internal connections are the first five from the above. According to figure 5 the external connections connect the outside individuals with an individual inside pump and the pump itself. 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 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 used to detect errors when a connection connects two 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.

5.3 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 subsystems 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 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 definitions of system instances with respect to the diagram of figure 5: *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 in section 3. As we can see from figure 5 it is error whenever a system of 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.

5.4 Errors detected

This layered specification can be executed to check whether various properties of the air pump system(figure 5) hold with respect to the ontological definitions of each of the three layers: mereological, topological and systems theory.

So, for example, we can check for specific mereological properties by asking whether *lever* is disjoint from *airLoad* giving the relevant Prolog query. As we can see from figure 5 we will get a, correct, positive answer.

If we want to check topological properties of the system we might ask which connections connect the *pump* system with components of the outside world by giving the Prolog query:
`connects(CONNECTION,COMPONENT,pump)?`
an answer to which will give us the possible connection/components set:
(powerSupply_to_coilMagnet, powerSupply),
(airSupply_to_valve1, airSupply),
(airLoad_to_valve2, airLoad).

Finally, we can execute the specification from the systems theory point of view that includes topological and mereological definitions and check, for example, which systems are considered to be closed systems. The answer set will consists of the: *powerSupply, airSupply* and *airLoad* systems.

However, 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 that A is not a mereological part of B the *disjoint* relation holds.

We can detect this sort of error at the mereological layer where it occurs by checking this layer's definitions against their error conditions. This is feasible with the multi-layer architecture since we can define the layer from which to start checking. So, if we ask the model whether *bellows* is disjoint from *bellows* we will get an erroneous positive answer. With the error conditions of layer 3 given above the error is detected and reported:

```
error_condition_satisfied(3,disjoint(bellows,bellows),
                          (equal(bellows,bellows) ∨
                          proper_part_of(Z,bellows) ∧
                          proper_part_of(Z,bellows)))
```

The error was detected because the condition `equal (bellows,bellows)` was proved by the meta-interpreter.

We can extent this layer checking to include the topological layer, layer 1 in the architecture. So, for example, we can check which components are connected by the `reservoir_X_valve2` connection. The answer will be, correctly, that `reservoir` and `valve2` compoments are connected via this connection. However, the erroneous definition of *disjoint* relation is trapped and reported:

```
error_condition_satisfied(3,disjoint(reservoir,reservoir),
                              (equal(reservoir,reservoir) ∨
                              proper_part_of(Z,reservoir) ∧
                              proper_part_of(Z,reservoir)))
error_condition_satisfied(3,disjoint(valve2,valve2),
                              (equal(valve2,valve2) ∨
                              proper_part_of(Z,valve2) ∧
                              proper_part_of(Z,valve2)))
error_condition_satisfied(2,connects(reservoir_X_valve2,
                                           reservoir,valve2),
            (connects(reservoir_X_valve2,reservoir,valve2) ∧
            (disjoint(reservoir,reservoir) ∧
            disjoint(reservoir,valve2)) ∧
            disjoint(valve2,reservoir) ∧
            disjoint(valve2,valve2)))
```

Three errors have been detected: two at the mereological layer with respect to the erroneous definition of *disjoint*, and one at the topological layer where the condition defined over the *connects* relation was proved by the 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 layer above.

The most interesting case is when we check the model from the systems theory point of view. We can ask, for example, whether the *pump* is an open system. We will get a, correct, positive answer. However, the hidden error is trapped and reported:

```
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))]
```

As in the previous case, three errors have been detected: two with respect to the erroneous definition of *disjoint* relation 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 multilayer architecture allows for check in the error conditions themselves and this enabled the detection of *disjoint* relation misuse. We include also the execution path we accumulate that helps to locate errors.

The detection of errors at the topological and systems theory layers, 1 and 0 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 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 Related work

In this section we provide pointers to relevant work in the area of ontological constraints management. Although our method is precise and comparatively straightforward, it touches on a broad range of different but related topics. We summarise each of these below.

In [GF95], 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 [Gua98]. 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 [VJBCS98] 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 [Gom96], the author discuss criteria that should be used to verify consistency and completeness of an ontology.

In the area of software development an example of describing commitments which must be met by system modules is described in [MTMS92]. The authors elaborate on the idea of constraints that each of the system modules has to conform to. These constraints are drawn from the underpinning ontology and the system that manages them (*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(in April of 1999): http://www.ie.utoronto.ca/EIL/tove/toveont.html.

References

[BAT97] P. Borst, H. Akkermans, and J. Top. Engineering Ontologies. *International Journal of Human-Computer Studies*, 46:365–406, 1997.

[BBWA96] P. Borst, J. Benjamin, B. Wielinga, and H. Akkermans. An Application of Ontology Construction. In *Proceedings of ECAI-96 Wokrshop on Ontological Engineering, Budapest,Hungary*, August 1996.

[Bri99] V. Brilhante. Using Formal Metadata Descriptions for Automated Ecological Modeling. In *Proceedings of the AAAI-99 Workshop on Environmental Decision Support Systems and Artificial Intelligence(EDSSAI99), Orlando, Florida, USA*, July 1999.

[CJB99] B. Chandrasekaran, R. Josephson, and R. Benjamins. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14(1):20–26, January 1999.

[FFPR96] A. Farquhar, R. Fikes, W. Pratt, and J. Rice. The Ontolingua Server: a Tool for Collaborative Ontology Construction. In *proceedings of the 10th Knowledge Acquisition Workshop, KAW'96,Banff,Canada*, November 1996. Also available as KSL-TR-96-26.

[GF95] M. Gruninger and M.S. Fox. Methodology for the Design and Evaluation of Ontologies. In *Proceedings of Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Quebec,Canada*, August 1995.

[Gom96] Gomez-Perez,A. A framework to Verify Knowledge Sharing Technology. *Expert Systems with Application*, 11(4):519–529, 1996. Also as Stanford's University, Knowledge Systems Laboratory, Technical Report, KSL-95-10.

[Gua98] Guarino,N. Formal Ontology and Information Systems. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy*, pages 3–15. IOS Press, June 1998.

[Kal99a] Kalfoglou,Y. and Robertson,D. 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*, June 1999. Also as: Research Paper No.927, Dept. of AI, University of Edinburgh.

[Kal99b] Kalfoglou,Y. and Robertson,D. Use of Formal Ontologies to Support Error Checking in Specifications. In D. Fensel and R. Studer, editors, *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management(EKAW99), Dagstuhl, Germany*, pages 207–220, May 1999. Also as: Research Paper No.935, Dept. of AI, University of Edinburgh.

[LGJ$^+$98] J. Lee, M. Gruninger, Y. Jin, T. Malone, A. Tate, G Yost, and other members of the PIF working group. The PIF Process Interchange Format and framework. *Knowledge Engineering Review*, 13(1):91–120, February 1998.

[MTMS92] W. Mark, S. Tyler, J. McGuire, and J. Schossberg. Commitment-Based Software Development. *IEEE Transactions on Software Engineering*, 18(10):870–884, October 1992.

[SIC95] SICStus. SICStus Prolog User's Manual. ISBN 91-630-3648-7, Intelligent Systems Laboratory - Swedish Institute of Computer Science, 1995.

[Ste94] Sterling,L. and Shapiro,E. *The Art of Prolog*. MIT Press, 4th edition, 1994. ISBN 0-262-69163-9.

[UCH$^+$98] M. Uschold, P. Clark, M. Healy, K. Williamson, and S. Woods. An Experiment in Ontology Reuse. In *Proceedings of the 11th Knowledge Acquisition Workshop, KAW98, Banff, Canada*, April 1998.

[Usc98] Uschold,M. Where are the Killer Apps? In Gomez-Perez,A. and Benjamins,R., editor, *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England*, August 1998.

[van98] van der Vet,P. and Mars,N. Bottom-Up Construction of Ontologies. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):513–526, 1998.

[VJBCS98] P.R.S. Visser, D.M. Jones, T.J.M. Bench-Capon, and M.J.R. Shave. Assessing Heterogeneity by Classifying Ontology Mismatches. In N. Guarino, editor, *Proceedings of 1st*

International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy, pages 148–162. IOS Press, June 1998.

[VRMS99] A. Valente, T. Russ, R. MacGrecor, and W. Swartout. Building and (Re)Using an Ontology for Air Campaign Planning. *IEEE Intelligent Systems*, 14(1):27–36, January 1999.