A case study in applying ontologies to augment and reason about the correctness of specifications

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

Abstract

In this paper we investigate how software specifications can benefit from the presence of formal ontologies to augment and enrich their context. This makes it possible to verify the correctness of the specification with respect to formally represented domain knowledge. We present a meta-interpretation technique that allows us to perform checks for conceptual error occurrences in specifications. We illustrate this approach through a case study: we augmented an existing formal specification presented by Luqi & Cooke with a formal ontology produced by the Information Sciences Institute at USC, the AIRCRAFT ontology. In addition, we explore how we can build and use application specific ontological constraints to detect conceptual errors in specifications.

1. Introduction

When we describe a chosen domain in the form of a software specification we can distinguish two kinds of errors: the ones that are related to the mathematical language underpinning the formal model and those that are related to the description of the domain itself. The first type of error is often detectable via normal debugging techniques. The latter type is difficult to detect since it requires subjective knowledge about correct forms of description to be included in the specification. We call this error a conceptual error. It can be pernicious in the software development process because it is introduced in early lifecycle phases and, if undetected, it may be propagated to subsequent phases of the development circle and affect the functionality of entire systems.

Ontologies have become popular in recent years in the area of knowledge-based systems as a means of standardising representation of domain knowledge. We investigate two potential uses of them: that of augmenting specifications with ontological constructs and that of using ontological constraints to check specifications for conceptual errors.

Most research in the ontological communities concentrates on tools and languages for building ontologies(see, for example, [9], [2], [1]) rather than on ways of deploying them. The main contribution of this paper is to show how the sorts of constraints now being used in ontology building can be deployed to test automatically whether parts of an executable specification stray outside the limits of these constraints. A practical benefit of this approach is that the specification preserves its original structure with augmentations being placed automatically on those parts where an error check is required. Moreover, as we will describe in section 4 the specification is enriched by using the formal constructs of the application domain ontology.

This approach is low-cost. The only burden for the specification engineer is to choose an ontology within the application domain to use in the specification. The specification itself does not need to be adapted to the ontological constructs and is executed in the usual way(in Prolog in our example). Checking for ontological errors is done automatically during testing of the executable specification.

This paper is organised as follows: section 2 describes the specification and section 3 the ontology we used in our case study. We elaborate on the augmentations we made to the specification in section 4 which are used in section 5 to detect conceptual errors. We discuss an enhancement to this approach at the end of section 5 and conclude our work in section 6.

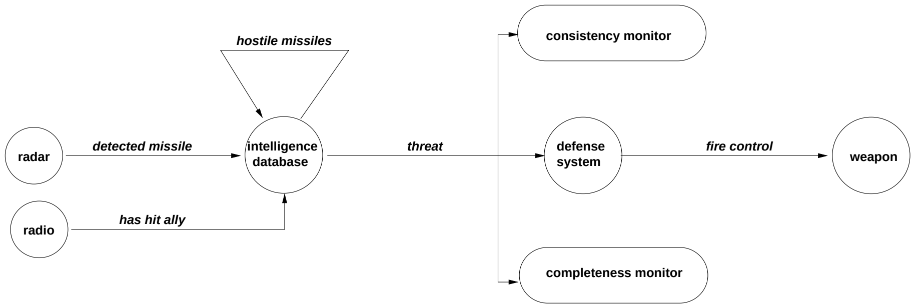

Figure 1. The missile defence prototype(from [6]): *Standard operators of the system are represented in circles(radar,radio,intelligent database, defense system, weapon), execution monitoring operators are represented in rounded boxes (consistency, completeness), data streams are represented as arrows that link the operators(detected missile,has hit ally, hostile missiles, threat, fire control).*

2. Example specification: the 'Luqi & Cooke' model

In our case study we borrowed a model, from Luqi & Cooke([6]), that tackles the problem of detecting context shifts in specifications expressed as Horn clauses. Their paper describes a prototype framework for supporting software evolution with the use of a nonmonotonic query mechanism. They used a problem originally suggested by [5] and based their model on the prototyping language PSDL(Prototype System Description Language).

2.1. Extended answering procedure

The answering mechanism presented by [6] realises the extended logic semantics for nonmonotonic programs that have a unique answer set. Nonmonotonic logics provide formalisms to handle beliefs and, furthermore, allow the retraction of beliefs when new information is presented which contradicts those beliefs. The answering procedure, in Prolog, taken directly from [6], is:

```
answer(P,inconsistency):-P,not(P).
answer(P,true):-P.
answer(P,false):-not(P).
answer(_,unknown).
not(not(P)):-P.
```

Any given goal to prove should be evaluated via the `answer/2` predicate. If both `p` and `not(p)` can be proved then an inconsistency is reported. If an attempt to prove a goal, `p`, succeeds then the answering mechanism will deduce that `p` is true. When `not(p)` can be proved then the mechanism will conclude that `p` is false. If none of the above succeed then it is reported that the answer is unknown. This way of defining a nonmonotonic query mechanism is a little under-defined because it relies on the normal order of selection of clauses(from top to bottom) acting as a sort of filter on the goal - testing first for inconsistency; then for truth; then falsity and finally defaulting to unknown. This means that, if forced to, any goal could be found to be unknown but we have not fixed this logical problem because is peripheral to our discussion. In addition, [6] defined the predicate `not(not(P))` to explicitly state that it means the same thing as `P` since the predicate `not/1`, which is used to express negation, is not a Prolog primitive. When it is known that `p` is false, `not(p)` is explicitly stated in the program[1].

2.2. The PSDL simulator and the missile firing problem

The full PSDL simulator has the following fundamental constructs: operators, data streams, timing constraints, control constraints and timers. The version presented in [6], is a simplified version with no mention of timing constraints and timers. A PSDL program is a network of operator and data streams, augmented by control constraints. The specifier must supply definitions of the initial values for any data streams representing state variables, and define the behaviour of the operators which is characterised by the data values each operator can write into its output streams. Those are used to produce a behavioural model that describes the intended behaviour of the system.

In addition, a context model must be defined to support the rules for computing output values. The context model usually contains facts known to affect the decisions made by the operators and rules which allow the system to discover dynamically when facts may change[2].

[1] For more details on the theoretical aspects of the answering procedure and a comparison of different implementations refer to the original paper of [6].

[2] The whole PSDL simulator is described in detail in [6].

The example used in [6] was originally suggested by [5] and involved British ships in the Falklands War. The software defending the ships was based on the assumption that an *exocet* missile is friendly. The assumption had been true until the Falklands War in which a British ship was sunk by an *exocet*. In [6] a prototype missile defence system for an allied vessel is presented. The major tasks of that system were: to shoot down all hostile missiles, keep track of the radar signatures for particular kinds of missiles, etc. A network of operators were defined, two of which were used to monitor the execution of the simulator and report possible inconsistencies and incompleteness. Those operators implement the answering procedure described above, the consistency operator is fired whenever the `threat` stream carries the value `inconsistency` and the completeness operator whenever value `unknown` is carried by the `threat` stream.

The model also included initial values of the data streams and a behaviour model that defines possible values for data streams that were simulated from external sensors based on random sampling. Another kind of simulation behaviour that was supported is based on computation rules that attached to the declarations of choices to add semantic information. For example, the stream `hostile_missiles` is modelled deterministically, by defining a list of missiles that are reported as having hit an ally.

The whole simulator program is too lengthy to include here but the reader can refer to the original paper [6] for a detailed explanation. However we include a diagrammatic version of it in Figure 1 borrowed from the original paper augmented with a short explanatory caption.

3. The AIRCRAFT ontology of USC/ISI

We have chosen the AIRCRAFT ontology, part of the Ontosaurus project implemented at the Information Sciences Institute(ISI) of the University of Southern California(USC), for our case study. The ontology, which is described in [8] and is electronically accessible from the web[3], covers the domain of military air campaign planning(ACP). It is constructed as a domain specific ontology based on a broad coverage generic ontology, the SENSUS. The construction methodology used is described in [8]; we will focus here on the characteristics that inspired our selection.

The AIRCRAFT ontology supports collaborative construction by system developers themselves. To quote [8]:

> "[. . .]rather than regarding the ontology as a separate resource that is updated periodically, the development and extension of the ontology should occur as part of system development. In this way, the ontology becomes a 'living document' that is developed collaboratively and, at any given time, reflects the terminology that is shared by developers within an effort"

[3] In March of 1999 the URL was: http://www.isi.edu/isd/ontosaurus.html

To maintain coherence of the ontology when changes occur in its structure the need for reasoners that can compare semantic differences was identified. This led to the use of a Description Logic classification system, the LOOM classifier, which was used in the development of Ontosaurus, the ontology server. When a new concept is added or an existing one modified, LOOM is used to classify the new description in its proper place in the ontology and to verify that the new description is coherent. The next section shows how the AIRCRAFT ontology can be used to augment the original model of Luqi & Cooke.

4. Augment the specification with ontology's constructs

The original model of Luqi & Cooke, uses *radar* and *radio* operators to detect information concerning an incoming missile to the vessel's airspace which is then processed by the *intelligence database* operator to determine whether the missile should be regarded as a threat. *Defense system* and *Defensive weapon* operators then treat the missile appropriately.

We adopted the original prototype system to treat hostile aircraft instead of missiles that enter an ally's airspace. The *radio* operator which categorised missiles in the original model, was replaced by an *ontology* operator which classifies aircraft according to the AIRCRAFT ontology. This allows the *intelligence database* operator to classify the form of threat posed by the aircraft. Figure 2 illustrates our augmented model.

We augmented the original simulator and behaviour model with additional constructs that gave us the ability to use ontological constructs in the specification. We summarise those changes along with their impact on the specification. The first of these was done simply because we thought it desirable and wasn't essential. The second and third are extensions to the model which allow it to exploit the ontology without changing the core of the model:

1. we changed the random choice of operators to be executed to a specific one. This simplifies the specification (since we dispense with a random selection function) but does not reduce the space of its possible behaviours because our non-random selection will still select every permitted operator if required. We classified the operators in two categories: the ones that sense and process information concerning the detected aircraft (*radar,ontology,intelligent database*); and those

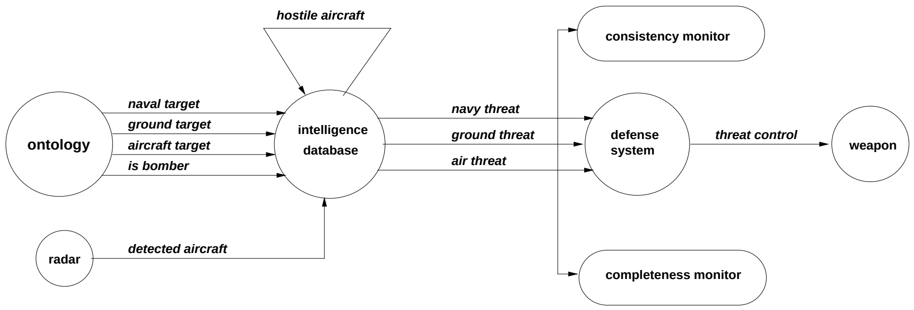

Figure 2. The augmented aircraft defence prototype

that used in order to protect the ally airspace from a potential threat (*defense system,defensive weapon*). We fire the operators in a predefined order: first the 'detect' operators and then the 'protect' operators. We kept the monitoring operators *completeness* and *consistency*, as originally defined;

2. the use of the AIRCRAFT ontology allowed us to enrich the treatment of a threat. We modelled three different kinds of threats (navy, ground, aircraft) in accordance with the formally defined in the ontology target types of the weapons that an aircraft stores;

3. we used the ontology's construct `bomber` to determine dynamically whether a detected aircraft, for which there is no information available about its targets, should be regarded as hostile and treated appropriately. This information is used to help the system to accumulate new contextual information.

The context model indicates how we can determine dynamically a threat(navy, ground or aircraft) and how aircraft is determined to be hostile when there is no information about its targets. Although we could define additional predicates to construct rules that govern those determinations we used constructs of the underpinning ontology for this purpose. By following this route we achieved:

- access to a repository of formally defined domain specific constructs, such as target types for a specific aircraft, missions it performs, etc.;
- ability to share this information with other systems using the same constructs;
- ability to reason about the correctness of the context model with respect to the ontology by using our meta-interpreter error checking mechanism(section 5).

For example, a detected aircraft is considered a navy threat when it stores weapons that target on naval units. The condition is given in logic:

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

Note that the interpretation of the concepts `aircraft/1`, `stores/2` and `target_type/2` is formally given in the ontology. This lightens the workload of the specifier since it has only to define a rule that will hold over these concepts, that of `navyThreat/1`. The selection of constructs to be used can be done by browsing the electronically available ontology.

The same strategy has been used to determine an air threat and a ground threat. The determination of an aircraft as hostile is based on a list that contains all the types of aircraft that are declared as bombers in the ontology.

In the next section we describe out meta-interpreter technique and its application to the model.

5 Detection of conceptual errors

Let us assume that the specifier of the context model defines(working with Horn clauses) erroneously a rule that determines whether or not an aircraft is a navy threat:

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

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

According to the ontology, an F-117 aircraft carries weapons that target on ground and aircraft units but not on naval units. The following fragment of execution demonstrates the erroneous behaviour of the system:

..

read_from(intelligence_database,F-117,detected_aircraft)

wrote_into(intelligence_database,true,navy_threat)
wrote_into(defense_system,shootToProtectShip,threat_control)
..

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

We can use the answering procedure introduced in the Luqi & Cooke model, to define that an F-117 is not a navy threat by stating it explicitly in the specification:
$not(navyThreat(\text{F-117}))$.
This will trigger the execution of the *consistency operator* which will alert the user:
read_from(consistency_monitor,inconsistency,navy_threat).
However, although this will do the job it has weaknesses. As the authors stated:

> "[...][the] goal of the specifier is to construct a specification that is complete in the sense of having a unique answer set [...] However, it is almost certain that the initial versions of the specifications for any real system will not be complete in this sense."

In other words, the declaration of the F-117 as not being a navy threat has to be given by the specifier but in a real defence system that tracks hundreds of types of aircraft it is difficult to maintain a complete catalogue of all classifications for individual aircraft types.

Moreover, the information that an inconsistency has occurred in trying to prove a particular goal is not enough for the specifier to understand the reason and locate the error that triggered the system to behave erroneously.

In the next section we present a mechanism which deals with this problem by utilising ontological constraints to detect conceptual errors.

5.1. Meta-interpreter error checking

Our mechanism, which is based on meta-interpretation, uses the products of ontological engineering such as ontological constraints to detect conceptual errors in specifications that are based on ontologies. An illustration of the approach is given in Figure 3.

Specification construction starts by adopting the syntax and semantics of the ontology. These ontology constructs are not the only ones that will make up the specification. In fact, it is normally impractical to construct an executable specification using only the ontology's constructs. Other constructs are normally added to customise the specification for the particular domain of application.

We use Horn clause logic 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 specification with the meta-interpreter mechanism to reveal potential conceptual errors.

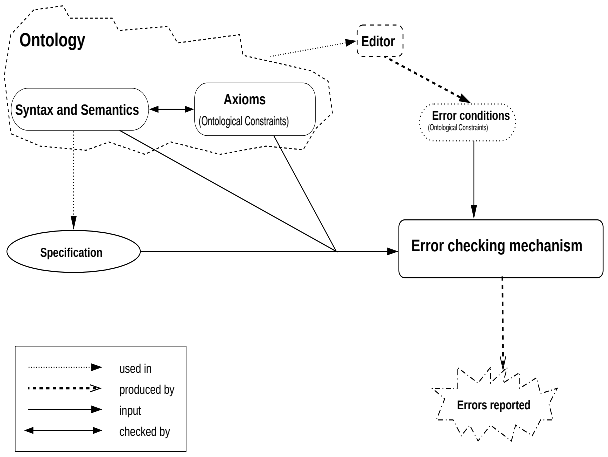

Figure 3. Error detection architecture

Ontological axioms are used to verify the correct use of ontological constructs in the specification. Whenever a statement in the specification will not satisfy the ontological axioms an error will be reported. Additional domain specific error conditions can be defined to facilitate customised error checking. These error conditions will exhibit some undesired behaviour of the specification. Once they will be satisfied during the error checking phase they will demonstrate an error occurrence in the specification. Few conventional ontological constraints are defined in the formalism we adopt and we used an editor to facilitate the task of writing them in the format manipulated by the meta-interpreter.

5.2. Meta interpreter

A meta interpreter is an interpreter for a language written in the language itself [7]. In the domain of Prolog programming, a meta interpreter is program written in Prolog which interprets Prolog programs. This makes it possible to represent in Prolog the search strategy used by Prolog interpreter and to adopt that strategy in various ways, which are often different from the standard interpreter. Among the best known and widely used meta interpreters is the 'vanilla' model. It describes the standard Prolog computation model of logic programs as goal reduction. The model is given below:

$solve(true).$
$solve((A, B)) \leftarrow solve(A) \wedge solve(B).$
$solve(A) \leftarrow clause(A, B) \wedge solve(B).$

This meta-interpreter program has the following declarative meaning: The first clause states that, by convention, the atom `true` is always satisfiable. The second clause states that a conjunction of literals A and B is true if A is true and B is true. The third clause states that A is true if there exists a clause $A \leftarrow B$ in the interpreted program such that B is true.

Our meta-interpreter augments the above model with treatment of disjunctive clauses, negation and builtin predicates. For brevity we describe here a fragment of the meta-interpreter that corresponds to the model above and includes the error checking procedure[4]. The three new clauses for `solve/3` correspond to each the earlier simpler ones, where a more general notion of built-in predicate replaces the use of 'true' and *specification(L,(A ← B))* replaces *clause(A,B)*:

```
onto_solve(Goal,Path):- solve(Goal,Path,0).
solve(X,_,_):-
   \+ expr(X),
   predicate_property(X, built_in),
   X.
solve((A,B),Path,Level):-
   solve(A,Path,Level),
   solve(B,Path,Level).
solve(X,Path,Level) :-
   \+ (expr(X); predicate_property(X,built_in)),
   specification(L, (X :- Body)),
   L =< Level,
   solve(Body,[X|Body],Level),
   NextLevel is Level + 1,
   detect_errors(X,Path,NextLevel).
detect_errors(X,Path,Level):-
   error(Level,X,Condition),
   solve(Condition,Path,Level),
   record_error(Level,X,Condition,Path,error),
   fail.
detect_errors(_,_,_).
...
```

The top level goal of the meta-interpreter is `onto_solve/2`. This will invoke `solve/3` for satisfying the given goal for test starting from the first layer of the specification. Each layer is checked against its ontological constraints - expressed either as axioms or error conditions - that belong to the layer above it. We will elaborate on the notion of a layer in section 5.4.

The declarative reading of the clause `solve(X,Path,Level)` is as follows: the given goal to test, that is `X`, is true if it is not a logical expression(i.e. a conjunctive, disjunctive or negative literal) or a builtin predicate and `X` is the head of a clause in the specification at level `L` which is beneath or in the same layer with the one we are testing(`Level`), and the `Body` of the clause is satisfiable from the enhanced vanilla model and we can check the head for conceptual error occurrences with respect to the layer above it(`NextLevel`). This check is implemented via the `detect_errors/3` goal.

[4]The whole meta-interpreter is described in [4].

The goal `detect_errors/3` is satisfiable if there exists an error condition, expressed as the 3rd argument in the `error(Level,X,Condition)` clause at same line, which is satisfiable from the meta-interpreter via the `solve(Condition,Path,Level)` goal. If so, then the error occurrence is reported via the goal `record_error/5`.

Error checking is recursive, so the proof that an error exists may itself generate errors. Those are checked against the ontological constraints exhaustively. We accumulate all the errors that are detected on given goals for testing as well as on their subgoals. We also accumulate information regarding the execution path that has been followed by the inference mechanism in proving a goal, the type of ontological constraint that has not been satisfied - axiom or error condition - and the layer in which the error has occurred.

The advantage of our meta-interpreter is that it does not interfere with the execution strategy of the specification, but detects when that strategy explores parts of the search space which are ontologically incorrect. This is done by applying ontological constraints to each relevant successful goal in the proof.

5.3. Applying the mechanism

Ontological constraints like the one in section 4 must be supplied by the engineer of the ontology. In addition, there are cases where the existing set of ontological constraints can be augmented by adding extra constraints. Those will be application specific constraints and are provided by the model checker. However, they should conform to the ontology vocabulary and be consistent with the existing constraints. We provide a structure editor to assist in editing these constraints which we describe in [3]. It is convenient to express some constraints in full first-order predicate calculus but they must be translated into a normal form appropriate for calling as Prolog goals. The editor takes care of this translation. For example, the constraint of section 4 is translated to:

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

which can be expressed via the editor to produce the appropriate format manipulated by the meta-interpreter:

error(1,navyThreat(A),
    (¬(aircraft(A) ∧ stores(A,W) ∧ target_type(W,'Naval-Unit')))).

The above constraint is an error condition used to express that it is an error when an aircraft that stores weapons which target naval units is not regarded as a navy threat. This condition makes it possible for the meta-interpreter of the previous section to detect the erroneous definition of navy threat.

We execute the simulator with the constraint described above. Assuming the erroneous definition of a navy threat given at the beginning of section 5 the error is detected and reported as follows:

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

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

5.4. Multi-layer checking

Ontological constraints are themselves specifications(of errors) and have as much chance of being ill-defined as the system models they are used to check. The notion of 'level' from section 5.2 is used to define layers of error checking, with the lowest being at model level and each successive layer checking the one below.

Since the mechanism used to execute the model is identical to the one used to check for errors, this allows error checking on the errors themselves by recursing through the appropriate goals at each level. This is the reason why `solve` and `detect_errors` in section 5.2 are mutually recursive.

The advantage of this approach is that we can check many specification layers 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.

6. Conclusions

In this paper we investigated two novel uses of ontologies in software development. Apart from their intended purpose of knowledge sharing and reuse, we demonstrated how they can be used to check specifications for conceptual error occurrences. This is done in a lightweight manner since the error checking mechanism can be applied without requiring manual alteration of the executable specification.

In addition, we argued that ontologies can augment existing specifications to enrich them with formally defined constructs that can be shared and reused in similar applications. The practical benefit of these approaches is that they are cost-effective since they don't intervene to the execution strategy followed in the specification.

Acknowledgements

The research described here 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.

References

[1] 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.

[2] N. Fridman Noy and C.D. Hafner. The State of the Art in Ontology Design: A Survey and Comparative Review. *AI Magazine*, pages 53–74, 1997.

[3] Kalfoglou,Y. and Robertson,D. Managing Ontological Constraints. Research Paper No.948, School of Artificial Intelligence, Institute for Representation and Reasoning, Division of Informatics, University of Edinburgh, March 1999.

[4] 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: University of Edinburgh, Dept. of AI, Research Paper No.935.

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

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

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

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

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