Use of Formal Ontologies to Support Error Checking in Specifications

Yannis Kalfoglou and David Robertson

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

Abstract. This paper explores the possibility of using formal ontologies to support detection of conceptual errors in specifications. We define a conceptual error as a misunderstanding of the application domain knowledge which results in undesirable behaviour of the software system. We explain how to use formal ontologies, and in particular ontological constraints, to tackle this problem. We present a flexible architecture based on meta interpretation in logic programming in which the specification is viewed as a multilayer design. We illustrate the significance of this approach for the software and ontology engineering community via an example case in the domain of ecological modelling.

1 Introduction

1.1 Specifications

The use of blueprints for guiding the development process of projects is common in many disciplines. In particular, in the field of software development, these blueprints are precise and independent descriptions of the desired program behaviour. They are crucial for the success of projects since they guide the way in which programmers will construct the desirable software. This has lead to the adoption of formal descriptions expressed in logic as a medium of blueprint, the purpose of which is to [9] "define all required characteristics of the software to be implemented, and thus form the starting point of any software development process". However, the precise role of formality in software development is a matter of debate([4]).

Formal methods support tests for some forms of completeness and consistency and there exist methods for methodological refinement of some types of formal specification into executable form, via appropriate interpreters. This provides an additional advantage, an executable specification represents not only a conceptual but also a behavioural model of the software system to be implemented [10], allowing early validation. Moreover, execution of the specification supplements inspection and reasoning as a means of validation. This might increase the correctness and the reliability of the software, and reduce development costs and time.

1.2 Conceptual errors

When describing a chosen domain we can make mistakes related to the mathematical language underpinning the formal model, like writing a non-terminating recursion using a logic programming language, or we can make mistakes in describing the domain, like defining an ecological model in which animals photosynthesise. The latter type of mistake is difficult to detect because it requires subjective knowledge about correct forms of domain description to be applied to the model description. We call this sort of mistake a conceptual error.

It is difficult, even with executable formal languages, to make models error free. In fact, it is easy(maybe easier) to make errors in this phase with pernicious side-effects for the remainder of the life cycle. This is because they may not be detected by those who use the formal model in subsequent design and may affect the functionality of entire systems by being propagated to subsequent design phases. The earlier the errors are detected the less serious are their consequences.

1.3 Our solution

Ontologies, which forge agreements on the use of terminology for particular domains are, potentially, a way of reducing this problem. Ontological engineers are beginning to supply information which helps in detecting conceptual errors. This information accompanies the formal ontology to which the specification should conform to and is often expressed in the form of axioms whose role is to restrict all possible interpretations of the ontology's constructs. In this paper we present a mechanism that makes the most of this information to allow us perform checks for conceptual errors in specifications.

1.4 Organisation of this paper

This paper is organised as follows: section 2 describes the field of formal ontologies with respect to ontological constraints, the core of our mechanism. In section 3 we present our detection mechanism and on section 4 we illustrate an example of its use. We elaborate further on a different use of the mechanism on section 5 where we conclude our work.

2 Formal ontologies

Ontologies have become popular in the recent years in the field of artificial intelligence. There exist different types of ontologies and numerous ways of constructing them ([8],[27]). The interpretation of the term varies across different communities and [15],[14] elaborates on terminological clarifications. The engineering community, and particular the KBS community has adopt a definition proposed by [11] and further elaborated by [27] and [24]. The type of ontologies in which we are interested in are the formal ontologies. A formal ontology is a language with a precisely defined syntax and semantics(which may be determined via model theory, proof theory or in terms of another formal ontology).

The inferences permitted in the language are constrained by one or more sets of proof rules accompanied by appropriate proof strategies. The forms of description allowed by those using the ontology are required to be consistent with a set of axioms limiting its use, which we call 'ontological constraints'. The aim of the ontology is to provide a language which allows a stipulated group of people to share information reliably in a chosen area of work.

A variety of ontologies have been reported in the literature with emphasis to their intended use in the area of knowledge sharing and reuse. There exist tools for browsing and editing ontologies(e.g.Ontolingua) as well as guidelines and methodologies to be followed on constructing them([27],[6],[2]). Although there have been efforts in applying ontologies(e.g.[5],[28],[21],[26],[16],[1]) as pointed out in [25] there is a dearth of well-developed applications based on formal ontologies. This contradiction is visible in the field of AI where the few applications that are discussed are intended applications which are yet to be built, or small research prototypes. According to [25] the reason for this is the lack of a rich representation of meanings which is contrary to the traditionally formal knowledge representation adopted by the AI ontology community.

Ontologies provide a set of characteristics that can be used in various ways. Apart from their intended purpose of knowledge sharing and reuse, we advocate that ontologies can be used in software design and in particular to support verification and formal evaluation of the early phases of it. This approach, although in its infancy, has already been explored in research experiments ([26]). Other researchers have used similar techniques([20]) and pointed out the benefits of using an ontology as a starting point in the design of a software product([19]).

By using ontologies as a starting point of software development we hope to gain a higher level of assurance that the specification is well defined and evaluated with respect to the real world it represents. This assumes that the syntax and semantics of an ontology can be checked and verified(arguably) against axioms. Note that, should one choose to follow this path it is not necessary strictly to use only the ontology's constructs in the specification. In fact, it is normally impractical to construct an executable specification by using only the ontology's constructs. Other constructs should be included as well, which do not directly benefit from the presence of ontological constraints but will be checked for errors using normal debugging techniques.

2.1 Ontological constraints

These are usually have the form of ontological axioms. We describe in textual form an axiom of a formal ontology, the PIF ontology [1] as presented in [21]:

> "The *participates-in* relation only holds between objects, activities, and timepoints, respectively"

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

[1] more on PIF can be found on [17]

$$participates\_in(X, A, T) \leftarrow object(X) \wedge activity(A) \wedge point(T).$$

The purpose of having formally defined this axiom is to allow reason about the various definitions of *participates-in* relation. So, whenever someone using the PIF ontology describes the relation in a way which does not conform to its axiomatised definition, this will reveal a potential discrepancy. For example, the following erroneous definition:

$$participates\_in(O1, O2, T) \leftarrow object(O1) \wedge object(O2) \wedge point(T).$$

is difficult to detect since it conforms to the ontology's syntax but reflects a misunderstanding of ontology's semantics [2].

The ontological axioms can be enhanced by adding more axioms or by introducing domain specific error conditions [3]. An error condition, which could be domain specific and added later on by the 'error checker' or software tester of the specification, is an erroneous condition which exhibits some undesired behaviour of the specification. Once this condition is satisfied during the error checking phase it will demonstrate an error occurrence in the specification. The erroneous definition of *participates-in* relation given above is an example of error condition. This technique will make the error checking domain specific and result in a customisation of the error detection process.

We call this sort of axiomatisation along with the domain specific error conditions, ontological constraints. Our approach explores a different angle in the use of ontological constraints. Moreover, it has been written that ontological constraints apart from the practical benefit of formal evaluation they provide([13]), they also verify that the ontology is consistent with respect to its conceptual coverage. This may facilitate mapping of ontologies that exhibit the same conceptual coverage of the real world([12],[3]), though conflicts may arise due to lack of correspondence on their top level division ([8]).

3 Error detection mechanism

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. The internals of the meta-interpreter will be described in detail at section 3.1. In this section we will focus on the general architecture we are adopting and the invention of a multilayered approach for error checking in specifications. The diagrammatic version of the mechanism is illustrated on Figure 1.

Specification construction starts by adopting the syntax and semantics of the ontology. We use Horn clause logic as a specification construction formalism [4],

[2] although this example is a typing error, not all ontological errors are simply defined in terms of types

[3] [26] elaborates on the need of additional axioms tailored to domain specific applications

[4] refer to [10] for the value of using declarative specifications

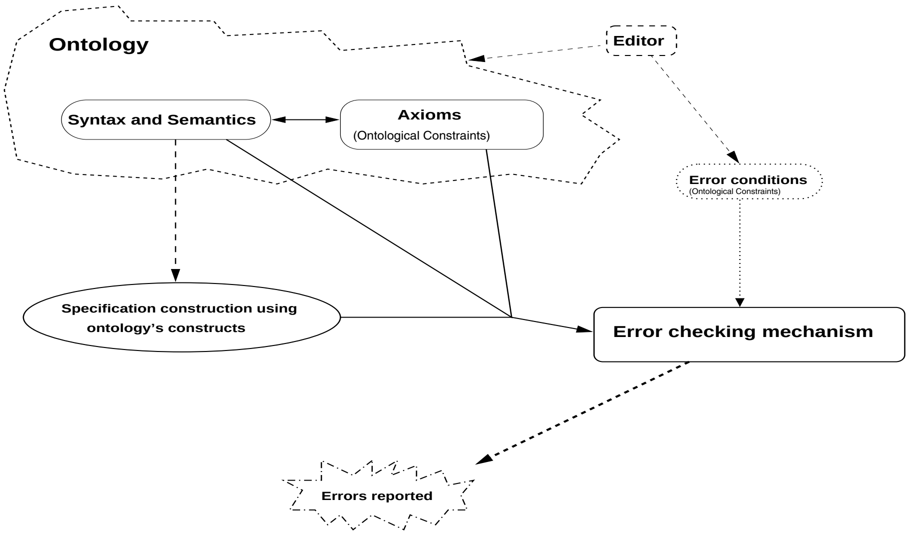

Fig. 1. Multi-layer architecture

with the normal Prolog execution model. To this we apply our checking mechanism and thus prevent the occurrence and propagation to subsequent phases of software development of harmful conceptual errors.

Few conventional ontological constraints are defined in this form so a transformation of them to the appropriate format is required. Although, the transformation could be done manually, we used an editor to facilitate the task of writing the constraints in the format manipulated by the meta-interpreter.

Additional domain specific error conditions can be defined to facilitate customised error checking. Assuming that an exhaustive check is required, then the ontological axioms will be the focal point of the mechanism. Whenever a statement in our specification will not satisfy the ontological axioms an error will be reported. Our mechanism can utilise both approaches to error detection, ontological axioms and error conditions, simultaneously and thus raise our confidence that the specification which will pass the test will be error-free.

3.1 Meta-interpreters

A meta interpreter is an interpreter for a language written in the language itself [23]. It gives access to the computation process of the language and enables the building of an integrated programming environment. 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 models the computation model of logic programs as goal reduction. Actually, the vanilla model reflects Prolog's choices of implementing the abstract computation model of logic programming. 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.

3.2 Error checking meta interpreter

Our aim in using the meta-interpreter technique is not only to purely replicate the computational model of Prolog. We are interested in utilising the products of ontological engineering - ontological constraints - to augment the meta-interpreter. This, in turn, will enable us to perform specific tests on selected goals of the specification with regard to the ontological constraints. The basis of the meta-interpreter is the standard vanilla model. In doing this we can explore the whole search space for a proof in the specification exactly in the normal way. So, the specification is actually executed in the normal way and checking for conceptual errors is performed on goals which have succeeded in the proofs. Thus, the maximum possible information is supplied for testing making sure that we wont loose crucial information on intermediate results.

The error checking is recursive, so the proof that an error exists may itself generate errors. Those are checked against the ontological constraints exhaustively. We cumulate all the errors that are detected on given goals for testing as well as on their subgoals. We also cumulate 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 that the error has occur. This notion will be explored in detail on section 5.1, we ignored for the time being since it does not affect the understanding of the algorithm.

We draw the attention of the reader to the specific format we are using for expressing specification statements as well as axioms and error templates. We use the standard logic notation $A \leftarrow B$, to denote that A is satisfiable if there exists a literal B which is satisfiable as follows:
specification(Index,(A $\leftarrow$ *B))*
where *Index* stands for the index of specification layer that the clause $A \leftarrow B$ belongs to. In cases where we have ground terms (clauses without subgoals) we use, by convention, the $A \leftarrow$ *true* notation. As far as concern the axioms and/or error conditions the format is:

ConstraintType(Index,Axiom,Condition)
where *ConstraintType* stands for either ontological axiom or error condition and denote the property that should hold, *Index* has the same meaning as before, and *Condition* is a condition(s) that has to be satisfiable in order for the ontological axiom to be satisfiable. The same format applies for error condition with the only difference being that the condition(s) must not be satisfiable.

The error checking meta-interpreter is given in Prolog notation at appendix A. We illustrate below in a pseudo-language the algorithm we apply:

for the given goal, **G***, for testing*
    while its layer, **L***, is not the last one*
        1. prove **G** *by applying the 'vanilla' model exhaustively*
        and check for conceptual error occurrences on goal **G**
        with respect to ontological constraints of layer **L+1**
        2. prove subgoal, **Gn***, if any, of* **G***, by applying*
        the same strategy
    exit while loop and cumulate information regarding execution
    path, conceptual errors found as well as the goal that are
    contained along with the ontological constraints that has not
    been satisfied

4 Error detection demonstration

In this section we will present, briefly, the practical use of our mechanism in an example case: an error detection in the ecological modelling domain. We use the following pattern in describing the case: an introductory part stating the problem description and relative domain knowledge opens the description which is followed by the specification of the problem. The ontological constraints are described in the sequel. This will help the reader to follow the test query and the conceptual errors detected based on the ontological constraints given, which close the case description.

4.1 Ecological model error checking

We have chosen ecology domain because in ecological modelling, being concerned with complex biological systems, it is difficult to decide how to represent the observed systems in a simplified form as simulation models. Furthermore, they are fraught with uncertainty and are prone to errors, especially conceptual errors. We demonstrate how our mechanism can alleviate this situation.

We have used a simple ecological model described in detail on [22]. The representation of this model in Prolog is given in the appendix B. We describe here the model in textual form: The model uses a "State Transition" approach to represent the passage of time during simulation. Suppose that we have 3 different animals(call them *a*,*b* and *c*) and that *a* prey on *b*;*b* prey on *c*; and *c* will prey on *a*. The area on which these animals live is represented by a grid with 3 squares along each side(thus 9 grid square grids in all). Animals move by shifting from the square in which they are currently situated to adjoining square. Each animal moves in the direction of potential prey(e.g. they actively hunt rather than browsing at random) but will not visit a square which it has occupied previously. If an animal is ever in the same square as its prey, the prey is eaten and thus removed from the simulation.

The specifier chooses to represent the states as follows: the initial state is named s0. New states of the system will be obtained whenever some aspect of the system changes so we require some way of linking the changes imposed on the system to the events which impose those changes. This could be achieved by the use of a nested term of the form:
`do(Action,State)`
where `Action` is a term representing some action which has been performed. `State` is either the initial state s0 or another term of the form:
`do(PreviousAction,PreviousState)`
The only action which it is necessary to represent in this model is the movement of an animal from one grid square to another. The specifier represents this action using the term `move(A,G1,G2)` where `A` is the name of some animal; `G1` is the location of the grid square at which the animal was located in the previous state and `G2` is its new location. Figure 2 illustrates a diagrammatic version of a move of animal *a* from square (1,1) which triggers a move of animal *b* to square (3,2).

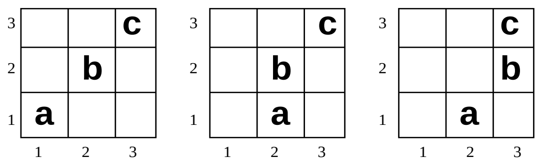

Fig. 2. State transition model - sequence of moves

Specification Although the model can generate any valid state of the system based on the constraints stipulated in the previous section for brevity and clarity we will focus on a fragment of the model that represents the treatment of locations for each animal in the system. However, before describing this chunk of code we will see how system generates the valid states: [5]:

```
specification(0,(possible_state(State) ←
              possible_state(s0,State))).
specification(0,(possible_state(State,State) ← true)).
specification(0,(possible_state(State,FinalState)←
              possible_action(do(A,State)) ∧
```

[5] the whole model is included at appendix B

```
5             possible_state(do(A,State),FinalState))).
```

For convenience of reading we have numbered the lines in correspondence with the code of appendix B. The declarative meaning of this top level goal of the specification is as follows: `State` must be a valid state of the system and this is defined by stating that `possible_state(s0,State)` must be true. This has effect of producing a valid `State`, starting with s0 as in initial state(line 1). This is true if `FinalState` is a valid state of the system which can be reached from `State`. In the simplest case, this is true if `FinalState=State`(line 2). Otherwise, it will be true if there is a possible action, `A`, which can be applied to `State` and the new state described by `do(A,State)` leads to `FinalState`.

In order to reason about the validity of various states of the system the specifier introduces a predicate, `holds(C,S)`, where condition `C` holds in state `S`. Three conditions are modelled: the location of an animal; whether it has been eaten; and which squares it has visited. For the purpose of demonstrating the error detection, we list here chunks of the specification that include potential error occurrences in describing the condition of animal location:

```
11 specification(0,(holds(location(a,(1,1)),s0) ← true)).
12 specification(0,(holds(location(b,(2,2)),s0) ← true)).
13 specification(0,(holds(location(c,(3,3)),s0) ← true)).
14 specification(0,(holds(location(A,G),State) ← ¬ State=s0 ∧
15              animal(A) ∧
17              last_location(A,State,G))).
```

At lines 11-13, the specifier defines the locations of the animals in the initial state, s0. In lines 14-17 defines the location of any animal in states other than s0. In such states an animal has a location determined by its most recent position in the sequence of actions. However, as we will see below there is a serious omission in this representation which will lead to undesirable behaviour of the model. This is detectable with the use of domain knowledge as expressed by the ontology.

Ontological constraints Although the specification is constructed based on the ontology's syntax and semantics, it should conform to various domain-specific constraints on the use of the ontology. For example, in order for an animal to exist at a particular location on the system it should not have been eaten in the meantime. Thus, a predator and a prey cannot be at the same square at the same state. We represent this constraint as follows:

```
60 axiom(1,holds(location(A,G),S),
61       (predator(A,B),
62       ¬ holds(location(B,G),S))).
```

Lines 60-62 represent the ontology's axiom. As we will see the specifier will have to redefine the `holds/2` clause with respect to animals location in order to conform to the ontological axiom given above.

Test query Assume a specification which has no errors, we can use the model by asking: 'Is there a state of the system in which animal, a, gets eaten?' giving the Prolog goal:

```
| ?- onto_solve((possible_state(S),holds(eaten(a),S)),[]).
```

The Prolog interpreter would then use the definitions of model structure [6] to solve this goal, instantiating S to a sequence of potential moves. The result is given below diagrammatically in figure 3 and in Prolog form:

```
S = do(move(c,(2,3),(2,2)),do(move(c,(3,3),(2,3)),
    do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0))))
```

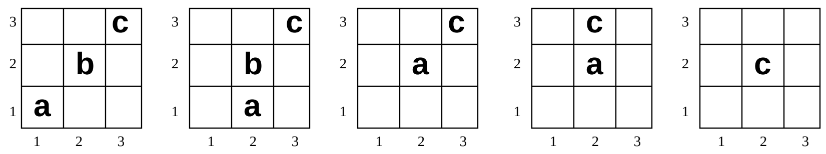

Fig. 3. Sequence of moves

As we can see, animal a has moved from its initial position (1,1) to (2,2) through square (2,1). Animal c, which preys on a, has moved from its initial position (3,3) to (2,2) and this satisfied the condition of holds(eaten(a),S). Note that animal b has removed from the simulation since its predator, animal a occupies the same square in the grid, that is (2,2).

However, assume the specification of our case(lines 14-17), on backtracking an erroneous answer will be returned to the same query. The Prolog answer is given followed by an illustration in Figure 4:

```
S = do(move(a,(3,2),(3,3)),do(move(b,(3,2),(3,3)),
    do(move(a,(2,2),(3,2)),do(move(b,(2,2),(3,2)),
    do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0))))))
```

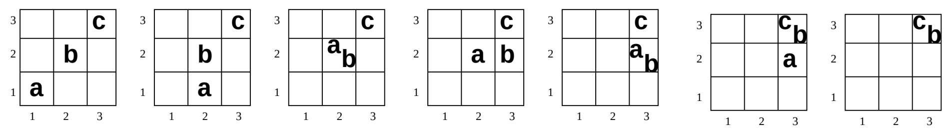

Fig. 4. Erroneous sequence of moves

[6] refer to appendix B for the representation of the model in Prolog

It is obvious that there is a problem with this answer. We observe a contradiction with the problem constraints: animal b continues to exist and actively moves although its predator, animal a, has visited its location to the grid, that is (2,2). This discrepancy is detected from the ontological axiom given above and explained in the next section.

Errors detected The error is detectable by the ontological axiom of lines 60-62. We illustrate the result diagrammatically in the form of a proof tree as shown in figure 5.

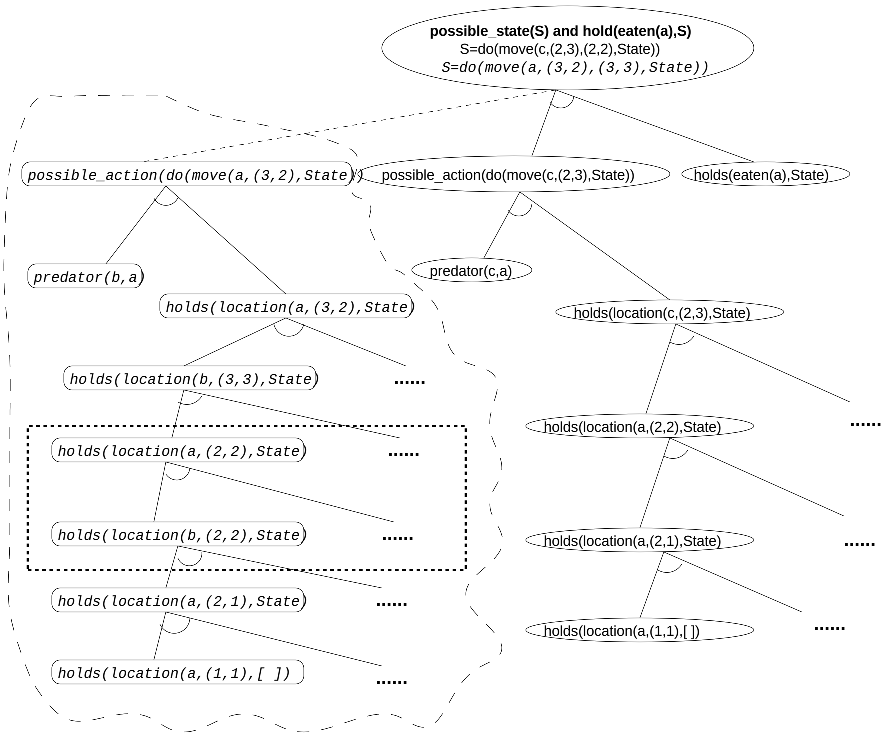

Fig. 5. Proof trees

The right part of the tree is the correct one while the left one is the ontologically erroneous path that has been followed. State variable `S` is instantiated to two values: the correct one is in plain font while the erroneous is in italics. In the right tree we have place in ellipses the goals that has been satisfied conjunctive arcs connecting them. The erroneous tree which is surrounded by a dashed line shows the correspondent satisfied goals within rectangle boxes. The rectangle box with a dashed line border represents the goal that does not conform to the axiom. This is reported by the mechanism as follows:

```
|?-onto_solve((possible_state(S),holds(eaten(a),S)),[]),report_errors.
axiom_violated(1,holds(location(a,(2,2)),do(move(a,(2,1),(2,2)),
              do(move(a,(1,1),(2,1)),s0))),
              (predator(a,X),
              ¬ holds(location(X,(2,2)),
              do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0)))))
```

we are using the reporting goal, `report_errors/5`(appendix A) to provide information concerning the axiom violated as well as the execution path that has been followed but we don't present it here for brevity.

In terms of meta-interpreter the discrepancy found, because the $\neg$ `holds(location(B,G),S)` clause was not satisfiable by the interpreter. As we pointed out earlier(3.1) axioms that are not satisfiable by the interpreter denote an error occurrence.

If we check the axiom that has not been satisfied we see that animal b continues to exists even after animal a visited its location. This is because animal b failed to satisfy the condition of line 62, in which 'an animal cannot hold the same position as its predator at the same state'. But what triggered this error?

If we examine carefully the specification of location condition we will discover an important omission: In order for an animal to keep a particular position on the grid at a particular `State` it should not get eaten by its predator at the same `State`. This could added to the specification by the statement:
$\neg holds(eaten(A), State)$
This statement, which is added to the specification manually by the specifier as a subgoal of `holds(location(A,G),State)` resolves the discrepancy. Our system, currently, does not support correction of conceptual errors.

5 Discussion

5.1 Checking the ontological constraints

How we can be sure that the ontological constraints are correct? Whether they are provided by ontological engineers in the form of ontological axioms or are domain 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 some core mechanism, the meta interpreter program, to check many specifications and their ontological constraints simultaneously. A key decision we made here is to use the same kind of augmentations to our meta interpreter model so that it can be used in many layers without the need of amendments. A diagrammatic version of the mechanism is given on Figure 6.

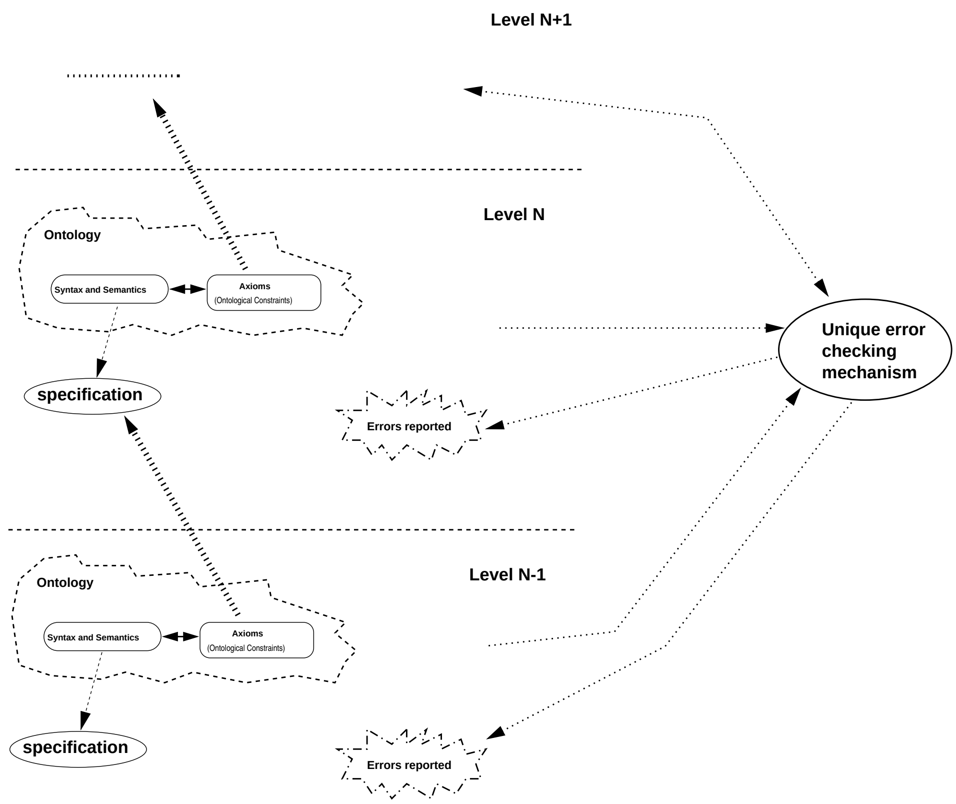

Fig. 6. Multi-layer architecture

This multi layered architecture is used as follows: assume that at the lower layer a specifier constructs the specification which we hope conforms to the syntax and semantics of the chosen ontology. The specification should also conform to the ontological constraints provided by the ontology - which can be checked with our mechanism as shown in the examples on a previous section(4). This will guarantee that the specification is correct with respect to the parts of it that conform to the ontological constraints.

However, if ontological constraint has been erroneously defined we can check this for error with our flexible mechanism. Ontological constraints are checked for errors against another set of constraints which can be viewed as meta-level constraints. They are part of the ontology and their use is to verify the correctness of the constraints. The result of this check will be the detection of an error, if any, in the ontological constraints. Ultimately, this layer checking can be extended to an arbitrary number of layers upwards, until no more layers can be defined.

The advantage is that we can capture a wide variety of errors occurring at different layers of the specification. It is possible to view the axioms introduced at each layer of error checking as an ontology and to check these for each query of the program by using the same mechanism.

Another use of this multi-layer architecture is in the area of ontology construction. It is often stated ([2]) that ontology construction can be viewed as a software design process. Moreover, the lack of rigorous evaluation methods during their construction will make prospective users reluctant to adopt an ontology. Assuming a middle-out or a bottom-up way of construction [7] this mechanism can be applied in order to detect discrepancies at various layers of the ontology or phases of its construction.

5.2 Conclusions

Our work contributes to existing work in error checking in specifications. The occurrence of conceptual errors which plague the specifications is of a great concern for the software engineering community and various attempts to tackle the problem have been made([20],[18],[7]). However, we are aware of no system that deploys ontological axioms to check for conceptual errors. In doing this we connect domain knowledge with the specification to facilitate error check for conceptual errors whereas other traditional techniques(e.g. debugging, program tracers) fail to reveal those errors. Our mechanism is flexible and can be used as a supplement to normal checking procedure without affecting the test strategy used.

Our work also contributes to ongoing work in the area of applications of ontologies. It proposes a different use of ontologies which diverge from the traditional ones(reuse, knowledge sharing). This use is(arguably) easier to apply since it relies on selective usage of ontological constraints tailored to domain specific problem descriptions. Furthermore, the multilayered approach we present is flexible to assist ontological engineers perform specific tests during the various phases of ontology construction with respect to domain knowledge.

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.

References

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

[7] see [8] and [27] for a discussion on various ways of ontology construction

2. M. Blazquez, M. Fernadez, J.M. Garcia-Pinar, and A. Gomez-Perez. Building Ontologies at the Knowledge Level using the Ontology Design Environment. In *Proceedings of the 11th Knowledge Acquisition Workshop, KAW98, Banff, Canada*, April 1998.
3. P. Borst, H. Akkermans, and J. Top. Engineering Ontologies. In *Proceedings of the 10th Knowledge Acquisition for Knowledge Based Systems Workshop,Banff, Canada*, 1996.
4. G. Cleland and D. MacKenzie. Inhibiting Factors, Market Structure and the Industrial Uptake of Formal Methods. In *Proceedings of Workshop on Industrial-Strength Formal Specification Techniques*, pages 47–61, Orlando(Florida) USA, April 1995. Boca Raton, Florida, USA.
5. Enterprise Integration Laboratory, University of Toronto, Canada. TOVE Project. available from http://www.ie.utoronto.ca/EIL/tove/ontoTOC.html, July 1995.
6. M. Fernandez, A. Gomez-Perez, and N. Juristo. METHONTOLOGY: From Ontological Arts Towards Ontological Engineering. In *Proceedings of the AAAI-97 Spring Symposium Series on Ontological Engineering, Stanford, USA*, pages 33–40, March 1997.
7. A. Finkelstein. Reviewing and Correcting Specifications. *Instructional Science*, 21:183–198, 1992.
8. 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.
9. N. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.
10. N. Fuchs and D. Robertson. Declarative Specifications. *The Knowledge Engineering Review*, 11(4):317–331, 1996.
11. T.R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.
12. M. Gruninger. Designing and Evaluating Generic Ontologies. In *Proceedings of the 12th European Conference of Artificial Intelligence*, August 1996.
13. 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.
14. N. Guarino. 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.
15. N. Guarino and P. Giaretta. Ontologies and Knowledge Bases: Towards a Terminological Clarification. *Towards Very Large Knowledge Bases*, 1995. IOS Press, Amsterdam.
16. Z. Jin, D. Bell, F.G. Wilkie, and D. Leahy. Automatically Acquiring Requirements of Business Information Systems by Reusing Business Ontology. 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.
17. 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.
18. 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.
19. W. Mark. Ontologies as Representation and Re-Representation of Agreement. In *Proceedings of the 5th International Conference on Principles of Knowledge*

Representation and Reasoning, KR'96, Massachusetts, USA, 1996. Position paper presented on the panel: *Ontologies: What are they and where's the research.*
20. W. Mark, S. Tyler, J. McGuire, and J. Schossberg. Commitment-Based Software Development. *IEEE Transactions on Software Engineering*, 18(10):870–884, October 1992.
21. S. Polyak, J. Lee, M. Gruninger, and C. Menzel. Applying the Process Interchange Format(PIF) to a Supply Chain Process Interoperability Scenario. In A. Gomez-Perez and R. Benjamins, editors, *Proceedings of Workshop on Applications of Ontologies and Problem Solving Methods, ECAI'98, Brighton, England*, August 1998.
22. D. Robertson, A. Bundy, R. Muetzefeldt, M. Haggith, and M. Uschold. *ECO-LOGIC Logic-Based Approaches to Ecological Modelling.* MIT Press, 1991. ISBN: 0-262-18143-6.
23. L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 4th edition, 1994. ISBN: 0-262-69163-9.
24. M. Uschold. Knowledge level modelling: concepts and terminology. *The Knowledge Engineering Review*, 13(1):5–29, February 1998.
25. M. Uschold. 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.
26. 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.
27. M. Uschold and M. Gruninger. Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, November 1996.
28. M. Uschold, M. King, S. Moralee, and Y. Zorgios. The enterprise ontology. *Knowledge Engineering Review*, 13(1), February 1998. Also available as AIAI-TR-195 from AIAI, University of Edinburgh.

A Error checking meta interpreter

```
onto_solve(Goal,Path):- solve(Goal,Path,0).
solve((A,B),Path,Level):- solve(A,Path,Level),
          solve(B,Path,Level).
solve((A;B),Path,Level):- solve(A,Path,Level) ;
          solve(B,Path,Level).
solve(\+ X,Path,Level):- \+ solve(X,Path,Level).
solve(X,Path,Level):- \+ logical_expression(X),
          predicate_property(X,(meta_predicate _Z)),
          solve_metapred(X,Call,Path,Level),!,
          Call.
solve_metapred(findall(X,Z,L),
          findall(X,solve(Z,Path,Level),L),Path,Level).
solve_metapred(setof(X,Z,L),
          setof(X,solve(Z,Path,Level),L),Path,Level).
solve(X,_,_) :- \+ logical_expression(X),
          predicate_property(X, built_in),
          X.
solve(X,Path,Level) :-
\+ (logical_expression(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(X,Path,Level):- axiom(Level,X,Condition),
        \+ solve(Condition,Path,Level),
        record_error(Level,X,Condition,Path,axiom),
        fail.
detect_errors(_,_,_).
record_error(Level,X,Condition,Path,Type):-
        \+ found_ontological_error(Level,X,Condition,Path,Type),
        assert(found_ontological_error(Level,X,Condition,Path,Type)).
report_errors:- show_errors,
        clear_errors.
show_errors:- found_ontological_error(L,X,C,P,T),
        ((T=error,
        write(error_condition_satisfied(L,X,C)),nl,
        write('path: '),write(P),nl);
        (T=axiom,
        write(axiom_violated(L,X,C)),nl,
        write('path: '),write(P),nl)),
        fail.
show_errors.
clear_errors :- retractall(found_ontological_error(_,_,_,_,_)).
logical_expression((_,_)).
logical_expression((_;_)).
logical_expression(\+ _).
```

B State Transition model

```
specification(0,(possible_state(State):-possible_state(s0,State))).
specification(0,(possible_state(State,State):-true)).
specification(0,(possible_state(State,FinalState):-
            possible_action(do(A,State)),
            possible_state(do(A,State),FinalState))).
specification(0,(possible_action(do(move(A,G1,G3),State)):-
            predator(A,B),
            holds(location(A,G1),State),
            holds(location(B,G2),State),
            move_in_direction(A,G1,G2,State,G3))).
specification(0,(holds(location(a,(1,1)),s0):-true)).
specification(0,(holds(location(b,(2,2)),s0):-true)).
specification(0,(holds(location(c,(3,3)),s0):-true)).
specification(0,(holds(location(A,G),State):- \+ State=s0,
```

```
            animal(A),
            \+ holds(eaten(A),State),
            last_location(A,State,G))).
specification(0,(holds(eaten(A),do(move(A,_,G),State)):-
            predator(P,A),
            holds(location(P,G),State))).
specification(0,(holds(eaten(A),do(move(P,_,G),State)):-
            predator(P,A),
            holds(location(A,G),State))).
specification(0,(holds(visited(A,G),s0):-holds(location(A,G),s0))).
specification(0,(holds(visited(A,G),do(move(A,_,G),_)):-true)).
specification(0,(holds(Condition,do(move(_,_,_),State)):-
            \+ Condition=location(_,_),
            holds(Condition,State))).
specification(0,(move_in_direction(A,(X1,Y1),(X2,Y2),State,(X3,Y3)):-
            (X1<X2,X3 is X1+1,Y3=Y1;
            X1>X2,X3 is X1-1,Y3=Y1;
            Y1<Y2,Y3 is Y1+1,X3=X1;
            Y1>Y2,Y3 is Y1-1,X3=X1),
            \+ holds(visited(A,(X3,Y3)),State))).
specification(0,(last_location(A,do(move(A,_,G),_),G):-true)).
specification(0,(last_location(A,do(move(A1,_,_),State),G):-
            \+ A=A1,
            last_location(A,State,G))).
specification(0,(last_location(A,s0,G):-
            holds(location(A,G),s0))).
specification(0,(animal(a):-true)).
specification(0,(animal(b):-true)).
specification(0,(animal(c):-true)).
specification(0,(predator(a,b):-true)).
specification(0,(predator(b,c):-true)).
specification(0,(predator(c,a):-true)).
specification(0,(adjoining_square((X1,Y1),(X2,Y2)):-
            max_x_square(MaxX),
            max_y_square(MaxY),
            min_x_square(MinX),
            min_y_square(MinY),
            (X2 is X1+1,X2=<MaxX,Y2=Y1;
            X2 is X1-1,X2>=MinX,Y2=Y1;
            X2=X1,Y2 is Y1+1,Y2=<MaxY;
            X2=X1,Y2 is Y1-1,Y2>=MinY))).
specification(0,(max_x_square(3):-true)).
specification(0,(max_y_square(3):-true)).
specification(0,(min_x_square(1):-true)).
specification(0,(min_y_square(1):-true)).
axiom(1,holds(location(A,G),S),
            (predator(A,B),
            \+ holds(location(B,G),S))).
```