Applying Ontological Engineering to Ecological Modelling: Detection of Conceptual Errors*

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

Abstract

Despite the fact that ontologies have gained great interest by the knowledge based systems community in the recent years, there is a dearth of applications based on formal ontologies. In the paper we shed light in an undiscovered corner in the use of formal ontologies: use of ontological constraints in combination with meta-interpretation in order to reveal potential conceptual errors in specifications. We applied this approach to ecological modelling and we report here 7 potential error cases that were revealed.

1 Introduction

Early phases of software development are fraught with uncertainty and the use of formal descriptions has been proposed(e.g.[10]). Those descriptions, often called specifications, are crucial for the success of every project since they guide the development process and describe in an independent and precise way the desired behaviour of the system. Errors that are made at these early steps of software development have serious side-effects to the project when remain undetected. This is because they may not be detected by those who use the formal description in subsequent design and affect the functionality of entire systems by being propagated to subsequent design phases. The earlier the errors are detected the less are their consequences.

Different approaches to the problem have been explored, an interesting one stem from the field of executable declarative specifications (see, for example, [11]). However, it is difficult, even with executable formal specifications to make blueprints error free. A type of errors that are difficult to detect is the conceptual errors: they reflect a misuse of a domain terminology in describing a problem. Their detection relies on knowledge about correct forms of domain description and normally this knowledge is not included in the specification.

The field of ontologies can provide this knowledge in a formal representation. It forges agreements among a group of engineers on the use of domain terminology for particular domains. The specification which will adapt to the ontology will reflect domain knowledge enabling checks for conceptual errors. We explore how ontologies can be used in order to perform this sort of testing.

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

1.1 Organisation of this paper

This paper is organised as follows: section 2 introduces the formal ontologies which are used by the error detection mechanism described on section 3. On section 4 we elaborate on ecological models which are described in detail on the subsequent sections 5, 6 and 7. Each of these models illustrate conceptual errors occurrences and detection in a standard format: description of the specification fragment that includes the conceptual error followed by the ontological constraints used to captured the error. The consequences of the erroneous specification are given in the form test queries followed by the errors that are reported by the error detection mechanism. We analyse the nature of the errors and elaborate on their ease of correction on section 8. Finally, section 9 concludes 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],[20]). 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([20],[6],[2]). Although there have been efforts in applying ontologies(e.g.[5],[21],[15],[19],[14],[1]) as pointed out in [18] 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.

However, 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 ([19]), though this wasn't the purpose of the experiment.

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, defining ontological constraints. Note that, should one choose to follow this path it is not necessary strictly to use only the ontology's constructs in the specification. We could include other constructs that don't belong to the ontology. This will enrich the specification, and will benefit from checks for conceptual errors on those parts of the specification which conforms to the ontology.

2.1 Ontological constraints

Ontological constraints play an important role in the ontology. 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]).

In this paper, we explore the use of ontological constraints from a different angle. Our approach is focused on the provision of a formal axiomatisation restricting the possible interpretations that the ontology's constructs could have. Those axioms will be used to verify that the specification constructs which are based on the underpinning ontology conform to the axiomatisation. Whenever a discrepancy is found this will reveal a potential conceptual error which is difficult to trap since there is nothing to prevent its occurrence: it is syntactically correct. But it is contrary to the axiomatisation and thus exhibits a misuse of domain knowledge as formally expressed by the ontology. In doing this we "bring" domain knowledge within the specification to facilitate error check for conceptual errors whereas other traditional techniques(e.g. debugging, program tracers) fail to reveal those errors.

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.

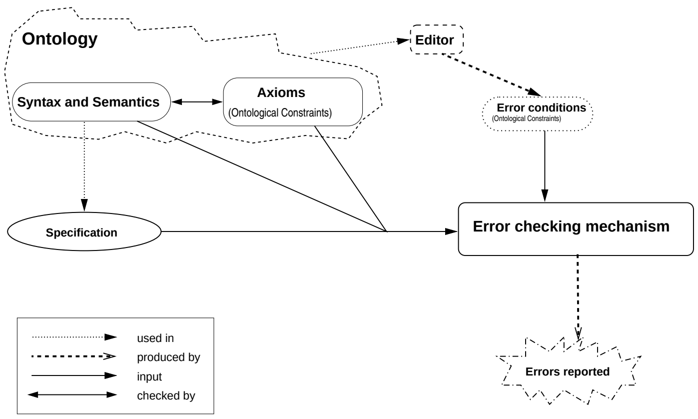

Figure 1: Error detection architecture

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 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.

We use Horn clause logic as a specification construction formalism[1], 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.

There are occasions where existing, published ontological constraints cannot be used directly. A transformation of them to the appropriate format is required. The format should be first order predicate calculus. 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.

The ontological constraints could be enhanced by adding more constraints or by introducing error templates [2]. An error template, 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. This technique will make the error checking domain specific and result in a customisation of the error detection process. Assuming that an exhaustive check is required, then the ontological constraints will be the focal point of the mechanism. Whenever a statement in our specification will not satisfy the ontological constraints an error will be reported. Our mechanism can utilise both approaches to error detection, ontological constraints and error templates, simultaneously and thus raise our confidence that the specification which will pass the test will be error-free.

A main contribution of our work is the invention of a multilayered approach for error checking. We can use the same 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 2.

We envisage the usage of this multi layered architecture as follows: assume that at the lower layer a specifier constructs the specification which he hopes 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. This will guarantee that the specification is correct with respect to the parts of it that conform to the ontological constraints. However, assume an error exists in the ontological constraints this will cause a false diagnosis with pernicious side effects. With our flexible mechanism, and the underlying uniform format, we are able to check the ontological constraints with respect to another set of axioms. This set of axioms itself is part of the ontology and its 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 termination of this upwards definition of layers depends on the level of abstraction that the domain can support. More on the technical implementation of the multi-layer architecture can be found on section 3.1.

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[3] this mechanism can be applied in order to detect discrepancies at various layers of the ontology or phases of its construction.

[1] refer to [11] for the value of using declarative specifications

[2] [19] elaborates on the need of additional axioms tailored to domain specific applications

[3] see [8] and [20] for a discussion on various ways of ontology construction

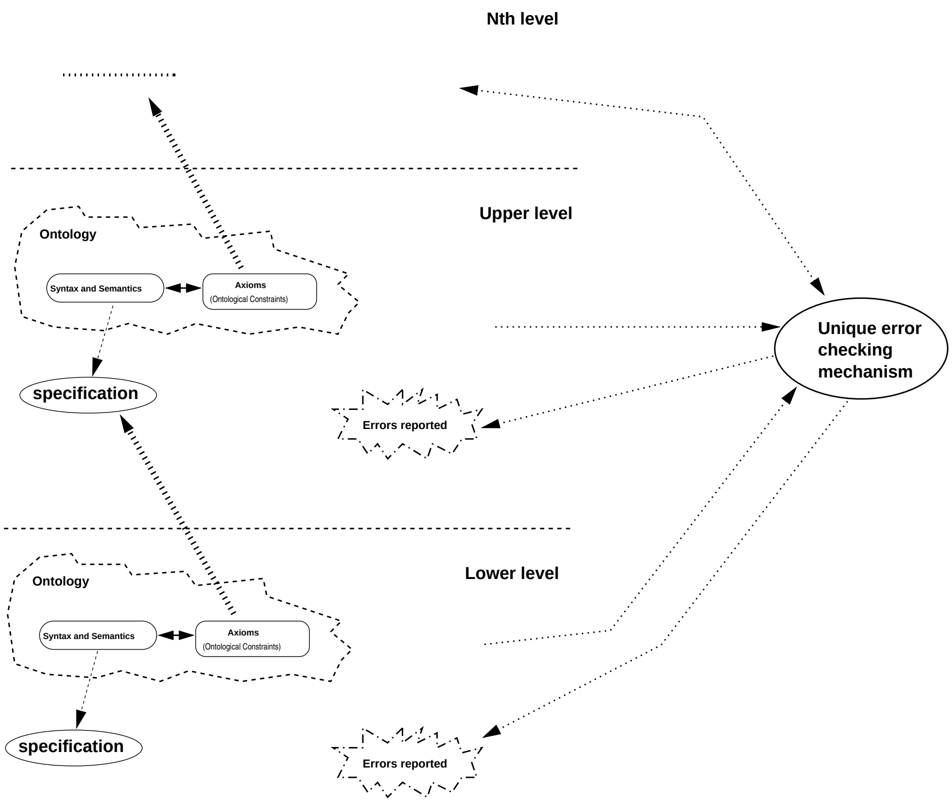

Figure 2: Multi-layer architecture

3.1 Meta-interpreters

A meta interpreter is an interpreter for a language written in the language itself [17]. 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. The model is given below:

```
solve(true).
solve(A,B):-solve(A),solve(B).
solve(A):-clause(A,B),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 is A is true and B is true. The third clause states that A is true if there exists a clause $A \rightarrow B$ in the interpreted program such that B is true.

Various implementations of meta interpreters can be found in the literature. A practical use of them is described with details in [16]. Two crucial decisions must be made in using the meta interpreter technique. First, you should choose the method for replicating the standard Prolog interpreter. This way should, potentially, be amenable to adaptation. Second, you should choose appropriate augmentations to the meta interpreter in order to achieve the desired efficiency. It is actually, those augmentations we have made that give the desired flexibility and efficiency to our mechanism, which will be described in the next section.

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 interesting to use 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. A fragment of the meta-interpreter is given below. The whole program can be found at appendix D. We have numbered the lines of code to ease their reading.

```
1 onto_solve(Goal,Path):- solve(Goal,Path,0).
2 ...
18 solve(X,Path,Level) :-
19 \ + (logical_expression(X); predicate_property(X,built_in)),
20          specification(L, (X :- Body)),
21          L =< Level,
22          solve(Body,[X|Body],Level),
23          NextLevel is Level + 1,
24          detect_errors(X,Path,NextLevel).
25 detect_errors(X,Path,Level):-
26          error(Level,X,Condition),
27          solve(Condition,Path,Level),
28          record_error(Level,X,Condition,Path,error),
29          fail.
30 detect_errors(X,Path,Level):-
31          axiom(Level,X,Condition),
32          \ + solve(Condition,Path,Level),
33          record_error(Level,X,Condition,Path,axiom),
34          fail.
35 detect_errors(_,_,_).
36 ...
```

We will describe in detail the augmentations we have made to the standard vanilla model with respect to ontological constraints used by the meta-interpreter.

We have replicated the vanilla model with enhancements for handling disjunctions of literals, extra logical features such as builtin predicates and meta predicates as well as negated literals. Those are included to the lines 2-17 - refer to appendix D. Other aspects, such as the treatment of non-declarative features of the language, like the cut(!) predicate sometimes need also be considered but these aren't important in our work. For the sake of brevity we will not explain these features since they are applicable to any generic meta-interpreter and will distract us from own main issue: how to use ontological constraints to detect conceptual errors in specifications.

The top level goal of the meta-interpreter is at line 1: `onto_solve/2`. This will invoke the `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.

The core of the meta-interpreter is given at lines 18-24. The declarative meaning of the clause `solve/3` at line 18 is as follows: `solve(X,Path,Level)` is true if the given goal for test, that is `X`, 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(lines 2-17) 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.

Lines 25-35 include the error checking. The goal `detect_errors/3` in line 25 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 of line 27. If so, then the error occurrence is reported via the goal `record_error/5` in line 28. The same mechanism applies to the second `detect_errors/3` goal which is used for detecting errors based on ontological axioms. The only difference is that the goal is satisfiable if there exists an axiom expressed as the 3rd argument in the clause `axiom/3` of line 29 which is not satisfiable from the meta-interpreter via the `solve/3` goal. The same error reporting method applies as before.

Our error reporting procedure cumulates information regarding the location of error occurrence in the execution tree, type of ontological constraint that has not been satisfied - axiom or error condition - the layer that the error has occur as well as the goal under test and the respective ontological constraint that has not been satisfied. The procedure is included at appendix D.

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 have replicated the standard Prolog notation of $A \rightarrow B$, denoting that A is satisfiable if there exist a literal B which is also satisfiable, to the following generic format:
`specification(Index,(A:-B))`
where `Index` stands for the index of specification layer that the clause `A:-B` belongs to. In cases where we have ground terms (clauses without subgoals) we use, by convention, the `A:-true` notation. As far as concern the axioms and/or error templates the format is:
`axiom(Index,Axiom,Condition)`
where `Index` has the same meaning as before, `Axiom` is a predicate denoting the axiom property and `Condition` is a condition(s) that has to be satisfiable in order for the `Axiom` to be satisfiable. The same format applies for the error templates with the only difference being that the condition(s) must not be satisfiable.

4 Ecological models

We have chosen the ecology domain to apply the ontology-based error checking because this domain is concerned with highly complex biological systems and needs a rational framework into which can be fitted descriptions of simulation models. Problems that are faced in ecology concern natural resources management, population dynamics, studying the flows of energy, etc. Some uses of models in ecology are: predicting future behaviour of some ecological system, replicating the real system's behaviour, form a scientific hypothesis to test a modellers understanding.

Furthermore, modelling of ecological systems is a classic example of the application of conventional simulation techniques to a poorly understood problem. The construction of models may be thought of as the formulation of an argument by the modeller about how he thinks the system works. The role of the prospective underpinning ontology, in principle, is to forge agreements among the modellers on the use of domain terminology. This will give them a better understanding of implementation choices which reflect subjective domain knowledge rather than governed by specific idioms of the implementation language.

It is stated that the use of logic to represent simulation models provides specific advantages. To quote [16]:

> "The use of logic to represent a simulation model provides the programmer with a radically different approach to the task of simulation. Instead of providing a sequence of instructions which tell the computer how to run the model, the programmer provides a set of Prolog clauses which describe the structure of the model. These are then processed, using a standard interpreter, to provide answers to users' questions about the structure of the model at any point in simulation."

In the sequel we present three simple simulation programs, borrowed from [16], each based on a different approach to simulation modelling. Each of these programs we assume that is based on a prospective ontology that formally represents domain terminology as well as ontological constraints that restrict the possible interpretations that ontology's constructs could have.

5 Model A: Rabbit-grass energy flow

The model we describe is based on a system dynamics model represented in [7] and adopted from [16] in the form of a logic program. The whole model is included at appendix A; here we describe the model in textual form while we illustrate a diagrammatic version of it in Figure 3 borrowed from [16]:

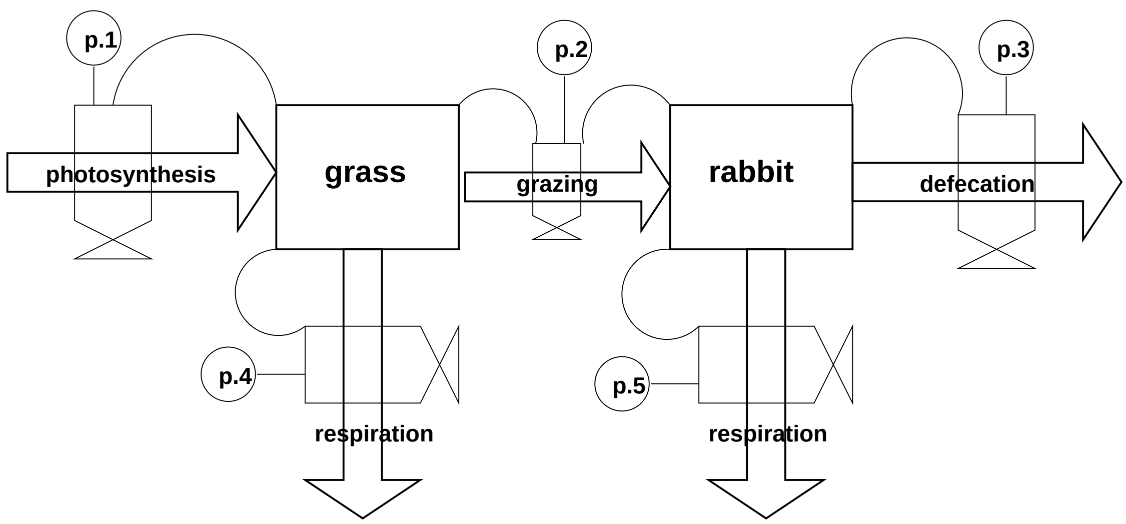

Figure 3: Rabbit-grass energy flow

As stated in [16]: "there exist two state variables, *grass* and *rabbit*. The changes to these state variables over time are affected by the flows of *photosynthesis*, *respiration*, *grazing* and *defecation*.

The *photosynthesis* flow is from the environmental "source/sink" into grass. The *grazing* flow transfers material from *grass* to *rabbit*. Flows of *respiration* empty out of *grass* and *rabbit*. An additional flow of *defecation* also exists from *rabbit*. The rates of transfer for each of these flows are controlled by an equation which takes as inputs a parameter and the values of the state variables involved in that flow". Before we start describing potential conceptual errors that could occur as well as how we detect them, we will explain briefly the focal point of the model: how state variables are represented and how their values are obtained. The fragment of the code that contains the top level goal is given:

```
1 specification(0,(state_variable(S,T,N):- initial_time(T),
2              initial_value(S,N))).
3 specification(0,(state_variable(S,T,N):- \ + initial_time(T),
4              previous_time(T,Tp),
5              state_variable(S,Tp,Np),
6              input_and_output_flows(S,Tp,Fi,Fo),
7              N is Np + Fi - Fo)).
```

The declarative meaning of this top level goal of the specification is as follows: at the initial time point of the simulation, a state variable will have whatever value was set as its initial value(lines 1-2). At times which are not initial times, a state variable will have the value, `N`, if: its value, `Np`, at the previous time point, `Tp`, is known; the sum, `Fi`, of all input flows at the previous time are known; the sum, `Fo`, of all output flows at the previous time are also known; and N equals `Np+Fi-Fo` (lines 3-7).

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

5.1 Error case 1

5.1.1 Specification

For the purpose of error demonstration we list below the chunks of code that include a potential error occurrence:

```
30 specification(0,(parameter(photosynthesis,grass,0.4):-true)).
31 specification(0,(parameter(grazing,rabbit,0.001):-true)).
32 specification(0,(parameter(respiration,grass,0.1):-true)).
33 specification(0,(parameter(respiration,rabbit,0.1):-true)).
34 specification(0,(parameter(defecation,grass,0.1):-true)).
```

At lines 30-34 the specifier defines the values of regulating parameters for *photosynthesis*, *grazing*, *respiration* and *defecation*. Each regulating parameter is associated with a state variable and denotes a flow of a parameter type attached to that variable. So, for example, the declarative reading of line 30 is: the value of one of the parameters that regulate the *photosynthesis* flow attached to state variable *grass* is `0.4`. As we will see on section **Errors revealed** there is a potential error in this specification fragment.

5.1.2 Ontological constraints

Although the specification is constructed based on the ontology's syntax and semantics, it should conform to the ontological constraints. We can express knowledge that is represented in the ontology, which states that "the defecation process holds only for animals", in a form of an axiom as follows:

```
39 specification(0,(animal(rabbit):- true)).
40 axiom(1,parameter(defecation,X,_),animal(X)).
```

Line 40 represents the ontology's axiom. In addition, the specifier has to include a statement in the specification that declares *rabbit* as *animal.* The axiom is added in the specification later on by the one who makes the error checking and indexed as layer 1 denoting that it monitors the layer beneath.

5.1.3 Test goal

Assume a specification which has no errors, we can use the model by asking: 'What is the value for the *grass* state variable at time 2?' giving the Prolog goal:

```
| ?- onto_solve(state_variable(grass,2,N),[]).
N = 999
```

The Prolog interpreter would then use the definitions of model structure[4] to solve this goal, instantiating N to 999.

However, assume the specification of our case, with the definition of the regulatory parameter *defecation* as described above(line 34), an erroneous answer will be returned to the same query, instantiating N to 998:

```
| ?- onto_solve(state_variable(grass,2,N),[]).
N = 998
```

This error, generated by the erroneous declaration of the regulatory parameter *defecation*, is detected by the mechanism as we describe in the next section.

5.1.4 Errors revealed

The error concerning the erroneous assignment of the regulatory parameter *defecation* to the *grass* state variable (line 34), is detectable via the ontological axiom of line 40 in combination with the specification statement of line 39 which declare the *rabbit* state variable as *animal.* The discrepancy found is reported:

```
| ?- onto_solve(state_variable(grass,2,N),[]),report_errors.
axiom_violated(1,parameter(defecation,grass,0.1),animal(grass))
path:  [flow(defecation,grass,source_sink,1,N)|
            ((defecation=respiration;defecation=defecation),
            parameter(defecation,grass,0.1),
            state_variable(grass,1,M),
            N is 0.1*M)]
```

[4] refer to appendix A for the representation of the model in Prolog

```
N = 998
```

we are using the reporting goal, `report_errors/5`(appendix D) to provide the following information: state the axiom that has been violated with the condition that triggered the error occurrence as well as the layer to which it belongs. The condition is the instantiation of variable `X` from line 40, to *grass* (`animal(grass)`). In addition, extra information is given such as the execution path that has been followed by the Prolog interpreter and includes the violated axiom.

The discrepancy was found, because the `animal(grass)` clause was not satisfiable by the interpreter. As we pointed out earlier(3.1), axioms that are not satisfiable by the interpreter are assumed to denote an error occurrence.

5.2 Error case 2

5.2.1 Specification

The specification fragment which contains the potential error is listed below:

```
17 specification(0,(flow(grazing,grass,rabbit,T,N):-
18              parameter(grazing,rabbit,P),
19              state_variable(grass,T,M),
21              N is P*M)).
```

The specifier introduces the predicate `flow(F,X,Y,T,N)` to denote that the value of the flow named `F` from object `X` to object `Y` at time `T` is `N`. Specific equations for calculating the *photosynthesis*, *grazing*, *respiration* and *defecation* flows are defined. We present here the calculation of flow *grazing*. As listed on lines 17-21, the value `N` of the *grazing* flow is calculated from the value `M` of grass multiplied by the value `P` of the *grazing* parameter. However, as we will see there is a serious omission here which affects the behaviour of the model.

5.2.2 Ontological constraints

The specification tester observes, based on domain knowledge expressed by the ontology, that the calculation of *grazing* flow is a slippery subject. It is the only flow that "connects" the two state variables of our model. It represents an *inflow* to *rabbit* from *grass* and simultaneously an *outflow* from *grass* to *rabbit*. Therefore, both values of state variables should be taken into account when the flow *grazing* is calculated. To prevent potential error, the 'error checker' defines a domain specific error template as follows:

```
41 error(1,flow(grazing,_,_,T,N),(parameter(grazing,_,P),
42         state_variable(rabbit,T,V1),
43         state_variable(grass,T,V2),
44         NewN is P*V1*V2,
45         \+ N = NewN)).
```

the declarative reading of this error template is: the value `N`, of the flow `grazing` at a given time `T` should be equal to the value `P` of `grazing` regulating parameter multiplied by the value `V1` of state variable `rabbit` at the same time, `T`, multiplied by the value `V2` of state variable `grass` at the same time `T`. The inequality of line 45 is used to denote that there is an error in the specification whenever this condition does not hold[5]. This error condition is added to the specification and

[5] although this condition is adequate to capture the specific error occurrence we elaborate on the idea of checking whether particular constituents that deemed to be necessary to prove a goal are part of the proof tree regardless the numerical expressions they involve. This will alleviate the error condition construction from being too specific as the one described here where an equation has to be declared as part of the error condition in order to evaluate the numerical comparison for revealing the error occurrence

indexed as layer 1 to monitor the layer beneath.

5.2.3 Test goal

Assume a specification with no errors, the following result is returned to the question: 'What is the value for the *rabbit* state variable at time 2':

```
| ?- onto_solve(state_variable(rabbit,2,N),[]).
N = 9
```

The Prolog interpreter satisfies the goal instantiating N to 9.

Let us now ask the same question in a model with the specification fragment(lines 17-21) described above:

```
| ?- onto_solve(state_variable(rabbit,2,N),[]).
N = 9
```

We observe that the answer is correct. However, our mechanism with the use of the ontological constraint described above will reveal a potential hidden error in the specification.

5.2.4 Errors revealed

The error is trapped by the condition of lines 41-45 which reveals a crucial omission in the calculation of the *grazing* flow. The specifier erroneously left out from the calculation the value of the *grass* state variable. The error reported as follows:

```
| ?- onto_solve(state_variable(rabbit,2,N),[]),report_errors.
error_condition_satisfied(1,flow(grazing,grass,rabbit,1,0.01),
                             (parameter(grazing,rabbit,0.001),
                             state_variable(rabbit,1,10),
                             state_variable(grass,1,1000),
                             10.0 is 0.001*10*1000,
                             \ + 0.01=10.0))
path:  [inflow(rabbit,1,0.01)|
        flow(grazing,grass,rabbit,1,0.01)]
N = 9
```

The same reporting goal(`report_errors/5` - appendix D) is used. The discrepancy is found, because the value of the `grazing` flow at time `1` was `0.01` while the correct value is `10.0` since both state variables are involved in the calculation. Whenever an error condition is satisfiable by the specification an error is revealed.

Note that this type of error is difficult to foresee since even with the erroneous specification we obtained correct result for the state variable *rabbit*. However, although the error does not affect directly the result of the top level goal `state_variable/3`, it does affect subgoals during the execution path, such as the `flow/5` subgoal.

5.3 Error case 3

5.3.1 Specification

The specification fragment that includes the potential conceptual error is given below:

```
11 specification(0,(inflow(S,T,N):- flow(F,S,X,T,N))).
12 specification(0,(outflow(S,T,N):- flow(F,X,S,T,N))).
```

The specifier uses the predicates `inflow/3` and `outflow/3` to distinguish between the different kinds of flows for our model. So, for example, the *grazing* flow is of type `inflow` and attached to *rabbit* state variable. It is also an `outflow` from the *grass* state variable. Lines 11-12 represent the specification of `inflow` and `outflow`. The declarative reading is that there is an inflow, `F`, to some object,`S`, if there is a flow from another object, `X`, to that object and there is an outflow,`F`, from some object, `S`, if there is a flow from that object to another object, `X`. However, as you may already have spotted there is flaw in the representation which will be revealed in the sequel.

5.3.2 Ontological constraints

Whenever, in the particular model, we have an inflow for the state variable *rabbit* this should origin from the *grass* state variable as an outflow. An error condition is used in order to express this knowledge:

```
error(1,inflow(S,T,N),(outflow(S1,T,N),
                       S=rabbit,
                       S1=source_sink)).
```

This condition states that it should not be satisfiable by the specification the case where an outflow from the *source sink* is attached to variable *rabbit* as an inflow. This will lead to undesirable situations because the only flow that comes from the *source sink* is *photosynthesis*. In other words, this will result in regarding the *photosynthesis* as an input flow for the state variable *rabbit*.

5.3.3 Test goal

Assume a specification with no errors. We ask the question: 'What is the value for *rabbit* state variable at time 2' by giving the Prolog goal:

```
| ?- onto_solve(state_variable(rabbit,2,N),[]).
N = 9
```

The Prolog interpreter correctly instantiates N to 9.

When the erroneous specification fragment of lines 11-12 is taken into account the model behaves as follows:

```
| ?- onto_solve(state_variable(rabbit,2,N),[]).
N = 11
```

An erroneous answer is returned instantiating N to 11. This error, triggered by the erroneous interpretation of `inflow` and `outflow` concepts is detected by the mechanism as explained in the next section.

5.3.4 Errors revealed

The error is detected and reported as follows:

```
| ?- onto_solve(state_variable(rabbit,2,N),[]),report_errors.
error_condition_satisfied(1,inflow(rabbit,1,1.0),
                            (outflow(source_sink,1,1.0),
                            rabbit=rabbit,
                            source_sink=source_sink))
path:  [input_and_output_flows(rabbit,1,Fi,Fo)|
        (findall(1.0,inflow(rabbit,1,1.0),Li),
        sum_elements(Li,Fi),
        findall(No,outflow(rabbit,1,No),Lo),
        sum_elements(Lo,Fo))]
N = 11
```

The reporting goal(`report_errors/5` - appendix D) shows the path information while the discrepancy is given by the `error_condition_satisfied` goal. This condition was satisfied by the interpreter by instantiating variables `S` and `S1` from lines 47 and 48 respectively to values `rabbit` and `source_sink`. This was provable by the equality relations of lines 47 and 48 but contrary to domain knowledge. As a result the specification tester has to check the initial definitions of `inflow` and `outflow` to find out that there is an erroneous ordering of the arguments of the `flow` predicate: instead of declaring that the `inflow(S,T,N)` is calculated by the `flow(F,X,S,T,N)` predicate the arguments are placed in a wrong order, `S` followed by `X`. The necessary rearrangements are applied to correct the error which are included to appendix A.

5.4 Error case 4

5.4.1 Specification

In this case we will elaborate further on the use of the multilayer approach we described in an earlier section.

```
54 specification(1,(lower_in_food_chain(S1,S,T):-
55              flow(Flow,S1,S,T,_),
56              eating_flow(Flow))).
57 specification(1,(lower_in_food_chain(S1,S,T):-
58              flow(Flow,S2,S,T,_),
59              eating_flow(Flow),
60              lower_in_food_chain(S1,S2,T))).
61 specification(1,(eating_flow(photosynthesis):-true)).
specification(1,(eating_flow(defecation):-true)).
```

In lines 54-60 the specifier defines the notion of *lower in food chain* that holds for the model's state variables. This is actually, a further articulation of the initial specification, it is added later on by the specifier, based on explicitly defined domain knowledge, and used to verify the model. This knowledge is formally represented in the underpinning ontology. It is regarded as a meta-level specification, and indexed as 1. The **Errors revealed** section describes the potential error that exists in this fragment of specification.

5.4.2 Ontological constraints

The specification tester defines the following domain specific error condition which will be used to detect the error:

```
49 error(1,state_variable(S,T,N),
50         (state_variable(S1,T,N1),
51         \ + S1=S,
52         N1<N,
53         lower_in_food_chain(S1,S,T))).
62 error(2,eating_flow(F), \ + F=grazing).
```

The interpretation of lines 49-53 is as follows: for any given state variable, S, that has the value, N, at time, T, there exists a state variable, S1, which is different than the previous, and has the value, N1, at the same time(T), and N1 is greater than N, then it is an error whenever S1 is lower in food chain than S at time T. This error condition, if satisfied, will demonstrate the presence of a potential conceptual error. The concept *lower in food chain* is already defined in the specification by the specifier. Note that this error condition is tailored to the specific application and holds for the existing values of the regulating parameters and only for the parameters defined in the model.

The second error template of line 62 is used by the specification tester to ensure a proper use of the *lower in food chain* concept. It actually demonstrates the usage of the multilayered approach since the error-checker in defining this error template protects the meta-level specification, layer 1, by introducing another level, layer 2, which monitors the statements of the previous layer.

5.4.3 Test goal

The test query that refers to the *lower in food chain* concept belongs to the meta-level, layer 1. So, we decided to exclude from testing the first level, layer 0, of the specification and start testing from the next level, layer 1. Assuming that a correct model is given, we could check whether the state variable *grass* is lower in food chain than the state variable *rabbit*:

```
| ?- solve(lower_in_food_chain(grass,rabbit,2),[],1).
```

The Prolog interpreter will solve the goal `lower_in_food_chain/3`. Note that we are using the goal `solve/3` in this test in order to define the layer from which we want to start testing.

However, assuming the erroneous specification of line 61, this goal will not be satisfied by the interpreter, due to misuse of the *eating flow* concept by the specifier. On the contrary, the reverse situation holds which leads to the erroneous answer:

```
| ?- solve(lower_in_food_chain(rabbit,grass,2),[],1).
```

The discrepancy is detected by the mechanism as we describe in the next section.

5.4.4 Errors revealed

The erroneous declaration of *eating flow* as expressed in line 61, is detectable via the error template of line 62 which defines that it is an error when the *eating flow* is not of type *grazing*. The discrepancy is reported as follows:

```
| ?- solve(lower_in_food_chain(rabbit,grass,2),[],1),report_errors.
error_condition_satisfied(2,eating_flow(photosynthesis),
                          \ + photosynthesis=grazing)
path:  [lower_in_food_chain(rabbit,grass,2)|
        (flow(photosynthesis,source_sink,grass,2,399.6),
```

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

The reporting procedure is used which gives the following information: two errors have been detected. The first one has instantiate the variable `F` of the `eating_flow/1` clause to *photosynthesis* and the second one to *defecation*. However, both of them satisfy the condition of line 62, that is the inequality with the *grazing* flow. As previously, extra information concerning the execution path is provided.

The discrepancy was found because the error condition was satisfiable by the interpreter, since there is not exist a definition of *eating flow* in the specification that matches to *photosynthesis* or *defecation*.

6 Model B: Structural growth

The second model of our scenario represents a structural growth process for a tree([16]). The whole model is included at appendix B. We describe it here in textual form: "the model represents the growth of a tree, starting with the initial shoot and ending with the branch structure, without leaves, obtained after some number of years". There are assumptions that hold for the particular model: "the number of new branches generated each year is determined as a function of the temperature that year and the length to which each branch grows during the following year is also a function of temperature". Further, we "assume that each branch grows only in the year after it is created and no branches are lost with age".

The specifier decides to represent the branching structure of the tree as a nested term of the form:

```
[b(Li,Bi),...,b(Lj,Bj)]
```

where, `b(Ln,Bn)` represents a subtree of the main tree with: `Ln` being the length of the main stem and `Bn` being a list of subtrees composed in the same way. The simulation model constructs a nested term of the form described above, based on the temperature values each year.

We describe here a fragment of the model that includes the calculation of the tree branches.

```
1 specification(0,(tree(Year,[]):- stoptime(Year))).
2 specification(0,(tree(Year,Branches):- stoptime(Stop),
3                 Year < Stop,
4                 temperature(Year,Temp),
5                 Dummy is round(0.1 * Temp),
6                 BranchNumber is integer(Dummy),
7                 NextYear is Year + 1,
8                 branches(BranchNumber,NextYear,Branches))).
```

The predicate `tree(Year,Branches)` defines the nested structure, `Branches`, of the subtree grown from a particular `Year`. The branches can grow in two ways: if we are in the final year of the simulation then there is no subtree and `Branches` is instantiated to an empty list(line

1). Alternatively, as shown from lines 2-8, if we are in a year earlier than the final, then the nested structure `Branches` can be obtained if the number of new branches, `BranchNumber` can be obtained as a function of the temperature `Temp`, for that year and each of these new branches can be developed by using the predicate `branches/3` given at appendix B.

6.1 Error case 5

6.1.1 Specification

The specifier choose the following erroneous representation for the structural growth model:

```
2 specification(0,(tree(Year,Branches):- stoptime(Stop),
3                 Year > Stop,
4                 temperature(Year,Temp),
5                 Dummy is round(0.1 * Temp),
6                 BranchNumber is integer(Dummy),
7                 NextYear is Year - 1,
8                 branches(BranchNumber,NextYear,Branches))).
17 specification(0,(stoptime(1):-true)).
```

The model presented here uses a different treatment for passing time during simulation. On section 5 the solutions were obtained by going *backwards* in time to obtain the value for a particular ecological variable. The tree growth model, as described in the previous section, goes *forward* in time, with each new branch forming the basis for additional subtrees at subsequent time points. However, in this case, the specifier decides to use the *backwards* implementation and adapts the model to represent this strategy. Lines 3, 7 and 17 reflect the changes that the specifier did in order to use the *backward* implementation to this model. However, as we will see at the next sections this will lead to erroneous results for the particular model.

6.1.2 Ontological constraints

One of the implications of applying the going *forward* in time strategy to this model given the assumptions we described above, is that for the same number of branches at two consecutive years in the simulation, the list of subtrees that can be obtained at the later year will always be smaller than the list of subtrees at a previous year. This is because as time progresses in the simulation, the tree becomes smaller until reaches to the stop time point(declared as 5 in the correct model).

The specification tester define the following error condition:

```
error(1,branches(Number,Year,List),
        (Number > 0,
        branches(Number,NextYear,NewList),
        NextYear > Year,
        smaller(List,NewList))).
```

the declarative reading of this error condition is: if the list, `List`, of subtrees developed at a particular year, `Year`, given the number of branches, `Number`, is smaller than a new list of subtrees, `NewList`, developed at a later year, `NextYear`, given the same number of branches, `Number`, then an error occurs in the specification where an erroneous strategy for passing time during simulation has been chosen.

The definition of this condition also includes some additional knowledge that is not part of the initial specification: definition of `smaller/2` for the list of subtrees which in turn introduces another auxiliary predicate, `count_branch/2` used to count the number of branches that stem from the first branch of the nested list. These two predicates added later on in the specification at a meta level. We include them in appendix B at lines 28-38 but they do not merit detailed scrutiny here.

6.1.3 Test goal

Assuming a specification with no error and the correct passing of time during simulation(*forward*). We can ask 'What will be the tree structure X, starting from year 3?' giving the Prolog query:

```
| ?- onto_solve(tree(3,X),[]).
X = [b(2.8,
          [b(1.6,[])]),
b(2.8,
          [b(1.6,[])])]
```

The query succeeds, instantiating X to the nested term given above. We illustrate diagrammatically this term for clarity in Figure 4, at the left side, where we present the result of tree formation from time 3 to time 5.

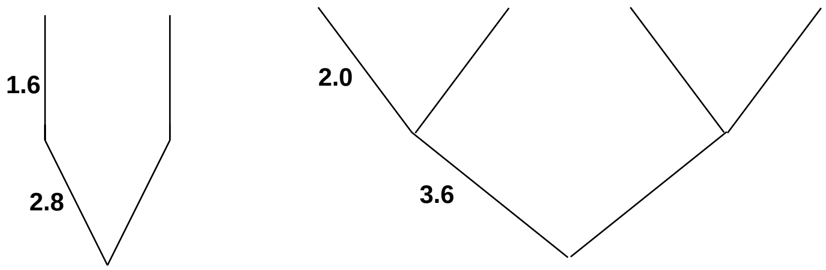

Figure 4: Tree structures growing from year 3 - correct version(left), erroneous(right)

However, assuming the specification with the erroneous passing of time, *backwards*, the result to the same query will be:

```
| ?- onto_solve(tree(3,X),[]).
X = [b(3.6,
          [b(2.0,[]),
          b(2.0,[])]),
b(3.6,
          [b(2.0,[]),
          b(2.0,[])])]
```

The variable X is instantiated to the nested term given above and illustrated at the right side of Figure 4. The erroneous declaration of time passing during simulation will be revealed in the next section with the use of the constraint described above.

6.1.4 Errors revealed

The result of error checking is given:

```
| ?- onto_solve(tree(3,X),[]),report_errors.
error_condition_satisfied(1,branches(2,2,[b(3.6,[b(2.0,[]),b(2.0,[])]),
                                         b(3.6,[b(2.0,[]),b(2.0,[])])]),
                                         (2>0,
                                         branches(2,3,
                                         [b(4.0,[b(3.6,[b(2.0,[]),b(2.0,[])]),
                                         b(3.6,[b(2.0,[]),b(2.0,[])])]),
                                         b(4.0,[b(3.6,[b(2.0,[]),b(2.0,[])]),
                                         b(3.6,[b(2.0,[]),b(2.0,[])])])]),
                                         3>2,
smaller([b(3.6,[b(2.0,[]),b(2.0,[])]),b(3.6,[b(2.0,[]),b(2.0,[])])],
        [b(4.0,[b(3.6,[b(2.0,[]),b(2.0,[])]),b(3.6,[b(2.0,[]),b(2.0,[])])]),
        b(4.0,[b(3.6,[b(2.0,[]),b(2.0,[])]),b(3.6,[b(2.0,[]),b(2.0,[])])])])))
path:  [tree(3,[b(3.6,[b(2.0,[]),b(2.0,[])]),b(3.6,[b(2.0,[]),b(2.0,[])])])|
        (stoptime(1),3>1,temperature(3,20),2.0 is round(0.1*20),2 is integer(2.0),
        2 is 3-1,branches(2,2,[b(3.6,[b(2.0,[]),b(2.0,[])]),
        b(3.6,[b(2.0,[]),b(2.0,[])])]))]
X = [b(3.6,[b(2.0,[]),b(2.0,[])]),b(3.6,[b(2.0,[]),b(2.0,[])])]
```

The discrepancy was found because the error condition defined in lines 23-27 was satisfied. In particular, variables `List` and `NewList` of lines 23 and 25 were instantiated to the list of subtrees for years 2 and 3 respectively. The condition of line 27, `smaller(List,NewList)`, was proved by the interpreter and thus, denotes an error in the specification. Note that we have pruned part of the answer since the meta interpreter exhaustively tests each step of a recursive structure. In fact, the list of errors detected is much greater than the one we present here, but the rest of the errors refer to the same query at different steps of recursion and they are not of interest for the purpose of our demonstration.

The interpretation of this error highlights the contradiction of using a *backward* time passing strategy in a model that is constructed to operate with a *forward* time passing strategy. This resulted in having a list of subtree at a later time point in the simulation that is greater than a list at an earlier time point. This result is contradictory to the initial assumption that a tree becomes smaller as time progresses towards the stop time point.

7 Model C: State transition

Our third model, borrowed from an ecological model described in detail on [16], uses a "State Transition" approach to represent the passage of time during simulation. The representation of this model in Prolog is given at appendix C. We describe here the model in textual form: "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 squares in all). Animals move by shifting from the square in which they are currently situated to an adjoining square. Each animal moves in the direction of a 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:

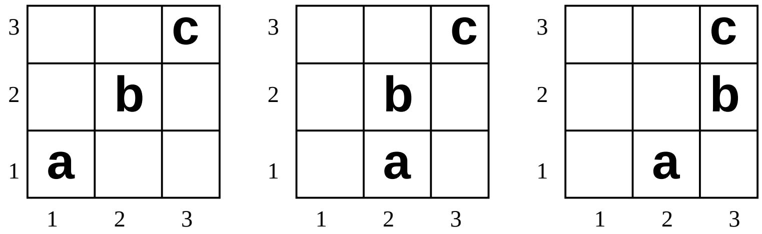

Figure 5: State transition model - sequence of moves

`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 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 5 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).

Let us describe how the model generates valid states. The relevant chunk of code is given:

```
1 specification(0,(possible_state(State):-possible_state(s0,State))).
2 specification(0,(possible_state(State,State):-true)).
3 specification(0,(possible_state(State,FinalState):-
4              possible_action(do(A,State)),
5              possible_state(do(A,State),FinalState))).
```

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 (partly)defined by stating that `possible_state(s0,State)` must be true. This has effect of producing a valid `State`, starting with s0 as an initial state(line 1). A state other than the s0, like the `FinalState`, is a valid state of the system when it 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`.

7.1 Error case 6

7.1.1 Specification

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 location of an animal:

```
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)).
```

```
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, lines 14-17 define 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 on the next section there is a serious omission in this representation which will lead to erroneous behaviour of the model. This is detectable with the use of domain knowledge as expressed by the ontology.

7.1.2 Ontological constraints

Although the specification is constructed based on the ontology's syntax and semantics, it should also conform to the ontological constraints. These state that in order for an animal to be able to hold a particular location on the system it should not have been eaten in the meantime. The specification tester chooses to utilise this ontological knowledge which states that a predator and a prey cannot be at the same square at a given state. Also, the model states that animals actively hunt and move in the direction of a potential prey. Thus, if a predator holds a position in the grid squares at some state `S` its prey cannot hold the same position in the same state `S`. The specification tester models following axiom:

```
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 on the next section the specifier has to redefine the `holds/2` clause with respect to animals location in order to conform to the ontological axiom given above.

7.1.3 Test goal

Assuming 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)),[]).
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))))
```

The Prolog interpreter would then use the definitions of model structure[6] to solve this goal, instantiating `S` to a sequence of potential 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)`. It is interesting to point out here that animal *b* has been removed from the simulation since its predator, animal *a* occupies the same square in the grid, that is (2,2).

However, assuming the specification of our case, with the erroneous definition of animals location in the grid given above(lines 14-17), the following answer will be returned to the same query:

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

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

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 sequel.

7.1.4 Errors revealed

The error is detectable by the ontological axiom of lines 60-62 and reported 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)))))
path:[possible_action(do(move(a,(2,2),State),do(move(a,(2,1),(2,2)),
      do(move(a,(1,1),(2,1)),s0))))|
      (predator(a,b),
      holds(location(a,(2,2)),
      do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0))),
      holds(location(b,G2),
      do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0))),
      move_in_direction(a,(2,2),G2,do(move(a,(2,1),(2,2)),
      do(move(a,(1,1),(2,1)),s0)),State))]
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))))))
```

The condition that triggered the error occurrence was the instantiation of variables `B` and `G` from lines 61 and 62 to animal *b* and to `do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0))` respectively. Additional information concerning the execution path that has been followed by the Prolog interpreter is also included as in previous cases.

The discrepancy was found because the `\ + 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. That is because animal *b* failed to satisfy the condition of line 62, that is, '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 `State` it should not get eaten by its predator at the same `State`. This can be added to the specification by the statement:

`\ + holds(eaten(A),State)`

This statement, included to the correct version of the specification at appendix C in line 16, resolves the discrepancy.

7.2 Error case 7

7.2.1 Specification

Another condition that the specifier uses to check the validity of system's states is to check the squares that an animal has visited. We list below the fragment of specification that represents this condition which includes a potential error:

```
24 specification(0,(holds(visited(A,G),s0):-holds(location(A,G),s0))).
25 specification(0,(holds(visited(A,G),do(move(A,G,_),_)):-true)).
```

The declarative meaning of the above statements which conform to the ontology is: all animals are considered to have visited their initial locations (line 24). Otherwise, an animal which has been moved to a new location has visited that square (line 25).

However, the statement of line 25 is not specified correctly. As we will see in the next sections there is a crucial misunderstanding of the concept 'visited a location'.

7.2.2 Ontological constraints

The specification tester takes into account the definition of the concept `holds(visited)`. He also considers the meaning of the `move(A,G1,G2)` action which states that `G1` is the location of a grid square at which the animal was located in the previous state and `G2` is the new location. Therefore, we have an error when a move of an animal from a previous state is regarded as a visited location. In order for a location to be regarded as a visited one it should be the new location that an animal visits. An error condition is used to detect the discrepancy:

```
63 error(1,holds(visited(A,G),S),
64          (\+ S=s0,
65           \+ S=do(move(A,_,G),_))).
```

Line 64 states that we have an error when the initial location of an animal is not considered as a visited location. In addition, line 65 states that it is also an error when the new location of an animal is not considered as a visited location.

7.2.3 Test goal

The model with the correct definition of the concept 'visited a location' is used to answer the question: "Is there a state of the system in which animal *a* has visited square (2,2)"? by giving the Prolog goal:

```
|?- onto_solve((possible_state(S),holds(visited(a,(2,2)),S)),[])
S = do(move(a,(2,1),(2,2)),do(move(a,(1,1),(2,1)),s0))
```

The interpreter satisfies the goal and instantiates `S` to the sequence of moves that animal *a* follows to reach point (2,2).

If we consider the specification with the lines 24-25 given above then the behaviour of the model to the same query is:

```
|?- onto_solve((possible_state(S),holds(visited(a,(2,2)),S)),[]).
S = do(move(a,(2,2),(3,2)),do(move(a,(2,1),(2,2)),
    do(move(b,(2,2),(3,2)),do(move(a,(1,1),(2,1)),s0))))
```

We observe that although the goal was proved by the interpreter, a redundant sequence of moves has been returned. This is, although animal *a* has visited square (2,2) following a move from square (2,1), the model satisfies the goal when animal *a* visits square (3,2) following a move from square (2,2). There is also a redundant move of animal *b* in the erroneous answer.

7.2.4 Errors revealed

Given the error condition above the discrepancy is detected and reported as follows:

```
|?- onto_solve((possible_state(S),holds(visited(a,(2,2)),S)),[]),report_errors.
error_condition_satisfied(1,holds(visited(a,(2,2)),do(move(a,(2,2),(3,2)),
                             do(move(a,(2,1),(2,2)),do(move(b,(2,2),(3,2)),
                             do(move(a,(1,1),(2,1)),s0))))),
                             (\ + do(move(a,(2,2),(3,2)),do(move(a,(2,1),(2,2)),
                             do(move(b,(2,2),(3,2)),do(move(a,(1,1),(2,1)),s0))))=s0,
                             \ + do(move(a,(2,2),(3,2)),do(move(a,(2,1),(2,2)),
                             do(move(b,(2,2),(3,2)),do(move(a,(1,1),(2,1)),s0))))=
                             do(move(a,_,(2,2)),_)))
path:  []
S = do(move(a,(2,2),(3,2)),do(move(a,(2,1),(2,2)),
    do(move(b,(2,2),(3,2)),do(move(a,(1,1),(2,1)),s0))))
```

Although the first part of the condition was not satisfied by the interpreter, viz. the state instantiated to S is not the initial location of animal *a*, the second one was satisfied and indicates the error. The sequence of moves that animal *a* follows to reach square (2,2) does not lead to this location but to a subsequent location in the grid: (3,2). This contradicts the definition of 'visited a location' concept. The discrepancy is resolved by declaring the new location of an animal as a visited location as shown in line 25 at appendix C.

8 Analysis of errors

In this section we attempt to make a classification of the errors described above according to their type. We will not investigate the cause of these errors but rather we categorise them with respect to their characteristics emphasising the ease or difficulty of correction. In the literature (e.g.[9]) it is reported that two categories of software errors dominate the field: syntactic and type checking errors. We refer to syntactic errors when we have a misuse of the programming language syntax. Those errors are, normally, detected during the parsing process of the program. There exist debuggers, tracers and other syntax checking tools that are available in almost every programming language, either as embedded tools or as separate systems. Usually, syntax errors are easy to capture and cause no damage to the program since their elimination is required in order for the program to run.

On the other hand, type checking errors are a slippery subject. They might reflect an erroneous assignment of a data type to a function, in which case advanced type checkers or program inspectors (e.g. [4]) are able to detect. However, when the data type is not part of the language itself but part of a domain specific application, then detection of a prospective error becomes harder. A further articulation of type checking errors are the conceptual errors to which we referred in previous sections. These are more difficult to detect because they not only reflect a misuse of data type but also a misunderstanding of the domain application. In the sequel, we classify the errors we described in previous sections by pointing out their ease or difficulty of correction.

Errors of cases 5.1 and 5.4 can be considered as type checking errors. They both return a wrong result to the test goal and reflect an erroneous type assignment. However, the error at case 5.4 is not detectable from the top level goal and belongs to a meta layer in the specification. It is part of an error condition used to monitor a meta layer specification which in turn is used to check the behaviour of the model. Furthermore, both errors require subjective knowledge about the correct types defined in the domain and their detection relies on correct understanding of domain knowledge. The detection of the 5.4 error demonstrates the flexibility of our mechanism with the ability to use it for checking meta level specifications.

Another error that can be classified under the same tag is that of case 5.3. It returns a faulty result to the test query due to erroneous declaration of *inflow* and *outflow* concepts. However, it is more difficult to detect than the previous type checking errors, since it requires knowledge concerning the representation of flows in the model. It is detectable with the use of a domain specific error condition that takes into account the implementation choices for the calculation of flows in the particular model.

Errors of cases 7.1 and 7.2 are considered to be conceptual errors. They might return a wrong result to the test query but they do not neglect or misuse a type assignment. Error of case 7.1 reflects an omission of a crucial condition for the model, that of *not holds(eaten)* used to calculate the *location* for a particular state variable. It is an omission of modelling assumption which results in situations where an animal continues to exist although its predator has visited its location. This error needs subjective knowledge of the domain to be captured and this is expressed as an ontological axiom. Error of case 7.2 seems to be more difficult to detect. Its answer is not entirely wrong. It is actually a redundant answer since the model returns more details with respect to prospective moves than required. It is a misunderstanding of a domain concept and is not detectable from a top level goal. We used a domain specific error condition to detect this error.

Finally, the last two errors, cases 5.2 and 6.1 are the most interesting ones. The error of case 6.1 is a conceptual error that reflects the wrong adoption of an implementation strategy concerning the passage of time during simulation. The strategy used (*backwards*) has been implemented correctly. However, it is not appropriate for the particular model. It returns an erroneous answer to the test query, and we used a domain specific condition to capture this error based on design decisions explicitly made to this model: trees are grown symmetrically and monotonically in time. The error of case 5.2 is the most difficult to detect. It returns a correct answer to the test query and implements correctly the calculation of *flow*. However, it erroneously omits a state variable from the calculation, which affects subgoals of top level goal. It reflects a misunderstanding of calculation of *flow* and a domain specific error condition is used based on ontological knowledge.

9 Discussion

In this paper we presented an approach to cope with a key engineering problem: the occurrence of conceptual errors. In particular, we used the domain of ecological modelling as a prospective exemplar for the application of ontological engineering products: ontological constraints. We advocate that the utilisation of these constraints could be of help for the formal evaluation of ecological models with respect to domain knowledge.

Moreover, we presented a multilayer approach for software testing that is capable of detecting discrepancies at a meta level. This will enable ontology engineers to verify the correctness of their constraints with respect to another set of constraints, so called meta-constraints. The unlimited definition of layers that this architecture can support will make easy its adoption to a variety of applications such as specification testing and ontological constraints testing.

We would like to mention an important issue in this approach. This is the ability to enhance the set of axioms with domain specific constraints. The work was done manually, so far, based on observations about the model behaviour. In future, we want to support this task by providing assistance to the error checker in defining the domain specific ontological constraints. We are currently working on an editor system that uses predefined domain specific templates to facilitate the task of ontological constraints construction.

Acknowledgements

The author would like to thank David Robertson for his comments, suggestions and valuable discussions on an early draft of this paper, as well as Virginia Biris-Brilhante for her comments and insights.

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.

[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] P. Brna, M. Brayshaw, A. Bundy, M. Elsom-Cook, P. Fung, and T. Dodd. An Overview of Prolog Debugging Tools. Research paper 398, Department of Artificial Intelligence, University of Edinburgh, 1988.

[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] J.W. Forrester. *Industrial Dynamics.* MIT Press, 1961. ISBN: 0-262-060035.

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

[10] N. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[11] N. Fuchs and D. Robertson. Declarative Specifications. *The Knowledge Engineering Review*, 11(4):317–331, 1996.

[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] 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.

[15] 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.

[16] 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.

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

[18] 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.

[19] 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.

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

[21] 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 Rabbit-grass energy flow model

```
specification(0,(state_variable(S,T,N):- initial_time(T),
             initial_value(S,N))).
specification(0,(state_variable(S,T,N):- \+ initial_time(T),
             previous_time(T,Tp),
             state_variable(S,Tp,Np),
             input_and_output_flows(S,Tp,Fi,Fo),
             N is Np + Fi - Fo)).
specification(0,(input_and_output_flows(S,T,Fi,Fo):-
             findall(Ni,inflow(S,T,Ni),Li),sum_elements(Li,Fi),
             findall(No,outflow(S,T,No),Lo),sum_elements(Lo,Fo))).
specification(0,(inflow(S,T,N):- flow(F,X,S,T,N))).
specification(0,(outflow(S,T,N):- flow(F,S,X,T,N))).
```

```
specification(0,(flow(photosynthesis,source_sink,grass,T,N):-
            parameter(photosynthesis, grass, P),
            state_variable(grass,T,M),
            N is P*M)).
specification(0,(flow(grazing,grass,rabbit,T,N):-
            parameter(grazing,rabbit,P),
            state_variable(grass,T,Mq),
            state_variable(rabbit,T,Mr),
            N is P*Mr*Mq)).
specification(0,(flow(F,S,source_sink,T,N):-
            (F=respiration; F=defecation),
            parameter(F,S,P),
            state_variable(S,T,M),
            N is P*M)).
specification(0,(initial_time(1):-true)).
specification(0,(initial_value(grass,1000):-true)).
specification(0,(initial_value(rabbit,10):-true)).
specification(0,(parameter(photosynthesis,grass,0.4):-true)).
specification(0,(parameter(grazing,rabbit,0.001):-true)).
specification(0,(parameter(respiration,grass,0.1):-true)).
specification(0,(parameter(respiration,rabbit,0.1):-true)).
specification(0,(parameter(defecation,rabbit,0.1):-true)).
specification(0,(previous_time(T,Tp):- T > 0,Tp is T - 1)).
specification(0,(sum_elements([],0):-true)).
specification(0,(sum_elements([_|Y],N):- sum_elements(Y,N1),
            N is N1 + 1)).
specification(0,(animal(rabbit):- true)).
axiom(1,parameter(defecation,X,_),animal(X)).
error(1,flow(grazing,_,_,T,N),(parameter(grazing,_,P),
            state_variable(rabbit,T,V1),
            state_variable(grass,T,V2),
            NewN is P*V1*V2,
            \ + N = NewN)).
error(1,inflow(S,T,N),(outflow(S1,T,N),
            S=rabbit,
            S1=source_sink)).
error(1,state_variable(S,T,N),
            (state_variable(S1,T,N1),
            \ + S1=S,
            N1<N,
            lower_in_food_chain(S1,S,T))).
specification(1,(lower_in_food_chain(S1,S,T):-
            flow(Flow,S1,S,T,_),
            eating_flow(Flow))).
specification(1,(lower_in_food_chain(S1,S,T):-
            flow(Flow,S2,S,T,_),
            eating_flow(Flow),
            lower_in_food_chain(S1,S2,T))).
specification(1,(eating_flow(grazing):-true)).
```

```
error(2,eating_flow(F), \ + F=grazing).
```

B Structural Growth model

```
specification(0,(tree(Year,[]):- stoptime(Year))).
specification(0,(tree(Year,Branches):- stoptime(Stop),
                Year < Stop,
                temperature(Year,Temp),
                Dummy is round(0.1 * Temp),
                BranchNumber is integer(Dummy),
                NextYear is Year + 1,
                branches(BranchNumber,NextYear,Branches))).
specification(0,(branches(0,_,[]):-true)).
specification(0,(branches(BranchNumber,Year,[b(Length,Branches)|Rest]):-
                BranchNumber > 0,
                temperature(Year,Temp),
                Length is 0.2 * Temp,
                tree(Year,Branches),
                NewBranchNumber is BranchNumber - 1,
                branches(NewBranchNumber,Year,Rest))).
specification(0,(stoptime(5):-true)).
specification(0,(temperature(1,10):-true)).
specification(0,(temperature(2,18):-true)).
specification(0,(temperature(3,20):-true)).
specification(0,(temperature(4,14):-true)).
specification(0,(temperature(5,8):-true)).
error(1,branches(Number,Year,List),
                (Number > 0,
                branches(Number,NextYear,NewList),
                NextYear > Year,
                smaller(List,NewList))).
specification(1,(smaller(List1,List2):-
                count_branch(List1,N1),
                count_branch(List2,N2),
                N1 < N2)).
specification(1,(count_branch([],0):-true)).
specification(1,(count_branch([b(X,L)|_],N):-
                islist(L),
                count_branch(L,N1),
                N is N1 + 1)).
specification(1,(islist([]):-true)).
specification(1,(islist([_H|T]):-islist(T))).
```

C 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))).
error(1,holds(visited(A,G),S),
        (\ + S=s0,
        \ + S=do(move(A,_,G),_))).
```

D 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(\+ _).
```