Error detection in specifications based on Ontologies

Ioannis Kalfoglou[†] David Robertson[†]
[†] Department of Artificial Intelligence, University of Edinburgh,
80 South Bridge, Edinburgh, EH1 1HN, Scotland
Email: {yannisk,dr}@dai.ed.ac.uk

Abstract

The problem of conceptual errors in the early stages of a software development is of a great concern for the engineering community. The side effects of these errors are pernicious because they reflect a misunderstanding of the application domain resulting in undesirable behaviour of the software system. They are not easily detected since they may be mathematically elegant but inappropriate to the domain. The field of Ontologies, which forge agreements on the use of terminology for particular domains is, potentially, a prospective solution to the problem. We present a method which is capable of detecting such errors in specifications by utilising ontologies as a supplement to the normal checking procedure.

1 Introduction

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 [6] "define all required characteristics of the software to be implemented, and thus form the starting point of any software development process". Such a description is often called a specification, though, this definition is not apply to representations of formal models.

The specifications can be informal, which they aim of being easier to read and write, but they don't have a precise meaning and are often overloaded with information, which makes hard to understand their meaning. By contrast, formal specifications have a precise meaning and thus may help dispel ambiguities by applying formal methods techniques. However, the precise role of formality in software development is still a matter of debate(see for example [12], [7]).

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[6], 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.

2 Errors in early phases

It is difficult, even with executable formal specifications, to make blueprints 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 blueprint in subsequent design and may affect the functionality of entire systems. By contrast, assuming a modular design, errors that occur in implementing a module at later phases affects only that module and can be corrected by reimplementing it. The earlier the errors are detected the less serious are their consequences. As pointed out in [2], "It is often stated that Prolog programs can be seen as executable specifications. If the programmer can get the specification right then there will be no program code errors. But it is not so simple to get the specification right..."

The problem of errors in the early phases of software development is treated with awe by the community[12]. Despite the various methods that have been proposed and used in error checking at these phases (see, for example, [5]) we are still far from an adequate and reliable solution that will alleviate this problem. The crux of the problem is the nature of errors that occur at these phases: conceptual errors. A conceptual error occurs when we misuse the terminology of a domain, thus describing a problem in a way which is inappropriate to that domain. Such errors are not detectable without knowledge of the domain itself and, since such knowledge isn't normally in the specifications, we can have deep conceptual errors in specifications which are mathematically elegant.

The Knowledge Based Systems(KBS) community has had to face this problem directly because it relies on describing domains. However, the methods used in KBS to help detect errors have been applied only sparingly to formal specifications-perhaps because these methods tend to be approximate and heuristic rather than deductive. In other words, they help suggest potential errors but don't prove their existence.

In the next few sections we demonstrate how we can improve this situation with the use of ontologies as a basis of our error-checking method.

3 Emergence of Ontologies

Ontologies have become popular in the recent years in the field of AI with applications in major areas such as knowledge representation. There are different types of ontologies and numerous ways for constructing them. Although the exact interpretation of the term is still a matter of debate(see [9]), concrete definitions have been proposed and adopted by the community (see, for example, [17] and [8]). The purpose of an ontology is to record a set of agreements by a group of engineers about how they describe a given domain. The benefits by using ontologies in systems engineering are mainly in three areas: reusability, reliability and specification. In particular in the area of reliability, the use of ontologies, through formal representations, makes possible the automation of consistency checking resulting in more reliable software. In addition, the ontological axioms can be used to help protect specifications against conceptual errors. However, we are aware of no system, which deploys ontological axioms to check for such errors. Our approach, which is based on this principle, uses the Process Interchange Format(PIF) ontology [10] for demonstrating its use in the domain of process modelling.

The aim of PIF is to develop an interchange format to help automatically exchange process descriptions among a variety of business modelling and support systems such as workflow software, flow charting tools, planners, process simulation systems and process repositories. The benefit of using PIF is that it acts as an interlingua among a variety of systems, reducing the number of translators needed from $O(n^2)$ to $O(n)$. The core of PIF consists of the minimal sets of constructs necessary to translate simple but non-trivial process descriptions. In addition, PIF can be extended to represent local needs of individual groups with the use of Partially Shared Views(PSV) described in [11]. The PIF ontology's focal point is a process, which is a set of activities that stand in certain relations to one another and to objects over timepoints.[1]

In the next two sections we describe the general framework of our approach followed by a detailed description of the implementation, and demonstrate its use through a scenario test case.

4 General framework for detection of conceptual errors

Our method, which illustrated in Figure 1, starts from the available syntax and axioms[2] of PIF. We can then follow either of two paths. The left path is followed by those who develop the implementation and the right path by those who will check it with respect to error occurrences.

The developers path starts with a manual implementation of the desired specification representing software processes. This implementation conforms to the syntax of PIF but may contain conceptual errors(some of which could be detectable via the PIF axioms). The specification is then translated to an executable formal specification, in our case using Prolog. We could test this specification in the normal way(by giving Prolog goals) but this would not include checks against the ontological axioms. To include these we need to look at the other elements of the diagram.

The right path starts with the transformation of the ontology's axioms to First Order Predicate Calculus(FOPC). We then translate automatically this axiomatisation to Normal Form using standard *Lloyd-Topor* transformations

[1] PIF examples can be found in [10] and in [15]

[2] The axiomatisation was proposed in [14]

[13], which we will describe in detail in the next section. Finally the translated axioms are stored in a database of constraints expressed as Prolog clauses.

The two paths converge at a meta-interpreter error-checking program, described in details in the next section, which is used in order to detect errors with respect to ontological constraints.

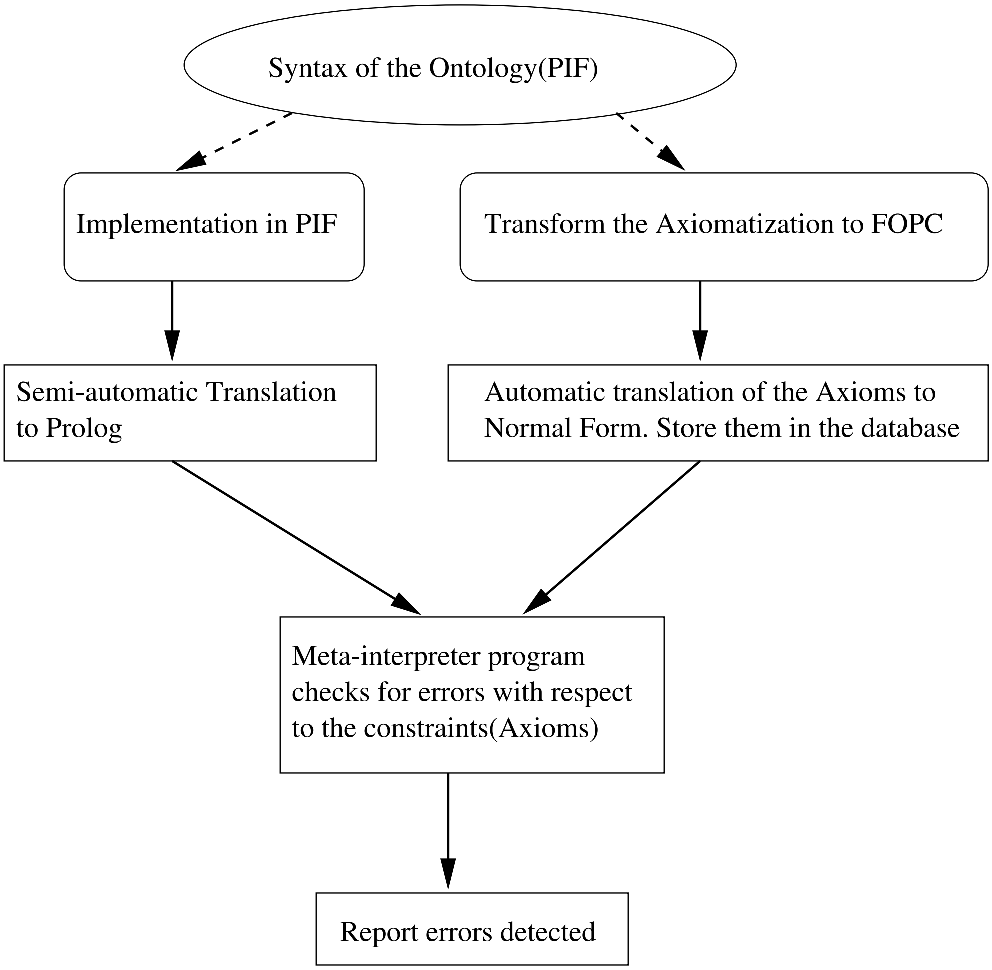

Figure 1: Method for error detection

The role of meta-interpreters in logic programming is normally to describe explicitly some pattern of inference. In our case, this description is the normal Prolog mechanism for satisfying Prolog goals - so our interpreter replicates the normal behaviour experienced by someone checking the specification. However, the mechanism is extended by checking, whenever the solution to a goal is contained in the execution, whether relevant ontological constraints apply and, if so, whether they are satisfied. Thus our interpreter can tell us, for each individual test, whether any part of it strayed outside the formalised ontological constraints.

Notice that this does not mean that our specification is error free. There could be conceptual errors which we fail to reveal because we choose the wrong tests. Our mechanism, as is normal, can show errors but cannot prove their absence.

In the following section we present technical details of the implementation along with a scenario test-case, simplified in order to demonstrate the operation of the method.

4.1 Translation of the axioms to Normal Form

The axioms of PIF are expressed in [14] in textual form along with their KIF-like syntax expressions. We transform these axioms to First Order Predicate Calculus(FOPC). The following are two examples:

Axiom1:
textual form:
The *before* relation holds only between timepoints
FOPC:
$before(P, Q) \rightarrow point(P) \wedge point(Q)$

Axiom2:
textual form:
Timepoint p is *BeforeEq* timepoint q if and only if is before or equal to q.
FOPC:
$before\_eq(P, Q) \leftrightarrow point(P) \wedge point(Q) \wedge (before(P, Q) \vee P = Q)$

By a similar way, we have transformed the rest of the PIF axioms to FOPC but these must be transformed to a normal form that is computable by our mechanism. There exist automatic transformations to normal form described in the literature(see, for example, [3]). We have build an automatic translator program in Prolog, tailored to the particular FOPC statements(see appendix A). The resulting NF is a disjunction of conjunctions of literals after a set of rewriting rules have been applied:

- removal of connectives except $\neg, \vee, \wedge$
- move $\neg$ inwards to propositions
- move $\wedge$ inside $\vee$
- reorganising within levels

The rules used are shown in table 1

in FOPC notation	in Prolog(as predicate `rewrite/2`)
$A \leftrightarrow B \Rightarrow ((A \rightarrow B) \wedge (B \rightarrow A))$	$rewrite((A =:= B), ((A :- B), (B :- A))).$
$A \rightarrow B \Rightarrow \neg(A \wedge \neg B)$	$rewrite((A :- B), (\backslash + (A, \backslash + B))).$
$\neg(A \vee B) \Rightarrow (\neg A \wedge \neg B)$	$rewrite((\backslash + (A; B)), (\backslash + A, \backslash + B)).$
$\neg(A \wedge B) \Rightarrow (\neg A \vee \neg B)$	$rewrite((\backslash + (A, B)), (\backslash + A; \backslash + B)).$
$A \wedge (B \vee C) \Rightarrow ((A \wedge B) \vee (A \wedge C))$	$rewrite((A, (B; C)), ((A, B); (A, C))).$
$(A \vee B) \vee C \Rightarrow A \vee (B \vee C)$	$rewrite(((A; B); C), (A; (B; C))).$
$(A \wedge B) \wedge C \Rightarrow A \wedge (B \wedge C)$	$rewrite(((A, B), C), (A, (B, C))).$
$\neg(\neg A) \Rightarrow A$	$rewrite((\backslash + \backslash + A), A).$

Table 1: Set of rewrite rules in FOPC and in Prolog

We apply these rules exhaustively[3]. So, *axiom1* listed above will be automatically translated as follows:
$NF = (\backslash + before(P, Q); point(P), point(Q))$

The transformation is applied to all of the axioms which, in continue, have been stored in a database as Prolog clauses of the form:
$contraint(NF).$

[3]The whole program as well as the description of the transformation algorithm are listed in the appendix A.

However, as we will see in the next section where we describe the meta interpreter mechanism, this is still not the final form of the axioms. The form of the axioms is closely related to the implementation of our meta interpreter, thus, the alteration of the format we have describe above is required.

Table 2 summarises the meaning of the operators we are using.

meaning of the operator	**Logical**	**SICStus Prolog**
logical equivalence	$\leftrightarrow$	=:=
implication	$\rightarrow$	:-
negation	$\neg$	\+
conjunction	$\wedge$	,
disjunction	$\vee$	;

Table 2: Operators and their meaning

4.2 Meta interpreter error checking program

Our meta interpreter mechanism reflects the standard 'vanilla' model, described in the literature [16], enhanced with an extra facility for detecting any discrepancies found against the axioms. The 'vanilla' model,reflects the computational model of logic programs as goal reduction and is as follows:

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

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 goal A is true if there exist a clause $A \rightarrow B$ in the interpreted program such that B is true.

We have adopt the above standard model with extra enhancements for handling disjunction of literals, non-logical features such as builtin predicates, and returning results from computations using the technique of accumulators. Figure 2 illustrates our approach.

```
ontosolve(true,_Path,Result,Result).
ontosolve((A,B),Path,Sofar,Result):-solve(A,Path,Sofar,New),solve(B,Path,New,Result).
ontosolve((A;B),Path,Sofar,Result):-solve(A,Path,Sofar,Result);solve(B,Path,Sofar,Result).
ontosolve(X,_Path,Result,Result):-builtin(X),X.
ontosolve(X,Path,Sofar,Result):- \+ (builtin(X);X=(A,B);X=(A;B)),
                                    clause(X,Body),
                                    solve(Body,[X|Path],Sofar,Result).
```

Figure 2: Enhancement of standard 'vanilla' model

The third clause of `ontosolve/4` handles the case of disjunction of literals: the disjunction A or B is true if either A is true or B is true. We are using two accumulators, one for accumulating all the errors detected so far in the interpreted program and another for accumulating the location that these errors were found.

However, the hard work of our meta interpreter is the facility for detecting the discrepancies found against the axioms. This is added as an extra subgoal to the normal 'vanilla'-based interpreter(Figure 3)

```
solve(X,Path,Sofar,Result):-ontosolve(X,Path,Sofar,New),
                            ontocheck(X,Path,New,Result).
ontocheck(X,Path,Sofar,Result):- constraint(X,C),
                                 \+ C,
                                 Result=[(C,Path)|Sofar],
                                 !.
ontocheck(_X,_Path,Sofar,Sofar).
```

Figure 3: Enhancement for detecting errors

The `ontocheck/4` clause is responsible for this kind of check. We require that our constraints must be satisfiable by the normal Prolog proof strategy. That means we need to select which part of the constraint should match to the subgoal we are checking. The NF of the axioms allows negated subgoals in the first part of the constraint, thus, our meta interpreter could fail to terminate if it chooses an inappropriate negated subgoal. Consequently, the way that this predicate works, reflects the way we represent the constraints in the database. So, the *axiom1*, described in the previous section, will be represented as a Prolog clause in the following way:

```
constraint(before(A,B),(point(A),point(B))).
```

which is used in the following condition in our meta interpreter of Figure 3:

```
constraint(X,C), \+C
```

and has the following interpretation:
For any given goal for testing, search the database of constraints to find a match with the first argument of the clause `constraint`. Whenever a match found in the database, try to evaluate the second argument of the clause. These are the constraints that should hold in order for the first argument(axiom) to be true. If a discrepancy found, then accumulate the error and its location, and recurse to the rest of the constraints, by exploiting the behaviour of the 'vanilla' model.

The above mechanism have made us to change the initial product of the automatic translator to NF, by adopting the following representation of the axioms:

constraint(<axiom description>,<constraints imposed by the description>)

This representation is (arguably) easier to read and write. Therefore, we anticipate that the user of the system would provide the axioms directly in this form. However, one often needs to translate from other notations because ontologies are domain-flexible. Having this need in mind we wrote a translator from PIF syntax [4]to Prolog clauses. This complementary tool facilitate our work and produce faster the implementation part for our method.

5 Scenario test-case

Having present in detail the way our method work we will describe its applicability to a 'real world' scenario. We have chosen one of the proposed scenarios for the PIF evaluation, the *"supply chain scenario"*. [5] The main context for this proposed scenario involves the development and coordination of supply chain processes between a manufacturer, retailer, distributor, warehouse company and transportation company[4]. For the sake of brevity we will present the detection of an error at the process 'process document request' in the transportation company. The description of this process is given in [15]:

[4]see [10] for more details on PIF syntax.
[5]This scenario was adopted from the Workflow Management Coalition's(WfMC) workflow interoperability demonstration presented at the 1996 Business Process and Workflow Conference in Amsterdam[4]

"...Shipping orders are received at the documentation department by a manager. The manager delegates the task to an employee. The delegated employee completes the shipping forms and the customs documents and returns them to the manager for approval. The manager then approves the forms and sends a notice of completion".

The diagrammatic version of this process, which illustrated in Figure 4, uses the UML activity notation [6]. Activities in this notation are represented via rounded box. A solid dot and a dot enclosed in a circle represent the begin and end points of the overall process, respectively. Arrows represent a simple ordering of the activity execution. A solid horizontal line represents an 'and' split or join in the activity network.

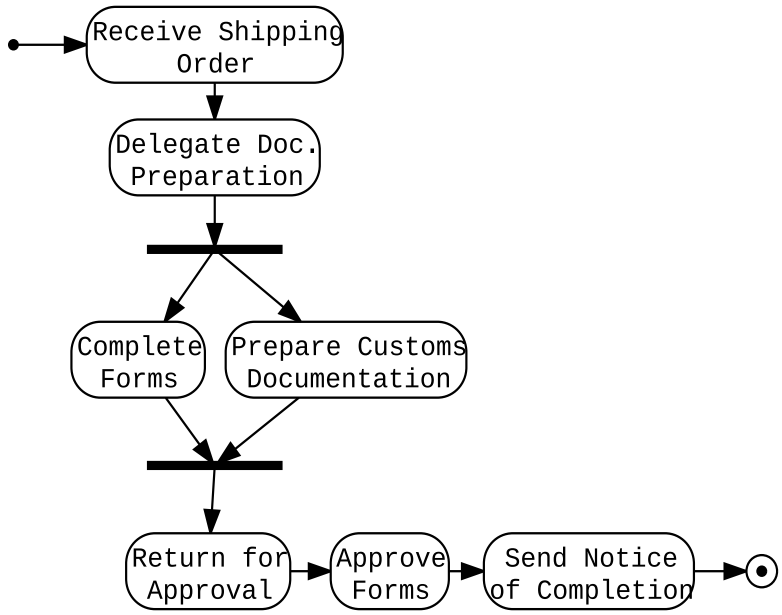

Figure 4: Process Document Request

One of the PIF axioms states that "the only agents which are allowed to perform an activity are those who have the appropriate capability that permits them to have a performs relation to that activity". The axiom is translated to FOPC as follows:

$performs(Actor, Activity) \rightarrow$
$object(Actor) \wedge activity(Activity) \wedge (capable\_of(Actor, Capability) \wedge refers\_to(Capability, Activity)).$

and stored in our database as follows:

```
constraint(performs(A,B),object(A),activity(B)).
constraint(performs(A,B),(capable_of(A,C),refers_to(C,B))).
```

It is noteworthy to say here that although we could translate the above axiom in one `constraint/2` predicate with description `performs` and constraints `object`,`activity`, `capable_of` and `refers_to`, we have choose to 'split' it in two predicates. The first represents the type-checking for the axiom, while the second specifies the properties that this axiom stipulates. This representation makes easier to analyse the output of our checking procedure and distinguish between 'low level' errors such as type-checking, and 'high level' ones such as conceptual errors by means of properties checking.

Another PIF axiom stipulates that a 'performs' relation must be defined over objects and activities. An actor is a specialisation of an object. Lets assume that the designer of this process chooses to represent the 'performs'

[6]for a detailed explanation of the notation see [1], and for a more detailed explanation of the diagram see [15]

relation in the following way:

$performs(Actor, Activity) \rightarrow object(Actor) \wedge activity(Activity).$

In addition, our program database of the process contains facts concerning the rest of the process entities in the following format:

$object(manager).$
$object(employee).$
$activity(approve\_forms).$
$activity(filling\_forms).$
$capable\_of(manager, approves).$
$capable\_of(employee, filling\_forms).$
$refers\_to(approves, approve\_forms).$
$refers\_to(filling\_forms, filling\_forms).$
$\ldots$

An error might occur in our specification, where we are allowed to assign an agent to perform an activity that it has not the capability to perform. This error could be demonstrated by giving the following Prolog goal:

$performs(employee, approve\_forms).$

This is satisfiable from the specification but it is contrary to the ontological constraints. Without our meta interpreter there is nothing to prevent the occurrence of this error, although it is syntactically correct. Assuming the ontological axioms given above, our meta-interpreter can detect this error. The discrepancy is reported as follows:

```
Conceptual error detected at:
(capable_of(employee,filling_forms),
refers_to(filling_forms,approve_forms)).
```

where the first line shows that an *employee* is capable of *filling_forms* and, if so, then it must has a 'performs' relation to the *filling_forms* activity according to the ontological axiom. However, as we can see from the second line a discrepancy detected where the *filling_forms* capability of the employee, does not refer to the correct activity, which is the *filling_forms* but to *approve_forms* which is the activity that the *manager* performs.

6 Further work and Conclusions

The method described in the above sections is still being developed and we believe, is a new angle on the use of ontologies. In particular, it offers a practical benefit in return for being formal in describing ontological constraints.

One of the weak points of the method is that, although it is guaranteed that the conceptual discrepancy will be detected, it relies on selecting the right goal to test. A further difficulty is in supporting "ontology engineers" in describing appropriate constraints and ensuring that the set of constraints supplied with an ontology have desirable properties(such as consistency). We are building a constraint editor with this aim in mind.

Finally, we would like to say that our method is capable of detecting conceptual errors as long as the designer of a system decides to use a formal syntax and the supported axiomatisation provided by an ontology. The method we have describe is "lightweight" to use because those testing specifications in Prolog give goals in the normal way, with checking for conceptual errors being given as a supplement to the results in satisfying the test goals. In other words, most of the formal "ontological engineering" can be hidden from the user of the system.

Acknowledgements

The research described in this paper is supported by a European Commission's Marie Curie Fellowship, under the TMR contract: ERBFMB-ICT-97-1955 for the first author and a EPSRC IT Advanced Fellowship for the second author.

References

[1] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Welsey, 1998. ISBN: 0-207-57168-4.

[2] 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.

[3] W Clocksin and C. Mellish. *Programming in Prolog*. Springer, 4th edition, 1994. ISBN: 0-387-17539-3.

[4] Workflow Management Coalition. Abstract Specification. WFMC-TC 1012, October 1996. Interoperability demonstration presented at the 1996 Business Process and Workflow Conference in Amsterdam.

[5] A. Finkelstein. Reviewing and Correcting Specifications. *Instructional Science*, 21:183–198, 1992.

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

[7] N. Gehani and D. McGettrick. *Software Specification Techniques*. Addison-Welsey, 1986.

[8] T.R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[9] N. Guarino and P. Giaretta. Ontologies and Knowledge Bases: Towards a Terminological Clarification. *Towards Very Large Knowledge Bases*, 1995. IOS Press, Amsterdam.

[10] J. Lee, M. Gruninger, Y. Jin, T. Malone, A. Tate, and G. Yost. The PIF Process Interchange Format and framework - version 1.1. Working paper No.194, MIT Center for Coordination Science, 1996.

[11] J. Lee and T. Malone. Partially Shared Views: A Scheme for Communicating between Groups Using Different Type Hierarchies. *ACM Transactions on Information Systems*, 8(1):1–26, 1990.

[12] B.H Liskov and J. Guttag. *Abstraction and Specification in Program Development*. MIT Press, 1986.

[13] L.W. Lloyd. Symbolic Computa. Springer-Verlag.

[14] C. Menzel. An Axiomatization of the PIF Core. 3rd PIF Workshop, Stanford, USA, August 1997.

[15] S.T. Polyak. A Supply Chain Process Interoperability Demonstration using the Process Interchange Format(PIF). Research Paper No.917, Department of Artificial Intelligence, University of Edinburgh, February 1998.

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

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

Appendix

A Translation program: `translator.pl`

In order to translate the given constraint in NF, check whether the constraint contains subterms, and if so, apply the set of rewrite rules, whenever applicable, to the subterm in order to produce the new subterm and recurse to the rest of the subterms until no more subterms are contained in the constraint. In case where a constraint does not contain subterm(s) then the `contains/4` predicate, will substitute the subterm with the given constraint in order to apply the set of rewrite rules.

```
%%% File:     translator.pl
%%% Author:   Yannis Kalfoglou <yannisk@ares.dai.ed.ac.uk>
%%% Created:  Thu Dec  4 1997

:-dynamic contains/4.
:-dynamic rewrite_expression/2.
:-dynamic rewrite/2.
:-dynamic contains_args/4.

rewrite((A=:=B),((A:-B),(B:-A))).
rewrite((A:-B),(\+(A,\+B))).
rewrite((\+ (A;B)),(\+ A,\+ B)).
rewrite((\+(A,B)),(\+A;\+B)).
rewrite((A,(B;C)),((A,B);(A,C))).
rewrite(((A;B),C),((A,C);(A,B))).
rewrite(((A;B);C),(A;(B;C))).
rewrite(((A,B),C),(A,(B,C))).
rewrite((\+ \+ A),A).

rewrite_expression(Constraint,Normalform):-    contains(Constraint,Subterm,Newconstraint,Newterm),
                                               rewrite(Subterm,Newterm),
                                               rewrite_expression(Newconstraint,Normalform).

rewrite_expression(Constraint,Constraint):- \+ (contains(Constraint,Subterm,_,_),
                                               rewrite(Subterm,_)).

contains(Term,TempTerm,New,New):-nonvar(Term),TempTerm=Term.
contains(Term,Subterm,Newterm,Newsubterm):-    nonvar(Term),
                                               Term =.. [F|Args],
                                               contains_args(Args,Subterm,Newargs,Newsubterm),
                                               Newterm =.. [F|Newargs].

contains_args([H|T],Subterm,[NewH|T],Newsubterm):-contains(H,Subterm,NewH,Newsubterm).
contains_args([H|T],Subterm,[H|R],Newsubterm):-contains_args(T,Subterm,R,Newsubterm).
```