Deploying Ontologies in Software Design

Yannis Kalfoglou

Ph.D.
University of Edinburgh
2000

Abstract

In this thesis we will be concerned with the relation between ontologies and software design. Ontologies are studied in the artificial intelligence community as a means to explicitly represent standardised domain knowledge in order to enable knowledge sharing and reuse. We deploy ontologies in software design with emphasis on a traditional software engineering theme: error detection. In particular, we identify a type of error that is often difficult to detect: conceptual errors. These are related to the description of the domain whom which the system will operate. They require subjective knowledge about correct forms of domain description to detect them. Ontologies provide these forms of domain description and we are interested in applying them and verify their correctness(chapter 1). After presenting an in depth analysis of the field of ontologies and software testing as conceived and implemented by the software engineering and artificial intelligence communities(chapter 2), we discuss an approach which enabled us to deploy ontologies in the early phases of software development(i.e., specifications) in order to detect conceptual errors(chapter 3). This is based on the provision of ontological axioms which are used to verify conformance of specification constructs to the underpinning ontology. To facilitate the integration of ontology with applications that adopt it we developed an architecture and built tools to implement this form of conceptual error check(chapter 4). We apply and evaluate the architecture in a variety of contexts to identify potential uses(chapter 5). An implication of this method for deploying ontologies to reason about the correctness of applications is to raise our trust in the given ontologies. However, when the ontologies themselves are erroneous we might fail to reveal pernicious discrepancies. To cope with this problem we extended the architecture to a multi-layer form(chapter 4) which gives us the ability to check the ontologies themselves for correctness. We apply this multi-layer architecture to capture errors found in a complex ontologies lattice(chapter 6). We further elaborate on the weaknesses in ontology evaluation methods and employ a technique stemming from software engineering, that of experience management, to facilitate ontology testing and deployment(chapter 7). The work presented in this thesis aims to improve practice in ontology use and identify areas to which ontologies could be of benefits other than the advocated ones of knowledge sharing and reuse(chapter 8).

Acknowledgements

I should start with the most significant acknowledgement which goes to my excellent supervisors David Robertson and Austin Tate. Their guidance and support was a second-to-none experience for me. David in particular has devoted a good deal of time to help me materialise the promising ideas and separate them from the less promising ones. His guidance and encouragement helped me go through the most arduous phases of my research. His countless questions and comments helped me convey my ideas in a more coherent manner. Austin's sense of direction and general understanding of how things fit together in the big picture of science has always been an inspiration for me.

I should also thank my thesis examiners John Lee and Enrico Motta. They gave me the pleasure to enjoy a thought-provoking VIVA. A successful research project is never carried out in complete isolation. My membership in the Software Systems and Processes(SSP) research group in Edinburgh was a great apprenticeship for me. It has been a continuous source of inspiration and i enjoyed working along with bright people who have also been very good friends. Steve Polyak, Renaud Lecoeuche, Edjard Mota, Alberto Castro, Stefan Daume, Jessica Chen-Burger, Virginia Biris-Brilhante, Daniela Carbogim, João Cavalcanti, Chris Lin, Josef Leung, Wamberto Vasconcelos are all past and present incarnations of this magnificent group. In particular, Steve's and Virginia's common interests in the world of ontologies was an endless source of fruitful discussions, Renaud's ideas on requirements engineering helped me formulate early ideas, Stefan and João shared visions with me for the Internet era, Daniela and Virginia provided much feedback in the computational logic realm. One of the great experiences being member of the SSP group is probably its unparallelled family feeling. I made great friendships with most of my colleagues for the years to come which are difficult to forget. I will especially remember Daniela's permanent smile and amazingly good mood which was an oasis during those endless and murky winter days in Edinburgh.

I have also benefited from being a member of the Department of Artificial Intelligence(AI) in Edinburgh. It is probably the best research centre in Europe and among the best in the world for AI. The countless seminars and talks that i attended is only a small fraction of the excellent facilities provided. I should also mention that the Department is located in one of the most beautiful cities in the world: Edinburgh is a stunning city.

Thanks to other members of the infamous E17 "PhDs room" in South Bridge for their friendship and support throughout the duration of the project: Daqing He, Hua Cheng, Gerhard Wickler, Hasan Kamal, Finlay Smith, Sonia Schulenburg, Elias Biris to mention just a few.

I enjoyed presenting parts of this thesis in conferences, workshops and i had the pleasure to give invited talks and participate in a panel discussion. I thank all those colleagues with whom i enjoyed discussions and collaboration over the last three years: Tim Menzies, Klaus-Dieter Althoff, Enrico Motta, Dieter Fensel, Frank van Harmelen, Richard Benjamins, Derek Sleeman, Alun Preece, Mike Uschold were all supportive with their constructive criticism and insight.

This research was supported by a European Commission's Marie Curie Research Fellowship under the programme Training and Mobility of Researchers. Their financial support made this project materialise.

Finally i would like to thank all my personal friends and my family back in Athens, Greece who have been marvellously supportive and patient all those years: Spyros, Leonidas, Nikos, Alexis, Dimitris, Vivi, and Pavlin. I am especially indebted to my family Thodoris, Nina and Tassos who made me feel like i had never left home.

Declaration

I hereby declare that I composed this thesis entirely myself and that it describes my own research.

Yannis Kalfoglou
Edinburgh
June 13, 2000

Contents

List of Figures

Chapter 1

Introduction

In this thesis we investigate the relation between ontologies and software design. These two themes are studied, primarily, in the artificial intelligence(hereafter, AI) and software engineering(hereafter, SE) communities, respectively. Despite the bulk of work in ontologies from AI researchers and in software design from SE researchers, there is still too little discussion on the interaction of these two core issues. We are motivated by recent advances on modelling domain knowledge, stemming from AI research, and at the same time curious to apply certain types of that knowledge to the early phases of design, an area traditionally researched by SE scholars.

Software design is still a young field, and we are far from having a clear articulation of the relevant principles. Winograd gives the succinct definition: "[...] is often used to characterise the discipline that is also called software engineering - the discipline concerned with the construction of software that is efficient, reliable, robust, and easy to maintain."([Winograd 96]). Although work has begun in engineering software design with the emergence of methodological approaches([Potts 96]), guidelines, and incorporation of rationale([Moran & Carroll 96]), there is still an area that remains relatively unexplored: bringing into the design process explicit knowledge regarding the domain on which the system to be developed will operate.

The study and modelling of that knowledge is a core theme in AI research. Having their roots in knowledge representation, knowledge engineering methods and techniques gave AI researchers a powerful tool for transforming contextual knowledge into machine-readable form to enable mechanised reasoning about a domain of interest. Ontologies are such a form of domain knowledge. However, the main focus of the ontological community is on deploying ontologies to support knowledge sharing and reuse.

We advocate that these are not the only areas which can benefit from ontologies. In this thesis we support this claim by deploying ontologies in software design with an emphasis on a traditional SE theme: error detection. In section 1.1 we elaborate on the kind of errors in which are interested and in section 1.2 we describe the role that domain knowledge plays in their detection. The solution we propose is further analysed in section 1.3 along with a brief summary of our results. In section 1.4 we give the organisation of this thesis and we conclude this introductory chapter with a summary in section 1.5.

1.1 Systems design

The use of blueprints for guiding the development process of projects is common in many disciplines. In particular, in the field of software systems design, these blueprints are precise and independent descriptions of the desired system behaviour. Those descriptions, often called specifications, are crucial for the success of every project since they guide the way in which programmers will construct the desired software. Errors that are made at these early steps of software development have serious side-effects on the project when they remain undetected. This is because they may not be detected by those who use the 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 stems from the field of formal descriptions expressed in logic as a medium of blueprint, the purpose of which is to: "define all required characteristics of the software to be implemented, and thus form the starting point of any software development process"([Fuchs 92]). However, even with executable declarative specifications([Fuchs & Robertson 96]), it is difficult to make blueprints error-free.

When we describe a chosen domain in the form of a software specification we can distinguish two kinds of errors: the ones that are related to the mathematical language underpinning the formal model and those that are related to the description of the domain itself. An example of the first type of error is when we write a non-terminating recursion using a logic programming language. These are often detectable via normal debugging techniques. On the other hand, the latter type is difficult to detect since it requires subjective knowledge about correct forms of domain description to be included in the specification. We call this error a *conceptual error*. For example, in an ecological model we may define animals that photosynthesize or in a process model we may assign an activity to an agent that is not capable of performing it. These kinds of errors reflect a misunderstanding of domain knowledge although they may be mathematically elegant. This makes their detection difficult.

1.2 The role of domain knowledge

The notion of knowledge has been studied in many disciplines ever since the early days of science. Nowadays, the term domain knowledge has become popular in the AI community, and specifically in the areas of knowledge acquisition, modelling, and management with end products such as intelligent systems. It is centred upon the notion of knowledge - a definition of which is offered by Uschold([Uschold 98a]): "Knowledge is anything that can be known or believed about a real or hypothetical world." Although this definition is generic, when applied to a domain it refers to specific knowledge regarding that domain. It can have many forms, like a textbook for engineering mathematics or a conceptual model of a process in a manufacturing plan. It can be explicit or tacit since the ideas may reside in someone's head.

In the last decade we have witnessed the emergence of computational forms of domain knowledge, mainly from the AI research community. The most widely known and used are ontologies and Problem-Solving Methods(hereafter, PSMs). Ontologies are explicit representations of a shared understanding of the important concepts in some domain of interest. PSMs are reusable, application-independent descriptions of problem solving behaviour. In this thesis we focus on ontologies but give pointers to PSMs references in chapter 2.

As the above working definition of ontologies implies, they forge agreements on the use of shared terminologies that will enable a stipulated group of people to share and reuse knowledge regarding a domain. They are the cornerstone for knowledge sharing and reuse among intelligent systems. The type of ontology that attracts our attention is that of formal ontology. A formal ontology is distinguished from others in that it provides a set of axioms which are intended to restrict the possible interpretations an ontological construct could have. These axioms are used by ontological engineers during the construction of an ontology as a way to formally define the relations between ontological constructs.

Recall from the previous section where we hinted at the problem of conceptual error occurrences: we argued that their detection relies on the provision of correct forms of domain description to be applied to the system description. Ontologies provide these forms of domain description whereas ontological axioms enforce their correctness. In the next section we will see how this idea can be harnessed to detect conceptual errors in specifications.

1.3 Solution proposed and results

In our work we deployed formalised domain knowledge(i.e., ontologies), to check the early phases of systems design(i.e., specifications) for misuse(i.e., conceptual errors) of domain knowledge. To implement this idea we apply formal ontologies to specifications. The latter is connected to the former by using ontological constructs. We then deploy the ontological axioms to verify that ontological constructs are not misinterpreted in the specification. This is implemented through logic programming techniques, and in particular meta-interpretation as we will explain in chapter 4.

We applied and extended this idea in various ways. Although the main focus was on existing specifications and ontologies, we also used this approach in cases where either the ontology or the specification did not exist. In the former case the ontology was derived by the specification, in the latter we devised an exemplar specification which adopts the existing ontology. Moreover, we applied this approach to ontologies themselves. As we describe in the next chapter, there is an acknowledged need for verifying ontologies before releasing them. By using a novel, multi-layer architecture for deploying ontologies in applications we were able to use exactly the same techniques for conceptual error detection to verify the correctness of ontologies themselves. We list below the claimed contributions of this thesis:

- *Deployment of ontologies in software design.* This helps to narrow the gap between domain knowledge and applications. It is often argued that most applications are not explicitly connected to their domain, and even if they are, no means for verifying their conformance to domain knowledge is provided. In this thesis, we provide a novel approach to deploying and verifying such knowledge in applications.

- *Worked examples of how ontologies can be deployed for purposes other than knowledge sharing and reuse.* The majority of ontology applications deal with knowledge sharing and reuse. In this thesis, we deploy them to improve the reliability of systems with respect to consistency checking. We provide a variety of working examples of how this can be done: in business process modelling, in ecological modelling, in air campaign planning, in systems dynamics, and other cases described in chapters 5 and 6.

- *A neutral architecture with supporting tools for deploying ontologies.* We invented a novel, multi-layer architecture, which makes it easy to deploy complex ontological structures in applications. In addition, we support this architecture with a variety of tools: an editing system for defining ontological axioms and relations that hold between concepts, a translator for automatically transforming them to the format manipulated by a designated error checking meta-interpreter, an integrated front-end which provides access to other program-synthesis specific tools for constructing specifications written in Prolog. These are described in chapter 4 and applied in the working examples of chapters 5 and 6.

Throughout the remainder of the thesis, and in particular in chapters 4, 5, 6 and 7 we will present implementations used to support these claims. We revisit the claims in chapter 8 where we justify the contributions made.

1.4 Organisation of this Thesis

The remainder of this thesis is organised as follows:

- *Chapter 2: Literature review*: in this chapter we survey the literature with emphasis on software testing as practiced in the SE and AI communities. We pinpoint the major contributions of these two fields and then scrutinise the field of ontologies. In particular, as ontologies are the main vehicle of our work we are interested in surveying all aspects related to their development, deployment and maintenance as well as exploring existing implementations, analysing the trends in the ontological community and focusing on their potential contribution to software design.

- *Chapter 3: Domain knowledge in systems design*: in this chapter we further analyse the role of domain knowledge, as deployed by ontologies, in the early phases of software design. These are described from the SE perspective and we highlight the importance of specific type of errors that often remain undetected: conceptual errors.

- *Chapter 4: Deploying domain knowledge*: this chapter is the heart of this thesis. We provide the theoretical background of all implementations following in chapters 5 and 6. In particular, we explain how logic programming(section 4.4) can play a crucial role in the deployment of ontologies to the early phases of software design in order to support error checking. To implement this we need formally to specify the conceptual error detection(section 4.2), as well as a standard way of representing ontological axioms, which we call *ontological constraints*(4.1). We also present a neutral architecture(section 4.5) for executing ontologies and specifications that adopt them in an integrated environment. We then extend this architecture(section 4.6) in order to verify the correctness of ontological constraints themselves, thus opening the root for checking ontologies for correctness. An example of this, is presented in chapter 6. However, this approach has some limitations which are listed in section 4.8.

- *Chapter 5: Evaluating the approach*: in this chapter, we materialise the theory presented in the previous chapter by applying the approach to various scenarios. We first describe three possible instantiations of the ontology/application pair where a conformance check with respect to the correct use of the ontology in the application is required(section 5.1). In the sequel, we provide examples of implementations of these cases in the areas of: business process modelling(section 5.2), ecological modelling(section 5.3), and air campaign planning(section 5.4). Some additional cases are presented in section 5.5.

- *Chapter 6: Evaluating the multi-layer approach*: while in the preceding chapter we focussed on the use of the standard(single-layer) architecture, here we evaluate the multi-layer approach which is suitable for manipulating complex ontological structures. We walk through a detailed example in the area of systems dynamics by applying the respective ontology to two exemplar systems. We also elaborate on the use of this approach to check ontologies themselves, as we did in this case which resulted to the detection of an ill-defined construct in the original ontology(section 6.4).

- *Chapter 7: Extensions*: in this chapter we speculate on future research directions by presenting implementations in two major areas: linking this work with the area of program synthesis(section 7.3), and exploring the role of the emerging *experienceware*, stemming from the SE community, in ontology deployment(section 7.1). We also analyse the impact of the approach in verifying knowledge models, in general.

- *Chapter 8: Contributions*: we close this thesis by revisiting the contributions made, listed in the previous section. We analyse and justify them by linking them with the implementations of chapters 5, 6 and 7.

In figure 1.1 we present a tabular form of this thesis contents. We place horizontally the main thematic areas tackled in this thesis: *deploying ontologies*, *worked examples* and *multi-layer architecture/tools* and vertically the chapters' numbers. This diagram can be used as a "road-map" to find out in which chapters the main thematic issues are tackled. For example, we describe the deployment of ontologies in chapters 3 to 6 inclusively, with the first two chapters giving the theoretical background followed by the implementation part in the last two.

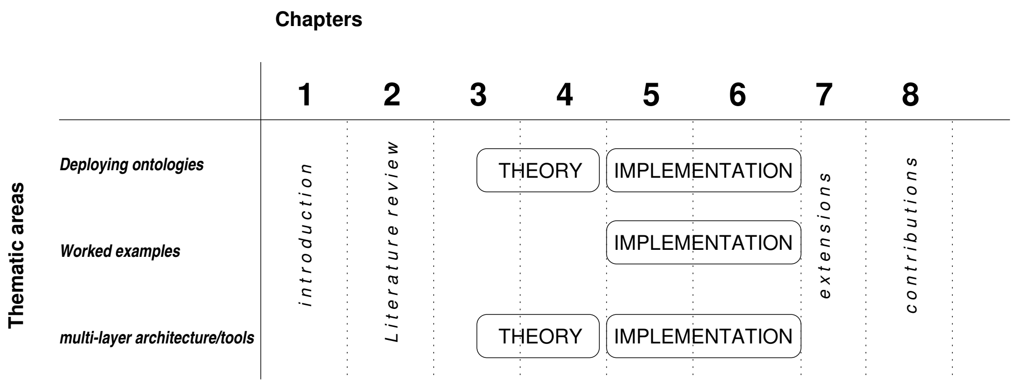


Figure 1.1: The organisation of this Thesis.

1.5 *Chapter summary*

This chapter introduced a novel idea in ontology deployment. We argued that despite the dominant interest of the community in the areas of knowledge sharing and reuse other areas can benefit from ontologies. We identified a potential area of applications in the context of conceptual errors occurring at the early phases of software design. We also described, briefly, this idea and set up the scene for unfolding it, accompanied by implementation in the following chapters. Initially though, we survey the field of software testing from the SE and AI points of view angle along with a detailed overview of the field of ontologies in chapter 2.

Chapter 2

Literature review

In this chapter we highlight the problem of errors that occur in the early phases of design. As we stated in the introductory chapter, we are interested in a specific type of error which we call *conceptual errors*. An area which encompasses any error checking theme is that of software testing. We review the field of software testing in section 2.1 before proceeding to survey contributions made in this field from the AI community(section 2.2). Lastly, we will scrutinise the foundation of our work, the field of ontologies in section 2.3.

2.1 Software testing

Software testing is a time-consuming and costly activity in the software development life-cycle. Among the early contributions to the field is the work of [Meyers 79] where software testing is conceived as a way of "ensuring the correctness of a program against its specification by executing the program on a test case and comparing the result with the expected results". Over the years the term has expand its scope as we see in the working definition provided recently by Friedman and Voas:

> "Software testing is a verification process for software quality assessment and software quality improvement. Software testing assesses the correctness for both the syntactic and semantic views of software."([Friedman & Voas 95])

Testing is intended to reveal failures or to provide confidence that failures do not occur, where a failure is the observable result of erroneous program behaviour. This is typically done by selecting test data, executing the program on that data, and comparing the results to some test oracle, which determines whether the results are correct or erroneous. During the eighties and early nineties a lot of work was focussed on test data selection and determination of testing criteria.

Many testing criteria([Richardson & Thompson 93]) select test data focussed on detecting failures caused by particular fault types, where a fault is a syntactic defect in the source code. This approach is also called 'fault-based testing'. Fault-based testing criteria fall into two broad categories: (a) measurement of the adequacy of pre-selected test data and, (b) guidance of test data selection. Among the first test data measurement criteria was the *mutation analysis* introduced in [DeMillo *et al.* 78]. *Mutation analysis* seeds single-token faults into the source code to produce 'mutant' programs. The system then executes the original and mutant programs on the pre-selected data and determines which mutants are 'killed', that is, which produce different output results from the original for at least one test datum. Then, the tester augments the test data set iteratively to eliminate the seeded faults that have not been distinguished and that are determined not to be equivalent to the original code. More recent fault-based measurement criteria are those introduced by Morell and Zeil. Morell's work is based on a fault-based testing model([Morell 90]) which has been used to symbolically represent faults that would not be detected by execution on a pre-selected test data set(this is also referenced as *symbolic fault-based testing*([Morell 88])). Zeil's work resulted in the *perturbation testing*([Zeil 89]) which identifies faults of a particular functional class that would not introduce an incorrect state.

While the above criteria focus on data measurement there are others that guide the test data selection process. Early work in this area is the *error-sensitive test case analysis*([Foster 80]) which consists of conditions sufficient to distinguish expressions that may contain a fault from the correct expression for several fault classes. Howden's work augmented this approach by concentrating on the low level 'functions'(e.g., statements) giving birth to the notion of *fault-based functional testing*([Howden 87]). An example of this is the *Mothra* mutation analysis system([DeMillo & Offutt 91]) which defines constraints on a test data set required for the set to be mutation adequate.

A modern example of fault-based testing is the RELAY model, which is a fault detection model that provides a framework within which other testing criteria's capabilities can be evaluated. The RELAY([Richardson & Thompson 93]) model defines failure conditions which describe test data for which execution will guarantee that a fault originates erroneous behaviour that also transfers through computations and information flow until a failure is revealed. In addition, since the early nineties work has begun in the area of *fault-based analysis*. A representative in this area is the *PIE* software assessment model([Voas 92]) for software testability. In the model, testability is a function of three characteristics: the likelihood of location execution, the likelihood of a fault at that location causing infection of the data state, and the likelihood of a data state infection at that location propagating to the output state([Voas & Miller 95]).

The most basic distinction among types of testing, however, is between *black-box* versus *white-box* testing. *Black-box* testing is based on external specifications without knowledge of how the system is constructed([Perry 95]). It quantifies the quality of the software strictly as a function of the external behaviour of the code([Friedman & Voas 95]). In the SE community many conceive *specification-based* testing and *requirements-based* testing as classes of black-box testing because no actual code is considered in these kinds of testing([Voas & McGraw 98]). However, as we will see in section 2.2, this does apply to the (limited)use of executable formal specifications expressed in logic which are often viewed as normal programs.

The other kind of testing is *white-box*. Testing is based on knowledge of the internal code structure and logic and is usually logic driven([Perry 95]). White-box software analysis methods can largely be viewed in one of two ways: *structurally* and *semantically*. When you take the structural view of the software, you limit your knowledge about the software to the operators and operands that comprise it. When the semantic perspective is adopted, you instead attach meaning to the operators and operands; that is, you consider what transpires when an input value is fed to the software and an output value is produced([Friedman & Voas 95]). Despite the ability to reason about the semantics of the code, white-box testing techniques are usually applied only at the subsystem level, and not to complete systems([Voas & McGraw 98]).

Testing is not limited to the coding phases of development. Voas and McGraw explain why: "if you can create an oracle that is able to ascertain correct behaviour from incorrect behaviour, and is not directly based on the requirements or the specification, then software testing can actually reveal errors in earlier phases of the life-cycle."([Voas & McGraw 98]) But, errors found at the level of requirements and specifications are often expensive, because their discovery leads to redesign. Perry speculates on the costs involved in fixing these errors: "All errors are costly, but the later in the life-cycle that the error discovery is made, the more costly the error. An error discovered in the latter parts of the life-cycle must be paid for four different times. The first cost is developing the program erroneously, which may include writing the wrong specifications, coding the system wrong, and documenting the system improperly. Second, the system must be tested to detect the error. Third, the wrong specifications and coding must be removed and the proper specifications, coding, and documentation added. Fourth, the system must be retested to determine that is now correct."([Perry 95]).

As a response to the need for high quality, error-free software, formal methods have regained much attention by the community since the early nineties. Although formal methods might help to produce better code, they can in no way guarantee what will happen when an analysed program is placed in a messy real world. For instance, Friedman and Voas provide the following criticism with respect to formal verification:

> "Formal verification does not provide absolute assurance; for example, the formal specification could be wrong or the verification tool could contain a bug. The mapping between the informal requirements communicated by the customer and the formal specification, as well as the real world and an abstract model, can be imperfect. The assumptions made about the environment can be wrong."([Friedman & Voas 95])

The above quotation highlights the problem of 'mismatch' between the real world and the formal model that represents it. We will revisit this point later(section 2.3 and chapter 3) when we examine the role of ontologies. In general, the use of formal methods is a traditional point of contention in SE literature and recent reflections and analyses of trade-offs can be found in [Cleland & MacKenzie 95].

Nowadays, software testing is challenged with the emergence of more complex systems. In particular, the area of testing object-oriented software([Kung *et al.* 98]) has received much attention by the community. Early work in this area is the external class specification technique, proposed in [Meyer 88], which specifies different interface methods and associated preconditions, postconditions, and invariants. An extension to this approach is the *method sequence specification* technique([Kirani & Tsai 98]), which takes into account the causal relationship between methods of a class. This is based on the correct order in which the methods of a class can be invoked by the methods in other client classes. The underlying technology of these systems is state transition diagrams which are suitable for modelling the dynamic behaviour of classes. Modern expressions of this technology in the area of object-oriented testing are the *object-state diagrams*([Kung *et al.* 96]). These are based on object state testing which is concerned with testing the objects' state behaviour rather than the control structures of individual data. A recent example of the use of state based testing is the work of Tackett and van Doren who applied state transition technology to process control. In ([Tackett & vanDoren 99]), the authors presented a state-based process in the development of a system for missile warning deployed in Cheyenne Mountain in the US.

We summarise this section with a critique: we observe that the majority of work in software testing is focussed on perfecting the method rather than broadening the error typology. While the former is desirable for coping with the ever increasing complexity and size of today's systems the latter is equally important. No matter how perfect is the method we deploy, there will always be a type of errors that might creep through to the later phases of development: conceptual errors. In the following section we survey AI-based contributions to testing and error checking.

2.2 AI-based testing

We follow a bottom-up fashion in reporting AI-based contributions to testing: we review the area of logic programming with emphasis on debugging techniques before proceeding to explore thematic areas and applications in the verification and validation of intelligent systems. We also review some model checking techniques related to our interests.

Logic programming offers an attractive feature: the ability to use logic for both specification and computation. By using a logic programming language we are able to reason about the semantics of the desired system and focus on *what* the system is intended to do rather than *how* it will be implemented. In a comprehensive review of debugging tools for Prolog programs([Brna *et al.* 91a]), various aspects of debugging logic programs were investigated. We repeat them here along with major contributions in each area. The area of *improved monitoring* deals with monitoring the program's execution. Among the best known contributions in this area is the *Transparent Prolog Machine(TPM)*. The TPM([Eisenstadt & Brayshaw 88] was introduced as a medium for visual representation and animation of Prolog programs. The system provides a representation of the inner workings of the Prolog interpreter and gives details about clause head matching, variable naming and deals with non-logical features of the language such the 'cut' and meta-logical built-in predicates. Its contribution to debugging is that programmers can inspect the execution of a program and apply their own debugging skills.

Related to this area is the work done on *automated search*, which looks for specific events in a program's history. This gives the ability to track the bug symptom through its causal chain to the source of the problem. Its benefit is in locating bugs in the code but requires the deployment of additional tools, such as a cliche analyser, to semi-automate the process of locating bugs which are often organised in suspicious symptom clusters([Eisenstadt 85]). To facilitate the understanding of logic programs the area of *annotated programs* provides various pieces of information to the programmer at edit-time. These pieces of information describe the intended behaviour of the program and are collected in a *meta-database*. This comprises of facts about the program and its execution in the form of declarations edited by the user([Brna *et al.* 91a]). Then the static analyser infers extra declarations and adds them to the meta-database. This can, in turn, be examined by the user and by other parts of the system including the debugger. However, the meta-database approach requires maintenance from the user and usually deals with predicate-level bugs rather than program-wide ones. Complementary to this idea was the introduction of *technique-oriented* debugging tools. They use as a basis for the debugging process *programming techniques*([Brna *et al.* 91b]) and check whether these techniques have been implemented correctly in the constructed program. They require knowledge of which technique the user is intended to use. The bug report would then be limited to all and only the violations of that technique - wider intentions would not being taken into account.

Further user involvement is encountered in the area of *guided debugging*. The underlying idea is that there is an infallible source of knowledge about whether a goal should succeed and which variable instantiations should be obtained. This is also known as an *oracle* and is based on the work of Shapiro on *algorithmic debugging* ([Shapiro 83]). The system uses this information to guide the programmer to the program code error. However, this approach requires the programmer to identify whether the result is the expected one. It is a computationally expensive approach since it requires the oracle to answer a potentially large number of questions about which goals should succeed and what solutions should hold.

An attractive area of research is that of *automated debugging and program verification*. We need to show that the program meets the specification which effectively means that the system needs to know the program specification. This also has an implication that the program must be separated from the specification at run-time in order to automatically locate and correct the program code error. A system that uses a specification of a program, written in Prolog itself, to generate test cases, locate and correct a bug, is presented in [Dershowitz & Lee 87].

The idea of using logic to support all the activities concerned with debugging a program - designing a test case, detecting an error in the program, locating the error, and fixing it - is further explored in [Dershowitz & Lee 93]. The authors presented a methodology, called *logical debugging*, based on specifications expressed in logic to reason about the correctness of programs derived from these specifications. In particular they used executable input/output specifications to define the intended behaviour of a program and to generate test cases for bug discovery. They then employed the execution mechanism of a Prolog machine to locate bugs, using specifications to validate computation results. They also used heuristics to analyse bugs and suggest fixes, and used techniques in deductive theorem proving and inductive synthesis to mechanise the bug correction process, also with the help of specifications. As their work relies heavily on the specifications they assume that these are correct. The main contribution of their work is probably the provision of an automated debugging environment, the *constructive interpreter*. The interpreter is constructive in the sense that is assumes an active role during the debugging process and tries to complete the construction of the program being debugged, all with very little user involvement.

Recently, Naish presented a *declarative debugging* scheme([Naish 97]) which can be used to diagnose bugs while hiding the complexities of the execution from the programmer. The scheme relies on a way of representing a computation as a tree and a way of determining the correctness of a subcomputation represented by a subtree. By suitably instantiating these two parameters, the scheme can diagnose multiple classes of bugs in a variety of languages. Declarative debugging can be thought as searching a tree for a buggy node. To determine if a node is erroneous, an oracle is used. Often the oracle is the programmer, who is asked questions about the validity of atoms.

However, even with these techniques stemming from the realm of logic programming the problem remains unsolved. To quote [Brna *et al.* 91a]:

> "It is often stated that Prolog programs can be seen as executable specifications. Consequently, if the programmer can get the specification right then there will be no program code errors. Unfortunately, it is not so simple to get the specification right."

We now shift our attention to an application area where most of the techniques discussed above are applied either directly or indirectly: verification and validation of knowledge based systems(hereafter, KBSs).

The verification and validation of KBSs is a long-standing theme in the AI research agenda. *Verification* is the process aimed at demonstrating whether a system meets its specified requirements. This is often referenced as "building the system right" ([O'Keefe *et al.* 87]). *Validation* is a process aimed at demonstrating whether a system meets its user's true requirements, also refered to as "building the right system".

Verification and validation can be viewed as a set of requirements and an associated process in which the techniques are applied, as part of the whole development process([Meseguer & Preece 95]).

Much of the work done in verification and validation of KBS is based on the assumption that, because KBSs are difficult to specify, practical verification and validation techniques should not depend upon the existence of detailed specification documents([Meseguer & Preece 95]). This is seen in the work of verification by anomaly checking([Preece *et al.* 92]). The authors provide a set of verification properties and compare a set of tools against them. In particular, they check KBSs for domain-independent anomalies such as inconsistency and incompleteness. More verification properties are presented in [Preece & Shinghal 94] along with the detection of errors in actual KBSs. However, most of the validation approaches developed for KBSs assume to work on an implemented system, like a prototyping. This late validation process has some drawbacks the most important being that when an acceptable system is deemed to have been produced, it is not clear what the system actually does or it may fail to do([Meseguer & Preece 95]). In addition, verification of KBSs against some domain-independent properties does not solve the problem of demonstrating its correctness. As Meseguer and Preece report:

> "An example of a domain-independent property is consistency, which means that from a consistent input the KBS cannot produce a contradictory output. Domain-independent properties appear as prerequisites for adequate functioning of a KBS, and they should be tested. However, although they are necessary they are not sufficient because they say little about the actual KBS correctness"([Meseguer & Preece 95])

To overcome these problems with the early validation and cope with the verification of KBS correctness, the use of formal specifications([Meseguer & Preece 95]) has been investigated. Formal specification techniques provide levels of description which support both verification and validation while verification and validation techniques feed back to assist the development of the specifications. Their contribution was studied from the viewpoint of the dominant techniques for verification and validation of KBSs. We summarise them below as reported in the review of Meseguer and Preece: *inspection*, *static verification*, *empirical testing*, and *empirical evaluation*. *Inspection* refers to the detection of semantically incorrect knowledge in the KB and is usually performed manually by human experts. *Static verification* checks the KB for anomalies. These are static patterns that suggest the presence of an error in the encoded knowledge. *Empirical testing* involves execution of the system on sample data sets to check its correctness. For complete correctness the testing has to be exhaustive but this is not computationally feasible in real systems. Two kinds of testing are identified: *structural* testing which aims at executing as many of the KBS components as possible, and *functional* testing which checks the function of a KBS by comparing its observed input/output relationship with that specified in the requirements. *Empirical evaluation* deals with issues such as technical performance, acceptability, inclusion in the organisation, responsibility issues. It is a human activity which is highly application-dependent.

In the inspection process, formal specifications of a KBS can alleviate the problem of ambiguity regarding the interpretation of informal specifications, and used for direct validation from domain experts. However, it is argued that the formal notations employed in the specifications impose a comprehension problem for domain experts who are unfamiliar with them. As a solution the inspection of the semi-formal conceptual model which underpins the KBS development has been proposed.

Static verification is an area where the formal specification techniques have been applied most. They allow verification to be performed on the same specification and to check either the specification or domain-dependent properties. In addition, some formal specification languages provide additional opportunities for verification. For example, the modular architecture and declaration of hierarchies of types provided by languages such as DESIRE([Treuer & Wetter 93]), KARL and $(\mathrm{ML})^2$([Fensel & vanHarmelen 94]) allow to check for additional properties, like violation of modularity, and type mismatches.

Formal specifications in empirical testing allow developers to test the specification itself. This helps them to assess whether they are specifying the intended system. It might require symbolic execution of the specification but specially designed specification languages([Fensel & vanHarmelen 94]) provide this feature. However, this does not waive the need to test the implemented system because the transformation from specification to implementation is usually done manually and new errors can be introduced.

The area of empirical evaluation is one which seems to have benefited little from the use of formal specifications. The is no experience of applications of formal specification in this verification and validation aspect, but given the benefits that formal specifications may offer, it is foreseeable that they may be applied in the evaluation process of KBSs.

Although the role of formal specifications in the KBSs development has been praised by many and more uses are envisaged in the foreseeable future there is an area that more work is required. As Meseguer and Preece report:

> "[...] the question of verifying the specifications themselves is still open, as few of the existing special purpose KBS specification languages have well developed proof techniques at present. If general purpose specification languages are used, then their existing proof techniques apply, but these have been found to entail a great deal of labour when KBS are specified"[Meseguer & Preece 95]

We will revisit this point in chapter 3 when we will explore the early phases of system development in more detail.

Lastly, we sample the area of model checking. Model checking is a technique that relies on building a finite model of a system and checking that a desired property holds in that model. The check is performed as an exhaustive state space search that is guaranteed to terminate since the model is finite. Although model checking has been applied primarily in hardware the current trend is to apply this technique to analysing specifications of software systems. For example, in [Heitmeyer *et al.* 96], the authors used model checking techniques to automatically check the consistency of system's requirements. In [Clarke & Wing 96], a summary of this field is presented in the wider setting of formal methods. The role of logic programming in model checking is further investigated in the context of the LMC project([Cui *et al.* 98]). Model-based Diagnosis(MBD) ([deKleer & Williams 87]) is a neighbouring area that is traditionally used for identifying faults in physical systems. The basic assumption of MBD is that if the underlying principles of a particular domain can be captured in a model, that model can then be used to locate faults in arbitrary systems that are composed of the components of that domain([Reiter 87]). Although the main focus of MBD is on physical systems attempts have been made to apply them in software debugging. For example, in [Console *et al.* 93] the authors focussed on pure logic programs. These were logical theories themselves, and suggested the use of clauses as components for diagnoses. Since clauses in a logic program are independent, alterations of the program are trivial to produce without violating the semantics of the language. Recently, a model-based approach has been applied to conventional programming languages. In [Mateis *et al.* 99], a model-based approach for debugging Java programs was presented.

We summarise this section of AI-based software testing by highlighting the need to check for correctness the blueprints of systems, namely the specifications. As we reported previously, this is acknowledged by logic programming researchers(see, section on logic programming above) and by the verification and validation of KBSs community(see relevant section above). In the sequel we change our perspective and scrutinise the field of ontologies.

2.3 Ontologies

Ontologies are studied by many scholars who belong to different communities. In order to effectively give a comprehensive review of this intricate field, we explore it from the following angles: design, deployment, and tradeoffs. Design issues are explored in sections 2.3.1 to 2.3.5 where we introduce the term, explain the internal structure of ontologies, and describe ways of construction and methodologies used. Deployment issues are described in sections 2.3.6 to 2.3.7 with emphasis on applications, ways of deployment, and references to influential projects from both industry and academia. Lastly, we discuss potential problems, tradeoffs and solutions along with pointers to resources for further reading in sections 2.3.8 and 2.3.9.

2.3.1 Definitions

We start our review by explaining what an ontology stands for. Although a single definition will usually suffice, ontologies have a peculiar characteristic: there are a number of different definitions proposed and used. Even nowadays there are people who argue about the actual meaning of the term. A reason for this is, probably, the fact that ontologies are studied, developed, and applied by people with diverse backgrounds and interests. We do not subscribe to this debate over the meaning of the term in this thesis nor we will introduce yet another definition. Rather, we briefly review the most commonly used definitions found in the literature in order to explain what an ontology stands for.

One of the early definitions appeared in [Neches *et al.* 91]. The authors define an ontology as: "the basic terms and relations comprising the vocabulary of a topic area as well as the rules for combining terms and relations to define extensions to the vocabulary". This definition introduced the idea that ontologies can be viewed linguistically, as extensible vocabularies regarding a topic area. In the context of knowledge sharing, Gruber offered a short definition which became the most widely cited in the literature: "an ontology is an explicit specification of a conceptualisation"([Gruber 93]). This definition was further enriched by Borst and his colleagues in [Borst *et al.* 97], where they argued that the specification is actually formal and the conceptualisation is shared. Studer and colleagues analysed the terms used in the definition and provide the following explanation: "Conceptualisation refers to an abstract model of some phenomenon in the world by having identified the relevant concepts of that phenomenon. Explicit means that the type of concepts used, and the constraints on their use are explicitly defined. Formal refers to the fact that the ontology should be machine-readable. Shared refers to the notion that an ontology captures consensual knowledge, that is, it is not primitive to some individual, but accepted by a group"([Studer *et al.* 98]). Uschold offers a working definition which hints at the purpose of having ontologies: "An ontology is virtually always the manifestation of a shared understanding of a domain that is agreed between a number of agents. Such agreement facilitates accurate and effective communication of meaning, which in turn leads to other benefits such as interoperability, reuse and sharing"([Uschold 98a]). Others, consider ontologies as domain theories([Fikes & Farquhar 99]), as vocabularies([Skuce 95]), as standards([Mark 96]), etc. In [Guarino & Giaretta 95] the authors offer a clarification of terminological issues regarding the various definitions founded in the literature.

Based on the definitions quoted above we summarise what an ontology stands for: an explicit representation of a shared understanding of the important concepts in some domain of interest. The role of an ontology is to support knowledge sharing and reuse within and among groups of agents(people, software programs, or both). In their computational form, ontologies are often comprised by definitions of terms organised in an hierarchy lattice along with a set of relationships that hold among these definitions. These constructs collectively impose a structure on the domain being represented and constrain the possible interpretations of terms. The latter is the heart of our work and Gruber offers a neat explanation:

> "Ontologies are often equated with taxonomic hierarchies of classes, class definitions, and the subsumption relation, but ontologies need not be limited to these forms. Ontologies are also not limited to conservative definitions, that is, definitions in the traditional logic sense that only introduce terminology and do not add any knowledge about the world. To specify a conceptualisation one needs to state axioms that do constrain the possible interpretations for the defined terms."([Gruber 93])

2.3.2 Design principles

A number of design criteria have been proposed, originally analysed in [Gruber 95]. In the following list we recapitulate them:

1. *Clarity*: this refers to the effective communication of the intended meaning. Formalism has been proposed as a means to dispel ambiguities. For example, when a definition can be stated as a logical axiom, it should be. However, all definitions should be documented in natural language.

2. *Coherence*: that means that the ontology should sanction inferences that are consistent with the definitions. This not only applies to the defining axioms that

should be logically consistent but to the concepts that are defined informally, such as documentation and examples. If a sentence that can be inferred from the axioms contradicts a definition or example given informally, then the ontology is incoherent.

3. *Extendibility*: the ontology should be designed to anticipate the uses of a shared vocabulary. One should be able to define new terms for special uses based on the existing vocabulary, in a way that does not require the revision of the existing definitions.

4. *Minimal encoding bias*: an encoding bias results when representation choices are made purely for the convenience of notation or implementation. Encoding bias should be minimised, because knowledge-sharing agents may be implemented in different representation systems and styles of representation.

5. *Minimal ontological commitment*: an ontology should make as few claims as possible about the world being modelled, allowing the parties committed to the ontology freedom to specialise and instantiate the ontology as needed.

We should note that the above criteria are not always possible to meet by ontology designers. A number of tradeoffs have been identified([Gruber 95]), and ways of compromising between well designed ontologies and applicability have been investigated ([Borst *et al.* 97]). We will not expand on this issue in this thesis because it is peripheral to our topic: deployment of ontologies. To support this we shift our attention to the notion of ontological commitment which plays an important role in our work.

2.3.3 Ontological commitment

Ontological commitment refers to agreement on the use of the shared vocabulary by the agents commit to the ontology at question. When we say that an agent commits to an ontology we mean that its observable actions are consistent with the definitions in the ontology([Gruber 95]). It has been said that commitment to a common ontology is a guarantee of consistency but not of completeness, with respect to queries and assertions using the vocabulary defined in the ontology.

Guarino describes the role of ontological commitment in software: "ontological commitment should be made explicit when applying the ontology in order to facilitate its accessibility, maintainability, and integrity. This will lead to an increase of transparency for the application software which based on that ontology".([Guarino 98a]). These commitments are often encoded as axioms that enforce the syntactical consistency of the definitions used. This thesis, however, has been challenged by Guarino and colleagues who argue for a greater role of ontological commitment. In [Guarino *et al.* 94], the authors continue, an ontological commitment should capture and constrain a set of conceptualisations. They propose a formalisation of ontological commitments which: "offers a way to specify the intended meaning of [a logical language] vocabulary by constraining the set of its models, giving explicit information about the intended nature of the modelling primitives used and their a priori relationships". The work of Guarino and colleagues is focussed on the design phases of ontology. However, closer to our interests is the deployment of ontological commitments as described by Waterson and Preece in [Waterson & Preece 99]. They used ontological commitments to verify the correctness of a knowledge base that commits to a given ontology. We will revisit this issue in detail in chapter 3 where we explain how we use ontological commitment in our work along with references to similar efforts.

2.3.4 Methodologies

The construction of an ontology is a time-consuming and intricate task. Although, there are no standards to obey when building an ontology, various design guidelines and methodological approaches have been proposed and used. In particular, in a comprehensive review of the field([Uschold & Gruninger 96]) the authors report on two methodologies used in the context of the *Enterprise* ontology([Uschold *et al.* 98c]) and the TOVE project([EIL 95]). In the former, a skeletal methodology has been proposed([Uschold & King.M. 95]) which identifies five main steps: (a)identify purpose and scope, (b)build the ontology, (c)evaluation, (d)documentation, and (e)guidelines for each phase. Step (b) is further divided into ontology capture, coding, and integration of existing ontologies. This skeletal methodology was used in the construction of the *Enterprise* ontology but does not explicitly deploy a formal evaluation procedure. This was the main focus of the methodology used in the context of the TOVE project([EIL 95]). In particular, Gruninger and Fox used a formal methodology that supported evaluation of the ontology using the notion of *competency questions* ([Gruninger & Fox 95]). The underlying philosophy is to define a set of queries that the ontology can answer. These queries help to assess the ontology's competence. They evaluate the expressiveness of the ontology which is required to represent these questions and characterise their solutions. These queries are drawn from a number of motivating scenarios which are story problems or examples which are not adequately addressed by existing ontologies.

Apart from the work on evaluation and construction methodologies by Uschold, Gruninger and colleagues, others have focussed on the preliminary phases of construction. In [Fernandez *et al.* 97] the authors presented a system, called *METHONTOLOGY*, which provides support for the entire life-cycle of ontology development. A distinguishing characteristic of the METHONTOLOGY framework is that it is tailored to support the early phases of development by employing the notion of *intermediate representations*. These are representations independent of the implementation language in which the ontology will be developed. The system that support the use of these representations is the Ontology Development Environment(ODE)([Blazquez *et al.* 98]). An overview of methodologies used in AI projects along with a comparison with standards from SE literature is given in [Fernandez 99].

2.3.5 Types

The development methodologies reported above were used in some of the ontologies which will be described in sections 2.3.6 and 2.3.7. Before we proceed to survey actual implementations of ontologies we describe various types of them as reported in the literature. Ontologies can be classified in terms of genericity. For example, broad ontologies like CYC([Lenat & Guha 90], model generic notions that forms the foundations for knowledge representation across various domains. These are also called top-level ontologies([Chandrasekaran *et al.* 99]), like Sowa's ontology([Sowa 00]). On the other hand, small-scale, domain-specific ontologies are carefully tailored to the domain at question. Examples of this type are the *PhysSys* ontology([Borst *et al.* 97]) which captures knowledge regarding physical system processes, the *AIRCRAFT* ontology ([Valente *et al.* 99]) used to represent air-campaign planning knowledge, the *PIF* ontology([Lee *et al.* 98]) used for business process modelling, etc.

Another classification of ontologies is concerned with their purpose. There exist *task* ontologies([Mizoguchi *et al.* 95]) that capture task-related knowledge independently of the domain that the task is defined. Complementary to these are the *method* ontologies([Chandrasekaran *et al.* 98]) which provide definitions of the relevant concepts and relations used to specify a reasoning process to achieve a particular task. Yet another type of ontologies is the *knowledge representation* ontologies. The most representative example is the Frame ontology([Gruber 93]) which captures the representation primitives used in frame-based languages. It allows other ontologies to be specified using frame-based conventions, as implemented by the Knowledge Interchange Format(KIF)([Genesereth & Fikes 92]).

Most ontologies, however, are placed under the tag *domain* ontology. These are designed to support a specific domain and applications defined within that domain. For example, the *PIF* ontology is concerned with the business process modelling domain and supports the exchange of information among a variety of business process modelling applications.

There is another type of ontology, the *linguistic* ontologies. The most illustrative examples are the Generalised Upper Model(GUM)([Bateman *et al.* 95]), WordNet ([Miller 90]), and SENSUS([Knight & Luk 94]). However, these usually have the form of a vast collection of terms which led to another classification with regard to the level of formality. These sort of ontologies are often called "terminological" ontologies whereas ontologies like TOVE are called "axiomatised" ontologies.

In their overview of the field, Uschold and Gruninger identified the following types with respect to the degree of formality: *highly informal*, *semi-informal*, *semi-formal*, *rigorously formal*([Uschold & Gruninger 96]). In the informal cluster we see definitions in natural language or at most in a structured form of natural language. In the formal cluster we have ontologies implemented in an artificial formal language(i.e., *Ontolingua*), or in first order theories with formal semantics, theorems and proofs of such properties as soundness and completeness(i.e., TOVE).

2.3.6 Engineering

Although many argue that engineering of ontologies is still in its infancy the first comprehensive reports covering all aspects of ontology construction and deployment began to emerge few years ago. We selectively refer to some of these efforts by highlighting their contributions to the field. In an experiment of ontology reuse([Uschold *et al.* 98a]), researchers working at Boeing were investigating the potential of using an existing ontology for the purpose of specifying and formally developing software for aircraft design. The application problem addressed was to enhance the functionality of a software component used to design the layout of an aircraft stiffened panel. They describe from start to finish a process that used an existing ontology, residing on the Ontolingua ([Farquhar *et al.* 97]) server, the *EngMath*([Gruber & Olsen 94]) ontology, which was then translated to the target specification language and integrated to an engineering software component. They then executed that component and demonstrated the benefits of reusing an existing knowledge component in the development process. The lessons learned from that experience is that ontology reuse can be pursued on a large scale and, under certain circumstances, it can be a cost-effective approach. We will revisit the tradeoffs identified by Uschold and colleagues in their experiment in section 2.3.8 while we continue here by reporting two studies that were focussed on the whole spectrum of engineering ontologies: the *AIRCRAFT* project, and the *PhysSys* project.

In [Valente *et al.* 99] the authors describe how they achieved reuse among ontologies themselves. The resulted ontology, *AIRCRAFT*[1], contains knowledge about types of US military aircraft, including data about the engines, pods, and fuel tanks that these aircraft can carry. The distinguishable feature of this ontology is how it has been developed in the first place. The process, which is described in [Swartout *et al.* 96], was based on the use of a large-scale, linguistic ontology, the *SENSUS*([Knight & Luk 94]). A characteristic of *SENSUS* is that it is actually constructed from extracting and merging information from existing electronic resources(like the *WordNet*, dictionaries,

[1] A demonstration version is electronically available from the URL: http://www.isi.edu/isd/ontosaurus.html

GUM ontology). The authors, used this broad coverage ontology and then devised a semi-automatic method which made it possible to identify terms in the original ontology that were relevant to their particular domain, and then pruned the ontology so that it included only those terms. In addition, they enhanced the newly emerged ontology with terms tailored to the domain of air campaign planning. These were military terms. The resulting ontology, *AIRCRAFT*, is accessible through an ontology development environment, the *ontosaurus* browser which supports the idea of "ontology developed collaboratively by the system developers themselves"([Swartout *et al.* 96]).

In [Borst *et al.* 97] a general and formal ontology, called *PhysSys*, is presented. It covers the domain of dynamic physical systems and it is composed by seven different ontologies. This work explored a new idea in ontology engineering, that is *ontology projections*: "a flexible mechanism to link and configure ontologies into larger ones." Three kinds of projections demonstrated in the paper, *include-and-extend*, *include-and-specialise*, and *include-and-project*. The latter was used to link an ontology developed by the group of *PhysSys* authors to an outsourced ontology, the *EngMath*. The *PhysSys* ontology was used as the foundation for the conceptual database schema of a library of reusable engineering model components, the *OLMECO* library. The library was evaluated by modelling and numerically simulating the existing heating system of a general hospital in Schiedan, the Netherlands([Borst *et al.* 97]). We will not describe further this project because it is analysed in detail in chapter 6 were we use the *PhysSys* ontology set for the evaluation of our work.

In the context of the *Plinious* project([van derVet & Mars 93]), the bottom-up method in ontology development is discussed([van derVet & Mars 98]). In contrast with the majority of approaches in ontology construction which fall into two categories, top-down and middle-out(analysed in [Uschold & Gruninger 96]), the bottom-up way "proposes to lay down the meaning of complex concepts by means of primitive meaning constituents." It has been applied to the domain of ceramic materials and covers their properties and the processes to make them. It was found that this approach was suitable for such a domain because, the authors continue, it is impossible to exhaustively predict in advance which concepts will be needed to express the knowledge founded in the texts. As this domain covers chemical substances, it was argued that listing all these substances is an open-ended task. As such, keeping track of the regular updates in a top-down designed ontology was impractical since it requires substantial effort and is error-prone. Consequently, the approach used supports reasoning along two orthogonal hierarchies: "the partonomy formed by substances and their constituents and the taxonomy formed by concepts and superconcepts"([van derVet & Mars 98]).

Other projects which provide an insight in the engineering process are the re-engineering effort of implemented ontologies, described in [Gomez-Perez & Royas-Amaya 99], and the collaborative effort in developing a common ontology for the knowledge acquisition community([Benjamins & Fensel 98]). In particular, Gomez-Perez and Rojas-Amaya describe a re-engineering process of retrieving and transforming a conceptual model of an existing ontology into a new one. The work of Benjamins and Fensel describes the *Knowledge Annotation Initiative of the Knowledge Acquisition Community* ontology(in short, KA^2), which models the knowledge acquisition community and forms the basis to annotate its documents on the web[2] in order to enable intelligent access.

2.3.7 Applications and projects

A complete listing of applications of ontologies is impossible. The literature references are huge and citing lengthy lists is not practical. However, we provide pointers to various resources in section 2.3.9 whereas here we selectively report the most representative ones. To do this effectively we cluster them according to their application domain.

We start with the area of *enterprise modelling*. In this area we found the *Enterprise* ontology([Uschold *et al.* 98c]), which captures the organisational structure of an enterprise with emphasis to activities and processes. The ontology is developed in a structured text form and a translation in *Ontolingua* is also available. In the same line is the TOVE ontologies set([EIL 95]) which shares the same aims with *Enterprise*, but has been developed in a formal computational form and uses different underlying principles([Fox & Gruninger 97]). The differences between these two representative ontologies for enterprise modelling are highlighted in [Uschold & Gruninger 96]. A relevant application area is that of *business process modelling*. The Process Inter-

[2] The ontology is accessible online from the following URL: http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.html

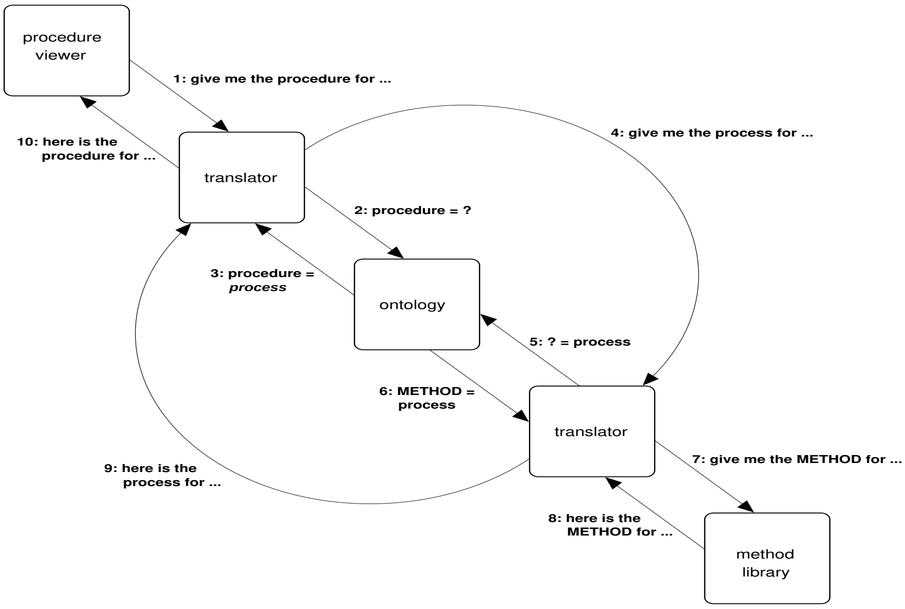


Figure 2.1: Interchange format example: the *procedure*, used by one tool is translated into the term, *method* used by the other via the ontology, whose term for the same underlying concept is *process*.

change Format(PIF)([Lee *et al.* 98]) is among the best known in this area. 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 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 [Lee & Malone 90]. The PIF framework has been applied in a *supply chain scenario*([Polyak 98]) which was adopted from the Workflow Management Coalition(WfMC)([WfMC 96]). An example of an interchange format is illustrated in figure 2.1.

Ontologies have also been applied to *medical applications*. For example, a methodology for integrating medical terminologies was presented in [Gangemi *et al.* 98]. This is the aim of the *ONIONS* methodology([Gangemi *et al.* 96]) developed by the same group. In the same context, the European project *GALEN*[3]([Rector *et al.* 95]) which

[3] The project is electronically accessible from the URL: http://www.ca.man.ac.uk/mig/galen

aims at capturing information from the clinical domain. In [Abernethy *et al.* 99], the authors present a system, called *Sophia* which acts as a knowledge server for web-based medical applications. An ontology for bioinformatics(*TAMBIS*) is presented in [Baker *et al.* 99]. Most of the applications in this area are based on terminological resources like the *GUM* ontology ([Bateman *et al.* 95]), the *CYC* ontology ([Lenat & Guha 90]), the *Unified Medical Language System(UMLS)* ([NLM'97 97]), etc.

Another area to which ontologies have been applied is that of *ontology-based brokering.* These are specifically designed agent systems which serve as brokers between heterogeneous systems. They use ontologies to facilitate the information brokering task. Representative applications are: the *Ontobroker*([Decker *et al.* 99]) which was used, among others, in the KA^2 project([Benjamins & Fensel 98]); the *onto2agent*([Aspirez *et al.* 98]) used to select publicly available ontologies on the web for a given application based on a *Reference Ontology* developed by the same group to classify candidate ontologies; the *OBSERVER*([Mena *et al.* 98]) system used to provide semantically rich information to a user who subscribes to an information management system on the web which supported by selected ontologies; the *IMPS(Internet-based Multi-agent Problem Solving)*([Crow & Shadbolt 99]) system which uses software agents to conduct knowledge acquisition on-line using distributed resources. Terminological ontologies(like *WordNet*) were used to underpin the whole process.

A related area of applications is that of *knowledge management* with emphasis on knowledge retrieval. A representative application in this area is the *PlanetOnto* which provides an integrated set of tools to support news publishing based on ontology-driven document enrichment([Domingue & Motta 00]). Two ontology-specific tools were developed: *Tadzebao* and *WebOnto*, both described in [Domingue 98]. The former aims to support a dialectical approach in ontology design and maintenance while the latter provides editing and browsing facilities. The goal of *Tadzebao* is to provide guidance for knowledge engineers around ongoing dialogues for designing ontologies. This can be used as a negotiation tool for proposed changes in an ontology with the additional flexibility that *Tadzebao* offers: the integration of discussion about an artefact and its implementation in the same visual metaphor. Another application in this area is the *knowledge-enhanced search* approach used in the *FindUR* project([McGuinness 98]). McGuinness describes a search tool, deployed at the AT&T research labs, which uses ontologies to improve the search experiences from the perspectives of recall and precision as well as ease of query formation. A similar approach which deploys *content matching* techniques is described in [Guarino *et al.* 99] where the authors present the *OntoSeek* system designed to support content-based access to the web.

A large area of applications of ontologies is that of *systems engineering*. In section 2.3.6 we already described systems like the *AIRCRAFT* and the *Boeing* experiment with the use of the *EngMath* ontology which was also used in the construction of the *PhysSys* ontologies set. Other representative applications are the *ATOS(Advanced Technology Operations System)*([Jones *et al.* 95]) system which was designed to meet specific needs of *spacecraft operations* such as the need for coordination of different agent applications who had to commit to a common ontology. In [Fillion & Menzel 96], the authors describe the *Integrated Development Support Environment(IDSE)*, a commercial computational environment that supports the integration of enterprise models. The integration is underpinned by axioms representing semantic constraints and relationships between different tools which are interpreted and enforced semi-automatically. This information is contained in a method ontology, the *IDEF1X*[4], accessed by a truth maintenance system that enforces rules and constraints defined in the method. There are other applications of ontologies in the area of systems engineering with emphasis in software design but these will be described in the next chapter where we further elaborate on the role of ontologies in software design.

There are also a number of applications related with projects undertaken by various organisations involving academic and industrial partners. We already mentioned some of them in the previous sections. We complete our coverage here by describing one of the first projects in this area which was the *Knowledge Sharing Effort(KSE)*([Neches *et al.* 91]) aimed to realise the benefit of sharing and reusing large knowledge bases. The distinguishable contribution of this project was the Knowledge Interchange Format(KIF) framework. Other projects are the *High Performance Knowledge Bases(HPKB)* programme ([Cohen *et al.* 98]) which aims at fostering the development of technologies

[4] Electronically accessible from the URL: http://www.idef.com/overviews/idef1.html

that can increase the rate at which we can write knowledge bases. The *Intelligent Brokering Service for Knowledge-Component Reuse on the World-Wide Web(IBROW^3)* project investigates means for supporting comprehensive reuse. The idea behind this project is to provide a brokering service that plays the role of a mediator between customers and PSM providers to support the configuration of customised knowledge systems that solve customers' problems. A library of reusable components([Motta *et al.* 99]) has been constructed based on the work of Motta in parametric design ([Motta 99]). The *Knowledge Reuse and Fusion/Transformation(KRAFT)* ([Preece *et al.* 99]) aimed to enable the sharing and reuse of information contained in heterogeneous databases and knowledge bases. In the area of planning the *SPAR*([Tate 98]) project draws on the range of previous work in planning activity ontologies to create a practically useful *Shared Planning and Activity Representation*.

2.3.8 Problems, tradeoffs, and solutions

Despite the fact that ontologies have been applied with success in a variety of fields there are reported problems and attempts have been made to identify tradeoffs and find potential solutions. We report on the problems first. In [O'Leary 97] the author discusses impediments in the use of ontologies. He points out the difficulty in library ontologies, scale-up, interfacing and raises the issue of formality in ontology development. O'Leary argues also for the difficulty in establishing a consensus: "ontologies are chosen after a political decision had been made, therefore it is impossible to choose an ontology that maximises the utility of all agents in process and the group."([O'Leary 97]). Other problematic areas have been identified: Uschold and colleagues raise the issue of lack of translators when the representation formalisms used are not the same in the context of their experiment for ontology reuse([Uschold *et al.* 98a]). They argued that, "the translation activity involved was an intensive one and lack of automatic support is an important disadvantage". The issue of ease of reuse was also the focal point of an empirical study performed in the context of the *HPKB* project. In [Cohen *et al.* 99], the authors report that ease of reuse is closely related to the type of ontology: it was found that very generic ontologies provide less support and are less useful than domain-specific ones. The latter scored a constant 60% rate of reuse in the

HPKB study in contrast with the poor 22% rate of reuse scored by broad ontologies. However, as the authors argue, these results should not undermine their role in structuring ontologies: "Although the rate of reuse of terms from very general ontologies may be significantly lower, the real advantage of these ontologies probably comes from helping knowledge engineers organise their knowledge bases along sound ontological lines."([Cohen *et al.* 99]).

Another important drawback is the lack of rigorous evaluation techniques for ontologies. For example, in an experiment of extending the *HPKB* upper ontology([Aitken 98]) the author states: "[...]validation remains an important issue, i.e.: the *PhysSys*, *EngMath* and *topology* ontologies are capable of being validated by reference to literature in their application fields[...] but ontologies such as the *HPKB* upper level and *SPAR* do not capture knowledge in such well understood fields, therefore this form of validation is not possible". We will revisit the evaluation of ontologies issue in chapter 3 where we review existing techniques and argue that the novel mechanism we present in chapter 4 can complement and enhance existing practices. The issue of maintenance has also been acknowledged and studied by many. Robertson neatly summarises the points made: "the cost of producing an ontology is not just in inventing the domain-specific formal language but in maintaining it once the system is deployed, since perfect ontologies cannot be guaranteed. Over-commitment to perfecting an ontology causes failure either during development(through irreconcilable arguments over what the ontology should be) or after deployment(through inappropriate human interpretation of inference system inputs or outputs)"([Robertson 98]).

However, there are ways to alleviate the situation and solve some of the problems mentioned above. For example, with respect to the problem of library ontologies, made by O'Leary, the online libraries of ontologies(i.e.: *Ontolingua*) are a potential solution especially when the maintenance and update facilities that are envisaged ([Fikes & Farquhar 99]) will be fully integrated. The issue of interfacing has attracted a lot of attention by the community. It is seen from different angles: 'integration', 'merging', 'mapping', etc. A summary of these approaches is given in [Pinto *et al.* 99] whereas Visser and colleagues analyse the nature of the problem in [Visser *et al.* 98]. Some of the solutions proposed and applied are the *ontological mediation algorithms*

([Campbell & Shapiro 98]), the *ontology clustering*([Visser & Tamma 99]), as well as the approaches used in projects like the creation of the *AIRCRAFT* ontology and the *Boeing* experiment([Uschold *et al.* 98b]). In addition to these, the *OBSERVER* ([Mena *et al.* 98]) and *ONIONS*([Gangemi *et al.* 98]) systems, the *Partially Shared Views(PSV)* scheme([Lee & Malone 90]), the *encapsulation and composition* technique ([Jannink *et al.* 98]) in the context of the *Scalable Knowledge Composition(SKC)*[5] project provide alternative solutions.

Even with this plethora of techniques the situation remains unsettled. There is no comparative analysis which identifies potential advantages and important drawbacks and no common practices to be followed. This has started to change with the proposal of frameworks that characterise ontologies, like the one originally presented by Uschold in [Uschold 98b] which was further analysed in [Uschold & Jasper 99]. These frameworks can be used to share experiences, discuss tradeoffs, and disseminate knowledge regarding attempts to apply ontologies. A small example of this is the instantiation of Uschold's framework, made by Kalfoglou and Robertson in the context of the *PhysSys* ontologies set([Kalfoglou & Robertson 00] and chapter 7). Another source of information is the comparative analyses. For example, in [Fridman-Noy & Hafner 97], the authors compare and analyse the state-of-the-art in ontology design. Ushcold and Jasper present a cost-benefit analysis of three commonly used approaches in knowledge sharing([Uschold *et al.* 99]). In a larger context, Kalfoglou and colleagues, compare various meta-knowledge types, analyse their cost-benefits, and identify pragmatic aspects in using meta-knowledge([Kalfoglou *et al.* 00a]). In similar fashion Menzies and colleagues analyse issues with meta-knowledge in [Menzies *et al.* 00] and Kalfoglou speculates on the role of formal ontologies in knowledge maintenance([Kalfoglou 99]).

2.3.9 Resources

As we stated earlier, an exhaustive review of the ontologies field is impractical and overwhelming for the reader. However, for the sake of disseminating up-to-date information on ontologies we have selected and include here pointers to publicly available online resources. These are:

[5] Electronically accessible from the URL: http://www-db.stanford.edu/SKC/

- a comprehensive collection of ontology-related research in alphabetical order, maintained regularly by Peter Clark:
  http://www.cs.utexas.edu/users/mfkb/related.html
- a similar collection maintained by Enrico Franconi:
  http://www.cs.man.ac.uk/franconi/ontology.html
- a list maintained by Adam Farquhar:
  http://ksl-web.stanford.edu/kst/ontology-sources.html
- a catalogue with classified information on ontologies prepared by Yannis Kalfoglou for a panel debate that took place in the SEKE'99 conference:
  http://www.dai.ed.ac.uk/daidb/people/homes/yannisk/seke99panelhtml.html

In addition to these periodically updated online resources there are several overviews in the literature. These are, the Uschold and Gruninger review ([Uschold & Gruninger 96]), the comparative review of Fridman-Noy and Hafner([Fridman-Noy & Hafner 97]), the survey of ontology research([Chandrasekaran *et al.* 99]), an overview of ontologies and PSMs([Gomez-Perez & Benjamins 99]), and a review of planning ontologies([Tate 98]). There are also special issues in referred journals devoted to ontology research: with respect to their role in IT([Guarino & Poli 95]), their involvement in KBSs([vanHeijst *et al.* 97]), and their uses([Uschold & Tate 98]). In addition, we should mention the volume edited by Guarino in [Guarino 98b].

2.4 *Chapter summary*

In this chapter we provided a review of two main areas used in our research: testing and ontologies. In particular, we reviewed the field of software testing and explored contributions made by the SE and AI communities. We highlighted the weaknesses of SE methods, namely, that the majority of software testing research is focussed on how to perfect the method used rather than broadening the types of errors to be captured. Although, a perfect method is a powerful tool to automate the process the limited typology hides an important caveat: a type of errors that are related to misinterpretations of domain knowledge, which we call *conceptual errors*, can be very damaging for the system to be developed if they remain undetected. The AI community has offered some insights in this problem since it is concerned with modelling domain knowledge. However, even in this community, we didn't found any particular method that deploys domain knowledge to detect these sort of errors. The closest approach used is, probably, the work on verification and validation of knowledge-based systems. We then surveyed the foundation of our work: the field of ontologies. We reviewed all the aspects involved in ontology engineering: design issues, deployment efforts, applications, problems, tradeoffs and proposed solutions. We didn't refer explicitly to an area that there is little discussion of, so far, in the community: evaluation of ontologies. This area will be scrutinised in the next chapter where we introduce our idea in deploying ontologies in software design.

Chapter 3

Domain knowledge in systems design

In this chapter we concentrate on the early phases of design and mention various attempts that have been made to engineer those phases. These are described in section 3.1 whereas in section 3.2 we elaborate on a specific type of error that occurs during early design and often remains undetected, the conceptual errors. We present a motivating example of a conceptual error before proceeding to analyse the role of domain knowledge in its detection in section 3.3. This section introduces the idea of deploying ontologies in early design to alleviate the problem of conceptual error occurrences.

3.1 Early phases of systems design

The early phases of design are often fraught with uncertainty which makes them difficult to engineer. In recent years we have witnessed a growing interest in engineering the requirements that have to be met by the system to be built. Usually, these are relevant to the system's potential users[1] and we, as system developers, use them to build the system that meet their needs. Although there are still a lot of open issues in requirements engineering([Zave & Jackson 97]), we have seen practice guides([Sommerville & Sawyer 97]) emerge and methodological approaches (CORE - [Mullery 79]) in use. However, the problem is not in engineering these requirements

[1] We use the term 'users' in a broad sense: we don't explicitly mean human users, they might be other software systems or requirements posed by the environment(i.e., when we update a system to follow the state-of-the-art).

and methodologically transforming them to computational artifacts, such as specifications, in order to start the system development processes. Neither is the problem in the technology we use: various modelling approaches(e.g., *conceptual modelling*) and techniques(e.g., the 'viewpoints' literature ([SEJ96])) have emerged and been used as well as tools that help us to automate the process of engineering the requirements.

The problem is in the domain. Requirements engineering people often refer to this as the system's "environment"[2]. In [Sommerville & Sawyer 97], the authors argue that to help understand the requirements, you should develop one or more models of the system's environment. These are models of the context in which the system is used. But modelling the environment is not an easy task. Sommerville and Sawyer explain why:

> "The environment of a system is usually very complex. There are often subtle, implicit relationships between different parts of that environment. It can be difficult to decide whether or not parts of the environment are relevant for the system being developed. Decisions about the system's environment may not be finalised and the environmental information may be unstable."([Sommerville & Sawyer 97])

This is the source of the uncertainty. Normally, we have to make various assumptions about the environment, or in a broader sense, the domain. Then, we need to make these assumptions explicit and verify their correctness. Here is the crux of the problem: against what are we going to verify their correctness? Since there are few well-defined, consensual, and transferable domain models, we centre our correction activities upon the mathematical formalisms used to represent these assumptions.

There have been several efforts to perform this sort of correctness check. They apply to both informal and formal models. We do not offer an exhaustive list of contributions here but we mention two indicative examples. On the informal side, Finkelstein presents a scheme for reviewing and correcting informal specifications([Finkelstein 92]). The scheme deploys marking techniques to annotate the textual specification to sup-

[2] In [Zave & Jackson 97], 'environment' is defined as: "the portion of the real world relevant to the software development project".

port the process of correction. On the formal side, Fuchs and Robertson argue for the executability of formal specifications written in logic([Fuchs & Robertson 96]), which gives us not only a conceptual, but also a behavioural model of the system to be implemented. This helps in observing undesired behaviour and doing the appropriate modifications before the actual system exists.

These approaches give us a way to verify that we modelled the assumptions correctly. We can also prove that they are mathematically elegant and methodologically transform them to encoding bits to be included in later phases of system development. However, they don't tell us anything about whether we made the correct assumptions with respect to the domain. This is where the conceptual errors emerge. They represent a misunderstanding of the domain, but they are mathematically elegant and thus, cannot be detected easily. In the next section we describe such an error.

3.2 Conceptual errors

To make clear the notion of conceptual error, we present an artificially-defined example which we introduced in the context of the air campaign planning domain. The whole case is described in detail in section 5.4. Here we briefly present the error, elaborate on its implications and argue for the difficulty of its detection.

The scenario is as follows: we have a prototype system designed to protect an allied vessel from hostile aircraft attack. The system detects the aircraft which enters the allied vessel's airspace and determines whether or not it is a threat. If the detected aircraft is determined to be a threat the appropriate weapon is fired and the airplane is shot down. The system developers have define the following rule which determines whether an aircraft should be regarded as a navy threat:

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

The above rule classifies an aircraft as a navy threat according to the nature of the mission it performs. Whenever this is a combat mission the detected aircraft will be regarded as a navy threat.

Although the above rule is mathematically elegant and we can check its correctness with respect to types it represents a conceptual error. Here's why: an aircraft that carries anti-aircraft and anti-ground weapons will also be treated as navy threat even if it carries no anti-naval weapons.

For example, when an aircraft of type F-117 enters the allied vessel's airspace, it can be detected by the system and shot down because it is capable of performing combat missions. But it is not carrying weapons that can harm a naval vessel. This may have pernicious side-effects in a real defence system because it causes loss of defensive weaponry, lack of concentration to real navy threats, etc.

The error occurred because the system developers under-defined the notion of navy threat. They made a wrong assumption with respect to the domain and the environment that the defence system is used. In order to correct this error, the correct assumption has to be made. For example, the following rule enriches the definition of navy threat and solves the problem:

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

This rule states that a detected aircraft is considered a navy threat when it stores weapons that target on naval units. It does not focus on the nature of the mission that an aircraft can perform, as the previous rule did. It is more specific and realistic since it is focussed on the threat that can harm a vessel: anti-naval weapons targeting it. In our example with the F-117 aircraft, the system will ignore it as it does not carry weapons that target naval units. But where does this knowledge concerning the correct formulation of rules for determining navy threats come from? How can we detect this sort of error? The answer to the former question is given in the next section where we examine the role of domain knowledge which is used to answer the latter question as we demonstrate in section 5.4.

3.3 The role of domain knowledge

Before explaining how domain knowledge is being deployed we shall describe what a domain is. This word has several meanings. As Jackson points out: "[...] the domain of a mathematical function is the set of argument values for which the function defines result values. For a practitioner of domain analysis, a domain is a general class of system for an application area such as resource management, or airline reservations, or banking, or production control"([Jackson 95]). In our work we consider the following definition of domain given by Shlaer and Mellor:

> "In building a typical large software system, the analyst generally has to deal with a number of distinctly subject matters, or *domains*. Each domain can be thought of as a separate world inhabited by its own conceptual entities, or objects."([Shlaer & Mellor 92])

Therefore, a domain is a particular part of the world that can be distinguished because it is conveniently considered as a whole, and can be considered separately from the other parts of the world. Once we have identified a domain we have to acquire information about it. This is an acknowledged need for the early phases of system design as Zave and Jackson point out:

> "[...] we are free to collect and record interesting information about the environment even before we are sure it will be needed. This freedom is clearly necessary for collecting libraries of reusable information about important portions of real world."([Zave & Jackson 97])

Recall from the previous chapter where we defined ontologies as explicit representations of some domain of interest(section 2.3.1). These are the "libraries of reusable information about important portions of real world" that Zave and Jackson call for. Knowledge regarding a domain can be captured in an ontology(see section 2.3.5). Thus, ontologies are computational forms of domain knowledge and theories that tell us how to appropriately describe a domain. How can such knowledge be of help to our problem of the erroneous assumption regarding the correct definition of navy threat introduced in the previous section? If we had at our disposal a well engineered computational form of the military domain and especially the aircraft and ordnance types used we could have solved the problem. That is because we could use that knowledge in order to make the correct assumptions with respect to the aircraft types domain. In fact, such knowledge already exists in a real ontology. This is the AIRCRAFT ontology which captures knowledge from publicly available resources, principally the US Air Force and US Navy Fact Sheets, with respect to military aircraft types. As we describe in chapter 5, section 5.4, it is possible to deploy this kind of knowledge in order to help the designers of the defence system make the correct assumptions.

This way of applying ontologies has been identified in the review of Uschold and Gruninger as beneficial for system engineering([Uschold & Gruninger 96]). It is referred to as the reliability benefit where the formal representation of domain knowledge enables the use of consistency checking resulting in more reliable systems. It is not only the knowledge sharing benefit that we achieve by consulting the AIRCRAFT ontology for aircraft and ordnance types. It is also the reliability benefit which interests us and we can achieve this by deploying the AIRCRAFT ontology in a way which enables us to check the correctness of assumptions with respect to aircraft and ordnance types. We describe how this can be done in chapters 4 and 5 while here we mention the few efforts found in the literature for achieving the reliability benefit ontologies can provide. We summarise these below.

One of the early contributions was that of the `Comet`([Mark *et al.* 92]) and `Cosmos` ([Mark *et al.* 94]) systems. Both systems aim at developing knowledge bases by capturing the set of ontological commitments that define the interdependencies among key terms in the ontology. Their role is to assess the impact of changes in their world and provide context-specific guidance to their users on what modules may be relevant to include in the design, and what design modifications will be required in order to include them([Mark *et al.* 95]). The key idea behind this work was to make use of the ontological commitment expressed by the underlying ontology in the system's development process. However, the issue of where these commitments come from and, most importantly, how they are verified was not addressed.

In the context of the TOVE project([EIL 95]), ontological commitments played a key role in the definition of an ontology's competence: a set of queries that the ontology can answer. These questions, called competency questions, were used to evaluate the expressiveness of the ontology that is required to represent them and characterise their solutions([Gruninger & Fox 95]). They do not generate ontological commitments but are used to evaluate them. The TOVE ontologies are built on top of foundational theories such as situation calculus. Foundational theories provide the semantics for the ontology and their axioms serve as a basis for the implementation of competency questions. Despite the fact that foundational theories were used to reason about the terminology of the ontologies using their axiomatisation they are generic and were only applied within the scope of ontologies development. This does not contribute to the problem of how to evaluate ontologies when applying them as Fox and Gruninger pointed out in [Fox & Gruninger 97]: "[...] by providing formal definitions of objects and their relations and attributes, we make it possible for users to understand their intended meaning. Though this does not guarantee that programs that access an ontology will interpret the results correctly".

In the **DISCOVER** project([Waterson & Preece 99]), the role of ontological commitment was further analysed and operationalised. The authors state that ontological commitment is a key issue for knowledge sharing and reuse and they applied existing verification techniques from the KBSs literature to check the commitment of a knowledge base to an ontology. In that project the role of the ontology was to act as a background body of knowledge against which a knowledge base can be validated.

In the area of evaluation of ontologies Gomez-Perez investigates criteria to verify knowledge sharing technology([Gomez-Perez 96]). The importance of evaluation is stressed by the same author: "[...] it is unwise to implement a software application that relies on ontologies written by others without first evaluating and assessing their definitions and axioms"([Gomez-Perez 95]). She provides guidelines on how to prove consistency of an ontological definition by proving that the definition is internally and metaphysically consistent. Internal consistency requires the informal and formal definitions of an ontology to have the same meaning. The most interesting case of consistency is the metaphysical one which is defined by Gomez-Perez as: "proving that there is no contradiction in the interpretation of the formal definition with respect to the real world." That is, to prove compliance of the world model with the world modelled formally. It is claimed that this sort of consistency is difficult to enforce in a mechanised manner so the model is checked manually by using the ontological commitments explicitly made by the ontology.

Even with these ways of deploying ontologies we do not solve the problem of conceptual error occurrences. In the next chapter we present a generic representation formalism for ontological axioms to enable their use in the multi-layered architecture discussed later in the same chapter. This can solve the problem with conceptual error occurrences and alleviate the situation with some of the problems discussed above: the partial verification of ontological commitments recognised by Mark and colleagues([Mark *et al.* 95]) can be done by regarding them as a layer in the layering structure(section 4.6) we propose which is then checked exhaustively against meta-layer constraints; the problem of guaranteeing the correct interpretation of formal definitions by programs that access an ontology faced by Fox and Gruninger can be explored by transforming the ontological axioms to the constraints format we adopt(section 4.1) and placing them along with the programs in the layering structure to enable automatic checking of a program's conformance to the axiomatisation. The metaphysical consistency checking described by Gomez-Perez is impossible to implement in a fully automated way. However, the layering approach we propose may help engineers to judge the correctness of their models with respect to meta-models in an integrated environment(section 4.5).

To visualise the scope of our research and relate it with previous work we have produce the motivating figure 3.1, shown in page 47. The left and middle parts are taken directly from the literature, namely on ontologies, and verification and validation of KBSs respectively. The right part is a visualisation of our research as a result of combining the other two parts.

The left part is taken from the work of Valente and Breuker([Valente & Breuker 96]) on core ontologies. It shows the interrelations between different elements in the ontology domain. As the authors describe: "Conceptualisations and domains are abstract things (represented by irregular polygons) while knowledge representations and ontologies are concrete things (represented by rectangles), for example, sentences in some symbolic language. Their interrelations are represented as ovals, and the direction of reading is indicated by the direction of the arrows. These interrelations are as follows: (i) an ontology describes a conceptualisation using definitions of the elements of the conceptualisation (objects, concepts, relations); (ii) a conceptualisation provides ontological commitments which are used in elaborating knowledge representations, and these commitments are embedded either in the knowledge representation language, or in the knowledge base; (iii) a knowledge representation represents a domain (a part of the world).".

The middle part is taken from the work of Meseguer and Preece on the use of formal specifications in the KBS development([Meseguer & Preece 95]). The description is as follows: "[...] the informal description - typically called the 'conceptual model' in various methodologies - is iteratively refined to create a more precise formal specification, with both descriptions undergoing gradual modification during the process. The formal specification keeps the structure and vocabulary of the conceptual model. It is kept to make easier the communication with domain experts in the development and validation process. The formal description will form the basis for the implementation.".

We combine these two parts to produce a single diagrammatic view of our research. As we can see from the right part, the role of the ontology has not changed: it describes a conceptualisation using definitions of the elements of that conceptualisation. This corresponds to the "conceptual model" of Meseguer and Preece's diagram. It provides ontological commitments which are embedded in the knowledge base. It can also be iteratively refined to a formal specification as Meseguer and Preece describe in their diagram. In addition, our research allows for verification of these commitments in the formal specification. As in Meseguer and Preece's diagram it forms the basis for the implementation of a KBS which represents a domain according to Valente and Breuker's diagram. That is because the whole process of constructing the initial "conceptual model" started from a given ontology, which by definition captures domain knowledge. The dashed line surrounding the elements(ontology, conceptualisation, ontological commitments, formal specification and KBS) denotes the scope of our multi-layer architecture(section 4.6). By using this mechanism we are able to verify the ontological commitment in various phases of ontology deployment: in the ontology itself by checking the correctness of its definitions against the ontological axioms, down to the formal specification of the system to be implemented by verifying its conformance to the underlying ontology. This will result in a system that includes domain knowledge where verification methods have raised our confidence that it has been represented appropriately.

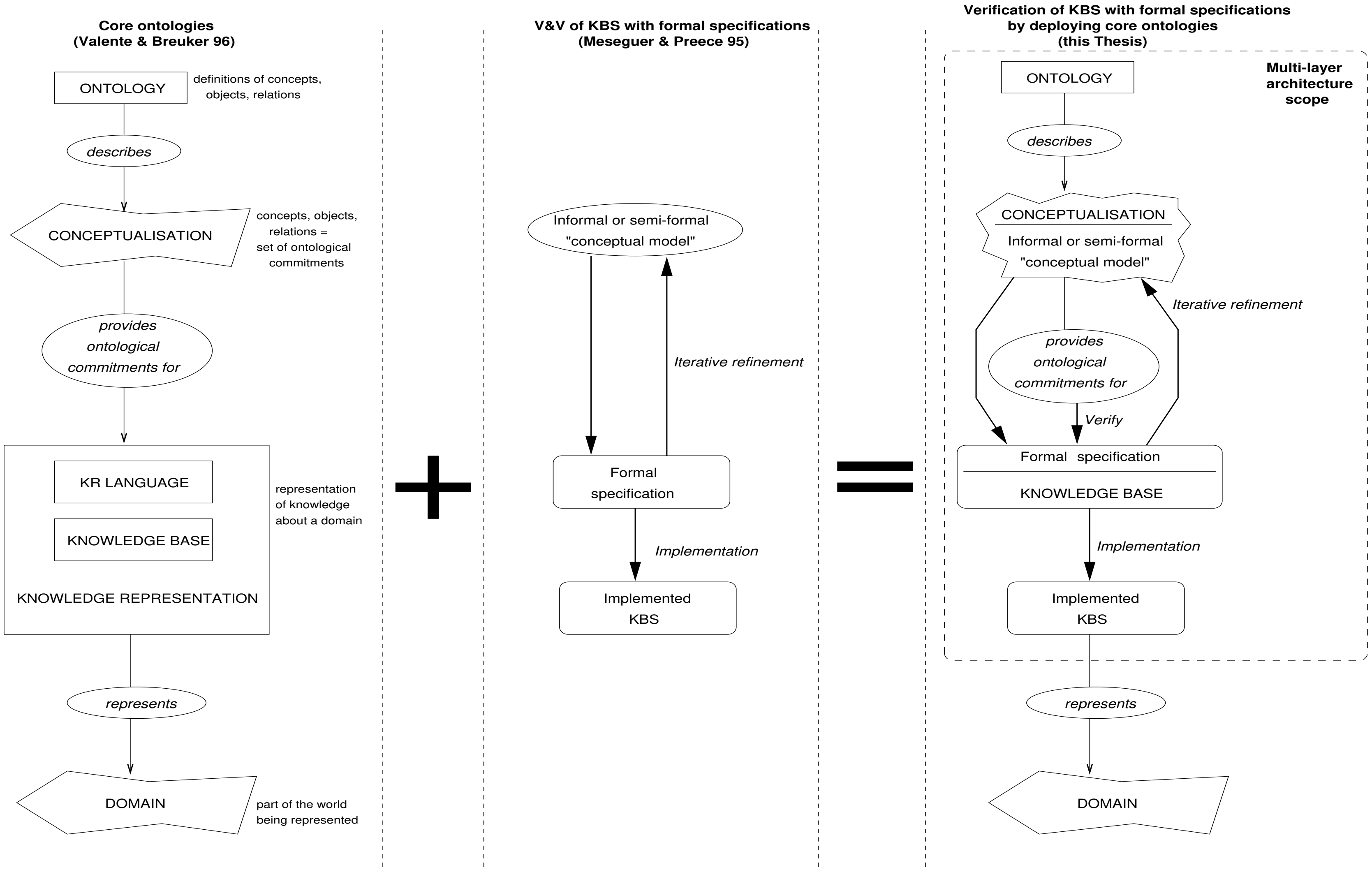


Figure 3.1: Verification of KBSs with formal specifications by deploying core ontologies.

3.4 *Chapter summary*

In this chapter we argued for the importance of the role that domain knowledge plays in early system design. We have seen evidence for this from the requirements engineering community, an area traditionally focussed on the early phases of design. To stress the difficulty of detecting errors in these phases we presented a motivating example of conceptual error. We advocated that computational forms of domain knowledge(i.e., ontologies), can alleviate the situation and detect this sort of error. We presented similar efforts, made by the ontology community, in this direction but we highlighted the weaknesses of these approaches. We argued that some of these weaknesses can be eliminated with our way of deploying ontologies, presented in the next chapter. We also presented a diagrammatic version of the scope of this research in relation with the ontologies and V&V literature. We argued that the use of formal specifications in the development of KBS, and the deployment of core ontologies have many similarities which we bring together in our research. In the next chapter we present the theoretical foundations of our research before proceed to chapters 5 and 6 where we present the implementations.

Chapter 4

Deploying domain knowledge

This chapter clarifies the core of our research. We provide the theoretical foundations for implementing our approach in deploying domain knowledge as explicitly represented in an ontology. In particular, we present a formal representation of ontological axioms in the constraints format we adopt(section 4.1), followed by the theoretical foundations of conceptual errors detection in section 4.2. To execute the error detection we employ a generic inference engine(section 4.3) and an error detection mechanism which we specify in section 4.4. These are the basis for a detection architecture(section 4.5) which is extended in section 4.6 to a multi-layer variant. This makes it possible to test for correctness the ontologies themselves, as we describe in chapter 6. The implementations of these theories are given in the form of Prolog programs in section 4.7 whereas in section 4.8 we discuss potential limitations of the approach.

4.1 Ontological constraints

The role of ontological axioms is to: "constrain the possible interpretations for the defined terms"([Gruber 93]). Hence, we call them *ontological constraints.* We describe below how we formally represent them in a format tailored to detect conceptual errors.

They are derived from ontological axioms which adopt the following formal representation:

(α) $\forall X1, ..., Xn.G \rightarrow C$

where G is a unit goal and $X1, ...Xn$ are all variables in G, and C is a condition composed of logical connectives($\wedge, \vee, \neg$) and/or unit goals. The condition C must be composed of valid ontological constructs and it must be true when the unit goal G is true.

So, for example, an axiom of a formal ontology, the Process Interchange Format(PIF) ontology([Lee *et al.* 98]) given in textual format:

> "An object can *participate in* an activity only at those timepoints at which both the object exists and the activity is occurring"

can be written according to formula (α) in predicate calculus as follows:

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

The role of this axiom is to restrict possible interpretations of the ontologically defined relation *participates_in*. So, whenever someone using the PIF ontology describes the relation in a way that does not conform to the axiomatised definition this will reveal a potential discrepancy.

In cases where the unit goal G is a composite one we transform it to a monadic literal since we are interested in having monadic predicates as the preconditions of axioms. For instance, in the CHEMICALS ontology([Aguado *et al.* 98]) the axiomatised definition of the melting point for halogens given in textual format:

> "The highest melting point for halogens is 302C"

is written in predicate calculus, taken directly from [Aguado *et al.* 98], as follows:

$$\forall H, M.halogen(H) \wedge melting\_point(H, M) \rightarrow M \leq 302 * degree\_celsius.$$

We apply the following rewrite rules:

$A \wedge B \rightarrow C \Rightarrow A \rightarrow (B \rightarrow C) \Rightarrow A \rightarrow \neg(B \wedge \neg C)$

to break the conjunction in the precondition. Note that $A \wedge B$ is symmetrical which means that we could have as a result of the above rule, B in the precondition instead of A. The '$\Rightarrow$' operator used to denote the rewrite of the original part to the transformed part. The axiom is now written, according to formula (α), as follows:

$\forall H, M.halogen(H) \rightarrow \neg(melting\_point(H, M) \wedge \neg(M \leq 302 * degree\_celsius))$.

When we check for conceptual error occurrences, we are interested in proofs over existentially quantified goals, so the formula (α) is automatically transformed via a tool described in section 4.7 into a conjunctive normal form:

$$(\beta) \quad \forall X1, ..., Xn.G \rightarrow C \Rightarrow \neg\exists X1, ...Xn.G \wedge \neg C$$

We then identify the predicate G derived from the left hand side of the original implication of formula (β) and lose the existential quantifier and outer negation. These two expressions will be used to test for errors on goals in the proofs as we describe in section 4.7 and we define the clause `error/2` to represent them. The transformation from the right part of formula (β) will be:

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

So, for the ontological axiom on the melting point of halogens given above a translation to normal form will be:

$$(\delta)\ \forall H, M.halogen(H) \rightarrow \neg(melting\_point(H, M) \wedge \neg(M \leq 302 * degree\_celsius))$$
$$\Rightarrow \neg\exists H, M.halogen(H) \wedge (melting\_point(H, M) \wedge \neg(M \leq 302 * degree\_celsius)).$$

As in formula (γ), we automatically transform this via a tool described in section 4.7 formula (δ) to the specific error format:

$$(\epsilon)\ error(halogen(H), (melting\_point(H, M) \wedge \neg(M \leq 302 * degree\_celsius))).$$

To use this error condition, which we call an ontological constraint, we do the following: whenever this condition is satisfied with respect to the program that uses the CHEMICALS ontology we have a conceptual error occurrence. That is, we have an error for a halogen H, we can prove that its melting point M, is not less than or equal to 302 degrees as it must be.

In the same manner, for the PIF axiom on the definition of the *participates_in* relation given above, we derive its ontological constraint in the form of an error condition:

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

This condition can be given a declarative reading: we have an error occurrence with respect to the *participates_in* relation used in a program that adopts the PIF ontology, whenever we can prove that an object O participates in an activity A at a timepoint T, and it is not provable that object O exists at activity A, occurring at timepoint T.

4.2 Error detection theory

In order to reason about the conceptual errors found in programs that adopt ontologies and the correctness of ontological constraints(in the error condition format described above) we deployed to detect them, we developed a logical theory. In this theory we have the following standard constructs which we assume to be constants in the expressions which follow:

Let Gs be the set of all goals in the program(i.e., a specification) defined as follows:

$Gs = \{G1, ..., Gn\}$

Let Es be the abstract set of all possible errors on goals in the specification. This set refers to all possible violations of the ontological constructs used in the specification. It is impossible to know beforehand what these might be, but we represent them as: $e(G,E)$ where E is an error for the goal G. Every goal in the specification can have an arbitrary number of subgoals and their associated errors.

We define the clause: $error(G,E)$ to denote that E is an error condition defined on goal, G. This is the specific error condition format introduced in the previous section, and it represents an ontological constraint defined over the goal G.

We then identify three cases in the reasoning process: the first one refers to situations where no conceptual errors are found in the specification; the second cover cases where we found conceptual errors; and the third one is an elaboration of the two previous cases: we found conceptual errors but they might be erroneously reported depending on the correctness of error conditions(i.e., ontological constraints) and/or specification. Every case is textually described and defined formally in predicate calculus.

4.2.1 Case A: no conceptual errors found

Textual description:

We found no conceptual errors. We cannot prove - via the inference engine described in section 4.3 - the error condition(s)(even the negative ones) defined on goal(s) given for proof.

Definition A:

There does not exist an error condition, E, defined on the given goal, G, such that E is provable:

$\forall G, E.\ G \in Gs \rightarrow (\neg(error(G,E) \wedge provable(E)) \rightarrow e(G,E) \notin Es)$

4.2.2 Case B: conceptual errors found

Textual description:

We found conceptual errors. We can prove - via the inference engine described in section 4.3 - the error condition(s) defined on goal(s) given for proof.

Definition B:

There exists an error condition, E, defined on the given goal, G, such that E is provable:

$\forall G, E.\ G \in Gs \rightarrow ((error(G,E) \wedge provable(E)) \rightarrow e(G,E) \in Es)$

4.2.3 Case C: conceptual errors found but maybe erroneously reported

Textual description:

We found conceptual errors. We can prove - via the inference engine described in section

4.3 - the error condition(s) defined on the given goal(s). However, these condition(s) might be erroneously defined or their dependent goals might be erroneous. Therefore we identify two cases: (a)we found conceptual errors that are erroneously reported because of an erroneous error condition and, (b)we found conceptual errors and erroneous error conditions.

From cases A and B we infer the consequences given below. In appendix A.1 we describe how we obtain these logical consequences.

Consequence A: (erroneous error condition)

1. There exists an error condition, $E1$, defined on the given goal, G, which is not provable, but there exists an error condition, $E2$, defined on that condition, $E1$, such that $E2$ is provable:

   $$\begin{aligned}&\forall G, E1, E2.\ G, E1 \in Gs \rightarrow\\ &\qquad (\neg\ (error(G,E1) \wedge provable(E1)) \wedge (error(E1,E2) \wedge provable(E2)) \rightarrow\\ &\qquad e(G,E1) \notin Es \wedge e(E1,E2) \in Es)\end{aligned}$$

2. There exists a negative error condition, $\neg E1$, defined on the given goal, G, which is not provable, but there exists an error condition, $E2$, defined on condition, $E1$, such that $E2$ is provable:

   $$\begin{aligned}&\forall G, E1, E2.\ \ G, E1 \in Gs \rightarrow\\ &\qquad (\neg\ (error(G,\neg E1) \wedge provable(\neg E1)) \wedge (error(E1,E2) \wedge provable(E2)) \rightarrow\\ &\qquad e(G,\neg E1) \notin Es \wedge e(E1,E2) \in Es)\end{aligned}$$

Consequence B: (erroneous error condition and conceptual error reported)

1. There exists an error condition, $E1$, defined on the given goal, G, which is provable, and there does not exist an error condition, $E2$, defined on that condition, $E1$, such that $E2$ is provable:

   $$\begin{aligned}&\forall G, E1, E2.\ G, E1 \in Gs \rightarrow\\ &\qquad ((error(G,E1) \wedge provable(E1)) \wedge \neg\ (error(E1,E2) \wedge provable(E2)) \rightarrow\\ &\qquad e(G,E1) \in Es \wedge e(E1,E2) \notin Es)\end{aligned}$$

2. There exists an error condition, $E1$, defined on the given goal, G, which is provable, but there also exists an error condition, $E2$, defined on that condition, $E1$, such that $E2$ is also provable:

$$\forall G, E1, E2.\ \ G, E1 \in Gs \rightarrow$$
$$((error(G, E1) \wedge provable(E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$$
$$e(G, E1) \in Es \wedge e(E1, E2) \in Es)$$

In this case, consequences A1 and A2 refer to erroneous error conditions. As a consequence, the conceptual errors found based on these conditions are erroneously reported. These conditions are erroneously defined ontological constraints - derived from ontological axioms as we described in the previous section - and their detection relies on the provision of meta-axioms, expressed as additional error conditions in the consequences. Similarly, consequences B1 and B2 refer to erroneous error conditions and reported conceptual errors. Their detection also relies on the availability of meta-level axioms. In section 4.6 we present the multi-layer architecture which makes it possible to integrate layers and meta-layers. In chapter 7 we present examples of this case where we elaborate on its implementation which is included in appendix C.1.

Now that we have defined a representation formalism for ontological axioms and a theory to reason about conceptual errors based on those axioms, we need an inference engine that allows us to implement those checks in an automated way. In the next section we describe such an engine.

4.3 Inference engine

Our aim in developing an inference engine to allow mechanised reasoning about conceptual errors in programs was to have the engine integrated in the programming environment where the programs to be checked are implemented. Moreover we were interested in an engine that will have clearly defined semantics based on computational logic; will be simple to use and easy to augment. As we describe in section 4.7 these augmentations give us the ability to perform the conceptual error checking. A technique from logic programming satisfies these requirements, that of meta-interpretation.

While we give the implementation details in section 4.7 we describe here the inference engine as a logic program. A common inference strategy in trying to establish truth when proving goals is goal reduction. This is made explicit through meta-interpretation based on the standard 'vanilla' model[1]. The predicate $provable/1$ introduced in section 4.2 is used as follows: $provable(Goal)$ is true if $Goal$ is true with respect to the program being interpreted. The inference engine is given in Horn Clause notation:

(1) $provable((A \wedge B)) \leftarrow provable(A) \wedge provable(B).$

(2) $provable((A \vee B)) \leftarrow provable(A).$

(3) $provable((A \vee B)) \leftarrow provable(B).$

(4) $provable(\neg X) \leftarrow \neg provable(X).$

(5) $provable(X) \leftarrow meta(X, M) \wedge provable(M).$

(6) $provable(X) \leftarrow builtin(X) \wedge X.$

(7) $provable(X) \leftarrow clause(X, B) \wedge provable(B).$

The interpreter has the following declarative reading: line (1) states that a conjunction of goals $A \wedge B$ is true when both A and B is true. Lines (2) and (3) state that a disjunction $A \vee B$ is true when either A or B is true. Line (4) deals with negative goals; it states that $\neg X$ is assumed true if we cannot prove X(so we are using closed world negation). In lines (5) and (6) specific idioms of the implementation language are treated; if X is a meta logical expression(such as $setof(E, p(E), S)$) then X is true if after applying successfully the relation $meta(X, M)$ to obtain a new goal M, M is provable(in the case of $setof$ the meta definition is $meta(setof(E, G, S), setof(E, provable(G), S))$). If X is a built-in expression then it is always true. In line (7) the strategy of goal reduction is realised: goal X is true if there is a clause $X \leftarrow B$ in the interpreted program such that B is true.

The interpreter defined above describes the way of executing a Horn Clause specification. In the next section we show how this can be overlaid with an error checking mechanism.

[1] Described in [Sterling & Shapiro 94].

4.4 Error checking mechanism

We specify an error checking mechanism tailored to detect conceptual errors. It interprets the inference mechanism described above(via clause (C) below) and uses ontological constraints of the form given in section 4.2. The mechanism is given in Horn Clause notation:

(A) $onto\_prove((X1 \wedge X2), E) \leftarrow onto\_prove(X1, E1) \wedge onto\_prove(X2, E2) \wedge$
$$E = E1 \cup E2.$$
(B) $onto\_prove(\neg X, []) \leftarrow \neg provable(X).$

(C) $onto\_prove(X, E) \leftarrow clause(provable(X), B) \wedge onto\_prove(B, Eb) \wedge$
$$error\_check(X, Ex) \wedge E = Ex \cup Eb.$$
(D) $onto\_prove(X, []) \leftarrow system\_call(X) \wedge X.$

(E) $error\_check(X, S) \leftarrow setof(errors\_found(X, E, Es), conceptual\_error(X, E, Es), S).$

(F) $error\_check(X, []) \leftarrow \neg conceptual\_error(X, \_, \_).$

(G) $conceptual\_error(X, E, Es) \leftarrow error(X, E) \wedge copy\_term(E, E1) \wedge$
$$onto\_prove(E1, Es).$$

We use the following predicates in the mechanism:

- $onto\_prove(Goal, Errors)$ to denote that in trying to prove a goal, $Goal$, we discovered conceptual errors, $Errors$;
- $error\_check(Goal, Set)$ to denote that the error check we performed on goal, $Goal$, yield the set of conceptual errors, Set;
- $conceptual\_error(Goal, Error, Errors)$ to denote that we found a conceptual error, $Error$, and its dependent errors, $Errors$, in trying to prove goal, $Goal$.

The error checking mechanism can be given a declarative reading: line (A) states that a conjunction of goals $X1 \wedge X2$ will yield error, E, if we can find error $E1$ in trying to prove $X1$, and error $E2$ in trying to prove $X2$, and E is the union of $E1$ and $E2$. Line (B) states that we get no errors(identified from the empty list, []) on a negated goal X when we cannot prove it via the inference engine. Line (C) realises the error checking strategy: a goal X will yield error E if in applying the inference engine - which realises the goal reduction strategy - to prove goal X, we can find a new error Eb on its subgoal B; and we can perform an error check on goal X to yield error Ex; and E is the union of Ex and Eb. In line (D) we define the predicate $system\_call(X)$ to denote that we get no errors on goal X when it is a solution given directly by the implementation language.

Lines (E) and (F) implement the error checking: in (E) we declare that an error check on goal X will yield a set of errors S if we can find conceptual errors E and its dependent errors Es on X and set S is composed of all occurrences of E and Es represented as the template $errors\_found(X, E, Es)$. This is an implementation of a standard predicate given as built-in in several Prolog implementations. For example, in SICStus Prolog, the predicate $setof(Template, Goal, Set)$ has the following meaning: "Set is the set of all instances of $Template$ such that $Goal$ is satisfied, where that set is non-empty"([SICStus 95]). In (F) we declare that we get no errors on goal X if there is no conceptual error on that goal.

Line (G) implements the interface to ontological constraints. It states that there exists a conceptual error E and its dependent errors Es on goal X, if an ontological constraint on goal X is satisfied identifying the error E, and we can find its dependent errors Es regarding $E1$ as a new goal to check which is a replica of E except that all its variables are new. We need this replication to check the specification exhaustively for all possible instantiations of the given goal. The replication is implemented as a standard built-in predicate, $copy\_term(Term, CopyOfTerm)$ which has the following meaning: "$CopyOfTerm$ is a renaming of $Term$, such that new variables have been substituted for all variables in $Term$"([SICStus 95]).

This way of specifying the inference engine(section 4.3) separately from the error checking mechanism(section 4.4) makes it easier to understand, and if we needed to, we could plug-in another inference engine(expressed as $provable/1$) without changing the error checking mechanism(expressed as $onto\_prove/2$). An implementation of this approach is given in appendix C.1 and we elaborate on its usage in section 7.1. However, as we will describe in section 4.7, for practical reasons we also chose to integrate these two in a single mechanism which can incorporate multiple layers in more complex ontology checking.

4.5 Detection architecture

In this section we glue together the previous sections in a designated architecture to support error detection. In figure 4.1 we illustrate the approach in a single layer, which is extended in the next section to multi-layers.

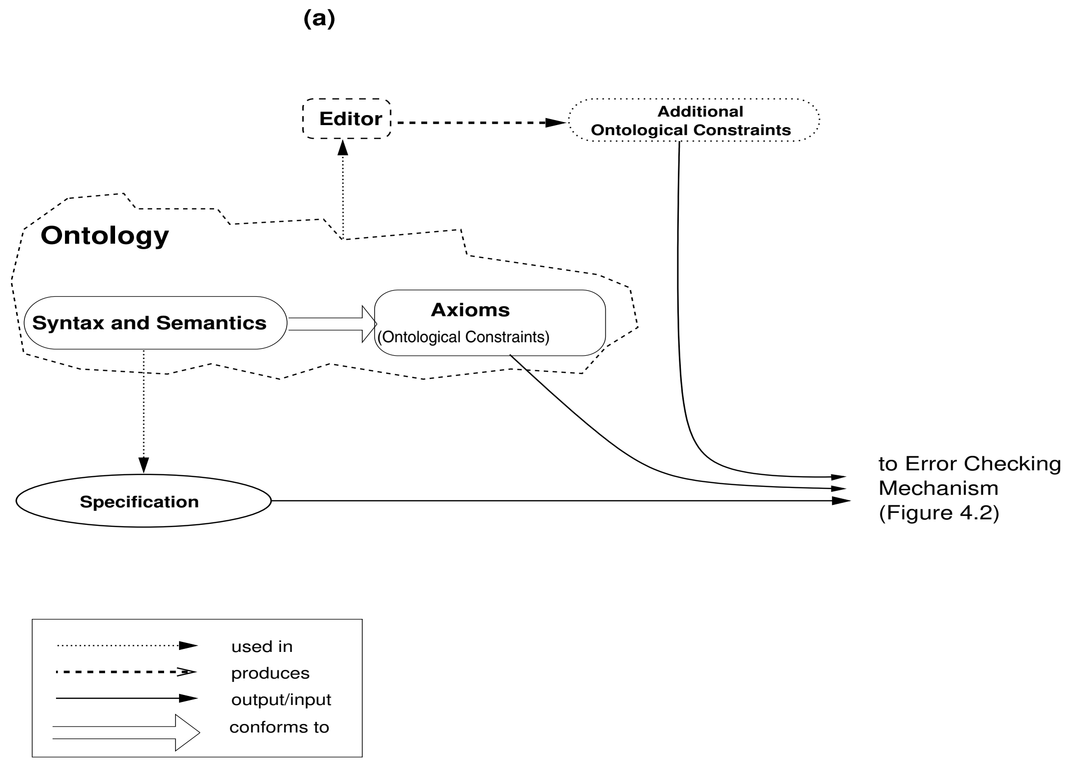


Figure 4.1: The Multi-layer architecture(part a): a single layer. See Figure 4.2 to clarify details of multiple layers.

We operate the architecture as follows: we have a formal ontology with its syntax and semantics which conform to ontological axioms given in the ontology. A specifier builds a specification by using the ontology. That is, the specification construction starts by adopting the syntax and semantics of the ontology. We use Horn clauses as a specification formalism with the normal Prolog execution model. This allows us to interpret the specification declaratively based on the underpinning computational logic while the procedural interpretation makes it possible to check the correctness of the specification automatically by using the error checking mechanism presented in the previous section.

The ontological constructs will not be the only parts of the specification. In fact, it is normally impractical to construct an executable specification by using only the ontology's constructs. However, the implementation of our error checking mechanism, the meta-interpreter, is tolerant to this: we can check all the ontological parts of the specification regardless of how they interact with other parts of the specification. Moreover, the meta-interpreter can parse any specification regardless of the presence or absence of ontological parts.

The existing ontological axioms can be enhanced by adding extra, application-specific constraints[2]. These are used to detect discrepancies tailored to the application that adopts the ontology, an example of which is given in section 5.4. In the same section we describe an editor we developed which facilitates the construction of additional constraints.

The specification along with the ontology constructs and any additional ontological constraints are parsed from the error checking mechanism, implemented as a meta-interpreter (section 4.7). Ontological constraints are used to verify the correct use of ontological constructs in the specification. Their role is to ensure that the correct interpretations of ontological constructs will be given. Whenever a statement in the specification which uses ontological constructs does not comply to the ontological constraints an error will be reported.

4.6 Multi-layer architecture

The architecture we described above does not solve a potential problem: the correctness of ontological constraints themselves. Whether they are provided by ontological engineers in the form of ontological axioms or are application specific error conditions they may be erroneously defined. This could lead to an erroneous error diagnosis.

To tackle this problem we invented a multi-layer architecture in which it is possible to check many specifications and their ontological constraints simultaneously. The

[2] Uschold and colleagues argue for the need of additional axioms tailored to domain specific applications in [Uschold *et al.* 98a].

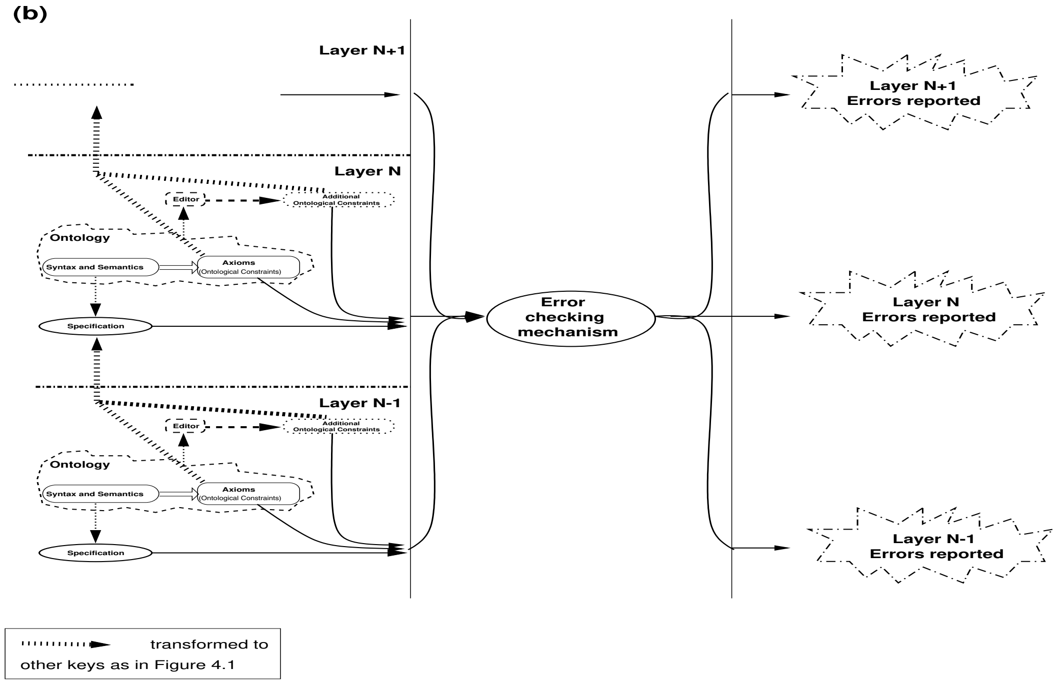


Figure 4.2: The Multi-layer architecture(part b): multiple layers. See Figure 4.1 to clarify details in a layer.

architecture is composed of an arbitrary number of layers each similar to the one described in the previous section. The use of the architecture, illustrated in figure 4.2, is as follows: assume that at the lower layer a specifier constructs the specification by using constructs from the chosen ontology. The specification should also conform to the ontological constraints provided by the ontology. This can be checked with our error checking mechanism as described in the previous section which will show that the specification is correct with respect to the parts of it that conform to the ontological constraints.

However, if an ontological constraint has been erroneously defined we can check this for error with our flexible mechanism. This is possible because we treat ontological constraints as a specification in a layer above the one to which we applied the same checks as for specifications. Ontological constraints are checked for errors against another set of constraints which can be viewed as meta-level constraints. They are part of the ontology and their use is to verify the correctness of the constraints. The result of this check will be the detection of an error, if any, in the ontological constraints. Ultimately, this layer checking can be extended to an arbitrary number of layers upwards, until no more layers can be defined. Although the multi-layer architecture looks similar to the traditional view of modularity in software development it differs in the sense that each new layer(or module) that is added to the indexed lattice is used to verify the correctness of the layer beneath it. Consequently, another layer can be added at the top of it to check its correctness. Hence, there is no restriction as to how many layers can be defined in the architecture.

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 using the same mechanism. This implies that the ontology in each layer need not be the same. In fact, in chapter 6 we describe an application of this approach to a set of ontologies where each layer represents a different ontology. Assuming that the ontologies placed in the architecture are somehow related (for example, via an inclusion lattice), we can check for correctness an ontology against another one. In chapter 6 we elaborate on the potential advantages of the approach via an example case whereas in the next section we describe how we implemented the architecture.

4.7 Implementation

In the domain of Prolog programming, a meta-interpreter is a program written in Prolog which interprets Prolog programs. Various implementations of meta-interpreters can be found in the literature(eg: [Clocksin & Mellish 94], [Sterling & Shapiro 94]). A practical use of them is described in detail in [Robertson *et al.* 91]. Two crucial decisions must be made in using the meta-interpreter technique. First, one must choose the method for replicating the standard Prolog interpreter. This should, potentially, be amenable to adaptation. Second, one must choose appropriate augmentations to the meta-interpreter in order to achieve the desired functionality.

4.7.1 Error checking meta-interpreter

The whole error checking meta-interpreter is included in appendix B.1 in Prolog. Here we describe the parts that are concerned with the implementation of the multi-layer approach and error detection. We also list below the predicates we use in the multi-layer architecture followed by part of the meta-interpreter in Horn clause notation:

- $specification(Level, (A \leftarrow B))$: we use this designated predicate to denote that the Horn clause $A \leftarrow B$ is part of the specification, and belongs to layer $Level$ in the architecture.

- $ontologicalDefinition(Level, (A \leftarrow B))$: this predicate denotes that the Horn clause $A \leftarrow B$ is defined in the ontology of layer $Level$ in the architecture.

- $error(Level, X, Condition)$: this predicate used to denote that the ontological constraint $(X, Condition)$ belongs to layer $Level$ in the architecture. This is the format used in section 4.1 for representing ontological constraints augmented with the index $Level$.

- $axiom(Level, X, Condition)$: this predicate is the opposite of the $error/3$ predicate. It represents ontological axioms directly, rather than transforming them to the constraints format of section 4.1.

$$
\begin{array}{l}
onto\_solve(Goal, Path) \leftarrow solve(Goal, Path, 0).\\
solve((A \wedge B), Path, Level) \leftarrow solve(A, Path, Level) \wedge solve(B, Path, Level).\\
solve((A \vee B), Path, Level) \leftarrow solve(A, Path, Level) \vee solve(B, Path, Level).\\
solve(\neg X, Path, Level) \leftarrow \neg\ solve(X, Path, Level).\\
solve(X, Path, Level) \leftarrow \neg\ logical\_expression(X) \wedge\\
\qquad\qquad predicate\_property(X, (meta\_predicate\ \_Z)) \wedge\\
\qquad\qquad solve\_metapred(X, Call, Path, Level) \wedge\\
\qquad\qquad Call.\\
solve\_metapred(findall(X, Z, L), findall(X, solve(Z, Path, Level), L), Path, Level).\\
solve\_metapred(setof(X, Z, L), setof(X, solve(Z, Path, Level), L), Path, Level).\\
solve(X, \_, \_) \leftarrow \neg\ logical\_expression(X) \wedge\\
\qquad predicate\_property(X, built\_in) \wedge\\
\qquad X.\\
solve(X, Path, Level) \leftarrow \neg\ (logical\_expression(X) \vee predicate\_property(X, built\_in)) \wedge\\
\quad ((specification(L, (X \leftarrow Body)) \wedge L =< Level \wedge solve(Body, [X|(L, Body)], Level)) \vee\\
\quad (ontologicalDefinition(L, (X \leftarrow Body)) \wedge solve(Body, [X|(L, Body)], L))) \wedge\\
\quad NextLevel\ is\ Level + 1 \wedge\\
\quad proofcheck(X, \_, ProofList, Body, ProofBody, Level) \wedge\\
\quad proofmember(ProofBody, ProofList, NextLevel) \wedge\\
\quad detect\_errors(X, Path, NextLevel).\\
\ldots\\
detect\_errors(X, Path, Level) \leftarrow error(Level, X, Condition) \wedge\\
\qquad\qquad\qquad solve(Condition, Path, Level) \wedge
\end{array}
$$

```
35                                 record_error(Level, X, Condition, Path, error)∧
36                                 fail.
37 detect_errors(X, Path, Level) ← axiom(Level, X, Condition)∧
38                                 ¬ solve(Condition, Path, Level)∧
39                                 record_error(Level, X, Condition, Path, axiom)∧
40                                 fail.
41 detect_errors(_, _, _).
```

The first part, lines 1 to 20, is concerned with the implementation of the standard 'vanilla' model augmented with extra features for handling meta-predicates, built-in predicates, negative clauses, and the designated predicates described above. This part has the following declarative meaning: the top level goal of the meta-interpreter is `onto_solve/2`. This will invoke `solve/3` for satisfying the goal given for test starting from the first layer of the architecture, layer 0. Each layer is checked against its ontological constraints - expressed either as axioms or error conditions - that belong to the layer above it. In line 2 we state that a conjunction of goals $A \wedge B$ is true(i.e.: solvable through the meta-interpreter), if A is true and B is true. Line 3 treats disjunctive goals, where $A \vee B$ is true, if either A or B is true. In line 4 we treat negated goals: $\neg X$ is true if we cannot solve it through the meta-interpreter. In lines 5 to 8 we treat non-logical features like meta-predicates. Meta-predicate X is true if after applying successfully the designated predicate $solve\_metapred/3$ to obtain a new goal, $Call$, $Call$ is true. In lines 9 and 10 we treat two such meta-predicates provided by the Prolog language: $findall/3$ and $setof/3$. In lines 11 to 13 we treat built-in predicates which are always true. In lines 14 to 20 we implement the error check in the multi-layer architecture. It has the following declarative reading: (line 14)the goal given for testing, that is X, is true if it is not a logical expression(i.e. a conjunctive, disjunctive or negative literal) or a built-in predicate; (line 15)and X is the head of a clause in the specification at layer L which is beneath or in the same layer with the one we are testing($Level$), and the $Body$ of the clause is solvable in the layer we are testing($Level$) from the enhanced vanilla model described above; (line 16)or X is the head of an ontological definition at layer L and its $Body$ is solvable in the same layer(L) by the vanilla model given above; (line 17)and we can check the head for conceptual error occurrences with respect to the layer $NextLevel$ which is above to the one we are testing($Level$). This check is implemented via the $detect\_errors/3$ goal in line 20. The predicates $proofcheck/6$ and

$proofmember/3$ in lines 18 and 19, respectively, are used to implement a different style in error detection and will be explained in detail through an example case in section 5.3.

The second part, lines 33 to 41 deals with the error detection. In lines 33 to 36, we state that the goal $detect\_errors/3$ is solvable if there exists an error condition, expressed as the 3rd argument in the $error(Level, X, Condition)$ clause(line 33), which is solvable from the meta-interpreter via the $solve/3$ goal(line 34). If so, then the error occurrence is reported in line 35, via the predicate $record\_error/5$. In lines 37 to 40 we check for error occurrences by using the same technique but this time checking for violations of axioms, given as the $axiom/3$ predicate described above.

The error checking is recursive(via the mutually recursive $detect\_errors/3$ and $solve/3$ predicates), so the proof that an error exists may itself generate errors. Those are checked against the ontological constraints exhaustively(by forcing the $detect\_errors/3$ to backtrack in lines 36 and 40 before eventually succeed in line 41). We accumulate all the errors we detected on given goals for testing as well as those on their subgoals(by recording them in lines 35 and 39). We also accumulate information regarding the execution path that has been followed by the inference engine in proving a goal(in lines 15 and 16 by instantiating the $Path$ variable with a list containing the path followed), the type of ontological constraint that has not been satisfied - axiom or error condition - and the layer in which the error has occurred(information recorded via the $record\_error/5$ predicate - not shown here).

The multi-layer approach, which is implemented in lines 15 to 17 in the meta-interpreter above, is based on the principle of 'climbing' layers in order to prove the correctness of statements in the specification. Intuitively, the specification statements at a given layer N, are included in layer N-1. That is why we force the layer we choose for testing to be beneath or equal with the one we are currently in(line 15). Furthermore, the idea of separating the specification statements from the ontological constraints used to check them, made us to place ontological constraints at a layer above. This is reflected by our strategy in implementing the meta-interpreter where we first increase the layer index(line 17) and then call the error conditions, if any, on that layer. This implies that the ontological constraints of layer N are defined over specification statements of layer N-1. Notice that, unlike specifications which we constrain to apply only at levels below the one currently being considered, ontological definitions are used irrespective of the layer(predicate $ontological\_definition/2$ in line 16). This gives us the freedom to check ontological definitions from several ontologies that are not necessarily defined in neighbouring layers(i.e.: layer N and N+1). This is common in complex ontological structures where a definition at layer 0 might use another definition that belongs to layer 5. An example of this is given in chapter 6 where we explore the use of the approach in the context of 7 ontologies related via an inclusion lattice. Despite the different approach in treating ontological definitions, we can still apply the same principle with regard to error checking: ontological definitions will be checked against ontological constraints which belong to the layer above them(as for specifications).

4.7.2 Transformation to constraints format

As we mentioned in section 4.1, existing ontological axioms are transformed automatically to the constraints format we adopt. This is done after applying a set of rewriting rules and then parsing the produced formula with a Definite Clause Grammar(DCG) program to produce the designated predicates format described in the previous section. In particular, we produce a conjunctive Normal Form(NF) after applying the following set of rewriting rules:

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

the rules used are given in Predicate logic:

$A \leftrightarrow B \Rightarrow ((A \rightarrow B) \wedge (B \rightarrow A))$

$A \rightarrow B \Rightarrow \neg(A \wedge \neg B)$

$\neg(A \vee B) \Rightarrow (\neg A \wedge \neg B)$

$A \wedge (B \vee C) \Rightarrow ((A \wedge B) \vee (A \wedge C))$

$(A \vee B) \vee C \Rightarrow A \vee (B \vee C)$

$(A \wedge B) \wedge C \Rightarrow A \wedge (B \wedge C)$

$\neg(\neg A) \Rightarrow A$

The program that implements the transformation in NF is given in appendix D.1. So, for example, the axiom on *navy threat* described in a later chapter(in section 5.4), given here in Prolog format:

`navyThreat(A):- aircraft(A),stores(A,W),target_type(W,vessel)`

will be transformed to the following NF(see appendix D.1):

`\+ (navyThreat(A), \+ (aircraft(A),stores(A,W),target_type(W,vessel)))`

From here we use an auxiliary program to identify the definition to be checked(the `navyThreat/1` predicate), and we employ the DCG program listed in appendix D.2 to produce the designated constraints format. In particular, let's assume that we want to produce an error condition which will be placed at layer 1 of the architecture, and we have a specification written in a file called 'aircraft.pl', and the NF given above is stored in a file called 'constraints.pl'. The following instantiations of the top level goal $translate/5$(described in appendix D.2):

`translate('aircraft.pl','constraints.pl','output.pl',0,errors)`

produces the following error condition with respect to the NF given above:

`error(1,navyThreat(A), (\+ (aircraft(A),stores(A,W),target_type(W,vessel)))).`

which is written along with the specification, masked with the $specification/2$ designated predicate explained in section 4.7, in the 'output.pl' file. It is placed at layer 1, to monitor the specification statements of layer 0, as indicated by the 4th argument of $translate/5$ goal.

4.7.3 The *Ontological Constraints Manager(OCM)*

We have written a `Java` application, which we call *Ontological Constraints Manager(OCM)*, that acts as the front-end of several tools implemented in Prolog(some of which are included in the appendices). We used the `Java` package `jasper` to link Prolog with the `Java` front-end, which is a bi-directional `Java` to Prolog interface developed from SICStus[3]. A screenshot of the entry point of the OCM is given in figure

[3] The package is documented online under the heading "Mixing Java and Prolog"' in the following URL(as of Feburary of 2000): http://www.sics.se/isl/sicstus/docs/latest/html/sicstus.html

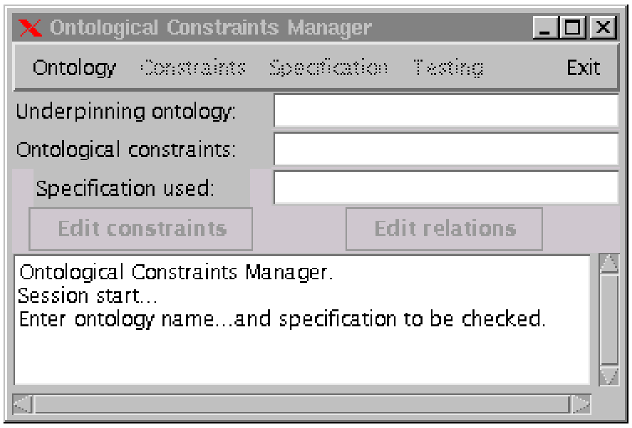


Figure 4.3: The *Ontological Constraints Manager(OCM).*

4.3.

We will not describe in detail the OCM's functionality but we list below tasks that can be accomplished by using this tool along with pointers to references in later chapters where these tasks are implemented.

- *Viewing/Selecting ontologies*: there are basic disk-browsing and file-viewing facilities embedded in the OCM to facilitate the ontology selection task.

- *Ontological relations editing*: in addition to the existing ontological relations, if any, the OCM user can build unary, binary, and ternary relations defined over the constructs of a given ontology by using a designated editing tool. An example of this is presented in section 5.2.

- *Ontological constraints editing*: in cases where ontological constraints are not provided, or we want to define extra ones, we have built an editing tool which allows the building of additional ontological constraints. It is based on a heuristic method and its operation is explained via an example case in section 5.4.

- *Specification construction aid*: as we describe in chapter 7, the OCM is linked to existing tools that allow semi-automatic construction of a specification written in Prolog. These tools stem from the *Prolog programming techniques* literature(see, for example, [Bowles *et al.* 94]). In addition, we also have written a simple free-text editor where the user can write a free-form specification and automatically choose ontological constructs to be included in it.

- *Error checking and reporting*: all the tools that were described in previous sections(meta-interpreter, transformation to NF, DCGs for producing the designated formats) are accessible via the OCM, which acts as a single entry point to conceptual error checking.

4.8 Limitations

Our approach has some limitations. We summarise these in three main areas: the error checker, ontological constraints editor, and the meta-interpreter technique. In particular, the error checking approach we adopt cannot guarantee that it will reveal all conceptual error occurrences in the specification and/or ontology. This stems from the fact that, on one hand, we rely on the provision of the right ontological constraint to detect the error and, on the other, we hope that we chose the right goal to test for errors. As far as the provision of the ontological constraints is concerned, to our surprise in most of the ontologies we have encountered no rich axiomatisation was provided. Even in some of the most carefully designed ontologies, we had to improvise a constraint in order to enhance the existing axiomatisation and enable trapping of subtle error occurrences. This might be a considerable workload and overwhelm those we choose to follow this root in conceptual errors checking. Nevertheless, the gain can be worthwhile as we demonstrate in the chapters 5 and 6 where we applied the approach successfully and reveal subtle errors not only in the specifications but in existing, published ontologies too. The problem of choosing the right goal to test is a concern in our method. That is because executing the 'wrong' goal will fail to reveal the error. However, the way that our error checking is deployed will guarantee that when we check the fallible goal the error will be detected. That is because our error checking strategy does not interfere with the execution of the specification. The specification is executed normally and checks for errors are done on goals that have succeeded in the proof. This means that any errors in the exploration of the failed proof are ignored. This sort of testing, amid its limitations, is sufficient for the purposes of our checks: to ensure that whenever we execute a goal in the specification this conforms to the underpinning axiomatisation. We achieve this guarantee by exhaustively checking the goal against the ontological axioms. As a consequence we raise an issue of computational cost here because although the goal requires one proof, the constraints are checked exhaustively. Lastly, another potential limitation might be the lack of automatic error correction. This is an area that we have not explore in depth. This decision stems from the nature of the errors we are dealing with. We argue that there is no foreseeable way to automate the process of correcting conceptual error occurrences. For example, if we apply the mechanism to check an ontology(as we are doing in chapter 6) and we manage to reveal an erroneous ontological definition, then what should the correct one be like? The answer is not at all obvious especially when the revealed error is concerned with an erroneous axiom. It is hard to imagine an automatic way of building correct ontological axioms to replace erroneous ones.

We cope with some of the limitations by using the ontological constraints editor we built. Take for example, the problem of building extra axioms. We used this editor to build some of the axioms used in later chapters(i.e.: 5.4). However, the editor is not meant to be a full-fledged ontological axioms building tool that will support free-form editing. It actually uses a heuristic mechanism which forces the user to employ as many existing ontological constructs in the axiom as possible. The rationale behind this is that this way of building an axiom will make it easier to reuse it in other applications. That is because the axiom is composed of existing ontological constructs, whereas in free axiom editing freedom to use and define new constructs will inject update issues with respect to the ontology.

The third area of limitations is that of the meta-interpreter. Although we provide two different approaches in building these designated meta-interpreters there are some limitations in their operation. For instance, the treatment of non-logical features like cut(!), built-in predicates other than those we treat($findall$ and $setof$), and system calls(i.e., invocation of external procedures/systems). There are some solutions described in the literature with respect to treatment of built-in predicates(see [Robertson *et al.* 91]), but these are routine extensions to the pure Prolog interpreter and distract from our main concern. Rather, we concentrated on various ways of deploying meta-interpreters and integrating them with the conceptual errors theory provided in section 4.2. Hence, the separated inference engine and error checking procedure described in appendix C.1 and used in section 7.1.

4.9 *Chapter summary*

In this chapter we explained how we implement the idea of deploying domain knowledge to check the early phases of design. In particular, we gave a formal representation of ontological axioms and show how we automatically derive ontological constraints from it. We defined a theory for conceptual error detection which allows us to reason about error occurrences. This is done via an an error checking mechanism which is based on a generic inference engine. It is made operational through a generic architecture tailored to accommodate error checking. We also explained how this approach can be extended to check ontologies themselves. We provided the implementation details of the approach in the last sections of this chapter and identified potential limitations. In the next two chapters we will deploy this approach in a variety of example cases to demonstrate its usage and explore its potential.

Chapter 5

Evaluating the approach

In this chapter we apply the approach presented in the previous chapter. In particular, we wish to do this in a variety of contexts related to the deployment of domain knowledge. To uniformally evaluate the approach we devised a 'conformance check' method which is explained in section 5.1. This made it possible to identify potential uses which are demonstrated via example cases drawn from the application areas of business process modelling(section 5.2), ecological modelling(section 5.3), and air-campaign planning(section 5.4). We then experimented with other prospective uses which are summarised in section 5.5. An elaboration of the conformance check which employs the multi-layer architecture(presented in section 4.6) is described in the following chapter.

5.1 Conformance check

Ontologies are often deeply embedded in applications which makes their role and contributions difficult to identify. Uschold argued in [Uschold 98b] for the need to characterise ontologies in order to alleviate the situation. Similar to his aims is the conformance check we devised([Kalfoglou *et al.* 00b]). It is driven from the need to ensure that the ontology is not misused by the application that adopts it. We wanted to demonstrate that the application conforms to the ontology. Hence the conformance check. We designed it to be as generic as possible so that we can instantiate it in a variety of contexts. Therefore we used only three entities: ontology, conformance check, and application. The method is given diagrammatically in figure 5.1.

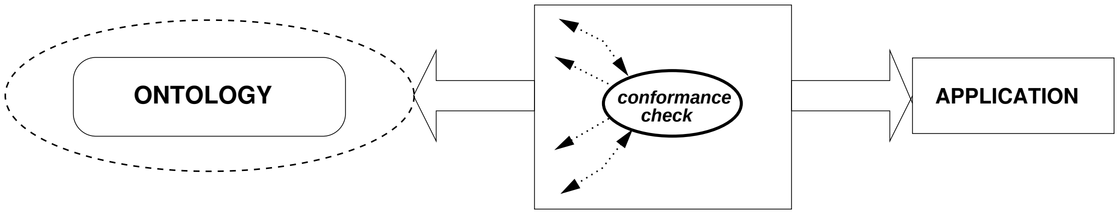


Figure 5.1: Conformance check of application to ontology.

The dashed line surrounding the ontology denotes the current ontologies' evaluation scope. As we mentioned in section 3.3 these are proposed guidelines that are applicable within the design phase of an ontology. Which means that they do not consider possible instantiations of an ontology in applications. The double arrow that connects the application with the ontology symbolise that the ontology is used in that application. As we can see from figure 5.1 the box placed between in the ontology and application entities denotes our mechanism, presented in section 4.6. Its role is to ensure that an application conforms to an ontology by means of checking its definitions for compliance to the ontological constraints.

There can be variations of this diagram with respect to possible instantiations of the ontology/application pair. These are: (i)applying the mechanism in a situation where the ontology exists but not the application, (ii)the application exists but not the ontology and, (iii)both the ontology and the application exist. In the first case, an application was designed according to a scenario described in the literature and supposedly conformed to the existing ontology. We then applied the mechanism to verify this conformance. In the second case, an existing application was used to derive potential ontological constraints applicable to a particular domain. We then used the mechanism to verify conformance of the application to these constraints. In the last case, the mechanism was deployed to verify the conformance of an existing application to an existing ontology. In all of these cases we devised and introduced artificial errors in the application in order to test the mechanism.

The cases are described in the next three sections whereas here we draw the attention of the reader to a fourth case which is related to the ontology evaluation scope. An elaboration of the third case is the extension of the evaluation scope we achieved after applying the mechanism to verify the conformance of exemplar applications to a given ontology. In doing this, we check both the ontology and applications while we consider application-specific constructs in our evaluation. The case employs the multi-layer architecture and we devote the whole of the next chapter to its description as it deserves special attention: we discovered real errors in the original ontology. Lastly, we should mention that there is also the case when neither the ontology nor the application exist. However, in this case the question should not be how to deploy the mechanism but what to build first - ontology or application - which is a matter of debate.

5.2 PIF case

In the first case we used an existing ontology, the Process Interchange Format(PIF) ontology([Lee *et al.* 98]). We took advantage of the rich axiomatisation of PIF to formulate our error conditions and we developed a specification, drawn from a publicly available scenario, that adopts the syntax and semantics of PIF. We then devised an erroneous specification statement that violates the PIF axiomatisation and deployed the mechanism to detect the error. In figure 5.2 we illustrate how figure 5.1 is instantiated with respect to this case.

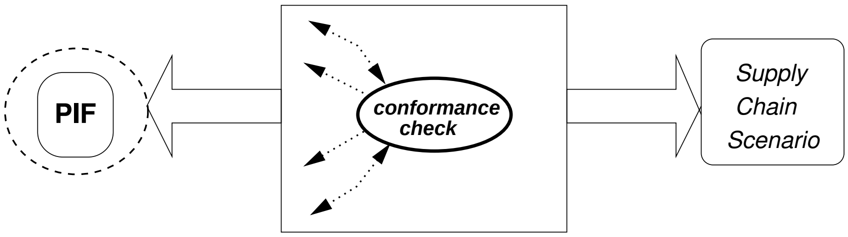


Figure 5.2: Conformance check of scenario-drawn application to the PIF ontology.

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 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 [Lee & Malone 90]. 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]

We have chosen a scenario proposed by others for the PIF evaluation: the *"supply chain scenario"*[2] which is concerned with the "development and coordination of supply chain processes between a manufacturer, retailer, distributor, warehouse company and transportation company"([Polyak *et al.* 98]). We will present the detection of an artificial error we introduced in a small part of the scenario.

Our specification fragment is given below and represents part of the "process document request" in the transportation company. We quote the description of this process from [Polyak 98]:

> "...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."

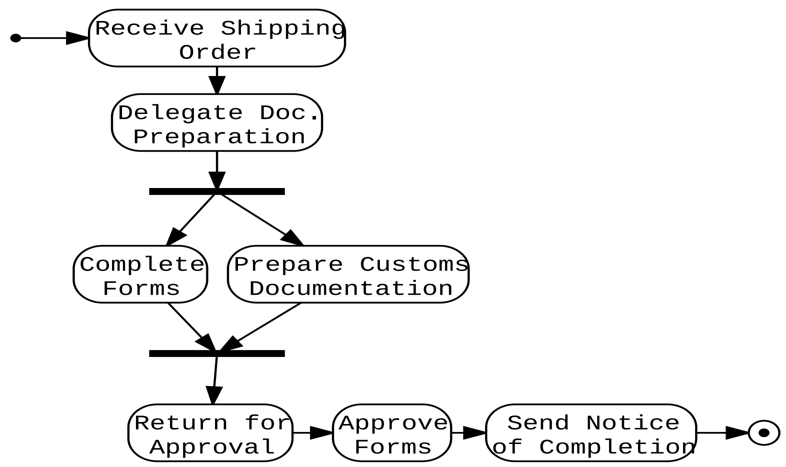


Figure 5.3: Supply chain scenario: Process Document Request.

A diagrammatic version of this process is illustrated in figure 5.3. It uses the UML

[1] Examples on the use of PIF can be found in [Lee *et al.* 98] and in [Polyak 98].

[2] 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 [WfMC 96].

activity notation[3]. Activities in this notation are represented via rounded boxes. 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.

The specification fragment with respect to two of the activities described above is as follows:

1 $specification(0, (object(X) \leftarrow agent(X)))$.
2 $specification(0, (agent(manager) \leftarrow true))$.
3 $specification(0, (agent(employee) \leftarrow true))$.
4 $specification(0, (activity(approve\_forms) \leftarrow true))$.
5 $specification(0, (activity(complete\_forms) \leftarrow true))$.
6 $specification(0, (performs(Agent, Act) \leftarrow object(Agent) \wedge activity(Act)))$.

The first line declares that, according to the PIF documentation([Lee *et al.* 96] and [Lee *et al.* 98]), an *agent* is a specialisation of an *object*. In lines 2 to 5 we have the declarations of agents and activities with respect to the process being modelled. The last line connects agents and activities by means of the *performs/2* relation. According to the PIF documentation it must be defined over objects and activities. Note that we adopt the specification format we introduced in section 4.7.1. Hence the first argument of the specification fragments given above denotes the layer to which they belong, layer 0 in the multi-layer architecture.

Although this sort of representation conforms to the syntax of PIF it hides an important caveat. If we are forced to, we can allow an erroneous assignment of activity to agent. That is, if we use the specification to answer the question: 'Will the agent *employee* perform the *approve_forms* activity?' the result will be a positive answer since there is nothing in the specification to prevent this error as both *employee* and *approve_forms* are valid constructs as parts of the given specification of *performs/2* relation.

To prevent such an error we take into account the existing PIF axiomatisation. We reinforce the definition of *performs/2* relation by introducing the notion of *capability* as expressed by the PIF ontology. According to PIF documentation "an agent is distinguished from other agents by what it is capable of doing or its skills"([Lee *et al.* 96]).

[3] For a detailed explanation of the notation see [Booch *et al.* 98], and for a more detailed explanation of the diagram see [Polyak 98].

This could be expressed as a binary relation that maps agents with their capabilities in the form of activities:

7 $specification(0, (capability(manager, approve\_forms) \leftarrow true)).$
8 $specification(0, (capability(employee, complete\_forms) \leftarrow true)).$

and the enhanced *performs/2* relation is now axiomatised as follows:

$$performs(Agent, Activity) \rightarrow capability(Agent, Capability) \wedge Activity = Capability.$$

We then transform this axiom to the constraints format we adopt(introduced in section 4.7.1) in the form of an error condition:

9 $error(1, performs(Agent, Activity), \neg\ (capability(Agent, Capability) \wedge Activity = Capability)).$

to express that it is an error when an activity performed by an agent is not one of which it is capable.

Assuming the ontological constraint of line 9, when we ask the question given above with respect to *employee* agent and *approve_forms* activity the error checking mechanism detects the error. The results of error detection are shown in figure 5.4 where we include a screenshot of the OCM tool, the Java front-end we mentioned in section 4.7.3.

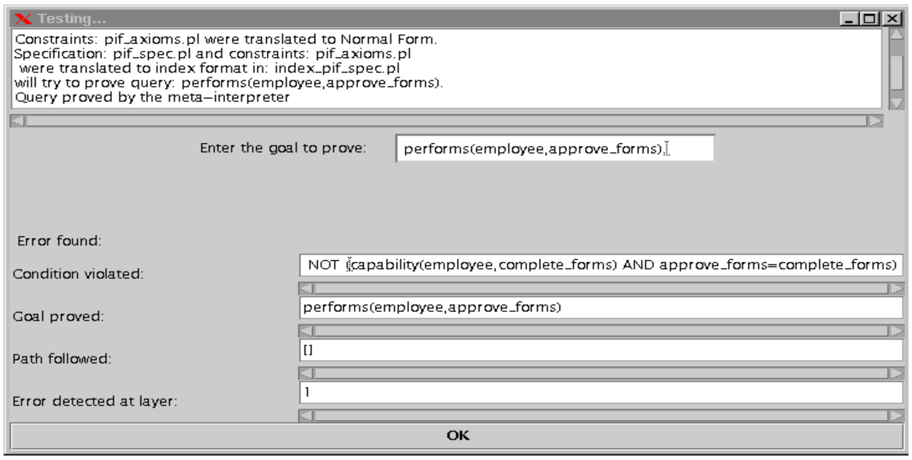


Figure 5.4: A screenshot of the OCM tool: an error detection dialog box.

As we can see from figure 5.4 the given goal requiring proof conforms to the definition of *performs* relation in line 6 and is provable by the meta-interpreter, but contradicts the ontological constraint of line 9. This is highlighted by the inequality of *employee's* capability(*complete_forms*) and the erroneous activity that intend to perform(*approve_forms*).

To facilitate the definition of ontological constraints such as the one described above we have built two editing tools which provide built-in checks for conflict and subsumption occurrence. Here we will describe the first tool which we call the 'relations editor' whereas the second tool, the 'constraints editor' is described in section 5.4 along with the example case of AIRCRAFT ontology.

With the relations editor, one can define unary, binary and ternary relations that hold over ontological concepts and choose logical connectives to link them. The collection of concepts from the ontology as well as the distribution of variables that will be shared among the literals is done automatically, the user only has to select the concepts he wants to use. For example, editing an axiom of the PIF core, which is described textually as: 'The *before* relation holds only between timepoints.", will result in the following Horn clause:
$before(A, B) \leftarrow point(A) \wedge point(B)$.
In figure 5.5 we include a screenshot of the relations editor at the point where the user will select the logical connective for the above relation.

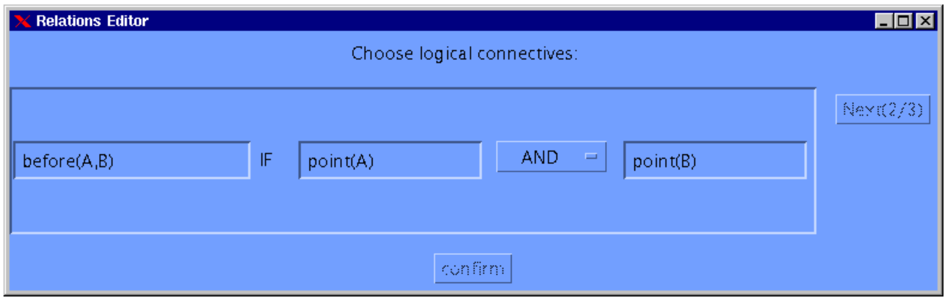


Figure 5.5: A screenshot of the OCM tool: the relations editor.

If we are interested in using the above relation as an axiom then at this stage we can add it to the existing ontology axiomatisation after conflict and subsumption checking is done. The sort of conflict check we apply declares two axioms as being contradictory to each other if they are variants but still not unifiable after having their variables temporarily bound. Two axioms are variants if they are the same although they might have different variable names. In our checks for variance we ignore the 'not' operator, if exist. This will be checked in the unification test in order to reveal potential conflict. For example, assume that the ontology already contains the axiom: $participates\_in(O, A, T) \leftarrow exists\_at(O, T) \wedge is\_occurring\_at(A, T)$, and we try to add the axiom: $participates\_in(Obj, Act, T1) \leftarrow exists\_at(Obj, Act) \wedge \neg is\_occurring\_at(Act, T1)$, this will cause a conflict alert because these two axioms are contradictory to each other. The subsumption check will ensure that for two axioms that their heads match, we won't let a more generic one subsume an existing detailed one. This is a limited form of subsumption check that will prevent specific information loss caused by a generic axiom. For example, assume the axiom above and that the ontology already contains the: $before(P, Q) \leftarrow point(foo(P)) \wedge point(foo(Q))$, this will cause a subsumption warning to the user since variables A and B from the new axiom will subsume predicates $foo(P)$ and $foo(Q)$, respectively. In both the conflict and subsumption check no action is taken by the system apart from warning the user because there are cases where we might want to include both axioms in the ontology.[4]

However, if the relation is to be used as an ontological constraint in the form of an error condition not to be satisfied by the specification then we apply the procedure described in section 4.1 to produce the following ontological constraint:
$error(1, before(A, B), \neg(point(A) \wedge point(B)))$.

5.3 EcoLogic case

The *EcoLogic* case gave us the ability to explore the second use of the approach. We used the semantically rich specifications included in the *EcoLogic*([Robertson *et al.* 91]) book to derive ontological constraints with respect to those specifications. These constraints represent broad ecological knowledge which could be applied to other models. Hence, they could be constructs of a prospective 'ecology ontology'. A similar approach in ontology building is discussed in [van derVet & Mars 98] where the authors elaborate on the bottom-up approach in ontology construction. In figure 5.6, we illustrate

[4] This is a debatable issue and Visser and colleagues elaborate on this in [Visser *et al.* 98].

the approach as an instantiation of figure 5.1.

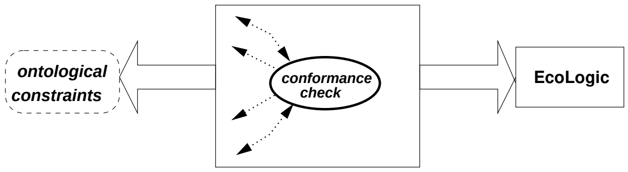


Figure 5.6: Conformance check of *EcoLogic* specifications to designated ontological constraints.

In particular, we dealt with ecological modelling in an area which is concerned with complex biological systems where it is difficult to decide how to represent them in a simplified form as simulation models. In the *EcoLogic* book Robertson and colleagues elaborate on the use of logic to represent those simulation models. We experiment with three simulation models and devised seven potential discrepancies which we were able to capture with our mechanism. All the cases are described in [Kalfoglou 98] whereas in [Kalfoglou & Robertson 99c] we discuss a single case. Here we will describe two representative cases originating from corresponding simulation models.

The first model to which we applied our approach was the "rabbit-grass energy flow" based on a system dynamics model represented in [Forrester 61]. The whole model is given in Horn logic in [Robertson *et al.* 91] while here we give a textual description and illustrate it in figure 5.7(borrowed from [Robertson *et al.* 91]).

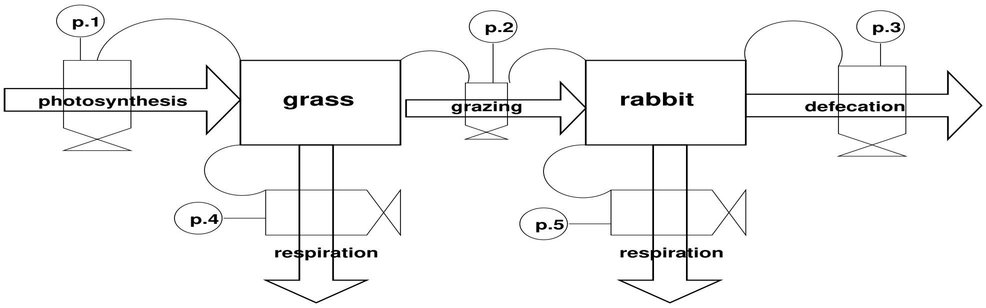


Figure 5.7: The *EcoLogic* case: Rabbit-grass energy flow model.

As stated in [Robertson *et al.* 91]: "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 exits 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." The recursive definition of a state variable is given in appendix E.1 while here we discuss the ontological constraints drawn from the model.

Using the model description of rabbit-grass energy flow - given in appendix E.1 - we devised four artificial errors and formulated ontological constraints to capture them. In particular we devised, (i)an erroneous assignment of regulating parameters to state variables(lines 30 to 34 in appendix E.1) which caused a miscalculation of the state variable's value, (ii)a deliberate omission of a state variable's value in the calculation of the *grazing* flow between state variables(lines 17 to 21 in appendix E.1) which did not affected directly the result of calculations but only the relevant subgoal, (iii)an erroneous interpretation of the *inflow* and *outflow* processes between state variables and the environmental *source_sink*(lines 11 and 12 in appendix E.1) which caused an erroneous calculation of a state variable's value. All these cases along with the ontological constraints we devised to capture them are described in [Kalfoglou 98]. Here we will present the fourth case which is concerned with an erroneous specification of an additional concept tailored to the particular model.

In detail, according to ecological modelling and simulation knowledge, for any given state variable S, that has value N, at time T, there does not exist state variable S1, which is different than the previous, has value N1 at the same time(T), N1 is greater than N, and S1 is lower in the food chain than S at that time(T). Note that this condition is tailored to the specific application and holds for the existing values of the regulating parameters(lines 30 to 34 in appendix E.1). The constraint is axiomatised as follows:

$$state\_variable(S,T,N) \rightarrow \neg(state\_variable(S1,T,N1) \wedge \neg\, S1 = S \wedge N1 < N \\ \wedge\, lower\_in\_food\_chain(S1,S,T)).$$

and transformed to normal form and subsequently to the error condition format:

$$error(1, state\_variable(S,T,N), (state\_variable(S1,T,N1) \wedge \neg S1 = S \wedge N1 < N \\ \wedge\, lower\_in\_food\_chain(S1,S,T))).$$

In addition, this constraint introduces the concept of *lower_in_food_chain*. This concept originates from domain knowledge and is regarded as an ontological definition in the context of our model. It is formalised as follows:

$$ontologicalDefinition(1, (lower\_in\_food\_chain(S1, S, T) \leftarrow flow(Flow, S1, S, T, \_) \wedge eating\_flow(Flow))).$$
$$ontologicalDefinition(1, (lower\_in\_food\_chain(S1, S, T) \leftarrow flow(Flow, S2, S, T, \_) \wedge eating\_flow(Flow) \wedge lower\_in\_food\_chain(S1, S2, T))).$$

The first clause states that whenever there is an *eating_flow* from a state variable S1, to another state variable S, at time T, then S1 is lower in the food chain than S. The second clause is a recursive definition of the above to enable the model to backtrack on all possible state variables. The definitions of *lower_in_food_chain* are placed in layer 1 of the architecture to be used in the ontological constraint mentioned above. However they include a new concept, that of *eating_flow*. This is then added in that layer and instantiated to the flows of the particular model:
$specification(1, (eating\_flow(grazing) \leftarrow true)).$

To ensure proper use of the *lower_in_food_chain* concept we introduce a constraint at a layer above, layer 2:
$error(2, eating\_flow(F), \neg\ F = grazing).$

This constraint states that it is an error whenever an *eating flow* is not of type *grazing*.

If we execute the simulation model with the correct definitions given above, and check how variables in the model are ordered in the food chain we will get the answer that state variable *grass* is lower in the food chain than state variable *rabbit*.
However, assume that the specifier defines, erroneously, the eating flow instances in the model:

$$specification(1, (eating\_flow(photosynthesis) \leftarrow true)).$$
$$specification(1, (eating\_flow(defecation) \leftarrow true)).$$

If we query the model again then we will get the following erroneous order: that *rabbit* is lower in the food chain than *grass*. With the ontological constraints given above the mechanism traps the error and reports it appropriately(given below in its native format, as a Prolog execution result):

```
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))]
```

Two errors have been detected. The first one has instantiated the variable `F` of the `eating_flow/1` clause to *photosynthesis* and the second one to *defecation*. However, both of them satisfy the error condition on *eating flow*, that is the inequality with the *grazing* flow. We also include extra information concerning the execution path.

The ontological constraint for the *state_variable* clause is tailored to the particular model. However, it uses the *lower_in_food_chain* concept which represent broad ecological knowledge and can be applied to similar applications. The same can be said for the *eating_flow* concept and the condition that monitor its use. We abstracted those concepts from the model, by placing them in another layer in the architecture which enabled us to check the domain knowledge without intervening in the execution of the model that uses it.

This case also gave us the opportunity to experiment with an alternative strategy in error checking. Recall from section 4.7.1 where we mentioned the predicates $proofcheck/6$ and $proofmember/3$ in the implementation of the meta-interpreter. These two are used to implement a different style of testing. Instead of defining a specific constraint over the goal to be checked, this approach implements the idea of checking whether specific parts that are deemed to be necessary to prove a goal belong to the proof tree. The way we represent this sort of error check is through the predicate $proof/3$ which has the following format: $proof(Goal,$`with_body/without_body`$,List)$, where $Goal$ is the goal we want to check, `with_body` and `without_body` are two identifiers we use to separate ground clauses from conditional ones, and $List$ is a list containing the parts that must be in the proof tree of $Goal$.

For example, the following clause:

$proof(lower\_in\_food\_chain(\_,\_,\_)$,`with_body`,$[eating\_flow(grazing)])$

is used to check if the ground clause $eating\_flow(grazing)$ is contained in the proof tree of the conditional clause *lower_in_food_chain/3*. This sort of check is different from the one described above in that we do not consider possible instantiations of subgoals for the *lower_in_food_chain/3* but only check whether the desired clause is in the proof tree.

These checks are implemented via the predicates $proofcheck/6$ and $proofmember/3$ in the meta-interpreter(lines 18 and 19 in section 4.7.1). They do not merit detailed description here but we point the interested reader to appendix B.1 where we include the whole listing of the meta-interpreter. The first predicate is used to convert the proof tree in a list which is traversed by the second predicate to check for list membership and report any missing elements.

The second model we dealt with, uses a state transition approach to represent the passage of time during simulation. It is included in appendix F.1 and explained in detail in [Robertson *et al.* 91]. In [Kalfoglou & Robertson 99c] we apply the mechanism to capture an artificial error we introduced in the model. Here we recapitulate that case.

Suppose that we have 3 different animals(call them a,b and c) and that a prey on b;b prey on c; and c will prey on a. The area on which these animals live is represented by a grid with 3 squares along each side(thus 9 grid square grids in all). Animals move by shifting from the square in which they are currently situated to adjoining square. Each animal moves in the direction of potential prey(e.g. they actively hunt rather than browsing at random) but will not visit a square which it has occupied previously. If an animal is ever in the same square as its prey, the prey is eaten and thus removed from the simulation.

The specifier chooses to represent the states as follows: the initial state is named *s0*. New states of the system will be obtained whenever some aspect of the system changes so we require some way of linking the changes imposed on the system to the events which impose those changes. This could be achieved by the use of a nested term of the form: $do(Action, State)$, where $Action$ is a term representing some action which has been performed. *State* is either the initial state *s0* or another term of the form: $do(PreviousAction, PreviousState)$.

The only action which it is necessary to represent in this model is the movement of an animal from one grid square to another. The specifier represents this action using the term *move(A,G1,G2)* where *A* is the name of some animal; *G1* is the location of the grid square at which the animal was located in the previous state and *G2* is its new location. Figure 5.8 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).

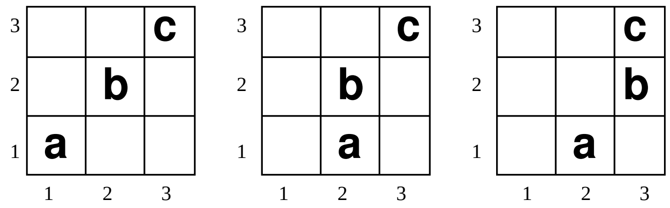


Figure 5.8: The *EcoLogic* case: State transition model - sample sequence of moves.

We will now focus on a fragment of the model that represents the treatment of locations for each animal in the system where we introduced our artificial error.

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 an erroneous part of the specification that includes the error occurrence in describing the condition of animal location:

11 $specification(0, (holds(location(a, (1, 1)), s0) \leftarrow true))$.
12 $specification(0, (holds(location(b, (2, 2)), s0) \leftarrow true))$.
13 $specification(0, (holds(location(c, (3, 3)), s0) \leftarrow true))$.
14 $specification(0, (holds(location(A, G), State) \leftarrow \neg\ State = s0 \wedge animal(A) \wedge$
15 $\qquad\qquad last\_location(A, State, G)))$.

At lines 11-13, the specifier defines the locations of the animals in the initial state, *s0*. In lines 14-15 defines the location of any animal in states other than *s0*. In such states an animal has a location determined by its most recent position in the sequence of actions. However, as we will see below there is a serious omission in this representation which will lead to undesirable behaviour of the model.

For example, in order for an animal to exist at a particular location on the system it should not have been eaten in the meantime. Thus, a predator and a prey cannot be at the same square at the same state. We represent this constraint as follows(in the format of an ontological axiom as introduced in sections 4.1 and 4.7.1):

$$axiom(1, holds(location(A,G),S),(predator(A,B), \neg\ holds(location(B,G),S))).$$

Assume a specification which has no errors, like the one included in appendix F.1, we can use the model by asking: 'Is there a state of the system in which animal, a, gets eaten?' giving the Prolog goal:

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

The Prolog interpreter would then use the definitions of model structure to solve this goal, instantiating S to a sequence of potential moves. The result is given diagrammatically in figure 5.9 and below as a Prolog result:

```
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))))
```

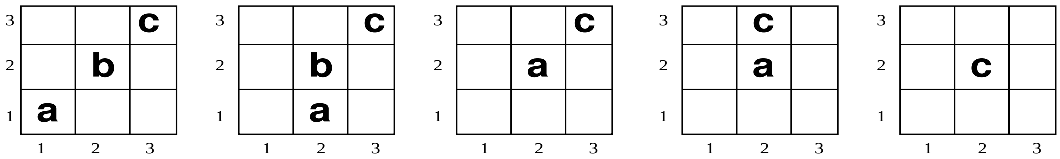


Figure 5.9: The *EcoLogic* case: State transition model - correct sequence of moves.

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

However, assume the specification fragment given above(lines 14 and 15), on backtracking an erroneous answer will be returned to the same query. The Prolog answer is given followed by an illustration in figure 5.10:

```
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))))))
```

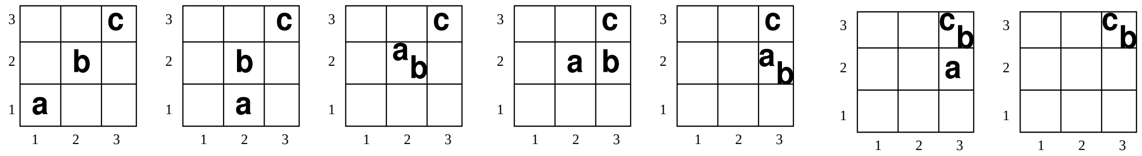


Figure 5.10: The *EcoLogic* case: State transition model - erroneous sequence of moves.

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 by the ontological axiom given above. We illustrate the result diagrammatically in the form of a proof tree as shown in figure 5.11:

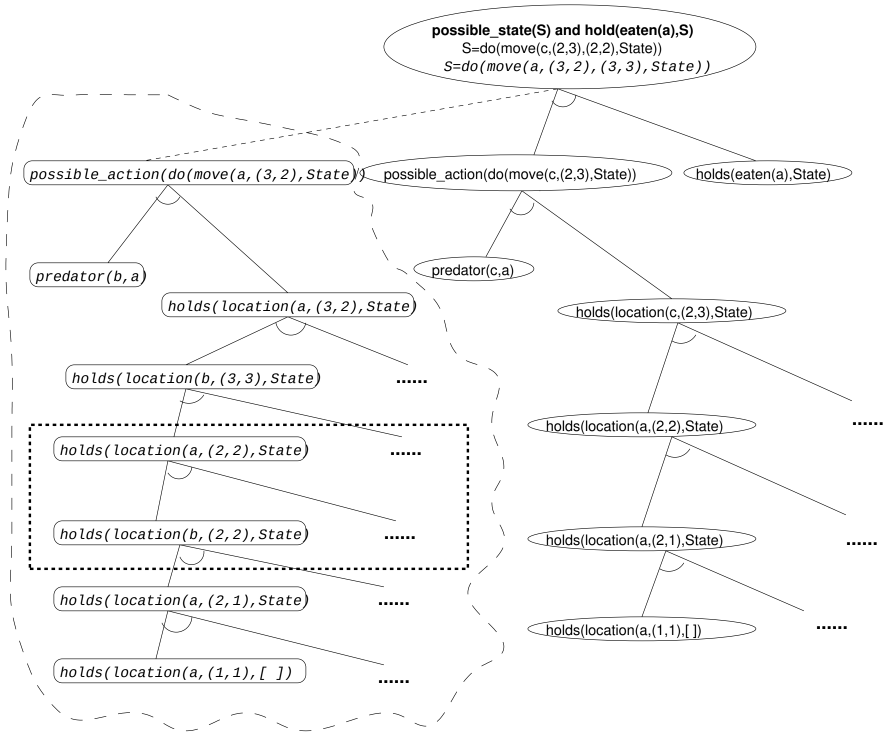


Figure 5.11: The *EcoLogic* case: State transition model - correct and erroneous proof trees.

The right part of the tree is the correct one while the left one is the ontologically erroneous path that has been followed. State variable S is instantiated to two values:

the correct one is in plain font while the erroneous is in italics. In the right tree we have placed in ellipses the goals that has been satisfied with conjunctive arcs connecting them. The erroneous tree which is surrounded by a dashed line shows the correspondent satisfied goals within rectangle boxes. The rectangle box with a dashed line border represents the goal that does not conform to the axiom.

In terms of the meta-interpreter the discrepancy found, because it could not satisfy the clause $\neg\, holds(location(B, G), S)$. In particular, animal b continues to exists even after animal a visited its location. This is because animal b failed to satisfy the condition: "an animal cannot hold the same position as its predator at the same state" expressed by the axiom given above. But what triggered this error?

If we examine carefully the specification of location condition we will discover an important omission: In order for an animal to keep a particular position on the grid at a particular *State* it should not get eaten by its predator at the same *State*. This could be added to the specification by the statement: $\neg\, holds(eaten(A), State)$ as it is shown in line 16 of appendix F.1.

We close this section by summarising in the following table the *EcoLogic* cases we experimented with along with references to corresponding publications:

model used	**error devised**	**consequences**	**described in**
rabbit-grass energy flow	erroneous assignment of regulating parameters to state variables	miscalculation of state's variable value	[Kalfoglou98a]
rabbit-grass energy flow	omission of state's variable value from calculation of a flow	inappropriate proof path followed	[Kalfoglou98a]
rabbit-grass energy flow	erroneous interpretation of flow processes	miscalculation of state's variable value	[Kalfoglou98a]
rabbit-grass energy flow	erroneous specification of additional concept	erroneous result returned	[Kalfoglou98a] section 5.3 of this Thesis [Kalfoglou et.al. 00b]
state-transition	omission of condition from an animal's location definition	erroneous state transition	[Kalfoglou98a] section 5.3 of this Thesis [Kalfoglou & Robertson 99c]
state-transition	erroneous interpretation of an animal's "visited a location" concept	redundant state transitions	[Kalfoglou98a]
structural growth	erroneous strategy in implementing the passage of time during simulation	erroneous result returned	[Kalfoglou98a]

Figure 5.12: The *EcoLogic* case resources.

5.4 AIRCRAFT case

The final case presents a different situation: both the ontology and the specification to which it was applied were available beforehand, described in publicly available resources. Our aim in deploying the mechanism in such a case was to show that existing ontologies can be applied to existing specifications. However, we had to make augmentations with respect to the ontology's axiomatisation. We enriched the AIRCRAFT([Swartout *et al.* 96]) ontology's axiomatisation with the help of an editing tool originally described in [Kalfoglou & Robertson 99b] and included later in this section. This resulted in application-specific constraints which made possible the detection of artificial errors we introduced in the application. Figure 5.13 illustrates this case as an instantiation of figure 5.1.

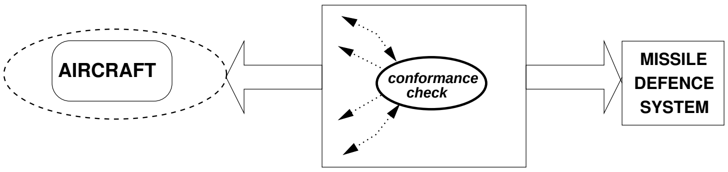


Figure 5.13: Conformance check of the Missile Defence System(MDS) application to the AIRCRAFT ontology.

This case differs from the previous ones in that it uses a pre-existing ontology and a pre-existing specification. In [Kalfoglou & Robertson 99a] we describe the case thoroughly whereas here we briefly summarise its important points. The specification we used is an implementation of the Prototype System Description Language(PSDL) simulator applied to the missile defence system(MDS), a prototype system designed to protect an allied vessel from missiles attack. The specification is documented in [Luqi & Cooke 95] along with a description of the MDS, originally suggested by Lehman in [Lehman 90], to which it was applied. The ontology we used is the AIRCRAFT ontology which was mentioned in a previous chapter(section 2.3.6).

In figure 5.14 we illustrate the MDS in two panes. Standard operators of the system are represented in circles, execution monitoring operators are represented in rounded boxes, data streams are represented as arrows that link the operators. Pane A shows the original model as described in [Luqi & Cooke 95]: two operators, *radio* and *radar* used to detect information concerning an incoming missile to the vessel's airspace which is then use the data streams *detected_missile* and *has_hit_ally* to allow the *intelligence_database* operator determine whether the missile should be regarded as a threat updating the data stream *hostile_missiles*. *Defense system* and *Defensive weapon* operators then treat the missile appropriately reading values from *threat* and *fire_control* data streams. In addition to the MDS, Luqi and Cooke in [Luqi & Cooke 95] had define two monitoring operators, *consistency* and *completeness* to detect context shifts in the specification in support of software evolution.

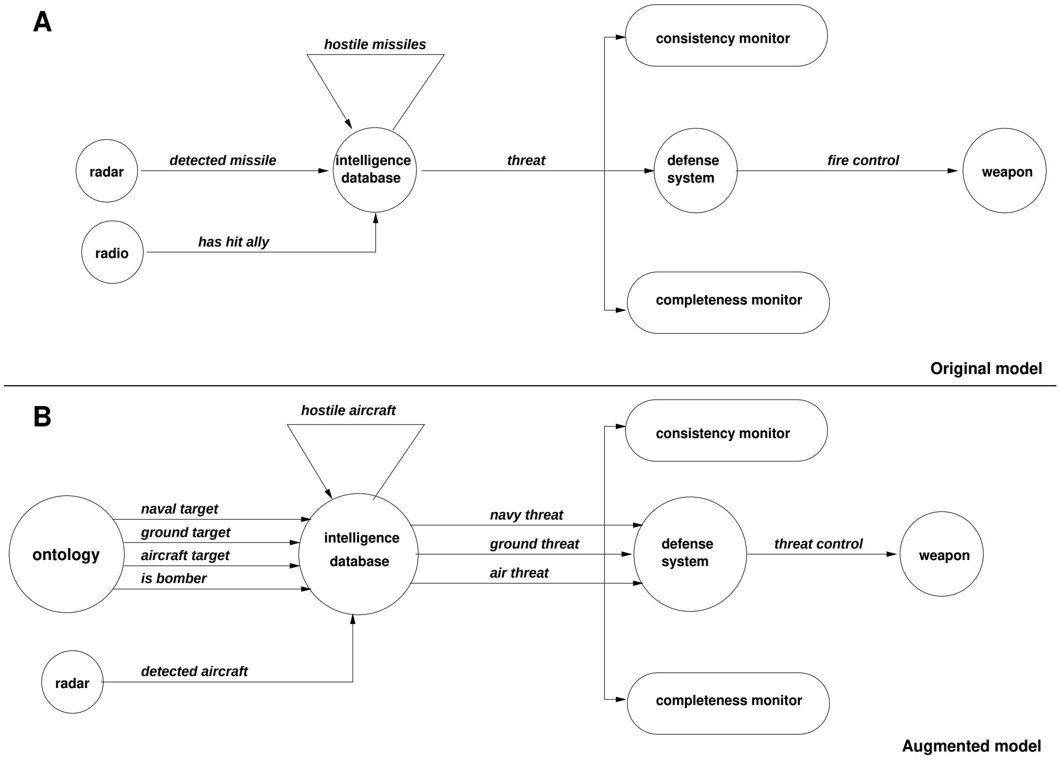


Figure 5.14: The Missile Defence System(MDS): original and augmented models.

In Pane B of figure 5.14 we illustrate the augmentations we made: we adopted the original prototype system to treat hostile aircraft instead of missiles that enter an ally's airspace. The *radio* operator which categorised missiles in the original model, was replaced by an *ontology* operator which classifies aircraft according to the AIRCRAFT ontology. This allows the *intelligence database* operator to classify the form of threat posed by the aircraft.

The use of the AIRCRAFT ontology allowed us to enrich the treatment of a threat.

We modelled three different kinds of threats(navy,ground,aircraft) in accordance with the formally defined, in the ontology, target types of the weapons that an aircraft stores. Moreover, we used the ontology's construct *bomber* to determine dynamically whether a detected aircraft, for which there is no information available about its targets, should be regarded as hostile and treated appropriately. This information belongs to the context model of the MDS and used to accumulate new contextual information. Although we could define additional predicates to construct rules that govern those determinations we used constructs of the underpinning ontology for this purpose. For example, a detected aircraft is considered a navy threat when it stores weapons that target on naval units. This can be axiomatised in the following ontological definition:

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

Note that the interpretation of the concepts *aircraft/1, stores/2* and *target_type/2* is formally given in the ontology. This lightens the workload of the specifier since it has only to define a rule that will hold over these concepts, that of *navyThreat/1.* To support this we developed an editing tool that provide assistance in constructing those rules.

It is the 'constraints editor' we mentioned earlier(section 5.2) which facilitates the construction of application-specific ontological constraints. It uses an heuristic for retrieving ontological relations as candidate parts of the constraint to be build. The taxonomy of concepts is taken into account to constrain the choices of the user in selecting candidate relations and to ensure that the maximum possible set of relations is retrieved. Apart from these relations there is a choice of augmenting the constraint with extra predicates or new relations to express complicated constraints whenever this is not possible with the available relations set. As in the relations editor of section 5.2 the distribution of variables that will be shared among the constraint's literals is done automatically.

We shall now look in detail the construction of the constraint given above to demonstrate the use of the constraints editor. In figure 5.15 we illustrate a selection of relations as described in the AIRCRAFT ontology. These are represented as rectangular boxes with the concepts that hold over stemming from them. We also include in the legend a subset of the ISA taxonomy declarations.

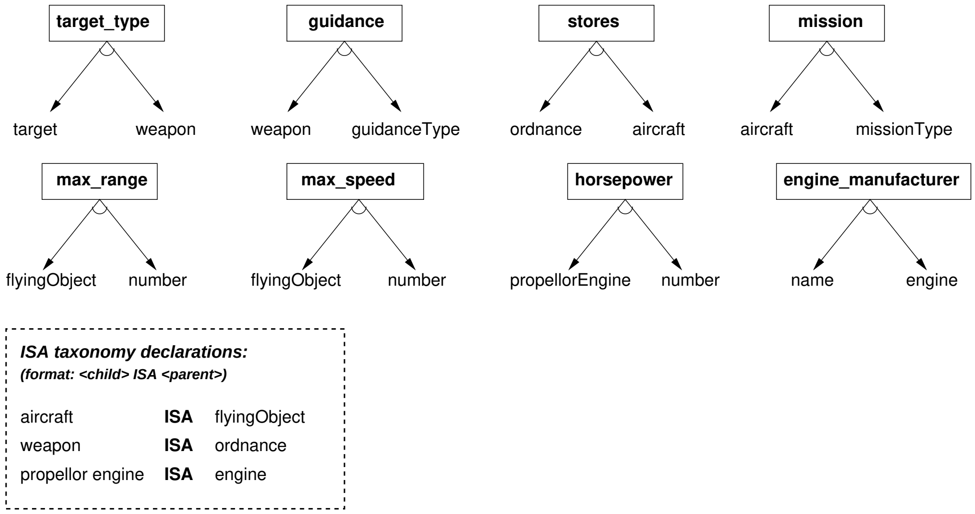


Figure 5.15: A selection of relations from the AIRCRAFT ontology.

The relations retrieval is initiated by typing a keyword which is an ontology concept chosen by the user. In our example this keyword was the concept *weapon*. As we can see from figure 5.15 the relations that hold directly over this concept are *target_type* and *guidance*. These will be the first to accumulate as candidate ones. However, by using the taxonomy of the ontology which declares that *weapon* is a type of *ordnance*, we will retrieve the relation *stores* as well. There are no more relations to be retrieved based on the particular concept, so the mechanism will proceed to retrieve potential relations based on associated keywords. Those will be the remaining concepts from the relations that already have been retrieved: *target*, *guidanceType* and *aircraft*. The same process is applied for each of the new concepts which will result in the retrieval of three more relations: *mission*, *max_range* and *max_speed*. Any new concepts that will accompany the new relations(e.g. *number*) will not be regarded as new keywords to try since this will result in retrieval of relations dissociated with the original keyword. Therefore, at this point the algorithm terminates since there are no more relations to retrieve nor there are concepts from the original retrieved relations set that haven't been checked yet.

Once the candidate relations have been retrieved the user selects the ones he wants to include in the constraint. In our case, those were: *stores* and *target_type*. The next step is define a name for the constraint to be build, in our case *navyThreat*, and the type of variables that will be used in the constraint's head. Those are selected from the ones that used in the constraint body. This step is illustrated in figure 5.16.

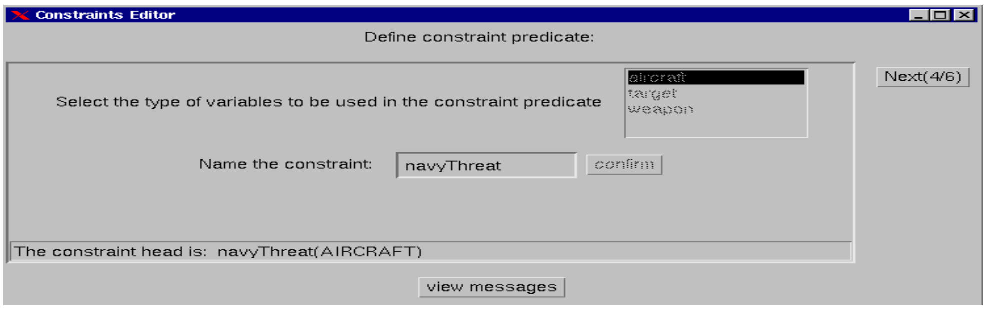


Figure 5.16: A screenshot of the constraints editor(part of the OCM tool):defining constraint's literals.

As we can see from the set of available variable types: *aircraft*, *target* and *weapon*, the user has choose to define one variable of type *aircraft* in the constraint's head. In the sequel, we bind the variables that will be shared among the relations. Again the taxonomy of concepts is taken into account to automatically bind variables that are of the same type or connected with an ISA relation. This is the case for the concept *weapon* which is child of *ordnance*. Their places in the relations will be occupied by the same variable. In figure 5.17 we include a screenshot of the constraints editor at this stage of constraint building.

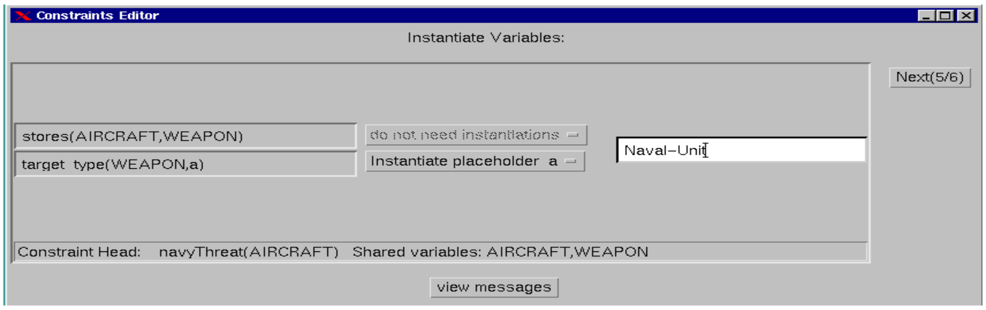


Figure 5.17: A screenshot of the constraints editor(part of the OCM tool):instantiating variables.

Only one variable has to be instantiated, the second variable of relation *target_type*. There can be an augmentation of the constraint with extra predicate or relations with respect to this variable, but in our case, this was bounded to the constant 'Naval-Unit'.

The final step is to link the constraint's literals with logical connectives($\neg, \vee, \wedge$). After the conflict and subsumption checking we described in section 5.2 is done the constraint is ready to add in the constraints base and transformed automatically in the error condition specific format:

$$\begin{aligned} error(1, navyThreat(AIRCRAFT), \neg\ (&aircraft(AIRCRAFT) \wedge \\ &stores(AIRCRAFT, WEAPON) \wedge \\ &target\_type(WEAPON,' Naval - Unit))). \end{aligned}$$

Recall from above the rule that determines whether or not an aircraft should be regarded a navy threat. Assume that the specifier of the context model, working with Horn clauses, defines erroneously that rule as follows:

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

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

According to the AIRCRAFT ontology, an F-117 aircraft carries weapons that target on ground and aircraft units but not on naval units. The following fragment of execution demonstrates the erroneous behaviour of the system, in the form of values written in the simulator's data streams:

```
..
read_from(intelligence_database,F-117,detected_aircraft)
wrote_into(intelligence_database,true,navy_threat)
wrote_into(defense_system,shootToProtectShip,threat_control)
..
```

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

However, with the ontological constraint described above, we capture this sort of error and report it as follows:

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

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

5.5 Other cases and further resources

Apart from the case studies we worked with, based on the conformance check method, we also experimented with other cases related to ontology exercise as well as elaborating on further uses of the approach. These are included below. Although they are not fully elaborated as the ones described in the previous sections and in the next chapter they focused on specific issues to facilitate ontology mapping, identify further uses of ontological constraints checking, and critiquing useful resources to support constraints building. In particular, in section 5.5.1 we experimented with a utility to facilitate ontology mapping; in section 5.5.2 we shift our attention to Object State Testing(hereafter, OST) and speculate on the role of ontological constraints checking in it; lastly, in section 5.5.3 we examine the usefulness of a rich axiomatisation that an existing ontology provides.

5.5.1 KA^2 and SHOE ontologies mapping

In this case study we worked with two publicly available ontologies: the KA^2 [5] and the SHOE[6] ontology. As we mentioned in section 2.3.7, the KA^2([Benjamins & Fensel 98]) ontology aims to represent the knowledge acquisition community(its researchers, topics, products, etc.) by annotating its Web documents in order to enable intelligent access to them. On the other hand the SHOE(Simple Html Ontological Extensions) ontology([Luke *et al.* 97]), despite its similar aims to provide a mechanism for Web authors to annotate their documents with semantic information it is not focused on a single research community. However, the ontology authors produced an exemplar

[5] Electronically accessible via the URL: http://www.aifb.uni-karlsruhe.de/WBS/broker/KA2.html

[6] Electronically accessible via the URL: http://www.cs.umd.edu/projects/plus/SHOE/

Computer Science department ontology which we used in our case. Similar to that ontology is the draft ontology proposed by the KA^2 developers.[7]

We were interested in building a mechanism of analysing information with regard to a potential mapping of these two ontologies, KA^2 and SHOE. In doing this, we did not aimed to complement and enhance current approaches to the ontology mapping problem mentioned in section 2.3.8. Rather, we compare this work with Dalianis's and Hovy's contribution([Dalianis & Hovy 98]) on revealing likely matches across concepts taken from different ontologies. However, their work was based on the underlying STEP/EXPRESS Schemata whereas in our case we worked with a hypothetical application of a University's academic department.

Assume that a specifier wants to check whether the ontological relation *researchInterest* which is common to both ontologies holds over the same concepts and if it does whether it could be mapped directly or there will be information loss. In the KA^2 ontology, the relation *researchInterest* holds over *researcher* and *researchTopic* concepts. In the SHOE ontology though, it holds over *person* and *research* concepts. We have build a mapping analysis program, which is given in appendix G.1.1, to assist the specifier in finding this information which returns the following result after asking for analysing the *researchInterest* concept:

```
Initial relationships:
researchInterest(person,research), for the SHOE ontology
researchInterest(researcher,researchTopic), for the KA2 ontology
Analysis of mapping:
researchInterest(person).
mapping of the first term with information loss for the ka2,
                                   lost concepts of ka2:  3
```

As we can see only the first concept can be mapped with information loss. In particular, the concept *researcher* of KA^2 ontology can be mapped to *person* concept of SHOE ontology. That is because in the ISA hierarchy of the KA^2 a *researcher* is defined as, eventually, a *person*. However, there is no direct connection between them in the KA^2 ISA hierarchy as a *researcher* is defined as being first *academicStaff*, then *employee* and then a *person*. The latter will allow our program to match the *person*

[7] Electronically accessible via the URL: http://www.aifb.uni-karlsruhe.de/WBS/broker/ka-onto.onto

concept of the SHOE ontology and propose the desired mapping. As far as the second concepts concerned, there is no connection between them as the *research* concept in the SHOE ontology is placed under the *work* and *shoentinty* concepts whereas in the KA^2 *researchTopic* under the *object* concept.

Our approach in analysing potential mapping of concepts is based on the available ISA hierarchies. It traverses these hierarchies in order to find matches of the given concepts. It keeps track of the concepts traversed which makes it possible to compute the information loss. If no match found, no messages are returned to indicate that mapping is not feasible.

Another way of searching for this kind of information is to look for relations that hold over user-specified concepts. For example, assume that we want to check whether there are any relations in both ontologies that hold over the concepts *person* and *organisation*. The result of executing the predicate *relation/2* described in appendix G.1.1 is given below:

```
common relation:
head(A,B) for SHOE,
employs(B,A) for KA^2 2
hold upon the same concepts:  person(A), organisation(B)
```

Two relations that hold over these concepts found: the *head* relation originating from SHOE ontology which defines a *person* as *head* of an *organisation* and the *employs* relation from the KA^2 ontology which defines that an *organisation employs* a *person*.

5.5.2 Object State Testing

In [Kung *et al.* 98], Kung, Hsia and Gao present a collection of approaches in object oriented software testing. Despite the increasing interest, this area is not fully explored yet. We will not analyse the reasons for the observed dearth of ideas and results as these are described in [Kung *et al.* 98]. Rather, we will reconstruct one of the proposed approaches for object oriented software testing using Horn logic and then argue for the role of ontological constraints checking in this fruitful area of research.

In particular, Kung and colleagues argue in [Kung *et al.* 96] for the need to test an object's behaviour. The authors observe that "an operation on an object may depend on the states of that object and other objects." This results in the so called Object State Testing. The states of an object could be defined by a subset of member data of the object class. An object changes its state when a member function is executed. In [Kung *et al.* 96] the authors presented an approach for implementing OST by capturing state dependent behaviours in an Object State Diagram(hereafter, OSD). They used a state transition machine to generate test cases with regard to object states. Two systems were presented as exemplars: a coin box of a vending machine and an implementation of a traffic light originally introduced by Leveson in [Leveson *et al.* 91].

In our case we worked with the second example, the traffic light implementation. We used a generic framework for solving problems by searching their state-space graphs presented in [Sterling & Shapiro 94] as our generic state transition machine. The implementation is given in appendix H.1 as a Prolog program. We had to make our choices with regard to the representation of states, as well as to how implement and check transitions between the states. According to the generic framework of [Sterling & Shapiro 94] these could be modelled by the predicates *move/2*, *update/3* and *legal/1*. Predicate *move/2* declares that a move is applicable to a state, predicate *update/3* checks if a new state is reachable after applying the *move/2* predicate to a state, and predicate *legal/1* checks for the validity of a state with respect to problem constraints. Their implementation as Horn clauses in Prolog format is given in lines 10 to 15 in appendix H.1. Here we focus on how we represented the states in the traffic light example.

In the traffic light(in short, TL) example, two TLs are being modelled: the *east* TL which controls the movement of vehicles in the east/west direction and the *north* TL controlling the north/south direction. Each TL has three possible modes which are coded as follows: 1 for red light, 2 for green light, and 3 for yellow light. We also use the construct *direction* to indicate the direction flow of vehicles in accordance with the TL which controls their movement(*east* or *north*). We use the following predicate to represent this information included in a list:

possible_state([`TLa(1/2/3)`,`TLb(1/2/3)`,*direction*(`east/north/_`),`nofinal/final`])

which has the following meaning: the first two arguments of the list refer to possible combinations of TLs and their modes, e.g, *east(3)* means that the *east* TL is yellow. The third argument represents the direction of a TL and the last is used to ensure termination of recursion over the possible states(by changing its value from `nofinal` to `final`).

Apart from this static information for representing states we modelled the notion of a member function which causes a TL to change its mode. For the traffic light example we modelled one such function: *setLight*. For example the following:
$memberFunction(setLight(east(1), north(2)))$.
sets the *east* TL to red and consequently *north* TL will be set to green. In addition we use two more predicates, *applicable/2* and *applyFunction/3* which check whether the member function *setLight* is applicable to a certain mode of a TL and if so then apply the function which changes the TL's mode. These predicates control the transition between states and prohibit undesired moves, e.g, changing the *east* TL from red to green without updating the *north* TL from green to red first.

We execute the program included in appendix H.1 to find a possible final state in the traffic light example and the following sequence of TL's returned: (2,1) to (3,3) to (1,2) to (3,3) and finally back to (2,1). For the sake of brevity we do not include the name of a TL in the parenthesis where the first number refers to the *east* TL and the second to the *north* TL. However, assume that the specifier erroneously defines the transitions between yellow and red/green modes. For example, a specification of the *applicable/2* and *applyFunction/3* predicates which does not takes into account the previous mode of a TL but only checks for a valid update will set a TL to red when the other is green and vice versa. Although this reasoning is sound is underspecified. Here is why: one can allow a TL to change from red/green to green/red without passing from warning yellow first. The following sequence reveals the erroneous behaviour: (2,1) to (3,1) to (1,2) to (1,3) and then back to (2,1). As we can see the *north* TL changes from red to green without passing from the warning yellow as it should.
We used the following constraint to detect such discrepancy:

```
error(1,applyFunction(setLight(_,_),[east(1),_,_,_],NewState),
                                        member(east(2),NewState)).
```

which can be given the following declarative reading: it is an error if we apply the member function *setLight* to set *east* TL to green(2), following a red(1) *east* TL. That is because we do not check for the yellow(3) mode first before changed it to either red or green.

This constraint is tailored to the particular implementation of the traffic light domain and we implemented and tested it with the detection mechanism described in earlier sections. Although we do not advocate that object oriented software developers should adopt this way of representing and testing object states we believe that this area provides insight for applying ontological constraints checking as we demonstrated with the traffic light example.

5.5.3 TOVE axiomatisation

In [Uschold *et al.* 98b], Uschold and colleagues argue that there is a dearth of rich axiomatisation in publicly available ontologies. This is the reality we faced in some of our cases as demonstrated in sections 5.2 and 5.4 were we presented an editing tool which we had to implement in order to facilitate construction of additional axioms. A notable exception is the TOVE project[8]. It provides a rich axiomatisation which could be used to support ontological constraints building or directly as constraints on time ontologies.

In appendix I.1 we include a set of 52 axioms, encoded as Horn clauses, with regard to treatment of time in TOVE ontologies[9]. These axioms, which constrain possible interpretations of time relations, are centred upon the notion of time points and time period. A time point represents an instant in time whereas a time period is a contiguous period of time.

We provide this information here as an example of what an ontology's axiomatisation should look like. However, only a few ontologies provide that rich axiomatisations. PIF and TOVE ontologies are among them but surprisingly are less used in real applications. Maybe this stems from the fact that often ontological axioms are too generic to

[8] Electronically accessible via the URL: http://www.eil.utoronto.ca/tove/ontoTOC.html

[9] The axiomatisation is provided online in the TOVE manual at the following URL: http://www.eil.utoronto.ca/tove/time/time47.html

support application-specific discrepancies. Hence the designated tools we described in earlier section which could help to alleviate the problem.

5.6 *Chapter summary*

In this chapter we applied the theoretical work presented in chapter 3 and further elaborated to implementation mechanisms in chapter 4 to several example cases. This gave us the opportunity to assess and evaluate the approach in a variety of contexts. In particular, we devised a uniform way of applying ontological constraint checking and then experimented with several example cases. These were drawn from such areas as business process modelling, ecological modelling, air-campaign planning, and we closed the chapter by elaborating on further uses in other areas. We also demonstrated, via examples, how various tools of the OCM are used in deploying the approach and argued for the importance of capturing conceptual errors. The latter were artificially introduced in our examples. However, as we explain in the next chapter this need not be the case whenever we want to apply the approach. The mechanism can actually detect and reveal real errors as we demonstrate in a complex example case in the following chapter.

Chapter 6

Evaluating the multi-layer approach

Although the multi-layer architecture is based on a single-layer approach its evaluation requires a different strategy. We no longer focus on errors that affect the definitions they misinterpret in a flat, self-contained model but we shift our attention to complex systems that are composed from several models often organised in layers and developed by different groups. It should be possible to check the behaviour of the system from the highest level of abstraction while not sacrificing the detection of subtle errors that might occur in the very detailed, lower layers of the architecture.

As in the single-layer evaluation we worked with publicly available resources to which we applied the multi-layer architecture. Although we can find numerous applications of ontologies in the literature(see, for example, the volume edited by Guarino in [Guarino 98b]) few of them realize the goal of knowledge sharing: to be composed from different ontologies(see, for example, [Valente *et al.* 99], [EIL 95], [Uschold *et al.* 98a] and [Gangemi *et al.* 98]). Even in this small set of potential candidates for our experiments it is hard to find clear definitions of the links between the ontologies being used along with technical details that implement those links and systems to which they were applied.

A notable exception is the ontologies set described in [Borst *et al.* 97] in the context of the PhysSys and `OLMECO` projects. It provides a complex inclusion lattice of seven ontologies used as the basis for two of the systems developed by the same group. The amount of implementation detail provided in that article and the complexity of the systems developed motivated our choice to use the PhysSys ontologies for evaluating the multi-layer architecture. Prior to evaluation we had to do a re-construction task which revealed some ill-defined terms in the original ontologies which we describe in the sequel.

This is the fourth case we mentioned in section 5.1 with respect to the conformance check. It is illustrated in the figure below as an instantiation of figure 5.1:

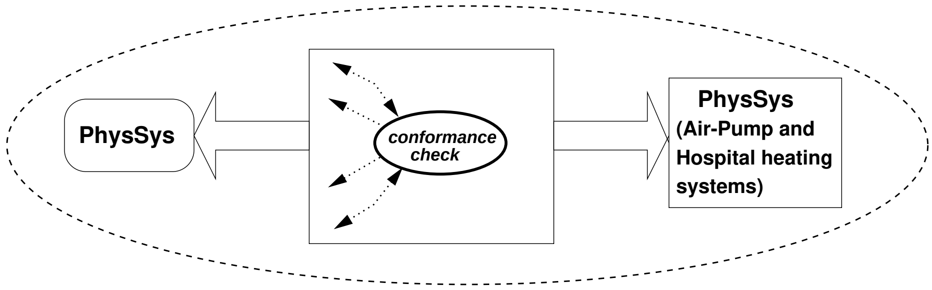


Figure 6.1: Extending the current evaluation scope and check the PhysSys ontology at the application level.

As we can see from figure 6.1 the applications used are actually derived from the ontology itself. This make it possible to extend the current evaluation scope denoted by the dashed line originally surrounding only the ontology part(figure 5.1) and now expanded to include the application part as well.

This chapter is organised as follows: we start by presenting the resources we used taken from the *"engineering ontologies"* article([Borst *et al.* 97]) in section 6.1. We then elaborate on the seven ontologies used in the PhysSys framework(section 6.2) and explore their interdependencies in section 6.2.6. The systems which we used for evaluating the ontologies are described in section 6.3 while the results of the evaluation presented in section 6.4. In particular, we elaborate on the detection of artificially introduced conceptual errors in section 6.4.1 whereas in section 6.4.2 we discuss misconceptions we discovered in the original ontologies. The implications of the multi-layer approach in ontology exercise are analysed in section 6.5. We summarise the chapter in section 6.6.

6.1 *"Engineering Ontologies"* resources

In [Borst *et al.* 97] the authors elaborate on the use of ontologies in the domain of engineering systems modelling, simulation and design. The ontology presented, PHYSSYS, is a formal ontology based upon system dynamics theory and express different conceptual viewpoints on a physical system. It consists of three engineering ontologies formalising these viewpoints: *component* ontology which represents the system layout; *process* ontology for the physical processes that underlie behaviour; and *EngMath* ontology which describes mathematical relations. The interdependencies between these ontologies are formalised as ontology projections and included in the PHYSSYS.

The *component* ontology itself is constructed from mereology, topology and systems theory. To quote [Borst *et al.* 97]:

> "In a separate ontology of mereology a *part-of-relation* is defined that formally specifies the intuitive engineering notion of system or device decomposition. This mereological ontology is then imported into a second separate ontology which introduces *topological connections* that connect mereological individuals. This topological ontology provides a formal specification of what the intuitive notion of a network layout actually means and what its properties are. The ontology of systems theory includes the topological ontology and defines concepts like(open or closed) systems, system boundary, etc., on top of it."

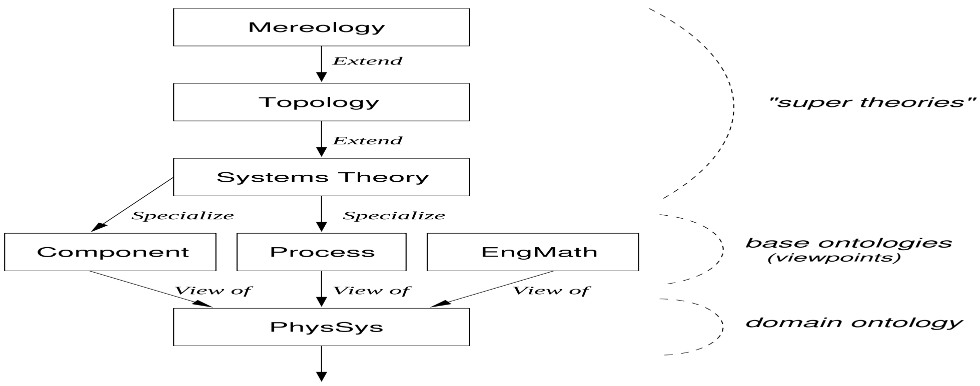


Figure 6.2: The inclusion lattice of PHYSSYS ontology.

In figure 6.2 we illustrate the inclusion lattice of PhysSys ontologies where the mereology, topology, and systems theory are 'super theories'; the component, process, and `EngMath` are base ontologies; and the produced PhysSys is a domain ontology.

PhysSys forms the base for OLMECO(Open Library for MEchatronic COmponents) which is a model component library for physical systems like heating systems, automotive systems and machine tools. It provides the foundation for designing the conceptual database schema of OLMECO. In the next section we will describe how we reconstruct the PhysSys ontologies set taken from [Borst *et al.* 97] to place them in the multi-layer architecture. This enabled us to examine them using the two systems described in [Borst *et al.* 97] presented here in section 6.3.

6.2 PhysSys ontology

All the ontologies were implemented, originally, in `Ontolingua` using the Ontology server([Farquhar *et al.* 96]). We translated them to the target language we use: Prolog. Although we could use the automatic translation provided by the server we chose to do this manually. This made it easier to preserve the syntactic elegance of the resulting Horn clauses. We rewrite, by convention, Ontolingua statements of the form: $A \leftrightarrow B$ as: $A \leftarrow B$ for definitions and $A \rightarrow B$ as ontological axioms which can be rewritten as error conditions(section 4.1). Intuitively, statements of the form $A \rightarrow B$ are already written as axioms and we had to find a way to produce elegant Horn clauses - as parts of the specification - from these statements. We elaborate on that way in the following paragraph.

Although we wanted to produce error conditions that would be consistent with the original axioms, there is no such prerequisite for the specification statements. These supposed to be part of a specification that, allegedly, conforms to the underlying ontology. Therefore the specifier can choose whichever way he wants to define his constructs. It is then up to our error checking mechanism and the available error conditions to check the conformance of those constructs to the given ontological axioms. In our re-construction task of the PhysSys ontologies, however, we chose to follow a more restricted form of specification construction. Since we were interested to utilise all of the ontologies' axioms in order to test specification statements we built a specification that is derived, whenever possible, from the available axioms. This was easy when we had a $A \leftrightarrow B$ ontological statement. That is because we split the double implication and obtain both an axiom and a specification statement. However, in situations where the ontological statement is of the form: $A \rightarrow B$ there is no guidance as to how the specification statement should be. In this case we improvise a specification statement that is tailored to the ontological axiom. To make this step clearer we treat some ontological axioms of the form $A \rightarrow B$ as double implications. This helped us not only to derive a specification statement but also to build the error conditions. That is because some of the $A \rightarrow B$ ontological statements were not appropriate for our tests: we wanted to produce an error condition with respect to a predicate that appeared in the post-condition of the original statement. Treating the statement $A \rightarrow B$ as $A \leftrightarrow B$ allow us to rewrite it as $B \leftrightarrow A$ and from here produce an error condition with respect to B. However, we are very carefull when we apply this trick because formula $A \rightarrow B$ is not logically equivalent to $A \leftrightarrow B$. We base our translation to the idea of *Predicate Completion* originally introduced by Clark in [Clark 78]. In *Predicate Completion* we can view each clause in a specification as part of the definition of a specific predicate. We can then the "only if" parts to *complete* the definition resulting in a full definition: "if and only if". We show how we use this translation in section 6.2.4 through an example case. We should also note that many of the ontological statements of that form had a composite precondition which means that we had to apply the rewrite rules described in section 4.1 in order to have a monadic predicate in the pre-condition. We will revisit those points in sections 6.2.1 to 6.2.6 where we describe the implementation details to make our case more precise.

In the sequel we describe six ontologies: mereology, topology, systems theory, component, process, and PhysSys, but not the `EngMath` ontology which also is part of PhysSys. The `EngMath` ontology is used to define the mathematics required to describe a physical process. It is documented in [Gruber & Olsen 94] but its usage is not necessary to deploy the remaining ontologies in the multi-layer architecture. Not using the `EngMath` ontology in our tests does not invalidate our conclusions.

6.2.1 Mereology - Layer 5

The top layer of the architecture, layer 5, consists of the mereology ontology. It provides definitions for mereological relations to specify decomposition and the properties that any decomposition should have. In figure 6.3 we present the original ontological definitions along with their correspondent Horn clauses masked with the `ontologicalDefinition/2` predicate described in chapter 4.

Mereology theory in *"Engineering Ontologies"*	**Prolog specification to test the Mereology ontology**
1 define-theory mereology	
2 define-class m-individual(X)	
a m-individual(X) <-> equal(X,X)	ontologicalDefinition(5,(m_individual(X) :- equal(X,X))).
3 define-relation proper-part-of(X,Y)	
a proper-part-of(X,Y) -> not proper-part-of(Y,X)	ontologicalDefinition(5,(proper_part_of(X,Z) :- part_of(X,Z))).
b proper-part-of(X,Y) and proper-part-of(Y,Z) -> proper-part-of(X,Z)	ontologicalDefinition(5,(proper_part_of(X,Z) :- part_of(X,Y), proper_part_of(Y,Z))).
4 define-relation direct-part-of(X,Y)	
a direct-part-of(X,Y) <-> proper-part-of(Y,X) and not exists Z: proper-part-of(Z,Y) and proper-part-of(X,Z)	ontologicalDefinition(5,(direct_part_of(X,Y) :- proper_part_of(X,Y), \+ (proper_part_of(Z,Y), proper_part_of(X,Z)))).
5 define-relation disjoint(X,Y)	
a disjoint(X,Y) <-> not(equal(X,Y) or exists Z: proper-part-of(Z,X) and proper-part-of(Z,Y))	ontologicalDefinition(5,(disjoint(X,Y) :- \+ (equal(X,Y); (proper_part_of(Z,X), proper_part_of(Z,Y))))).
6 simple-m-individual(X) <-> m-individual(X) and not exists Y: proper-part-of(Y,X)	ontologicalDefinition(5,(simple_m_individual(X) :- m_individual(X), \+ proper_part_of(_,X))).

Figure 6.3: Mereology ontology

The declarative reading of the Horn clauses corresponding to original definitions is the following: definition 2a realises the notion of mereologically individual. An individual `X` is a mereological individual if and only if `equal(X,X)` holds. The relation `equal(X,Y)` defines which individuals are considered to be mereologically equal. These are static definitions tailored to the system at question. Definitions 3a and 3b formalise the notion of *proper_part_of* and we devise the recursive Horn clause `proper_part_of/2` to represent them as part of the specification. Recall from the previous section where we hinted the problem of producing specification statements from $A \rightarrow B$ statements as the 3a and 3b here. We produce a corresponding Horn clause as follows: we define a base case which states that when an individual `X` is part of individual `Z` then the `proper_part_of` relation holds. The recursive definition of `proper_part_of` supplies the transitivity property. We use the `part_of/2` predicate to express static relations that hold with respect to a system that uses this ontology. Predicate `direct_part_of/2` corresponds to definition 4a and realises the direct relation of individual `X` to individual `Z` without the transitivity property taken into account. Definition 5a is written as clause `disjoint/2` which holds for individuals that are not mereologically equal or do not share a part. Finally, the `simple_m_individual/1` predicate states that an individual `X` is regarded as a simple mereological individual when it has no decomposition.

Apart from these ontological definitions we can also write down, directly from the Ontolingua syntax, ontological axioms in the form of error conditions given below as predicate `error/2`:

```
error(6,m_individual(X), ¬ equal(X,X)).
error(6,proper_part_of(X,Y),proper_part_of(Y,X)).
error(6,proper_part_of(X,Y),(proper_part_of(Y,Z), ¬ proper_part_of(X,Z))).
error(6,direct_part_of(X,Y), ¬ (proper_part_of(X,Y) ∧
                           ¬ (proper_part_of(Z,Y) ∧ proper_part_of(X,Z)))).
error(6,disjoint(X,Y), (equal(X,Y) ∨ (proper_part_of(Z,X) ∧
                           proper_part_of(Z,Y)))).
error(6,simple_m_individual(X), ¬ (m_individual(X) ∧
                                    ¬ proper_part_of(_,X))).
```

Notice that the `m_individual`, `direct_part_of`, `disjoint` and `simple_m_individual` error conditions are identical to the preconditions of corresponding definitions in the ontology. This is because each error/precondition pair was obtained by "splitting" a double implication, as described earlier. In such circumstances the definitions are guaranteed to be consistent with the error conditions but this can change if we add new definitions or adapt the existing ones. The `proper_part_of` error conditions are derived directly from original definitions 3a and 3b where they are given as ontological axioms.

According to the multi-layered architecture presented in chapter 4, the ontological definitions are placed in layer 5 while the error conditions that monitor them belong to a layer above, layer 6. In layer 5 we also found definitions of monadic predicates `equal/2` and `part_of/2` with respect to instances of the system at question. We will describe these in section 6.3.

6.2.2 Topology - Layer 4

At layer 4 of the architecture we found the topology ontology. This ontology provides the means to express that individuals are connected. Axioms ensure that only sound connections can be made. We apply the same principles to transform the Ontolingua syntax in Horn clauses. Figure 6.4 illustrates the topology ontology.

Topology theory in *"Engineering Ontologies"*	**Prolog specification to test the Topology ontology**
1 define-theory topology	
2 include-theory mereology	
3 define-class connection(C)	
a connection(C) <-> exists X,Y: connects(C,X,Y)	ontologicalDefinition(4,(connection(C) :- connects(C,_,_))).
4 define-relation connects(C,X,Y)	
a connects(C,X,Y) -> connects(C,Y,X)	ontologicalDefinition(4,(connects(C,X,Y) :- (connect(C,X,Y);connect(C,Y,X)))).
b connects(C,X,Y) -> not(part-of(X,Y) or part-of(Y,X))	
c connects(C,X,Y) and part-of(X,Z) and disjoint(Z,Y) -> connects(C,Z,Y)	
d connects(C,X1,Y1) and connects(C,X2,Y2) -> not(disjoint(X1,X2) and disjoint(X1,Y2))	

Figure 6.4: Topology ontology

As previously the correspondence of the original definitions on the left part of figure 6.4 to Horn clauses in the right part of figure 6.4 is straight forward for definition 3a: A connection `C` connects two individuals `X` and `Y`. However, definitions 4a to 4d are given as axioms so we had to devise a Horn clause to generate plausible solutions for the *connects* relation. This is the `connects/3` predicate which realises the symmetrical property that holds for the *connects* relation. It uses the predicate `connect/3` to express instances with respect to the system that uses the topology ontology. In section 6.3 we describe how these instances are defined.

The ontological constraints of this topological layer are the following:

```
error(5,connection(C), ¬ connects(C,X,Y)).
error(5,connects(C,X,Y), ¬ connects(C,Y,X)).
error(5,connects(C,X,Y), (part_of(X,Y) ∨ part_of(Y,X))).
error(5,connects(C,X1,Y1),(connects(C,X2,Y2) ∧
                           ((disjoint(X1,X2)∧disjoint(X1,Y2))
                           ∧ (disjoint(Y1,X2) ∧ disjoint(Y1,Y2))))).
```

The first two conditions are used to trap side-effects of the symmetrical property that holds for system's connections as well as invalid definitions of connections. They correspond to original definitions 3a and 4a. The third condition follows definition 4b and prohibits a part from being connected to itself or its whole. The last error condition, corresponding to definition 4d, is used to detect errors when a connection connects an entirely separated pair of individuals. It uses the mereological relation *disjoint* that has already been defined in layer 5. Note that this error condition makes it possible to check that all possible combinations of individuals of the given pairs are disjoint in contrast with the original definition 4d which neglects combinations with the second individual from the first pair. These conditions are placed in layer 5 of the architecture to monitor the topological statements of layer 4.

The definition 4c which according to [Borst *et al.* 97], is designated as an axiom to ensure that "when a part whose whole is disjoint with an individual connected to the part, the whole is also connected to that individual" is under-specified with respect to the error condition produced from that axiom. We deliberately neglect the definition and we will revisit this issue in detail in section 6.4.2 along with examples to illustrate its misbehaviour.

6.2.3 Systems theory - Layer 3

In layer 3, we place the systems theory ontology. It defines standard system-theoretic notions such as system, sub-system, system boundary, environment, openness/closeness, etc. Figure 6.5 shows the original definitions and the Prolog code of that ontology.

Systems theory in *"Engineering Ontologies"*

```
1 define-theory systems-theory
2 include-theory topology
3 define-class system(s)
a  system(S) -> m-individual(S)
4 define-relation in-system(X,S)
a  in-system(X,S) <-> proper-part-of(X,S) and
                      system(S) and not system(X)
5 define-relation in-boundary(C,S)
a   in-boundary(C,S) <->
      connection(C) and system(S) and
        exists X,Y: connects(C,X,Y) and
           in-system(X,S) and not in-system(Y,S)
6 define-relation subsystem-of(SUB,SUP)
a   subsystem(SUB,SUP) <-> system(SUB) and
      system(SUP) and proper-part-of(SUB,SUP)
7 define-class open-system(s)
a  open-system(S) <-> system(S) and
                exists C: in-boundary(C,S)
8 define-class closed-system(s)
a  closed-system(S) <-> system(S) and not open-system(S)
```

Prolog specification to test the Systems theory ontology

```
ontologicalDefinition(3,(in_system(X,S) :- proper_part_of(X,S),
                                         system(S), \+ system(X))).

ontologicalDefinition(3,(in_boundary(C,S) :- connection(C),
                                           system(S),connects(C,X,Y),
                           in_system(X,S), \+ in_system(Y,S))).

ontologicalDefinition(3,(subsystem(SUB,SUP) :- system(SUB),
                               system(SUP),proper_part_of(SUB,SUP))).

ontologicalDefinition(3,(open_system(S) :- system(S),
                                         in_boundary(_,S))).

ontologicalDefinition(3,(closed_system(S) :- system(S),
                                         \+ open_system(S))).
```

Figure 6.5: Systems theory ontology

The declarative reading of systems theory clauses is as follows: the `in_system/2` predicate corresponds to original definition 4a and holds for individuals that are in the system and are not sub-systems of it. The `in_boundary/2` predicate follows the 5a definition which requires a connection to be in the boundary of a system when it connects an individual in the system to an individual outside the system. Definition 6a is given as `subsystem/2` predicate which holds for individuals that are part of a system and must be a system themselves. The `open_system/1` predicate corresponds to definition 7a and declares a system to be open when a connection of that system belongs to its boundaries; and finally the `closed_system/1` predicate states that a system is a closed system when it is not an open system following the definition 8a. Apart from these clauses we also found constructs of the system that uses the systems theory ontology, like definitions of system instances as we will describe in section 6.3.

The ontological constraints of systems theory are:

```
error(4,system(S), ¬ m_individual(S)).
error(4,in_system(X,S), ¬ (proper_part_of(X,S) ∧ system(S) ∧ ¬ system(X))).
error(4,in_boundary(C,S), ¬(connection(C) ∧ system(S) ∧ connects(C,X,Y) ∧
                           in_system(X,S) ∧ ¬ in_system(Y,S))).
error(4,subsystem(SUB,SUP), ¬ (system(SUB) ∧ system(SUP) ∧
                           proper_part_of(SUB,SUP))).
error(4,open_system(S), ¬ (system(S) ∧ in_boundary(_,S))).
error(4,closed_system(S), ¬ (system(S) ∧ ¬ open_system(S))).
```

The first condition corresponds to definition 3a and prohibits a part which is not mereologically individual to be regarded as system. The remaining conditions are direct transformations from the given Ontolingua code used to detect possible misuse of the clauses given in the systems theory ontology. These conditions belong to layer 4 in the architecture to monitor the systems theory in layer 3.

6.2.4 Component - Layer 2

The component ontology defines the structural view of physical systems where components can have sub-components and terminals. The terminals are the interfaces of the components to the outside world. This ontology is placed in layer 2 of the architecture and depicted in figure 6.6.

Predicate `component/1` declares that mereological individuals are considered to be components in this ontology following the axiomatised definition in 3a. Recall our

Component view in *"Engineering Ontologies"*	**Prolog specification to test the Component ontology**
1 define-theory component-view	
2 include-theory systems-theory	
3 define-class component(C)	
a component(C) -> m-individual(C)	ontologicalDefinition(2,(component(X) :- m_individual(X))).
4 define-relation comp.subcomp(C,S)	
a comp.subcomp(C,S) <-> component(C) and component(S) and direct-part-of(S,C)	ontologicalDefinition(2,(comp2subcomp(Comp,SUBComp) :- component(Comp), component(SUBComp), direct_part_of(SUBComp,Comp))).
5 define-relation conn.term(CONN,TERM)	
a conn.term(CONN,TERM) and comp.term(COMP1,TERM) -> exists COMP2: connects(CONN,COMP1,COMP2)	ontologicalDefinition(2,(conn2term(Conn,Term) :- comp2term(Comp1,Term), connection(Conn), connects(Conn,Comp1,_))).
b component(COMP1) and component(COMP2) and connects(CONN,COMP1,COMP2) -> exists TERM: conn.term(CONN,TERM) and comp.term(COMP1,TERM)	
6 define-class phys-system(S)	
a phys-system(S) <-> system(S) and (in-system(C,S) -> component(C))	ontologicalDefinition(2,(phys_system(S) :- system(S), \+ (in_system(C,S), \+ component(C))).

Figure 6.6: Component ontology

method in devising the corresponding Horn clauses. We chose to treat the original definition as a double implication which allow us to define predicate `component/1` and generate component instances automatically to avoid the tedious task of defining them statically. Definition 4a is written as predicate `comp2subcomp/2` and uses the mereological relation *direct_part_of* to declare that components which are direct parts of other components are sub-components of them. The predicate `conn2term/2` is used to relate connections to terminals: components obtained through a topological connection with the *connects* relation are related to terminals given as predicate `comp2term/2`. These are static definitions tailored to the system that uses the component ontology as we will see in section 6.3. Finally, predicate `phys_system/1` corresponds to definition 6a and used to express that a physical system is a system of components.

The ontological constraints of this layer are as follows:

```
error(3,component(X), ¬ m_individual(X)).
error(3,comp2subcomp(Comp,SUBComp), ¬(component(Comp) ∧
                     component(SUBComp) ∧ direct_part_of(SUBComp,Comp))).
error(3,conn2term(Connection,Terminal), (comp2term(Comp1,Terminal) ∧
                                       ¬ connects(Connection,Comp1,_))).
error(3,conn2term(Connection,Terminal), (component(Comp1) ∧
                     component(Comp2) ∧ connect(Connection,Comp1,Comp2) ∧
                     ¬ comp2term(Comp1,Terminal))).
error(3,phys_system(S), ¬ (system(S) ∧ ¬ (in_system(C,S) ∧ ¬ component(C)))).
```

The first two conditions are written directly from original definitions 3a and 4a and used to capture misuses of *component* and *comp2subcomp* concepts. The next two conditions used to monitor the behaviour of *conn2term* relation. The first is written directly from definition 5a. As far as 5b concerned, however, we chose to treat it as a double implication. This allowed us to define an error condition with respect to `conn2term` predicate which appeared in the conclusion of the original definition. This happens because a consequence of treating a formula as a double implication is that we treat post-conditions as pre-conditions and that made it possible to identify predicate `conn2term` after breaking the composite pre-conditional part according to the rewrite rules of section 4.1. We could not do that before because we were not interested to test the predicates that belonged to the pre-condition of the original definition, which were: `component` and `connects`. The reason being, that we had already defined an error condition to test the former whereas the latter does not belong to this ontological layer. Finally, the last error condition follows definition 6a to monitor the use of *phys_system* concept. These conditions used to monitor statements of the components ontology in layer 2 and placed at a layer above, layer 3 in the architecture.

6.2.5 Process - Layer 1

The process ontology specifies the behavioural view of a physical system. It is based on system dynamics theory in which the dynamics of a system can be captured by looking at the change of different energy flows. This makes it possible to define physical behaviour in terms of these flows where physical mechanisms are applications of physical laws to one or more energy flows. The ontology is depicted in figure 6.7 along with its corresponding Horn clauses.

The declarative reading of the clauses in the right part of figure 6.7 is the following: predicate `mechanism/1` states that mechanisms are simple mereological individuals. This is given as an ontological axiom$(A \rightarrow B)$ in the original Process ontology whereas here we translate it as if there was a double implication$(A \leftrightarrow B)$ connecting the two literals, *mechanism* and *simple_m_individual*. We chose to do this in order to avoid the tedious task of writing static definitions of mechanisms as facts, with regard to the application at question, since there is nothing in the original ontology to indicate what

Process view in *"Engineering Ontologies"*	**Prolog specification to test the Process ontology**
1 define-theory process-view	
2 include-theory systems-theory	
3 define-class mechanism(M)	
a mechanism(M) -> simple-m-individual(M)	ontologicalDefinition(1,(mechanism(X) :- simple_m_individual(X))).
4 define-class energy-flow(EF)	
a energy-flow(EF) -> connection(EF)	ontologicalDefinition(1,(energy_flow(EF) :- connection(EF))).
5 define-relation ef.from-to(EF,F,T)	
a ef.from-to(EF,F,T) -> not ef.from-to(EF,T,F)	ontologicalDefinition(1,(energyFlowFromTo(EF,F,T) :- connects(EF,F,T))).
b ef.from-to(EF,F,T) -> connects(EF,F,T)	
c energy-flow(EF) and connects(EF,X,Y) -> ef.from-to(EF,X,Y) or ef.from-to(EF,Y,X)	
6 define-class process(P)	
a process(P) <-> system(P) and (in-system(M,P) -> mechanism(M))	ontologicalDefinition(1,(process(P) :- system(P), \+ (in_system(M,P), \+ mechanism(M)))).

Figure 6.7: Process ontology

the corresponding Horn clause for definition 3a should be. This allows us to generate dynamically the *mechanism* instances in the program and apply ontological axiom 3a to check its correctness. In the same fashion, we translated the original definitions 4a and 5b to their corresponding Horn clauses. The former is written as predicate `energy_flow/1` to express that energy flows are related to topological connections in a physical system. The latter, `energyFlowFromTo/3` declares an energy flow from one mechanism to another as a topological connection that connects those two mechanisms. The `process/1` predicate follows definition 6a where a process description can be defined as a system of mechanisms.

In this ontology we also include the following constraints in the form of error conditions:

```
error(2,mechanism(X), ¬ simple_m_individual(X)).
error(2,energy_flow(EF), ¬ connection(EF)).
error(2,energyFlowFromTo(EF,F,T), ¬ connects(EF,F,T)).
error(2,energy_flow(EF), (connects(EF,X,Y) ∧
                   (¬ energyFlowFromTo(EF,X,Y) ∧
                     ¬ energyFlowFromTo(EF,Y,X)))).
error(2,process(P), ¬ (system(P) ∧ ¬ (in_system(M,P) ∧ ¬ mechanism(M)))).
```

The first two conditions correspond to definitions 3a and 4a. The third condition follows the 5b definition used to trap misbehaviour of the `energyFlowFromTo/3` predicate. The next condition is a rewrite of definition 5c and used for monitoring the `energy_flow/1` predicate as does the second condition. The final condition is obtained directly from definition 6a used to check the execution of `process/1` predicate. Note that we do not use definition 5a, originally given as an axiom, in the form of error condition. We did this deliberately to avoid problems caused by the topological *connects* relation: recall from the topology ontology, the *connects* relation is formalised as a bi-directional one with respect to the parts that connects and since the *energyFlowFromTo* relation in this ontology relies on the output of *connects*, 5a could be satisfied in a logic program since we can force the *connects/3* predicate to check exhaustively the whole search space for an answer which would be inappropriate for the original 5a axiom.

These conditions are placed in layer 2 to monitor the ontological definitions of process ontology in layer 1.

6.2.6 PhysSys - Layer 0

The PhysSys ontology is used to perform projections on the three ontologies that imported to it: component, process and `EngMath`. The definitions in this ontology state that components are the carriers of physical processes that can be mathematically described with physical quantities and mathematical relations. Figure 6.8 illustrates the ontology.

PhysSys theory in *"Engineering Ontologies"*

```
1 define-theory PhysSys
2 include-theory component-view
3 include-theory process-view
4 include-theory EngMath
5 define-relation comp.proc(C,P)
a   component(C) and simple-m-individual(C) ->
        exists P: process(P) and comp.proc(C,P)
b   mechanism(M) -> exists C,P: process(P) and
        in_system(M,P) and comp.proc(C,P)
c   comp.proc(C1,P1) and comp.proc(C2,P2) and
        C1 != C2 -> disjoint(P1,P2)
6 define-relation conn.ef(C,EF)
a   conn.term(C,T1) and conn.term(C,T2) and
    comp.term(C1,T1) and comp.term(C2,T2) and
    comp.proc(C1,P1) and comp.proc(C2,P2) ->
          exists EF: conn.ef(C,EF) and
    in-boundary(EF,P1) and in-boundary(EF,P2)
b   energy-flow(EF) and process(P1) and
    in-boundary(EF,P1) and comp.proc(C1,P1) and
    process(P2) and in-boundary(EF,P2) and
                              comp.proc(C2,P2) ->
    exists C: comp.term(C1,T1) and conn.term(C,T1) and
                comp.term(C2,T2) and conn.term(C,T2)
```

Prolog specification to test the PhysSys ontology

```
ontologicalDefinition(0,(comp2proc(C,P) :- component(C),
                                            simple_m_individual(C),
                                            process(P), in_system(C,P))).

ontologicalDefinition(0,(conn2ef(C,EF) :- conn2term(C,T1),
                                          conn2term(C,T2), \+ T1 = T2,
        comp2term(C1,T1), comp2term(C2,T2), \+ C1 = C2,
                          comp2proc(C1,P1), comp2proc(C2,P2),
                                          energyFlowFromTo(EF,C1,C2),
                 in_boundary(EF,P1), in_boundary(EF,P2))).
```

Figure 6.8: PhysSys ontology.

In the original ontology, two main relations are formalised: components to processes and energy flows to connections in definitions 5 and 6 respectively. Definitions 5a and 5b axiomatise the role that *component* and *mechanism* play in the components to processes relation. In particular, 5a states that every atomic component must have a process description and 5b that each mechanism must be part of the process description of a component. As in the Process ontology, these definitions are given as ontological axioms and we translated them in a slightly different format as a compact Horn clause: recall definition 3a in Process ontology which relates *mechanism* with *simple_m_individual* this made possible to write them in one predicate, the `comp2proc/2`. It states that atomic components, which are also regarded as mechanisms, must be part of a process description. Definitions 6a and 6b used to say that energy flows between process descriptions of two components must go through a connection. In particular, 6a states that for each connection between components, the process descriptions of these components must interact via an energy flow. We rewrite this ontological axiom as predicate `conn2ef/2` which can be given the following declarative reading: A topological connection, C, is related to an energy flow, EF, if it connects two distinct terminals, T1 and T2, that are related to components C1 and C2, of which their process descriptions, P1 and P2 respectively, must include the energy flow EF. We use the predicate `energyFlowFromTo/3` to force the energy flow EF,obtained by the `in_boundary/2` predicate, be related to components C1 and C2. In addition, we used inequality checks for the terminals and components used which are missing from the original definition. This Horn clause is giving us solutions for the *connection to energy flow* relation and we deliberately neglect from our description definition 6b as it uses a different path to obtain the same set of solutions.

In this layer we also define the following error conditions as ontological constraints:

```
error(1,comp2proc(C,P),¬(component(C) ∧ simple_individual(C) ∧
                          process(P) ∧ in_system(C,P))).
error(1,conn2ef(C,EF), ¬ (conn2term(C,T1) ∧ conn2term(C,T2) ∧ ¬ T1 = T2 ∧
                         comp2term(C1,T1) ∧ comp2term(C2,T2) ∧ ¬ C1 = C2 ∧
                         comp2proc(C1,P1) ∧ comp2proc(C2,P2) ∧
                         energyFlowFromTo(EF,C1,C2) ∧
                         in_boundary(EF,P1) ∧ in_boundary(EF,P2))).
```

We did not translated the 5b axiom as it is defined over the predicate `mechanism/1` which we already check in an earlier ontological level. We also did not translated axiom 5c which is designated to check the components to processes relation because it caused problems which we describe in section 6.4.2. Similarly, we did not translated axiom 5a because even if we consider it as a double implication we cannot derive a sound error condition over the predicate `comp2proc/2`. The reason is because we want to produce the formula $A \rightarrow (B \wedge C \wedge D)$ from the available $A \wedge B \rightarrow C \wedge D$ but this is not logically sound. Therefore, we used tailored versions of constraints which do not follow strictly their corresponding axioms. This allowed us to check the specification statements by taking into account the extra constructs we used, like the `energyFlowFromTo/3` predicate in the *conn2ef* relation. As previously, these error conditions monitor ontological definitions of layer 0 and are placed at a layer above, layer 1.

6.3 Example systems

Two systems are described in this section: an *Air Pump* system and a *Hospital Heating System*, both adopted from [Borst *et al.* 97]. We depict them diagrammatically along with their static definitions in the multi-layer architecture.

6.3.1 *Air Pump* system

Figure 6.9 shows a structural-topological diagram of the *Air Pump* system. It emphasizes the topological connections of the system. Sub-components are drawn inside the area defined by their super-component. The small solid blocks are the interfaces through which components are connected.

Apart from the clauses described in the previous section, we need to express system-specific definitions to operationalize and execute the PhysSys ontologies set.

In the mereological layer, layer 5 in the architecture, we found the following predicates: `equal/2` and `part_of/2`. These are used to express mereological equality of a part and membership relationship, respectively. They have the following format:

```
specification(5,(equal(coilMagnet,coilMagnet):-true)).
specification(5,(part_of(coilMagnet,pump):-true)).
```

and use the clause `specification/2` to distinguish them from ontological definitions. In this system the: coilMagnet, lever, bellows, valve1, reservoir, valve2, airLoad, airSupply, powerSupply, pump and airPump are defined with the `equal/2` predicate as

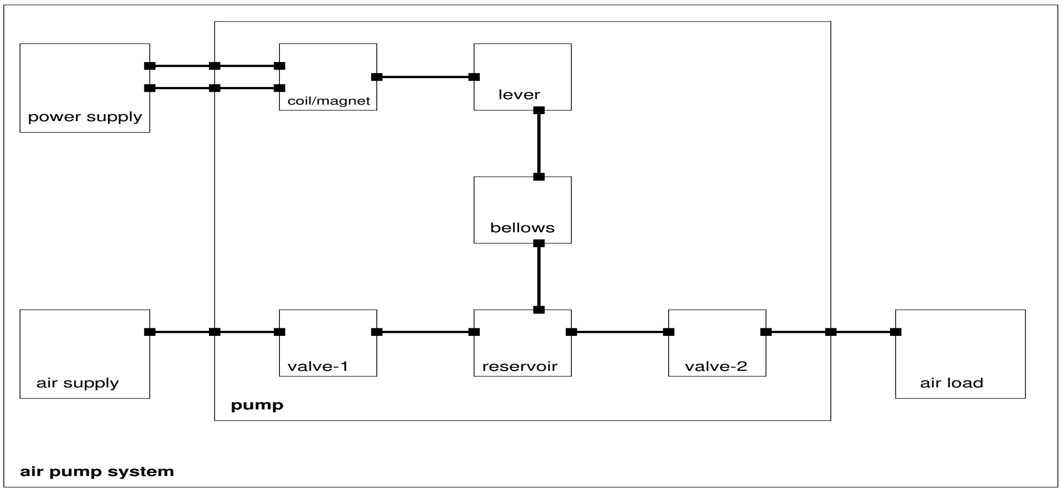


Figure 6.9: The structural-topological diagram of the *Air Pump System*.

shown above whereas the first six are regarded as part of pump using the `part_of` predicate; pump, airSupply, airLoad and powerSupply are all parts of airPump.

In layer 4, topological layer, we use the predicate `connect/3` to express static connections between components and their relative connector. A definition in this layer looks like this:

`specification(4,(connect(valve1_X_reservoir,valve1,reservoir):-true)).`

and the connectors defined in this way are: valve1_X_reservoir, reservoir_X_valve2, bellows_X_reservoir, lever_X_bellows, coilMagnet_X_lever, airSupply_X_valve1, powerSupply_X_coilMagnet, airLoad_X_valve2. All of them connect the components identified from their description except for the last three that also connect airSupply, powerSupply and airLoad to pump, respectively.

The only system-specific definitions in systems theory ontology, layer 3, are the declarations of pump, powerSupply, airLoad, airSupply and airPump as systems using the `system/1` predicate.

In layer 2, components ontology, we found the predicate `comp2term` which used to express components to terminals static relations. These are: coilMagnetT terminal for coilMagnet component, leverT for lever, and so on for the remaining of the air pump system components.

There are no more system-specific definitions in the remaining ontologies, process in layer 1 and PhysSys in layer 0.

6.3.2 ***Hospital Heating*** system

The system illustrated in figure 6.10 is a component view of a heating system in a general hospital in Schiedan, The Netherlands([Borst *et al.* 97]):

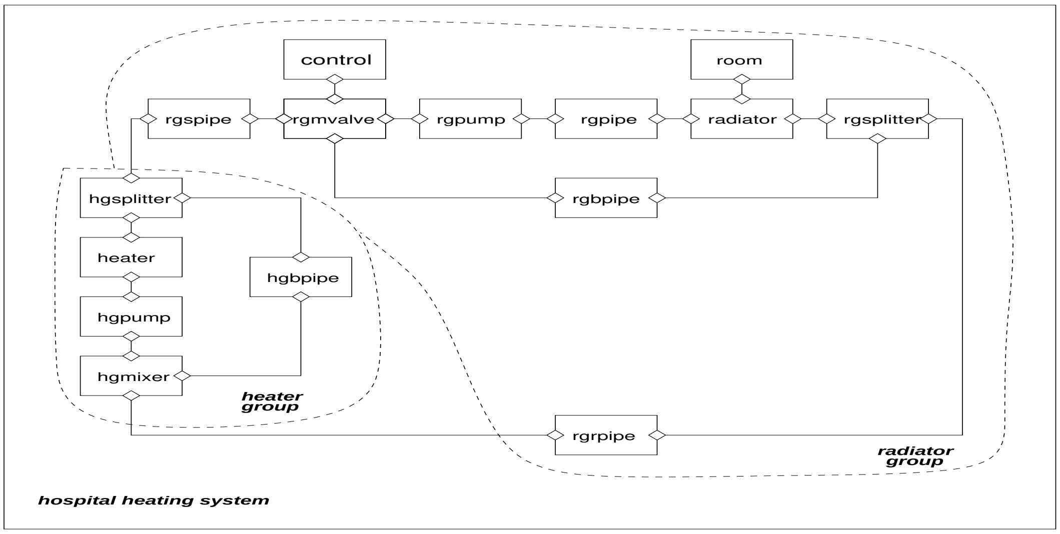


Figure 6.10: The component view of the *Schieden hospital heating system.*

The system consists of two subsystems: one around the radiator and the other around the heater(*radiatorGroup* and *heaterGroup* in figure 6.10). Its operation is briefly depicted in [Borst *et al.* 97] as follows: "This system heats up the water up to a desired temperature regulated by the radiator. When the heater is on, its temperature first increases quickly because the water that flows into it is of almost the same temperature as the water that flows out of it. As the temperature increases the amount of heat that flows out of the heater will become larger than the heat carried by the water that flows into it. This will cause the radiator to become hot, and the temperature of the water from the radiator group return pipe(*rgrpipe* in the diagram) will increase and so will the temperature of the water that flows back into the heater. This will cause the heater temperature to increase at a constant rate until the maximum heater temperature is reached and the heater is switched off."

As previously, in the mereological layer 5, predicate `equal/2` is used to express system-specific instances. We do not list them here for the sake of brevity but they can easily be identified as the square boxes in figure 6.10 plus the *heaterGroup* and *radiatorGroup* systems. The same applies for the `part_of/2` declarations where we identify from figure 6.10 ten components be part of *radiatorGroup* and five part of *heaterGroup*. These two are then part of the *hospital heating system*

In layer 4 the predicate `connect/3` is used to define the following connectors between the corresponding components identified from their name: rgspipe_X_rgmvalve, rgmvalve_X_rgpump, rgmvalve_X_control, rgpump_X_rgpipe, rgpipe_X_radiator, radiator_X_rgsplitter, radiator_X_room, rgsplitter_X_rgbpipe, rgsplitter_X_rgrpipe, rgbpipe_X_rgm valve, rgspipe_X_hgsplitter, rgrpipe_X_hgmixer, hgsplitter_X_heater, heater_X_hgpump, hgpump_X_hgmixer, hgbpipe_X_hgsplitter and hgbpipe_X_hgmixer.

In the systems theory layer, *radiatorGroup*, *heaterGroup* and *hospital heating system* are defined as systems using the `system/1` predicate.

In the component ontology layer, we use the `comp2term` to define the terminals that are related to system's components. These are given in the same format as in the previous system with a 'T' attached at the end of a component's name.

6.4 Evaluating the ontology

Firstly, we introduced artificial conceptual errors in various parts of the architecture and then used the error detection mechanism to check whether we can detect them. This is described in 6.4.1 with example cases from both systems. In 6.4.2 we describe how the translation of original ontological definitions to the executable error condition format led us to discover ill-defined concepts in the original set of PhysSys ontologies.

6.4.1 Detecting conceptual errors

This layered architecture can be executed to check whether various properties of the systems described in 6.3.1 and 6.3.2 hold with respect to the ontological definitions given in their respective layer.

So, for example, we can check for specific mereological properties with respect to the

Air Pump by asking whether *lever* is disjoint from *airLoad* giving the relevant Prolog query. As we can see from figure 6.9 we will get a, correct, positive answer.

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

From the systems theory point of view, which includes topological and mereological definitions, we can check which systems are considered to be closed systems with respect to the *Air Pump* system. The answer set will consists of the: *powerSupply, airSupply* and *airLoad* systems.

In the component layer, we can check which connections are related to terminal *rgbpipeT*(radiator group bypass pipe terminal) regarding the *Hospital Heating System*, by executing the *conn2term* relation. The answer will consists of the two possible connections: *rgsplitter_X_rgbpipe* and *rgbpipe_X_rgmvalve*.

In the process ontology, we check for energy flows with respect to the *Hospital Heating system* by using the *energy_flow* ontological definition. A set of answers produced where connections like the *rgspipe_X_rgmvalve* are returned as energy flows.

The correspondences between connections and energy flows can be found by executing the ontological definition *conn2ef* in the most abstract layer of the architecture, layer 0. So, if we ask which connection corresponds to energy flow *airSupply_X_valve1* with respect to the *Air Pump* system the answer will yield the same value as an energy flow.

However, assume that at the mereological layer, layer 5 of our architecture, the ontologist makes the following erroneous definition:

```
ontologicalDefinition(5,disjoint(A, B) ← ¬ proper_part_of(A, B)).
```

This definition states, erroneously, that for two individuals A and B, if A is not a mereological part of B, then the *disjoint* relation holds.

We can detect this sort of error at the mereological layer where it occurs by checking this layer's definitions against their error conditions. This is feasible with the multi-layer architecture since we can define the layer from which to start checking. So, if we ask whether *bellows* is disjoint from *bellows* we will get an, erroneous, positive answer. With the error condition defined over the *disjoint* relation(given in section 6.2.1) the error is detected and reported:

```
error_condition_satisfied(6,disjoint(bellows,bellows),
                    (equal(bellows,bellows) ∨
                    proper_part_of(Z,bellows) ∧ proper_part_of(Z,bellows)))
```

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

We can extend this layer checking to include the topological layer, layer 4 in the architecture. So, for example, although the connection/component couple (*airSupply_X_valve1, airSupply*) is the correct answer to the relevant query given above, the hidden error on *disjoint* relation is trapped and reported:

```
error_condition_satisfied(6,disjoint(airSupply,airSupply),
             (equal(airSupply,airSupply) ∨
             proper_part_of(Z,airSupply) ∧ proper_part_of(Z,airSupply)))
error_condition_satisfied(6,disjoint(pump,pump),
                        (equal(pump,pump) ∨
                        proper_part_of(Z,pump) ∧ proper_part_of(Z,pump)))
error_condition_satisfied(5,connects(airSupply_X_valve1,airSupply,pump),
             (connects(airSupply_X_valve1,airSupply,pump) ∧
             (disjoint(airSupply,airSupply) ∧ disjoint(airSupply,pump)) ∧
             disjoint(pump,airSupply) ∧ disjoint(pump,pump)))
```

Three errors have been detected: two at the mereological layer with respect to the erroneous definition of *disjoint*, and one at the topological layer where the condition defined over the *connects* relation was proved by the meta-interpreter. In particular the `disjoint(airSupply, airSupply)` and `disjoint(pump, pump)` that belong in the condition of *connects* relation are erroneous and reported at the layer above.

If we move one layer up, in systems theory level, the error is still detectable when we check the query on closed systems given above. The *powerSupply* is reported correctly as a closed system but the presence of the erroneous definition of disjoint revealed by the meta-interpreter. The three errors detected are the same as in the previous example and we do not repeat them here. However is interesting to see how we reached to the detection of the error from the *closed_system* relation: recall the error condition over *closed_system* relation from 6.2.3 we see that it uses the *open_system* relation. This in turn, uses the *in_boundary* relation which ultimately uses the *disjoint* where the error occurred. The multi-layer architecture allows for check in the error conditions themselves and this enabled the detection of *disjoint* relation misuse.

In the component ontology level, layer 2 in the architecture, we found the correct answer to the query on *conn2term* relation by revealing the hidden error. The error is the same since it is traceable from the error conditions defined over the *conn2term* relation. Recall from 6.2.4 these conditions use the topological definition of *connects* where the *disjoint* first appears. The following fragment of the error detection report shows the path followed to the *disjoint* relation when trying to prove that *rgspipe* is disjoint from itself:

```
error_condition_satisfied(6,disjoint(rgspipe,rgspipe),
            (equal(rgspipe,rgspipe) ∨
            proper_part_of(Z,rgspipe) ∧ proper_part_of(Z,rgspipe)))
path:  [connection(rgspipe_X_rgmvalve)|
                          (4,connects(rgspipe_X_rgmvalve,rgspipe,rgmvalve))]
```

In the process ontology we introduce another artificial error to check the behaviour of the mechanism with multiple error occurrences. The original definition of *mechanism* states that all *simple_m_individuals* are regarded to be mechanisms. However, assume the following change in the ontology:

```
ontologicalDefinition(1,mechanism(M) ← component(M)).
```

which erroneously states that mechanisms are dependent on components.

This error can be trapped by checking the definition of *mechanism* at this layer but we postpone it for the next layer. However, we execute the *energy_flow* query given above. The hidden error of mereology is detectable as before since an *energy_flow* relation is dependent on the *connection* definition in topology which gives access to the erroneous *disjoint* relation.

Finally, the last check is from the highest layer of abstraction in the architecture: the PhysSys layer. The query we were trying to answer is which energy flow is related to connection *airSupply_X_valve1*. The correct answer is yielded, along with the hidden errors. The errors were traceable because the definition of *conn2ef* is dependent on other relations like, the *conn2term*(component ontology) and *in_boundary*(systems theory ontology), which provide error conditions defined on the erroneous *disjoint* and *mechanism* concepts. For example, the following excerpt from the error detection output shows the *mechanism* error captured from the *comp2proc* relation and its dependent concept, *process*, which gives access to *mechanism*:

```
error_condition_satisfied(2,mechanism(pump),
         ¬ simple_individual(pump))
path:  [process(airPump) |
         (1,system(airPump),mechanism(pump),in_system(pump,airPump))]
```

6.4.2 Discovering ill-defined terms

In our challenging task to produce elegant Horn clauses from the given definitions in the PhysSys ontologies set we faced situations where the original definitions didn't had a direct counterpart in Horn logic or were ill-defined. While we described some of the former cases in previous sections we focus here on the latter: how the transformation of original definitions to the designated executable error condition format made it possible to discover serious omissions which we report below:

In section 6.2, in the description of topology ontology we raised the issue of under-specification of definition 4c with respect to the *connects* relation. In the sequel we explain our observations by using instances of the *Air Pump* system described in the previous section for clarity.

As mentioned in section 4.1, we are interested in clauses with monadic predicates in the precondition, so we apply the following rewrite rules:

$$A \wedge B \wedge C \rightarrow D \Rightarrow \neg(A \wedge B \wedge C) \vee D \Rightarrow \neg A \vee \neg B \vee \neg C \vee D \Rightarrow A \rightarrow \neg B \vee \neg C \vee D \Rightarrow A \rightarrow \neg(B \wedge C \wedge \neg D)$$

to break the conjunctive precondition of definition 4c which is now written as:

(1) connects(C,X,Y) $\rightarrow \neg$(part_of(X,Z) $\wedge$ disjoint(Z,Y) $\wedge \neg$ connects(C,Z,Y))

this is then transformed to its equivalent error condition according to the error theory presented in chapter 4:

error(connects(C,X,Y), (part_of(X,Z) $\wedge$ disjoint(Z,Y) $\wedge \neg$ connects(C,Z,Y))).

which is interpreted as follows: we have an error whenever a connection C, connects two parts, X and Y, of whom X is part of another part, Z, and that part is disjoint from part Y, and connection C does not connect parts Z and Y.

Let us examine this condition with respect to the *Air Pump* system illustrated in figure 6.9: if we check the validity of this relation with respect to connections that connect the parts *powerSupply*, *airSupply* and *airLoad* - which lie outside the borders of *pump* system - with parts of the *pump* system we found no problems since the definition of *disjoint* relation ensures that the 'outside' parts are disjoint with *pump*.

However, if we apply the condition to parts of *pump* system, like for example, *valve1* and *reservoir*, we found the following erroneous behaviour, after firing the meta-interpreter with the relevant Prolog query:

```
error_condition_satisfied(5,connects(valve1_X_reservoir,valve1,reservoir) ∧
           (part_of(valve1,pump) ∧ disjoint(pump,reservoir) ∧
           ¬ connects(valve1_X_reservoir,pump,reservoir)))
```

As we can see the condition is proved by the meta-interpreter although a quick look at the *Air Pump* system(figure 6.9) shows that *valve1_X_reservoir* is the right answer. Why this had happened? There are two plausible answers to this question: the first one suggests that the authors have under-specify the definition of *disjoint* relation. Although it prohibits two parts that are equal or do share a part to be regarded as disjoint no special care is taken when one of these parts is a sub-part of the other. For example, the *reservoir* part accounts as disjoint from *pump* system since it satisfies the conditions of *disjoint* relation. This has a bad effect in the condition of the *connects* relation since it allows the theorem prover to satisfy the *disjoint* relation and proceed to prove that *valve1_X_reservoir* connects *pump* with *reservoir* which is not the case as can be seen from figure 6.9, thus satisfying the condition and reporting, erroneously, the occurrence of an error.

The second answer suggests that definition 4c needs more careful consideration. As we can see from the transformations above even if we use formula (1) which a logical consequence of 4c after applying a number of rewrite rules no instance of the *Air Pump* system will satisfy the relation.

We faced similar problems with the definition 5c of the PhysSys ontology. That definition is designed to check the components to processes relation and ensures that a mechanism can only be part of one process description of one component.

As previously we are interested in clauses with monadic predicates in the precondition and we apply the rewrite rules given above to break the conjunctive precondition of original definition 5c which is now written as:

(2) comp2proc(C1,P1) $\rightarrow$ $\neg$(comp2proc(C2,P2) $\wedge$ $\neg$ C1 = C2 $\wedge$ $\neg$ disjoint(P1,P2))

this is then transformed to its equivalent error condition according to the error theory presented in chapter 4:

error(comp2proc(C1,P1), (comp2proc(C2,P2) $\wedge$ $\neg$ C1 = C2 $\wedge$ $\neg$ disjoint(P1,P2))).

the interpretation of which is as follows: we have an error whenever a component C1 is related to a process description P1, and there exists component C2, which is different from C1 and related to a process description P2 and these processes, P1 and P2, are not disjoint.

We check this condition with respect to the *Hospital Heating System* depicted in figure 6.10. If we consider that there are only two systems present, the *radiator group* and *heater group*, and try to prove that component *rgspipe*(radiator supply pipe) is related to process *radiatorGroup*, then we have an occurrence of an error which reported as follows:

```
error_condition_satisfied(1,comp2proc(rgspipe,radiatorGroup),
            (comp2proc(rgmvalve,radiatorGroup) ∧ ¬ rgspipe=rgmvalve
            ∧ ¬ disjoint(radiatorGroup,radiatorGroup)))
```

However, a quick look at the *Hospital Heating System*(figure 6.10) will reveal that there is nothing wrong with this query. In fact, *rgspipe* is legally related to process *radiatorGroup* since it is a part of it. But the definition 5c, given as an error condition, was proved because another component found, *rgmvalve*(radiator group mixing valve) which is not the same but has the same process description, the *radiatorGroup* and therefore does not qualify for the disjoint relation.

If we closely examine definition 5c we see that there a serious omission: the process descriptions P1 and P2 should not be equal before testing them for disjointness. This will alleviate the situation above but not entirely: when we have 'super systems' as part of the model then all components in their subsystems will not qualify for the *comp2proc* relation. For example, if we regard the *Hospital Heating System* as system, which entails *radiatorGroup* and *heaterGroup* systems, this will cause the condition to be satisfied: all components of *radiatorGroup* or *heaterGroup* have two process descriptions, one for the system that they belong to and one for the 'super system', the *Hospital heating system.* However, *radiatorGroup* or *heaterGroup* are not disjoint with *Hospital heating system* because they do share parts, thus satisfying the error condition for *comp2proc* relation. As previously we observe that in this case the disjoint relation needs more consideration with respect to the system/subsystem dependencies between the parts we want to check for disjointness. In the context of our case, a possible remedy to this would be to rewrite the conditional part of `disjoint/2` predicate given in section 6.2.1 as follows:

$$\neg\ (equal(X,Y) \vee (proper\_part\_of(Z,X) \wedge proper\_part\_of(Z,Y)) \vee$$
$$(proper\_part\_of(X,Y) \vee proper\_part\_of(Y,X))))).$$

This will prevent two parts, X and Y, which are related with a system/subsystem dependency to be checked for disjointness.

The detection of these under-specified definitions in the original PHYSSYS ontology set highlights the need for evaluating ontologies prior to releasing and using them. However, it also emphasizes the point that this evaluation should, ideally, done on exemplar systems where the actual ontology will be used. Although we didn't try we are not aware of other ways for detecting this sort of discrepancies apart from the method we followed: use ontological definitions in the exemplar systems and then execute them to monitor their behaviour. The efficiency of this method is that we take into account application specific design choices, like the Horn clauses we devised in our experiment, and thus performing a more pragmatic evaluation of the ontology in real applications instead of evaluating it at the development stage where no such choices are considered. It also gives an insight of possible extensions or alterations in order to meet specific applications needs. The disadvantage is on the dearth of available exemplar applications. Apart from those of PhysSys few publicly available ontologies provide examples of their use(i.e., in the PIF manual [Lee *et al.* 96]).

6.5 The multi-layer perspective

Using the multi-layer approach to reconstruct the PhysSys ontologies set provided us with advantages which we list below:

- the existence of an indexing mechanism to separate the different ontologies of the PhysSys set made them easier to understand and operationalize. We are able to execute each layer separately without sacrificing the notion of the overall structure that it belongs to;
- we are able to detect and locate discrepancies easier than before which facilitates the exercise of complex ontological structures;
- there is no upper bound defined in the multi-layer architecture. This makes it possible to extend current ontological structures by adding an arbitrary number of new ontologies to which we can apply the same error checking procedure to verify their correctness.

To demonstrate the complexity of error checking in a multi-level ontological structure as the PhysSys we illustrate in figure 6.11 the detection of the two errors occurred during the execution of the *conn2ef* relation described in 6.4.1:

In the left part of the figure we include the identification number of a layer, 0 for the PhysSys ontology, 1 for the Process and so on. In the middle we illustrate the detection of erroneous definitions, *mechanism* and *disjoint* starting checks from relation *conn2ef* in layer 0. The dashed lines starting from the subgoal of *conn2ef* denote the path followed in order to detect the errors. Those were occurred in layers 1 and 5 for

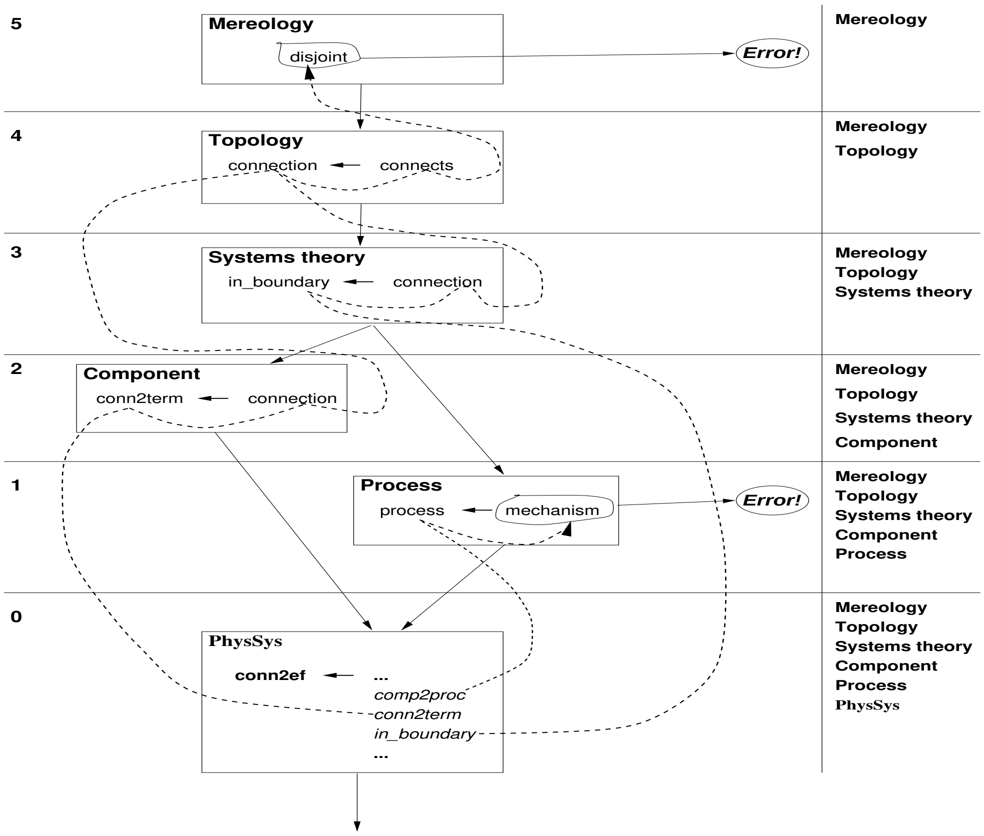


Figure 6.11: The multi-layer approach applied in the PhysSys ontology set.

the *mechanism* and *disjoint* respectively. The right part shows which ontologies are included in the error check at a given layer.

6.6 *Chapter summary*

In this chapter we presented a worked example of how the multi-layer architecture described in section 4.6) is used. We dealt with the PhysSys ontology set which we reconstructed to use in two exemplar applications. We highlighted the advantages of the approach in section 6.4.2 where we described the detection of real errors in the original ontologies. We illustrated the ease of deploying the multi-layer architecture by explicitly analysing translation steps and explaining design choices in sections 6.2.1 to 6.2.6. We applied the reconstructed ontologies in two exemplar applications(section 6.4.1). This made it possible to evaluate the ontologies at the application level. The complexity of this sort of evaluation was discussed in section 6.5. The discovery of real errors in the original ontology was the main driver for the work we present in the next chapter: how to evaluate and preserve the results of evaluation for later consultation in an automated manner. We borrowed ideas from the SE community to accomplish this goal.

Chapter 7

Extensions

In this chapter we discuss extensions to our work. In particular, we were interested in finding ways to improve ontology management. Our work in section 7.1 shows how we deployed an idea, originally conceived and implemented in the SE community, to manage experiences collected during ontology testing and deployment. We elaborate on the implications of this idea and further explore the role of ontologies in knowledge management in section 7.2. Lastly, as a means to augment the facilities provided by the tools described in earlier chapters, we employed techniques from the area of automated program synthesis to aid the construction of ontology-based specifications. These are discussed in section 7.3.

7.1 Managing experiences

As we discussed in section 2.3.8 there is dearth of solutions and tools for ontology maintenance. Despite the fact that ontology development tools cover a wide spectrum of issues and variety of them is available, little work has been done in the areas of deployment, evaluation, and maintenance. To deploy ontologies correctly we need not only self-contained, reuse-oriented tools and technologies as for example in the HPKB([Cohen *et al.* 98]) and IBROW([Fensel *et al.* 99]) projects. It is also necessary to record and organise our experiences in having applied them in order to improve future ontology deployment. It is hard to gain this sort of experience from the literature because few cases of comprehensive ontology reuse and deployment on a large scale are reported([Uschold *et al.* 98a], [Borst *et al.* 97], [Valente *et al.* 99]). Even those which are reported do not normally discuss the hidden assumptions and tradeoffs identified during testing.

In this thesis, so far, we presented a novel way of deploying ontologies which make it possible to evaluate them when applying them. In this section we will present how we can manage the experiences collected during this sort of deployment. Most of the material presented here originates from [Kalfoglou & Robertson 00] where Kalfoglou and Robertson discuss how experience management as practised in the SE community through the use of Experience Factories(hereafter, EFs) and their constituents Experience Bases(hereafter, EBs) can facilitate ontology deployment.

In the late eighties([Basili & Rombach 88]) and early nineties([Basili *et al.* 94]), EFs were investigated as a means to promote reuse of "all-kinds" of artefacts in a software organisation. In figure 7.1 we illustrate the organisation of an EF as implemented in [Althoff *et al.* 99b].

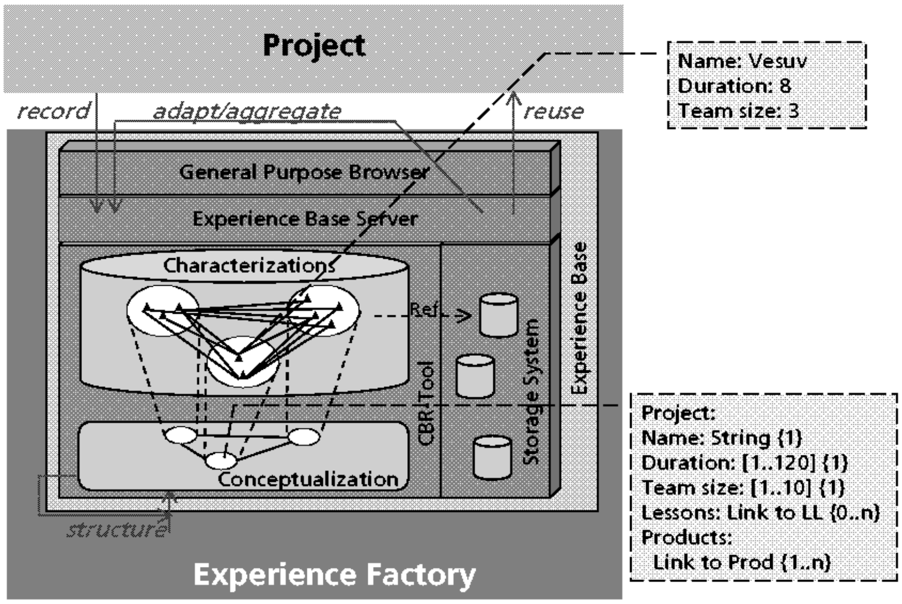


Figure 7.1: Main EF tasks and EB architecture.

This figure shows the main EF tasks and diagrammatically analyses the EB architecture. The EB structure consists of conceptualisations of experiences and their characterisations. Experiences are accessible through a general browser which facilitates the task of recording and reusing them. The details of implementation are given in [Althoff *et al.* 99b]. Here we elaborate on the idea behind EFs and EBs. The core of an EF is the EB which acts as an organisational memory. The key idea is to install an organisational memory to support exchange of all kinds of experiences in the life-cycle of a software project. This approach supports 'learning from experience' on a technology-independent organisational level. An EF stores the collected experiences in an EB. In [Althoff & Wilke 97] it was argued that Case-Based Reasoning(hereafter, CBR) plays an important role in the EF paradigm. As CBR provides both the technology and a methodology for 'learning from experience' in the context of case-based knowledge systems([Althoff *et al.* 99a]), it was natural to use it for implementing continuous learning in an EF style. CBR was also deployed for managing the retrieval and adaptation of experiences in an EF([Althoff *et al.* 99b]).

In the following section we focus on a particular aspect of managing experiences: how *experienceware*(i.e., EFs and their constituents EBs) can be deployed to facilitate ontology deployment and in particular, ontology testing.

7.1.1 Using EFs in ontology deployment

We argue that *experienceware* can be useful in ontology development and deployment as a way of managing the experiences collected from various agents participating in ontology building and usage. To make this idea more concrete, we built a simple architecture centred upon the notion of ontology verification. The implementation details are discussed in the sequel while here we focus on its operation. The architecture is given diagrammatically in figure 7.2.

The left-hand side of figure 7.2, depicts the task of verification. In particular, we are interested in verification of ontologies at the application level as we presented in the previous chapter. After applying our verification mechanism we accumulate, temporarily, the results in an EB. These are code-testing results and we regard them as experiences. The EB is then imported by an experiences editing tool which allows for further additions and modification of the description of existing experiences. It allows us to express information usually not obtainable through code-testing. We then select the experiences we want to validate and send them to a designated tool

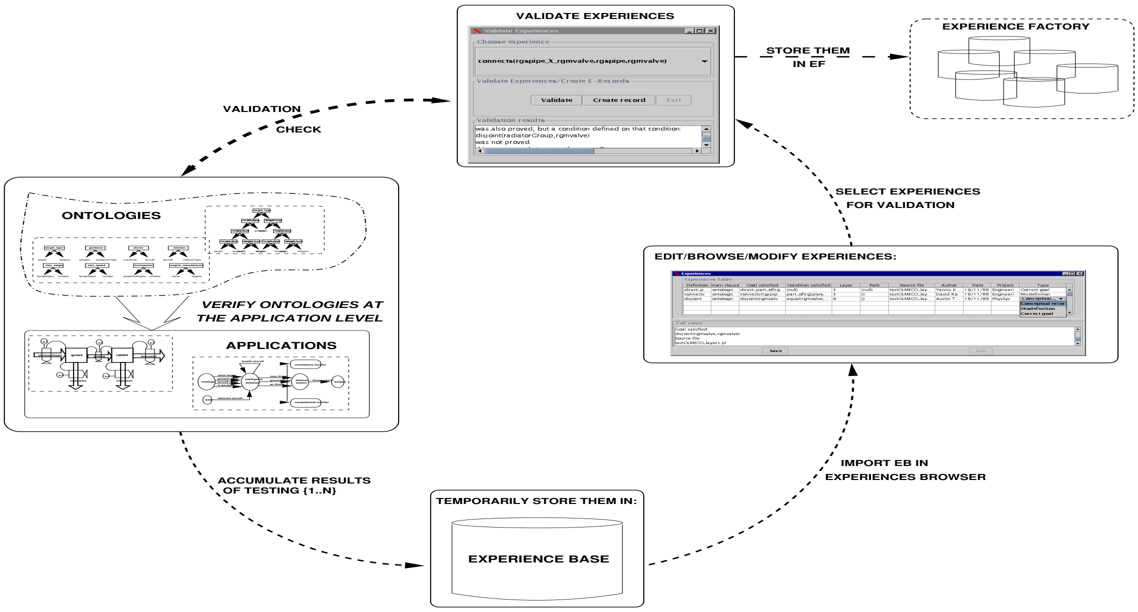


Figure 7.2: Experience-based architecture to support ontology verification.

for verifying their correctness with respect to test results. This tool embodies the verification mechanism we deploy in the first step but here we apply it to verify the correctness of the results themselves. After the selected experiences have been validated we store them in the final EB to be part of an EF.

This cycle can be repeated as many times as we wish in the same or other ontologies to collect and manage the knowledge accumulated during verification and testing. Ultimately, this will result in an EF of ontology verification and testing that can be deployed to similar projects in order to facilitate ontology use.

We tested this architecture using the PhysSys ontologies set as described in the previous chapter. Details of the structure of the PhysSys ontology are given in section 6.1 and the errors we found are explained in sections 6.2 to 6.4.2. In the rest of the current section, we focus on the tools comprising the architecture for managing and validating experiences and explain their operation by reference to the error cases discussed in the previous chapter.

Central to the system is a table which allows us to add extra information regarding the tests. A screenshot of the *experiences table* is given in figure 7.3.

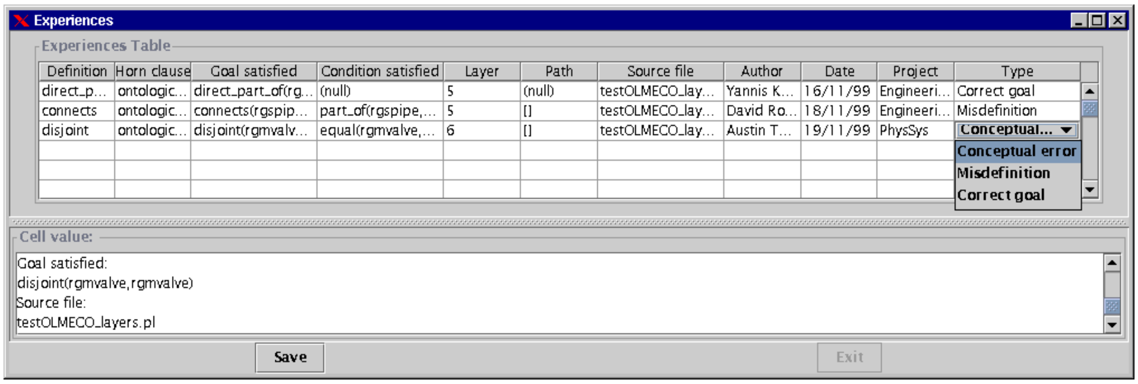


Figure 7.3: The Experiences Table.

Using this table we can add information with regard to the *author* of the experience, the *date* and the *project* used to derive the experience, and its *type*. We identify three possible types: *conceptual error*, for goals that violated an error condition; *misdefinition* for the ill-defined terms discussed in section 6.4.2; and *correct goal* for goals that did not violated error conditions. In figure 7.3 we present various results on testing. For example, the definition *disjoint* which belongs to the *testOLMECO_layers.pl* source file, is reported from experience author *Austin* on *19/11/99* as part of the project *PhysSys* and characterised as *conceptual error*. In the same manner definitions *direct_part_of* and *connects* are reported by authors *Yannis* and *David* as *correct goal* and *misdefinition*, respectively.

In addition to this direct information, we also found it useful to consider the following ontology characterisations. These were originally introduced in [Uschold 98b] and further analysed in [Uschold & Jasper 99]. Although we did not integrated this information with the table given above, we instantiated the proposed framework with two of the PhysSys ontologies: mereology and topology. Each item represents a category, followed by its instantiation in the particular context. We also include a short explanation of each category in a parenthesis following its name.

- `Purpose`:*(the purpose of the ontology)* In our tests, both mereology and topology provide the building blocks for PhysSys ontology. They formalise generic mereotopological relations as described in the literature(i.e., [Simons 87] and [Clarke 81]);

- `Representation languages and paradigms`:*(Ontolingua? Description Logics? FLogic? CycL? Prolog? Clips? XML?)* The mereology and topology ontology were implemented in Ontolingua, we translated them in the implementation language we use: Prolog;

- `Meaning and formality`:*(to what extent and how formal is the specification of the meaning of each term?)* The Ontolingua versions of mereology and topology provide primitive terms with axioms restricting their use by placing constraints on relationships between types of entities. The implemented versions we used(in Prolog), include this information along with application specific constructs;

- `Subject Matter`:*(is it domain-specific or general?)* Both mereology and topology provide generic relations and axioms to be used in the PhysSys ontologies set;

- `Scale`:*(how big is the ontology?)* Both ontologies are quite small, each of them formalises less than 10 relations;

- `Development`:*(the degree to which the application is specified, developed and/or fielded)* The mereology and topology ontologies are research prototypes but the PhysSys which includes them has been used in the context of the OLMECO project([Borst *et al.* 97]);

- `Conceptual architecture`:*(what are the main components in the ontology application, and how do they relate to each other)* Two research prototype systems were used, the *hospital heating system* and the *air pump system*, both implemented in Prolog, and include ontological constructs;

- `Mechanisms and techniques`:*(what specific mechanisms and techniques were invoked to make use of the ontology)* We deployed the multi-layered architecture(section 4.6) to embed ontological constructs in the applications. We translated ontological constructs to the target implementation language: Prolog, and translated ontological axioms to the constraints format used to detect discrepancies with respect to misuse of ontological constructs;

- `Implementation platform`:*(the particular implementation platform and context)* The ontologies are described textually in the literature and were also written in

> Ontolingua syntax,which we translated in Prolog. The testbed applications were also implemented in Prolog. The tests were executed through a `Java` front-end.

These characterisations provide ontology-specific information which could be used for organising and retrieving experiences on testing in similar ontology deployment efforts.

In a typical experiences recording framework we would proceed to store the information accumulated so far in an EB. However, in our approach we want to validate this information before storing it. This is a requirement in the ontology exercise where the existing ontological structures might be erroneously defined, as indeed happened in our case and demonstrated in chapter 6. We use the *Goal satisfied* for validating an experience. The remaining information will be part of the experiences record which will be created in the next phase.

We will use the erroneous definition 4c from topology ontology, analysed in section 6.4.2, to show how the validation mechanism captures its misbehaviour in a way which also relates the result returned to the error detection theory presented in section 4.2. As we saw in section 6.4.2, the way the topological axiom 4c is originally defined will allow the following goal:

connects(rgspipe_X_rgmvalve,rgspipe,rgmvalve)

to be reported as erroneous, although a quick look at figure 6.10 will reveal the opposite. The error was reported because the above goal satisfied the condition:

```
error(5,connects(C,X,Y), (part_of(X,Z) ∧ disjoint(Z,Y) ∧ ¬ connects(C,Z,Y))).
```

However there is another error condition:

```
error(6,disjoint(X,Y),(equal(X,Y) ∨ (proper_part_of(Z,X) ∧
                                     proper_part_of(Z,Y)))).
```

defined over the mereological definition 5a[1], *disjoint*, which is used as a subgoal of the *connects* condition given above and instantiated as follows:

```
disjoint(radiatorGroup,rgmvalve)
```

which is not provable.

[1] See section 6.2.1 for its description.

This corresponds to *consequence B1* described in section 4.2 where the definition of an erroneous error condition is given. We elaborate on this consequence here via our example. The first part of *consequence B1* states that: "there exists an error condition, *E1*, defined on the given goal for proof, *G*, which is provable". In our case, *E1* is instantiated to:

```
part_of(rgspipe,radiatorGroup) ∧ disjoint(radiatorGroup,rgmvalve) ∧
        ¬ connects(rgspipe_X_rgmvalve,radiatorGroup,rgmvalve))).
```

and *G* is instantiated to:

```
connects(rgspipe_X_rgmvalve,rgspipe,rgmvalve)
```

the second part of *consequence B1* states that: "and there does not exist an error condition, *E2*, defined on that condition, *E1*, such that *E2* is provable.". In our case *E2* is instantiated to:

```
equal(radiatorGroup, rgmvalve) ∨ (proper_part_of(Z,radiatorGroup) ∧
                                  proper_part_of(Z,rgmvalve))
```

Note that this condition is defined over a part of condition *E1* given above. This usually happens when we have a composite condition as in our case with three goals comprising condition *E1*: *part_of*, *disjoint*, and $\neg$*connects*. So, in the second part of *consequence B1* we use a part of condition $E1$ which refers to the goal *disjoint* and instantiated as follows:

```
disjoint(radiatorGroup,rgmvalve)
```

As can be seen from the instantiations, the error condition *E2*, defined on a part of *E1*, was not proved because the *radiatorGroup* is not *equal* to *rgmvalve* and neither do they share a part(satisfying the *proper_part_of* relation). Following the definition of *consequence B1* in section 4.2, we identify error condition *E1* of the topology ontology as erroneous. That is because another condition, *E2*, defined on one of the parts of *E1*(*disjoint*), was not proved. The side-effect is that we can prove that *radiatorGroup* is disjoint from *rgmvalve* and, consequently, condition $E1$ is provable since its third goal, $\neg$*connects(rgspipe_X_rgmvalve, radiatorGroup, rgmvalve)*, is also provable, as it should be. Consequently, the given goal G is reported as an error although it is actually correct.

To implement this sort of reasoning in error checking and relate the results to the error theory of section 4.2 we deployed a meta-interpreter, first mentioned in section 4.7.1, which separates the inference engine from the error checking procedure. This implementation follows the representations in sections 4.3 and 4.4 and is included in appendix C.1. We do not refer to the first two parts here since they are direct transformations of the given Horn clauses. In particular, as can be seen from appendix C.1, lines 5 to 14 implement the inference engine of section 4.3 enhanced with multi-layer architecture specific constructs(given as predicates $ontologicalDefinition/2$ and $specification/2$). In lines 15 to 47 we include an implementation of the error checking mechanism described in section 4.4. The first part, lines 15 to 29, correspond to lines (A) to (D) in the generic error checking mechanism of section 4.4 and does not merit detailed scrutiny here since it is a transformation of the given Horn clauses enhanced with constructs similar to those described above. However, we describe below the implementation of error check(lines 30 to 47) which is an augmented version of the generic representation in section 4.4(lines (E) to (G)):

```
30 error_check(X, S) ← setof(errors_found(X, E, Es), ontological_error(X, E, Es), S).
31 error_check(X, [('cannot prove the negated condition : ','E,' on goal :', X,
32                  'proceed to check for errors in :', Goal)|S]) ←
33          ¬ ontological_error(X, _, _)∧
34          error(_, X, E)∧
35          E = (¬ Goal)∧
36          ¬ error(_, Goal, ¬ X)∧
37          error_check(Goal, S).
38 error_check(X, [('cannot prove condition :', Goal,' on goal :', X)|S]) ←
39          ¬ ontological_error(X, _, _)∧
40          ¬ (X = (¬ _))∧
41          error(_, X, Goal)∧
42          ¬ error(_, Goal, X)∧
43          error_check(Goal, S).
44 error_check(X, []) ← ¬ ontological_error(X, _, _).
45 ontological_error(X, E, Es) ←error(_, X, E)∧
46                              copy_term(E, E1)∧
47                              onto_solve(E1, Es).
```

Line 30 follows the description of the generic representation in section 4.4: an error check on goal X will yield a set of errors S if we can find conceptual errors E and its dependent errors Es on X and set S is composed of all occurrences of E and Es represented as the template $errors\_found(X, E, Es)$. In lines 31 to 43 we augment this error checking by introducing a technique which we call 'skip-one'. Here is how it works: if we cannot find an ontological error on the given goal for proof, we 'skip' the goal and try to check for errors on the ontological constraints that are defined over it. In particular, in lines 31 to 37 we state that whenever we cannot find an ontological error(line 33) on the given goal for testing, X, we check if the error condition defined over it is a negated one(lines 34 and 35), and we proceed to check recursively for errors on that condition without the negation being taken into account(line 37). We also check for cyclic definitions of error conditions(line 36) in order to avoid infinite looping. In lines 38 to 43 we deal with situations where we cannot find an ontological error(line 39), and this is not happens because the given goal for proof X is a negated one(hence the check for a non-negated goal in line 40), in which case we check for error conditions defined over that goal(line 41), and apply recursively the same check on these conditions($Goal$ in line 43). As previously, we check for cyclic definitions of error conditions(line 42) in order to avoid infinite looping. In line 44 we terminate this recursive check and lines 45 to 47 correspond to the generic representation of section 4.4 with regard to the interface with ontological constraints.

We further elaborate on the application of this technique: the chunks of code in lines 31 to 37 and 38 to 43, are a way of treating negated goals. Recall from section 4.4, page 57, where we stated that "[...] we get no errors on a negated goal when we cannot prove it via the inference engine.". Furthermore, in section 4.8, page 69, we stated that "[...] any errors in the exploration of a failed proof are ignored.". In our implementation of the 'skip-one' technique we first verify two things: that we could not prove a goal to be erroneous(lines 33 and 39) and this wasn't because we didn't had available error conditions to check them. As it shown from lines 34 and 41, error conditions were indeed defined over the goals given for proof. We then apply the mechanism to check whether we failed to reveal errors because an error condition was itself erroneous. To do that we treat the condition as a goal given for proof and check it recursively as if it was a goal. We take care of two cases: when the condition to be checked is a negated one(first chunk, lines 31 to 37) and a non-negated(second chunk, lines 38 to 43). In both cases we check them as if they were goals given for proof with the difference being that we ignore the negation in the first case. This implementation also gives us a neat representation of the error theory consequences *A1*, *A2* and *B1*, described in section 4.2, page 54.

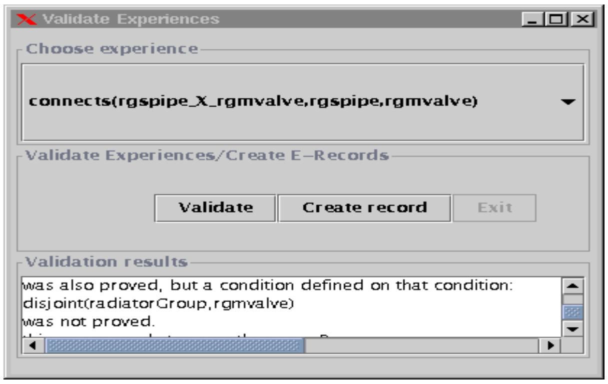


Figure 7.4: Validating experiences

Consequently, we were able to reason about error occurrences and relate them to the error detection theory. This meta-interpreter is used in the validation tool for experiences, a screenshot of which is given in figure 7.4 and shows the result returned from checking the error case described above.

At this step we collect the information returned from validation, like the corresponding error theory cases, the conditions that were found erroneous(if any), and information already accumulated before(in the *experiences table*), to build the experiences record.

Each record contains the following fields: *Ontological definition*, *Horn clause*, *Goal satisfied*, *Condition satisfied*, *Layer*, *Path*, *Source file*, accumulated from the verification task; *Condition not satisfied*, *Erroneous condition*, *Erroneous specification*, *Dependent condition*, *Error theory correspondence*, collected from the validation task; and *Author*, *Date*, *Project*, and *Report type* as entered in the *experiences table*.

In addition, each record is also linked to the characterisations of ontology entered in the experiences table. In the sequel, each record is stored in the final EB which in turn will be part of an EF on ontology verification.

Recalling the figure 7.1 where we presented an implementation of an EF as realised in [Althoff *et al.* 99b]. Here we reproduce that figure along with its instantiations in the context of our case: ontology verification.

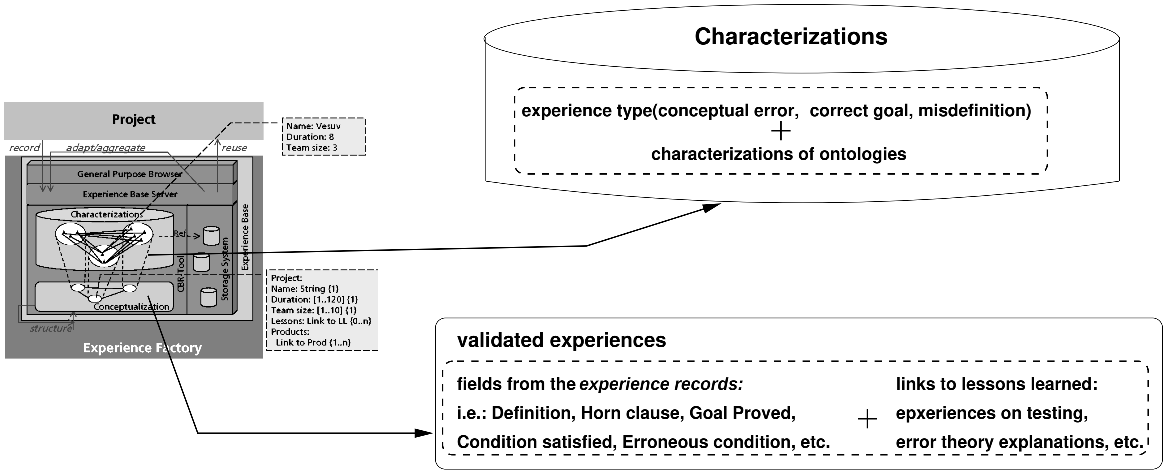


Figure 7.5: The *EF* instantiated with the ontology verification scenario.

As illustrated in figure 7.5, we have added information obtainable from our experiences with the ontology verification task. These are results from code-testing along with *links to lessons learned* where we provide links to the error theory of section 4.2. This is then contained in an experience record. In the characterisation phase we can give generic information, like the experience types with respect to conceptual error occurrences(i.e., *conceptual error*, *correct goal*, *misdefinition*). In addition, at this phase we also provide links to the ontology characterisations edited in the *experiences table*. This could help to browse through the collected experiences and ultimately supports the idea of "learning from experience" which is an acknowledged need in ontology deployment.

The benefits of deploying EFs to organise ontologies are two-fold: for software and ontology engineers. The former can use ontologies as a testbed for EFs which could lead to their improvement as ontology deployment brings interesting and intricate problems to cope with. On the other hand, ontology engineers can benefit from the use of EFs to facilitate ontology exercise as a means of collecting and organising experiences during ontology deployment. Many applications are possible in the ontology development and deployment life-circle. Apart from EBs on testing, an example of which was shown in this section, we envisage EBs on development, on maintenance, on reuse, etc. All these EBs can be part of an EF tailored to a particular ontology or project. Having these EFs available in online libraries, alongside with their ontology counterparts, may help us to develop them better, deploy them faster, and reuse them more easily.

7.2 Ontologies and knowledge management

Another area to which we extended our work was that of knowledge management (hereafter, KM) and organisational memories(hereafter, OMs). The work we describe here originates from [Kalfoglou 00] where Kalfoglou argues for the convergence of ontologies and EFs in order to support OMs and facilitate KM activities and tasks. In that paper, the author argues for the fact that KM and OMs are intertwined areas but the technologies used to implement and support them are not treated in the same fashion. Consequently, we cannot fully exploit their strength and integrate them easily. Two core technologies for developing OMs and deploying KM activities were identified: EFs as a means to support and provide the infrastructure for developing OMs, and ontologies which could be used to support KM activities and tasks. In this section we focus on the role of ontologies in KM and elaborate briefly on the convergence of ontologies and EFs. We point the interested reader to [Kalfoglou 00] for a discussion on the role of EFs in OMs.

KM refers to "the formal management of knowledge for facilitating creation, access, and reuse of knowledge"([O'Leary 98a]). Ontologies have been studied by many AI researchers as a means to support KM and O'Leary discusses their role in [O'Leary 98b] and argues that, "a KM system depends on ontologies to facilitate communication between its multiple users and links between multiple knowledge bases". O'Leary describes uses of ontologies in KM systems with example cases drawn from the private sector, in particular consulting firms([O'Leary 98b]) whereas Benjamins and colleagues argue for an ontology-based KM which results in an intelligent access to knowledge assets as they shown in their example case of KM in a virtual organisation([Benjamins *et al.* 98]).

We recapitulate here on the conclusions drawn from these reports and further investigate the potential of using ontologies to achieve effective KM. Firstly, we identify areas of KM where ontologies are suitable for exploitation. In his review, O'Leary identified three factors that lead organisations to use KM. These were classified as *environmental pressures* imposed by the increasingly competitive global market place, *technological* *advancements* arising from recent developments in Internet technology, and the ability to *create valuable information* by converting individually available knowledge into group or organisationally available knowledge([O'Leary 98a]). Whereas the *environmental pressures* and *technological advancements* give an organisation the reason and the means to pursue KM activities, *creating valuable information* is the goal of KM.

The latter is achieved by the *converting and connecting* processes as identified in [O'Leary 98b]. These are summarised as follows: convert (i)individual to group knowledge, (ii)data to knowledge, (iii)text to knowledge, and connect (iv)people to knowledge, (v)knowledge to knowledge, (vi)people to people, and (vii)knowledge to people. We argue that ontologies are present in most of these processes, either by playing a major role or by supplying the supporting infrastructure that helps an organisation to implement them. In the following paragraph we mention indicative examples from the ontology research literature to justify this claim.

In particular, ontologies provide part of the infrastructure for conversion processes(i to iii as listed above) and support the connection activities(iv to vii as listed above). Conversion processes (i) seem to benefit more from the presence of ontologies as this is the underlying principle in their construction. Methodological ([Uschold & Gruninger 96]) and collaborative approaches([Swartout *et al.* 96]) in ontology building([Benjamins & Fensel 98]), convert individual to group knowledge in the form of an ontology. Processes (ii) and (iii) use other AI technology like data and text mining techniques with ontologies being the guide to the 'right' data or text repository([Decker *et al.* 99]). Ontologies are more active in the connecting processes. Process (iv) is concerned with the so called, 'pull' technology, which aims at pulling knowledge residing in vast repositories to people. The means which used to pull that knowledge are, mainly, search engines and intelligent agents. Examples of ontology use in this area are given in [McGuinness 98] and [Guarino *et al.* 99]. Process (v) actually highlights the main contribution of ontologies: enabling communication and interoperability between systems. The best way to cite indicative work here is to point to the field reviews and collections which were mentioned in chapter 2. Process (vi) is not directly related to ontologies as it is more concerned with technological means such as Intranets. However, we should mention the work on collaboration and discussion aided by ontologies([Summer & Buckingham-Shum 98]).

In contrast with process (iv), process (vii) is concerned with 'push' technology. Means to achieve this are designated systems that focus on content and push knowledge to the user instead of waiting for the user to pull out that knowledge. As in (iv), ontologies play a major role here since they are concerned with content and semantically enriched information. Example uses are described in [Decker *et al.* 99].

The result of applying ontologies in these processes supports and improves KM in such areas as: formation of discussion groups on particular topics of interests, improved search capabilities by semantically-enriched query/answering facilities, filtering facilities to capture the desired knowledge, reusability of artefacts, and enabling communication between different systems and/or people by using an interlingua([O'Leary 98b]). Moreover, Benjamins and colleagues argued that four basic activities of KM, namely, knowledge gathering, organisation and structuring, refinement, and distribution could be effectively supported by ontologies as they shown in their report with the use of ontologies in each of these activities([Benjamins *et al.* 98]).

However, the development and use of KM ontologies hides important caveats as O'Leary reports in [O'Leary 98a]: "Each consulting firm we have been examining has built or is building its own ontologies. Because these enterprise ontologies are so costly to develop and maintain and are constantly changing, ontology or taxonomy issues are emerging as some of the most important problems in knowledge management". O'Leary analyses further these problems from the KBSs development viewpoint in [O'Leary 97]. In [Kalfoglou 00], Kalfoglou argues for the potential of converging EFs and ontologies to alleviate the situation. To explain the idea we illustrate the approach in figure 7.6 taken directly from [Kalfoglou 00].

On the left hand side of figure 7.6, we place within a box surrounded by a dash-lined border the OMs technologies. At the bottom of that box we place a candidate technology for supporting OMs implementation: EFs. The diagram included there is actually a reduction of figure 7.2 presented in section 7.1. It denotes an example use of EFs in the area of ontology verification as we described in that section. There can be other technologies to support the implementation of OMs. We point the interested reader to [Abecker *et al.* 98] were Abecker and colleagues identify various candidates.

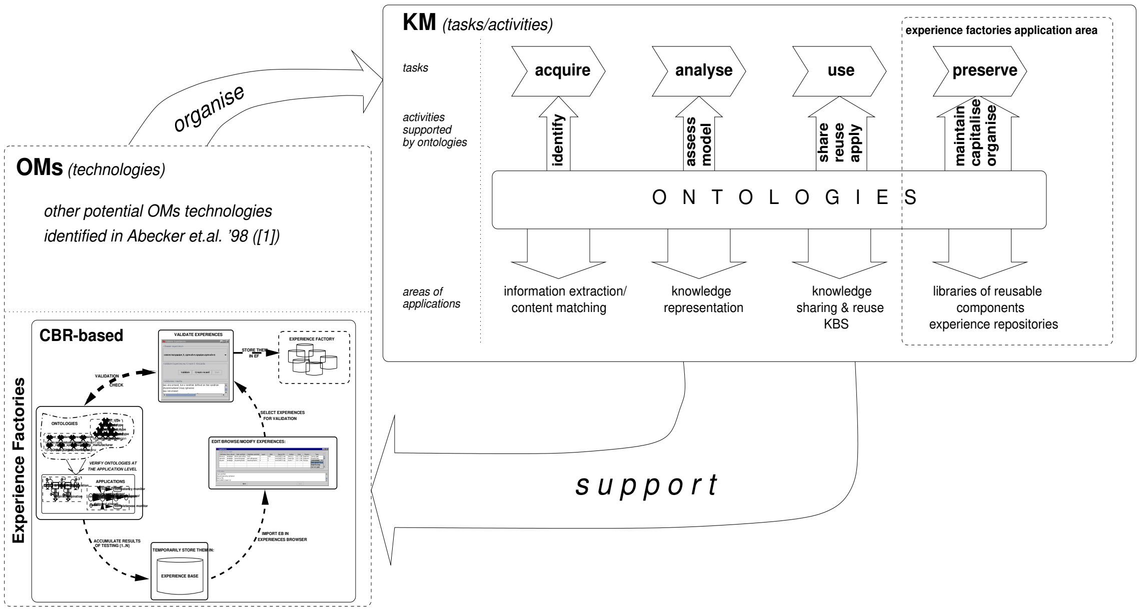


Figure 7.6: OMs in KM: OMs technology used to organise KM tasks/activities which in turn support the implementation of OMs.

On the right hand side of figure 7.6, we illustrate the main KM tasks and activities. We identify four main KM tasks: *"acquiring"*, *"analysing"*, *"using"*, and *"preserving"* knowledge. We argue that these tasks are accomplished by activities underpinned by ontologies. In particular, the knowledge acquisition task, is accomplished by *"identifying"* activities underpinned by ontologies. This results in the application area of information extraction and/or content-matching. In the same manner, ontologies in the area of knowledge representation are used to *"model"* and *"assess"* the environment, which are activities employed in the *"analysing"* knowledge task. The *"using"* knowledge task, includes the *"apply"*, *"share"*, and *"reuse"* activities, which are underpinned by ontologies with such application areas as knowledge sharing, reuse, and KBSs. The last task of the KM tasks/activities diagram is *"preserving"* knowledge. It is accomplished by activities such as *"organising"*, *"maintaining"*, and *"capitalising"* which are partially aided by ontologies. The resulting application area is that of libraries of reusable knowledge components and experience repositories. The knowledge preservation task and its accompanying activities along with the relevant ontologies are the area of overlap with EFs as denoted by the box surrounding the task in figure 7.6.

The way in which the two boxes of figure 7.6 are related summarises the linkage suggested in [Kalfoglou 00]. As we mentioned earlier, OMs and KM are intertwined areas. In this figure we illustrate how the technologies used to implement them can also be intertwined. As can be seen from the curly arrow connecting the OMs technologies with the KM tasks/activities box, technologies such as EFs can be employed to *"organise"* ontologies which underpin main KM tasks/activities. The latter, in turn, can *"support"* OMs implementation by *"acquiring"*, *"analysing"*, *"using"*, and *"preserving"* knowledge processed by the OM.

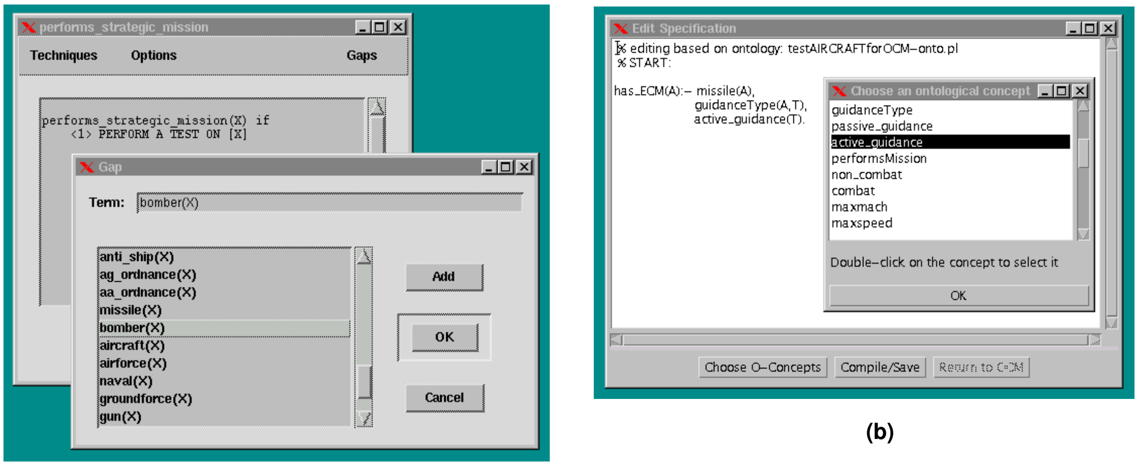
There are mutual benefits for integrating OMs technology in KM ontologies. On one hand, an OM framework could help to improve ontology development and deployment, facilitate understanding, and ease reuse. A better organised ontology could, in turn, overcome some of the problems identified in [O'Leary 97]('perfect' ontology hype, library ontologies, scale-up, interface, formality), and analysed from a cost-benefit point of view in [Kalfoglou *et al.* 00a]. A small example on the use of an EF-style organisation of ontology testing was presented in section 7.1. On the other hand, better ontologies could help to meet practical requirements for the implementation of OMs. These were identified by Abecker and colleagues in [Abecker *et al.* 98]: "(i)collection and systematic organisation of information from various sources, (ii)ability to minimise up-front knowledge engineering, by taking advantage of readily available information, (iii)exploiting user feedback for maintenance and evolution, (iv)integration into existing work environment, (v)active presentation of relevant information". Requirements (i),(iv) and (v) could benefit more from ontologies as we briefly described above and illustrated in figure 7.6. Requirements (ii) and (iii) could benefit more from the presence of an EF.

The contribution of our work is closely related to O'Leary's observation for the need of *knowledge harvesting* which: "must identify knowledge that it desirable to share, worth converting, and usable by others"([O'Leary 98b]). In our work we paraphrase this definition in the context of ontologies as follows: *Ontology harvesting must identify ontologies that are desirable to share, worth converting, and usable by others.* We argue that this *harvesting* process could be accomplished by employing OMs technologies like EFs.

7.3 Specification construction tools

As we mentioned in section 4.7.3 the OCM provides tools which help in building a specification in Prolog which adopts the ontology at question. The ultimate goal of this sort of tools is to ensure that as many ontological constructs as possible will be used in the specification. Although this can be done manually, the size of most ontologies requires some sort of automation. Having in mind this requirement, we built a free-style specification editor which allows the addition of ontological constructs, and we augmented with ontological constructs a semi-automatic method for building a Prolog specification, the *Techniques Editor*. These are described below using two small examples in the context of the OCM.

In figure 7.7 we include two screenshots from the specification construction tools used in the OCM: the *Techniques Editor* in the left hand side and a free-style text editor in the right hand side.



(a) (b)

Figure 7.7: Specification construction tools: (a) the *Techniques Editor*, (b) a free-style editor.

As we can see from figure 7.7, the free-style editor tool is a plain text editing window where the specifier can write a specification in Prolog. For example, assume someone writes a specification based on the AIRCRAFT ontology mentioned in an earlier chapter. The corresponding ontology file is loaded along with the editor. This file contains ontological constructs, such as relations and concepts of the AIRCRAFT ontology in the form of Horn clauses as a Prolog program. In our case, the specifier wants to define the notion of "electronic counter-measures"(in short, ECM) with regard to a missile's capabilities of avoiding jamming. These are electronic devices which enable the missile that uses them to bypass the "electronic measures"(in short, EM) emitted by the hostile aircraft in order to divert incoming missiles from their targets. The specifier chooses to define the binary predicate `has_ECM` which declares that a missile has ECM if its guidance type is active. Although the notion of ECM is new and does not exist in the AIRCRAFT ontology, the concepts *missile*, *guidance type* and *active_guidance* do. Therefore, the specifier can use them directly in the specification without having to define them from scratch. To facilitate their selection we have build a Prolog program which collects concepts from the underpinning ontology and presents them to the specifier in a small window as it is shown in figure 7.7. The specifier then browses the list to find the concept he wants to use and selects it by double-clicking to insert it in the specification. The concepts collection accumulates all the literals founded in a Prolog program excluding built-in predicates, number, variables and omitting duplicates. Although not fully-automated, this tool lightens the load for the specifier since it gives him a quick glance over the ontology's concepts thus avoid redefining already existing ones.

The second tool,*Techniques Editor*, shown in the left hand side of figure 7.7 is based on the work of Bowles and colleagues described in [Bowles *et al.* 94]. We provide the same linkage to ontological constructs as for the free-style editor described above. That is, we provide for the *Techniques Editor* user the option to choose from ontology constructs apart from those already provided by the editor such as built-in predicates. These constructs could then be used by the *Techniques Editor* user whenever he needs to fill-in a gap in the technique under construction. For example in the screenshot shown in figure 7.7, the specifier defines a new technique called `performs_strategic_mission` with regard to aircraft of type *bomber*. He has chosen to build this technique as a basic rule[2] which sets the unary predicate `performs_strategic_mission` as the post-condition of the designated Horn clause and leaves one gap to be filled-in in the pre-

[2] We should note that the whole spectrum of techniques provided by the *Techniques Editor* ranges from simple facts and rules to accumulators, counters, and meta-interpreters. For the sake of brevity we chose to present a simple one here: a basic rule.

condition with respect to the variable `X` as shown in figure 7.7. At this point we provide the link to ontological constructs. As can be seen from the overlapped window in figure 7.7 we present candidate ontological constructs to be included in the technique under construction. In our example, the *bomber* concept is chosen. In the sequel the *Techniques Editor* user can save this technique a normal Prolog code, namely a basic rule that declares a bomber to perform strategic missions.

7.4 *Chapter summary*

In this chapter we saw how our work can be extended in a variety of contexts. In particular, we showed how a method originating from the SE community, that of experience management, could help to organise ontologies during deployment. We worked with our PhysSys scenario case and demonstrated how the method of EF can fit in the ontology development life-cycle to collect and manage experiences accumulated during testing. We further elaborated on the impact of ontologies in the whole spectrum of KM and argued for the linkage of core technologies for OMs and KM. Potential candidates for the former are EFs whereas for the latter are ontologies. The key contributions of ontologies in KM were identified and we speculated for their influence in KM core activities and tasks. Finally, we presented two tools that aim to facilitate inclusion of ontological constructs in a specification. This is achieved via a simple extraction method and utilised by the two specification construction editors provided by the OCM environment.

Chapter 8

Contributions made

In the introductory chapter of this thesis, in section 1.3, we claimed three main contributions which we repeat below:

1. Deployment of ontologies in software design;
2. Worked examples of how ontologies can be deployed for purposes other than knowledge sharing and reuse;
3. A neutral architecture with supporting tools for deploying ontologies.

As we mentioned in section 1.3, we revisit these claims in this chapter and justify them by recapitulating on the content of chapters 2 to 7.

The main aim of the first claim is to narrow the gap between domain knowledge and applications. We argued in chapter 1 that most applications are not explicitly connected to their domain, and even if they are, no means for verifying their conformance to domain knowledge is provided. This was indeed emphasized in section 2.1 where we pointed out that the majority of work in software testing is focussed on perfecting the method rather than broadening the error typology. We also looked at AI-based software testing in section 2.2 which provides some partial solutions to the problem of checking the conformance to domain knowledge. This led us to scrutinise the role that ontologies could play in this problem. After having present them thoroughly in section 2.3 we devote the whole chapter 3 to justify the claim made above. In particular, we focussed on the difficulties of the early phases of design(section 3.1) and gave an example of a type of error which often remains undetected, the conceptual error(section 3.2). In section 3.3 we explained how domain knowledge can help to tackle this problem where we also illustrate in an inspiring figure the connection of our work with previous efforts(see figure 3.1).

The second claim listed above is an example of implementing the first one. We argued that the majority of ontology applications deal with knowledge sharing and reuse(section 2.3.7). Our aim in deploying them to applications is a different one: to improve the reliability of applications that adopt ontologies with respect to consistency checking. Throughout chapter 5 we described a variety of worked examples of how this can be done: in business process modelling(section 5.2), in ecological modelling(section 5.3), and in air-campaign planning(section 5.4). Each of these example cases gave us the opportunity to investigate deeply the connection between ontology and applications. In particular, we experimented with situations where, (i)we had an existing ontology(PIF) and we devised an exemplar application that adopts it(section 5.2), (ii)we had an existing application(EcoLogic) and we identified generic applications constructs to be parts of a prospective ontology(section 5.3), and (iii)we had both an existing ontology(AIRCRAFT) and an application(MDS) and our aim was to link them(section 5.4). The outcome of these worked examples is the *conformance check*(section 5.1) which covers all possible connections of ontology to application and enforces the application of our conceptual errors check method.

The implementation of that method is the result of the last claim. We described the theoretical background underpinning our implementation choices in chapter 4 where we presented an extension to our approach: the multi-layer architecture. This made it possible to deploy easier complex ontological structures in applications. We justified this claim by devoting the whole of chapter 6 to explaining how a complex lattice of 7 different ontologies(section 6.2) can be deployed in two exemplar applications(section 6.3) by using the multi-layer architecture. More interesting though, was the fact that we were also able to detect ill-definitions in the original ontologies(section 6.4.2). The use of this architecture is supported by a variety of tools, including an editing system for defining ontological axioms and relations(section 5.4), a translator for automatically transforming axioms to the preferred error condition format(section 4.7.2), specification construction aid tools(section 7.3), and an integrated front-end(section 4.7.3) written in `Java` for accessing the tools mentioned above.

Last but not least, our work also opens areas that seem to be fruitful for exploration. An interesting one is the convergence of ontologies and experience factories, a method conceived and deployed by software engineers to facilitate management of software projects. In section 7.1 we provide an example of how these two technologies can complement each other and argue for the benefit of such convergence for both communities that invented them, artificial intelligence and software engineering. Another area of potential exploration is that of knowledge management. In section 7.2 we argued for the role of ontologies in knowledge management as vehicles for supporting its core activities and tasks.

Bibliography

[Abecker *et al.* 98] A. Abecker, A. Bernardi, K. Hinkelmann, O. Kuhn, and M. Sintek. Toward a Technology for Organizational Memories. *IEEE Intelligent Systems*, 13(3):40–48, June 1998.

[Abernethy *et al.* 99] N.F. Abernethy, J.F. Wu, M. Hewett, and R.B. Altman. Sophia: A Flexible, Web-Based Knowledge Server. *IEEE Intelligent Systems*, 14(4):79–85, July 1999.

[Aguado *et al.* 98] G. Aguado, A. Banon, J. Bateman, S. Bernandos, M. Fernandez, A. Gomez-Perez, E. Nieto, A. Olalla, R. Plaza, and A. Sanchez. ONTOGENERATION: Reusing domain and linguistic ontologies for Spanish text generation. 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.

[Aitken 98] S. Aitken. Extending the HPKB-Upper-Level Ontology: Experiences and Observations. 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.

[Althoff & Wilke 97] K-D. Althoff and W. Wilke. Potential uses of case-based reasoning in experienced based construction of software systems and business process support. In *Proceedings of the 5th German Workshop on Case-Based Reasoning*, pages 31–48. LSA-97-01E, University of Kaiserslauten, March 1997.

[Althoff *et al.* 99a] K-D. Althoff, P. Bergmann, and L.K. Branting, editors. *Case-Based Reasoning Research and Development(ICCBR'99)*, number 1650 in Lecture Notes in Artificial Intelligence. Springer Verlag, July 1999.

[Althoff *et al.* 99b] K-D. Althoff, A. Birk, S. Hartkopf, W. Muller, M. Nick, D. Surmann, and C. Tautz. Managing Software Engineering Experience for Comprehensive Reuse. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 10–19, June 1999.

[Aspirez *et al.* 98] J. Aspirez, A. Gomez-Perez, A. Lozano, and S. Pinto. (onto)2agent: An ontology-based www broker to select ontologies. In *Proceedings of the Workshop on Applications of Ontologies and Problem-Solving Methods, ECAI'98, Brighton, England*, pages 16–24, August 1998.

[Baker *et al.* 99] P. Baker, C. Goble, S. Bechhofer, N. Paton, R. Stevens, and A. Brass. An ontology for bioinformatics applications. *Bionformatics*, 15(6):510–520, 1999.

[Basili & Rombach 88] V.R. Basili and H.D. Rombach. The TAME Project: Towards Improvement Oriented Software Environments. *Transactions on Software Engineering*, SE-14(6):758–773, June 1988.

[Basili *et al.* 94] V. Basili, G. Caldiera, and D. Rombach. Experience Factory. In J. Marciniak, editor, *Encyclopedia of Software Engineering*, volume 1, pages 469–476. John Wiley & Sons, 1994.

[Bateman *et al.* 95] J. Bateman, B. Magnini, and G. Fabris. The Generalized Upper Model(GUM) knowledge-base: Organisation and use. In N.J.I. Mars, editor, *Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB & KS'95), Twente, The Netherlands*, pages 60–72, Amsterdam, NL, 1995. IOS Press.

[Benjamins & Fensel 98] 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.

[Benjamins *et al.* 98] R. Benjamins, D. Fensel, and A. Gomez-Perez. Knowledge Management through Ontologies. In *Proceedings of the 2th International Conference on Practical Aspects of Knowledge Management(PAKM'98), Basel, Switzerland*, October 1998.

[Blazquez *et al.* 98] 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, Modelling and Management Workshop, KAW98, Banff, Canada*, April 1998.

[Booch *et al.* 98] G. Booch, J. Rumbaugh, and I. Jacobson. *Unified Modeling Language User Guide*. Addison-Welsey, 1998. ISBN: 0-207-57168-4.

[Borst *et al.* 97] P. Borst, H. Akkermans, and J. Top. Engineering Ontologies. *International Journal of Human-Computer Studies*, 46:365–406, 1997.

[Bowles *et al.* 94] A.W. Bowles, D. Robertson, W. W. Vasconcelos, M. Vargas-Vera, and D. Bental. Applying Prolog Programming Techniques. *International Journal of Human-Computer Studies*, 41(3):329–350, September 1994.

[Brna *et al.* 91a] P. Brna, M. Brayshaw, A. Bundy, T. Dodd, M. Elsom-Cook, and P. Fung. An Overview of Prolog Debugging Tools. *Instructional Science*, 20(2/3):193–214, 1991.

[Brna *et al.* 91b] P. Brna, A. Bundy, T. Dodd, M. Eisenstadt, C-K. Looi, H. Pain, D. Robertson, B. Smith, and M. vanSomeren. Prolog Programming Techniques. *Instructional Science*, 20(2/3):111–133, 1991.

[Campbell & Shapiro 98] A.E. Campbell and S.C. Shapiro. Algorithms for Ontological Mediation. Technical Report 98-03, Department of Computer Science and Engineering, State University of New York at Buffalo, January 1998.

[Chandrasekaran *et al.* 98] B. Chandrasekaran, J.R. Josephson, and R. Benjamins. The Ontology of Tasks and Methods. In *Proceedings of the 11th Knowledge Acquisition Modeling and Management Workshop, KAW'98, Banff, Canada*, April 1998.

[Chandrasekaran *et al.* 99] B. Chandrasekaran, R. Josephson, and R. Benjamins. What Are Ontologies, and Why Do We Need Them? *IEEE Intelligent Systems*, 14(1):20–26, January 1999.

[Clark 78] K.L. Clark. Negation as failure. In H. Gallaire and J. Minker, editors, *Logic and Databases*, pages 293–322. Plenum Press, 1978.

[Clarke & Wing 96] E. Clarke and J. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, December 1996.

[Clarke 81] B.L. Clarke. A calculus of individuals based on "connection". *Notre Dame Journal of Formal Logic*, 22:204–218, 1981.

[Cleland & MacKenzie 95] G. Cleland and D. MacKenzie. Inhibiting Factors, Market Structure and the Industrial Uptake of Formal Methods. In *Workshop on Industrial-Strength Formal Specification Techniques, Boca Raton, FL, USA*, pages 46–60, Orlando(Florida) USA, April 1995.

[Clocksin & Mellish 94] W. Clocksin and C. Mellish. *Programming in Prolog.* Springer, 4th edition, 1994. ISBN 0-387-17539-3.

[Cohen *et al.* 98] P. Cohen, R. Schrag, E. Jones, A. Pease, A. Lin, B. Starr, D. Gunning, and M. Burke. The DARPA High Performance Knowledge Bases project. *AI Magazine*, 19(4):25–49, 1998.

[Cohen *et al.* 99] P. Cohen, V. Chaudhri, A. Pease, and R. Schrag. Does prior knowledge facilitate the development of knowledge-based systems? In *Proceedings of the Sixteenth National Conference on Artificial Intelligence, AAAI'99, Orlando, FL, USA*, pages 221–226, July 1999.

[Console *et al.* 93] L. Console, G. Friedrich, and D. Theseider-Dupre. Model-based diagnosis meets error diagnosis in logic programs. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence(IJCAI'93),Chambery,France*, pages 1494–1499. Morgan Kaufmann, August 1993.

[Crow & Shadbolt 99] L. Crow and N. Shadbolt. Acquiring and Structuring Web Content with Knowledge Level Models. In R. Studer and D. Fensel, editors, *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management(EKAW'99), Dagstuhl, Germany*, pages 83–101. Springer Verlag, May 1999.

[Cui *et al.* 98] B. Cui, Y. Dong, X. Du, K. Narayan-Kumar, C.R. Ramakrishnan, I.V. Ramakrishnan, A. Roychoudhury, A. Scott, and S.D. Warren. Logic and

Programming and Model Checking. In C. Palamidessi, H. Glaser, and K. Meinke, editors, *Proceedings of PLILP/ALP'98-Principles of Declarative Programming, Pisa, Italy*, volume 1490 of *Lecture Notes in Computer Science(LNCS)*, pages 1–20. Springer-Verlag, September 1998.

[Dalianis & Hovy 98] H. Dalianis and E. Hovy. Integrating STEP Schemata using Automatic Methods. 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.

[Decker *et al.* 99] S. Decker, M. Erdmann, D. Fensel, and R. Studer. Ontobroker: Ontology Based Access to Distributed and Semi-Structured Information. In R & et.al. Meersman, editor, *Proceedings of DS-8, Semantic Issues in Multimedia Systems, Boston, MA, USA*, pages 351–369, 1999.

[deKleer & Williams 87] J. deKleer and B. Williams. Diagnosing multiple faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[DeMillo & Offutt 91] R. DeMillo and A. Offutt. Constraint-based automatic test data generation. *IEEE Transactions on Software Engineering*, 17(9):900–910, September 1991.

[DeMillo *et al.* 78] R. DeMillo, R. Lipton, and F. Sayward. Hints on test data selection: Help for the practicing programmer. *Computer*, 11(4), April 1978.

[Dershowitz & Lee 87] N. Dershowitz and Y-J.. Lee. Deductive debugging. In *Proceedings of the 1987 Logic Programming Symposium, San Fransisco, USA*, pages 298–306. IEEE Computer Society, 1987.

[Dershowitz & Lee 93] N. Dershowitz and Y-J. Lee. Logical Debugging. *Journal of Symbolic Computation*, 15(5/6):745–773, May 1993.

[Domingue & Motta 00] J. Domingue and E. Motta. Planet-Onto: From News Publishing to Integrated Knowledge Management Support. IEEE Intelligent Systems(In Press), 2000.

[Domingue 98] J. Domingue. Tadzebao and WebOnto: Discussing, Browsing, and Editing Ontologies on the Web. In *Proceedings of the 11th Knowledge Acquisition, Modelling and Management Workshop, KAW'98, Banff, Canada*, April 1998.

[EIL 95] Enterprise Integration Laboratory. EIL. TOVE Project, University of Toronto, Canada. available from http://www.ie.utoronto.ca/EIL/tove/ontoTOC.html, July 1995.

[Eisenstadt & Brayshaw 88] M. Eisenstadt and M. Brayshaw. The transparent Prolog machine(TPM): an execution model and graphical debugger for logic programming. *Journal of Logic Programming*, 5(4):277–342, 1988.

[Eisenstadt 85] M. Eisenstadt. Retrospective zooming: a knowledge based tracking and debugging methodology for logic programming. In *Proceedings of the 9th International Joint Conference on Artificial Intelligence(IJCAI'85), Los Angeles, CA, USA*. Morgan Kaufmann, 1985.

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

[Farquhar *et al.* 97] A. Farquhar, R. Fikes, and J. Rice. The ontolingua server: a tool for collaborative ontology construction. *International Journal of Human-Computer Studies*, 46(6):707–728, June 1997.

[Fensel & vanHarmelen 94] D. Fensel and F. van Harmelen. A comparison of languages which operationalise and formalise KADS models of expertise. *The Knowledge Engineering Review*, 9(2):105–146, 1994.

[Fensel *et al.* 99] D. Fensel, V.R. Benjamins, E. Motta, and B. Wielinga. UPML: A Framework for knowledge system reuse. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI'99, Stockholm, Sweden*, pages 16–21, August 1999.

[Fernandez 99] M. Fernandez. Overview for Methodologies for Building Ontologies. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, August 1999. Available from: http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

[Fernandez *et al.* 97] 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, CA, USA*, pages 33–40, March 1997.

[Fikes & Farquhar 99] R. Fikes and A. Farquhar. Distributed Repositories of Highly Expressive Reusable Ontologies. *IEEE Intelligent Systems*, 14(2):73–79, March 1999.

[Fillion & Menzel 96] F. Fillion and C. Menzel. Using ontologies to enable enterprise model integration. in Appendices of the paper: "Ontologies: principles, methods and applications", Uschold,M. and Gruninger,M., The Knowledge Engineering Review, 11(2), p.130-136, 1996.

[Finkelstein 92] A. Finkelstein. Reviewing and Correcting Specifications. *Instructional Science*, 21:183–198, 1992.

[Forrester 61] J.W. Forrester. *Industrial Dynamics.* MIT Press, 1961. ISBN: 0-262-060035.

[Foster 80] K.A. Foster. Error sensitive test case analysis(ESTCA). *IEEE Transactions on Software Engineering*, 6(3):258–264, May 1980.

[Fox & Gruninger 97] M.S. Fox and M. Gruninger. On Ontologies and Enterprise Modelling. In *Proceedings of International Conference on Enterprise Integration Modelling Technology 97.* Springer-Verlag, 1997.

[Fridman-Noy & Hafner 97] N. Fridman-Noy and C.D. Hafner. The State of the Art in Ontology Design: A Survey and Comparative Review. *AI Magazine*, 18(3):53–74, 1997.

[Friedman & Voas 95] M. Friedman and J. Voas. *Software Assessment: Reliability, Safety, Testability.* Willey & Sons, 1995. ISBN: 0-471-01009-X.

[Fuchs & Robertson 96] N. Fuchs and D. Robertson. Declarative Specifications. *The Knowledge Engineering Review*, 11(4):317–331, 1996.

[Fuchs 92] N. Fuchs. Specifications are (preferably) executable. *Software Engineering Journal*, pages 323–334, September 1992.

[Gangemi *et al.* 96] A. Gangemi, G. Steve, and F. Giacomelli. ONIONS: An Ontological Methodology for Taxonomic Knowledge Integration. In P. van der Vet, editor, *Proceedings of the Workshop on Ontological Engineering, ECAI'96, Budapest, Hungary*, August 1996.

[Gangemi *et al.* 98] A. Gangemi, D. Pisanelli, and G. Steve. Ontology Integration: Experiences with Medical Terminologies. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontology in Information Systems, FOIS'98, Trento, Italy*, pages 163–178, June 1998.

[Genesereth & Fikes 92] Computer Science Dept., Stanford University. *Knowledge Interchange Format*, 3.0 edition, 1992. Technical Report, Logic-92-1.

[Gomez-Perez & Benjamins 99] A. Gomez-Perez and R. Benjamins. Overview of Knowledge Sharing and Reuse Components: Ontologies and Problem-Solving Methods. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, August 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

[Gomez-Perez & Royas-Amaya 99] A. Gomez-Perez and MD. Royas-Amaya. Ontological Reengineering for Reuse. In *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modeling and Management,(EKAW99), Dagstuhl, Germany*, pages 139–157, May 1999.

[Gomez-Perez 95] A. Gomez-Perez. Guidelines to Verify Completeness and Consistency in Ontologies. In *Proceedings of the 3rd World Congress on Expert Systems*, pages 901–908, February 1995.

[Gomez-Perez 96] A. Gomez-Perez. Towards a Framework to Verify Knowledge Sharing Technology. *Expert System with Applications*, 11(4):519–529, 1996.

[Gruber & Olsen 94] T. Gruber and G. Olsen. An ontology for engineering mathematics. In J. Doyle, P. Torasso, and E. Sandewall, editors, *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning, San Mateo, CA, USA.*, pages 258–269, 1994.

[Gruber 93] T.R. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[Gruber 95] T.R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. *International Journal of Human-Computer Studies*, 43:907–928, 1995.

[Gruninger & Fox 95] M. Gruninger and M.S. Fox. Methodology for the Design and Evaluation of Ontologies. In *Proceedings of the IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Quebec,Canada*, August 1995.

[Guarino & Giaretta 95] N. Guarino and P. Giaretta. Ontologies and Knowledge Bases: Towards a Terminological Clarification. In *Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB&KS'95), Twente, The Netherlands*, April 1995.

[Guarino & Poli 95] N. Guarino and R.(eds.) Poli. The Role of Ontology in the Information Technology. *International Journal of Human-Computer Studies*, 43(5/6):623–965, 1995.

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

[Guarino 98b] N. Guarino, editor. *Formal Ontology In Information Systems*, Frontiers in Artificial Intelligence and Applications. IOS Press, June 1998. ISBN: 90-5199-399-4.

[Guarino *et al.* 94] N. Guarino, M. Carrara, and P. Giaretta. Formalizing Ontological Commitments. In *In Proceedings of the 12th National Conference on Artificial Intelligence(AAAI'94), Seattle, Washington, USA*, 1994.

[Guarino *et al.* 99] N. Guarino, C. Masolo, and G. Vetere. OntoSeek: Content-Based Access to the Web. *IEEE Intelligent Systems*, 14(3):70–80, May 1999.

[Heitmeyer *et al.* 96] C.L. Heitmeyer, R. Djeffords, and B.G. Labow. Automated consistency checking of requirements specification. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, 1996.

[Howden 87] W.E. Howden. *Functional Program Testing and Analysis*. Software Engineering and Technology. McGraw-Hill, 1987.

[Jackson 95] M. Jackson. *Software Requirements & Specifications: a lexicon of practice, principles and prejudices.* Addison-Wesley, 1995. ISBN: 0-201-87712-0.

[Jannink *et al.* 98] J. Jannink, S. Pichai, D. Verheijen, and G. Wiederhold. Encapsulation and Composition of Ontologies. In *Proceedings of the AAAI'98 Workshop on Information Integration, Madison, WI, USA*, July 1998.

[Jones *et al.* 95] M. Jones, J. Wheadon, D. Whitgift, M. Niezatte, M. Timmermans, R. Rodriquez, and R. Romero. An agent-based approach to spacecraft mission operations. In N.J.I. Mars, editor, *Proceedings of 2nd International Conference on Knowledge Building and Knowledge Sharing(KB&KS'95), Twente, The Netherlands*, pages 259–269. IOS Press, April 1995.

[Kalfoglou & Robertson 99a] Y. Kalfoglou and D. Robertson. A Case Study in Applying Ontologies to Augment and Reason about the Correctness of Specifications. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 64–71, June 1999. Also as: Research Paper No.927, Dept. of AI, University of Edinburgh.

[Kalfoglou & Robertson 99b] Y. Kalfoglou and D. Robertson. Managing Ontological Constraints. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/, August 1999. Also as: Research Paper No.948, Dept. of AI, University of Edinburgh.

[Kalfoglou & Robertson 99c] Y. Kalfoglou and D. Robertson. Use of Formal Ontologies to Support Error Checking in Specifications. In D. Fensel and R. Studer, editors, *Proceedings of the 11th European Workshop on Knowledge Acquisition, Modelling and Management(EKAW99), Dagstuhl, Germany*, pages 207–220, May 1999. Also as: Research Paper No.935, Dept. of AI, University of Edinburgh.

[Kalfoglou & Robertson 00] Y. Kalfoglou and D. Robertson. Applying Experienceware to support ontology deployment. In *Proceedings of the 12th International Conference on Software Engineering and Knowledge Engineering, SEKE2000, Chicago, IL, USA*, July 2000.

[Kalfoglou 98] Y. Kalfoglou. Applying Ontological Engineering to Ecological Modelling: Detection of Conceptual Errors. Technical Paper No.47, Department of Artificial Intelligence, University of Edinburgh, December 1998.

[Kalfoglou 99] Y. Kalfoglou. The Role of Formal Ontologies. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 401–405, June 1999. Position paper presented at the Panel: Knowledge Maintenance. Also as: Research Paper No.952, Dept. of AI, University of Edinburgh.

[Kalfoglou 00] Y. Kalfoglou. On the convergence of core technologies for knowledge management and organisational memories: ontologies and experience factories. In *proceedings of the ECAI 2000 Workshop on Knowledge Management and Organisational Memories, Berlin, Germany*, August 2000.

[Kalfoglou *et al.* 00a] Y. Kalfoglou, T. Menzies, K-D. Althoff, and E. Motta. Meta-knowledge in systems design: panacea...or undelivered promise? *The Knowledge Engineering Review(submitted)*, 2000.

[Kalfoglou *et al.* 00b] Y. Kalfoglou, D. Robertson, and A. Tate. Using Meta-Knowledge at the Application Level. *Journal of Artificial Intelligence Research, submitted*, 2000. Also as: Research Paper No.956, Dept. of AI, University of Edinburgh.

[Kirani & Tsai 98] S. Kirani and T.W. Tsai. *Method sequence specification and verification of classes*, pages 43–53. Testing Object Oriented Software. IEEE Computer Society, 1998. ISBN: 0-8186-8520-4.

[Knight & Luk 94] K. Knight and S. Luk. Building a Large Knowledge Base for Machine Translation. In *Proceedings of the American Association of Artificial Intelligence Conference-AAAI 94, Seattle, USA*, July 1994.

[Kung *et al.* 96] D. Kung, Y. Lu, N. Venugopalan, P. Hsia, Y. Toyoshima, C. Chen, and J. Gao. Object State Testing and Fault Analysis for Reliable Software Systems. In *Proceedings of the 7th International Symposium on Software Reliability Engineering*, 1996.

[Kung *et al.* 98] D. Kung, P. Hsia, and J. Gao. *Testing Object Oriented Software*. IEEE Computer Society, 1998. ISBN: 0-8186-8520-4.

[Lee & Malone 90] 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.

[Lee *et al.* 96] 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.

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

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

[Lenat & Guha 90] D.B. Lenat and R.V. Guha. *Building large knowledge-based systems. Representation and inference in the Cyc project*. Addison-Wesley, Reading, Massachusetts, 1990.

[Leveson *et al.* 91] N. Leveson, S. Cha, and T. Shimeall. Safety verification of Ada programs using software fault trees. *IEEE Software*, pages 48–59, July 1991.

[Luke *et al.* 97] S. Luke, L. Spector, D. Rager, and J. Hendler. Ontology-based Web Agents. In W.L. Johnson, editor, *Proceedings of the 1st International Conference on Autonomous Agents(AA'97)*, pages 59–66. ACM, 1997.

[Luqi & Cooke 95] Luqi and D. Cooke. How to combine nonmonotonic logic and rapid prototyping to help maintain software. *International Journal of Software Engineering and Knowledge Engineering*, 5(1):89–118, 1995.

[Mark 96] W. Mark. Ontologies as Representation and Re-Representation of Agreement. In *Proceedings of the 5th International Conference on Principles of Knowledge Representation and Reasoning, KR'96, Massachusetts, USA*, 1996. Position paper presented on the panel: *Ontologies: What are they and where's the research.*

[Mark *et al.* 92] W. Mark, S. Tyler, J. McGuire, and J. Schossberg. Commitment-Based Software Development. *IEEE Transactions on Software Engineering*, 18(10):870–884, October 1992.

[Mark *et al.* 94] W. Mark, J. Schlossberg, S. Tyler, and J. McGuire. Cosmos: A System for Supporting Engineering Negotiation. *Concurrent Engineering: Research and Applications*, 2:173–182, 1994.

[Mark *et al.* 95] W. Mark, J. Dukes-Schlossberg, and R. Kerber. Ontological Commitment and Domain-Specific Architectures: Experience with Comet and Cosmos. In N.J.I.Mars, editor, *Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB & KS'95), Twente, The Netherlands*, pages 33–45, April 1995.

[Mateis *et al.* 99] C. Mateis, M. Stumper, and F. Wotawa. Debugging of Java Programs using a model-based apporach. In *Proceedings of the 10th International Workshop on Principles of Diagnosis(DX'99), Loch Awe, Scotland*, pages 166–173, June 1999.

[McGuinness 98] L.D. McGuinness. Ontological Issues for Knowledge-Enhanced Search. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontology in Information Systems(FOIS'98), Trento, Italy*, pages 302–316. IOS Press, June 1998.

[Mena *et al.* 98] E. Mena, V. Kashyap, A. Illarramendi, and A. Sheth. Domain Specific Ontologies for Semantic Information Brokering on the Global Information Infrastructure. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontology in Information Systems(FOIS'98), Trento, Italy*, pages 269–283. IOS Press, June 1998.

[Menzies *et al.* 00] T. Menzies, K-D. Althoff, Y. Kalfoglou, and E. Motta. Issues with Meta-Knowledge. *International Journal of Software Engineering and Knowledge Engineering(to appear)*, 10(4), August 2000.

[Meseguer & Preece 95] P. Meseguer and A. Preece. Verification and validation of knowledge-based systems with formal specifications. *The Knowledge Engineering Review*, 10(4):331–343, 1995.

[Meyer 88] B. Meyer. *Object-Oriented Software Construction.* Englewood Cliffs. Prentice Hall, NJ,USA, 1988.

[Meyers 79] G. Meyers. *The Art of Software Testing.* Englewood Cliffs. Prentice Hall, NJ,USA, 1979.

[Miller 90] G.A. Miller. WORDNET: an online lexical database. *International Journal of Lexicography*, 3(4):235–312, 1990.

[Mizoguchi *et al.* 95] R. Mizoguchi, J. van Welkenhuysen, and M. Ikeda. Task ontology for reuse of problem-solving knowledge. In N.J.I. Mars, editor, *Proceedings of the 2nd International Conference on Knowledge Building and Knowledge Sharing(KB & KS'95), Twente, The Netherlands*, pages 46–57, Amsterdam, NL., 1995. IOS Press.

[Moran & Carroll 96] T.P. Moran and J.M. Carroll. *Design Rationale: Concepts, Techniques, and Use.* Lawerence Erlbaum Associates, 1996. ISBN: 0-8058-1567-8.

[Morell 88] L.J. Morell. Theoretical insights into fault-based testing. In *Proceedings of 2nd Workshop on Software Testing, Verification, and Analysis, Banff, Canada*, pages 45–62, July 1988.

[Morell 90] L.J. Morell. A theory of fault-based testing. *IEEE Transactions on Software Engineering*, 16(8):844–857, August 1990.

[Motta 99] E. Motta. *Reusable Components for Knowledge Models: Case Studies in Parametric Design Problem Solving*, volume 53 of *Frontiers in Artificial Intelligence and Applications.* IOS Press, 1999. ISBN: 1-58603-003-5.

[Motta *et al.* 99] E. Motta, D. Fensel, M. Gaspari, and R. Benjamins. Specifications of Knowledge Components for Reuse. In *Proceedings of the 11th International Conference on Software Engineering and Knowledge Engineering, SEKE'99, Kaiserslauten, Germany*, pages 36–43, June 1999.

[Mullery 79] G. Mullery. CORE: A method for Controlled Requirements Specification. In *Proceedings of the 4th International Conference on Software Engineering(ICSE'79), Munich, Germany*, 1979.

[Naish 97] L. Naish. A Declarative Debugging Scheme. *The Journal of Functional and Logic Programming*, 3, 1997.

[Neches *et al.* 91] R. Neches, R.E. Fikes, T. Finin, T.R. Gruber, T. Senator, and W.R. Swartout. Enabling Technology for Knowledge Sharing. *AI Magazine*, 12(3):36–56, 1991.

[NLM'97 97] National Library of Medicine, Bethesda, Maryland, USA. *UMLS Knowledge Sources*, 1997.

[O'Keefe *et al.* 87] M. O'Keefe, O. Balci, and E. Smith. Validating expert system performance. *IEEE Expert*, 2(4):81–90, 1987.

[O'Leary 97] D. O'Leary. Impediments in the use of explicit ontologies for KBS development. *International Journal of Human-Computer Studies*, 46(2):327–337, 1997.

[O'Leary 98a] D. O'Leary. Knowledge Management Systems: Converting and Connecting. *IEEE Intelligent Systems*, 13(3):30–33, June 1998.

[O'Leary 98b] D. O'Leary. Using AI in Knowledge Management: Knowledge Bases and Ontologies. *IEEE Intelligent Systems*, 13(3):34–39, June 1998.

[Perry 95] W. Perry. *Effective Methods for Software Testing.* Willey & Sons, 1995. ISBN: 0-471-06097-6.

[Pinto *et al.* 99] S. Pinto, A. Gomez-Perez, and J. Martins. Some Issues on Ontology Integration. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, August 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

[Polyak 98] 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.

[Polyak *et al.* 98] 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.

[Potts 96] C. Potts. Supporting Software Design: Integrating Design Methods and Design Rationale. In

P.T. Moran and M.J. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, chapter 10, pages 295–321. Lawrence Erlbaum Associates, 1996.

[Preece & Shinghal 94] A. Preece and R. Shinghal. Foundation and application of knowledge base verification. *International Journal of Intelligent Systems*, 9(8):683–702, 1994.

[Preece *et al.* 92] A. Preece, R. Shinghal, and A. Batarekh. Principles and practice in verifying rule-based systems. *The Knowledge Engineering Review*, 7(2):115–141, 1992.

[Preece *et al.* 99] A. Preece, K. Hui, P. Gray, P. Marti, T. Bench-Capon, D. Jones, and Z. Cui. The KRAFT Architecture for Knowledge Fusion and Transformation. In *Proceedings of the 19th SGES International Conference on Knowledge-based Systems and Applied Artificial Intelligence(ES'99), Cambridge, England.* Springer Verlag, December 1999. Best Technical Paper Award.

[Rector *et al.* 95] A.L. Rector, W.A. Nowlan, and the GALEN Consortium. The GALEN Project. *Computer Methods and Programs in Biomedicine*, 45:75–78, 1995.

[Reiter 87] R. Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[Richardson & Thompson 93] J.D. Richardson and C.M. Thompson. An Analysis of Test Data Selection Criteria Using the RELAY Model of Fault Detection. *IEEE Transactions on Software Engineering*, 19(6):533–553, June 1993.

[Robertson 98] D. Robertson. Pitfalls of Formality in Early System Design. In *Proceedings of the 1998 ARO/NSF Monterey Workshop on Increasing the Practical Impact of Formal Methods for Computer-Aided Software Development, Carmal, California*, 1998.

[Robertson *et al.* 91] 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.

[SEJ96] Special issue on Viewpoints in Requirements Engineering. *Software Engineering Journal*, 11(1), January 1996.

[Shapiro 83] E.Y. Shapiro. *Algorithmic Program Debugging.* MIT Press, 1983.

[Shlaer & Mellor 92] S. Shlaer and S. Mellor. *Object Lifecycles: Modeling the World in States.* Prentice-Hall, 1992. IBSN: 0-136-29940-7.

[SICStus 95] SICStus. SICStus Prolog User's Manual. ISBN 91-630-3648-7, Intelligent Systems Laboratory - Swedish Institute of Computer Science, 1995.

[Simons 87] P. Simons. *Parts: A Study in Ontology*, pages 5–128. Oxford: Clarendon Press, 1987.

[Skuce 95] D. Skuce. Viewing Ontologies as Vocabulary: Merging and Documenting the Logical and Linguistic Views. In *Proceedings of the IJCAI'95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Quebec, Canada*, August 1995.

[Sommerville & Sawyer 97] I. Sommerville and P. Sawyer. *Requirements Engineering: a good practice guide.* John Wiley & Sons, 1997. ISBN: 0-471-97444-7.

[Sowa 00] J. Sowa. *Knowledge Representation: Logical, Philosophical, and Computational Foundations.* Brooks Cole Publishing Co., Pacific Grove,CA,USA, 2000. ISBN: 0-534-94965-7.

[Sterling & Shapiro 94] L. Sterling and E. Shapiro. *The Art of Prolog.* MIT Press, 4th edition, 1994. ISBN 0-262-69163-9.

[Studer *et al.* 98] R. Studer, V.R. Benjamins, and D. Fensel. Knowledge engineering, principles and methods. *Data and Knowledge Engineering*, 25(1-2):161–197, 1998.

[Summer & Buckingham-Shum 98] T. Summer and S. Buckingham-Shum. From Documents to Discourse: Shifting Conceptions of Scholarly Publishing. In *proceedings of the CHI'98: Human Factors in Computing Systems, Los Angeles, CA, USA*, pages 95–102. ACM Press, 1998.

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

[Tackett & vanDoren 99] D.B. Tackett and B. vanDoren. Process Control for Error-Free Software: A Software Success Story. *IEEE Software*, 16(3):24–29, May 1999.

[Tate 98] A. Tate. Roots of SPAR - Shared Planning and Activity Representation. *The Knowledge Engineering Review*, 13(1):121–128, 1998.

[Treuer & Wetter 93] J. Treuer and T. Wetter. Formal specification of Complex Reasoning systems. Ellis-Horwood, 1993.

[Uschold & Gruninger 96] M. Uschold and M. Gruninger. Ontologies: principles, methods and applications. *The Knowledge Engineering Review*, 11(2):93–136, November 1996.

[Uschold & Jasper 99] M. Uschold and R. Jasper. A Framework for Understanding and Classifying Ontology Applications. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm, Sweden*, August 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

[Uschold & King.M. 95] M. Uschold and King.M. Towards a methodology for building ontologies. In *Proceedings of the IJCAI-95 Workshop on Basic Ontological Issues in Knowledge Sharing, Montreal, Canada*, 1995.

[Uschold & Tate 98] M. Uschold and A.(eds.) Tate. Putting Ontologies to Use. *The Knowledge Engineering Review*, 13(1):1–128, 1998.

[Uschold 98a] M. Uschold. Knowledge level modelling: concepts and terminology. *The Knowledge Engineering Review*, 13(1):5–29, February 1998.

[Uschold 98b] 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.

[Uschold *et al.* 98a] 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.

[Uschold *et al.* 98b] M. Uschold, M. Healy, K. Williamson, P. Clark, and S. Woods. Ontology Reuse and Application. In N. Guarino, editor, *Proceedings of the 1st International Conference on Formal Ontology in Information Systems(FOIS'98), Trento, Italy*, pages 179–192. IOS Press, June 1998.

[Uschold *et al.* 98c] M. Uschold, M. King, S. Moralee, and Y. Zorgios. The enterprise ontology. *The Knowledge Engineering Review*, 13(1), February 1998. Also available as AIAI-TR-195 from AIAI, University of Edinburgh.

[Uschold *et al.* 99] M. Uschold, R. Jasper, and P. Clark. Three Approaches for Knowledge Sharing: A Comparative analysis. In *Proceedings of the 12th Knowledge Acquisition, Modelling and Management Workshop, KAW'99, Banff, Canada*, October 1999.

[Valente & Breuker 96] A. Valente and J. Breuker. Towards Principled Core Ontologies. In *Proceedings of the 10th Knowledge Acquisition of Knowledge-Based Systems Workshop(KAW'96),Banff,Canada*, November 1996.

[Valente *et al.* 99] A. Valente, T. Russ, R. MacGrecor, and W. Swartout. Building and (Re)Using an Ontology for Air Campaign Planning. *IEEE Intelligent Systems*, 14(1):27–36, January 1999.

[van derVet & Mars 93] P. van der Vet and N. Mars. Structured system of concepts for storing, retrieving, and manipulating chemical information. *Journal of Chemical Information and Computer Science*, 33:564–568, 1993.

[van derVet & Mars 98] P. van der Vet and N. Mars. Bottom-Up Construction of Ontologies. *IEEE Transactions on Knowledge and Data Engineering*, 10(4):513–526, 1998.

[vanHeijst *et al.* 97] G. van Heijst, A. Schreider, and B.(eds.) Wielinga. Using Explicit Ontologies in KBS Development. *International Journal of Human-Computer Studies*, 46(2/3):183–292, 1997.

[Visser & Tamma 99] P.R.S. Visser and V.A.M Tamma. An Experiment with Ontology-based Agent Clustering. In *Proceedings of the IJCAI-99 Workshop on Ontologies and Problem-Solving Methods(KRR5), Stockholm,Sweden*, August 1999. http://sunsite.informatik.rwth-aachen.de/Publications/CEUR-WS/Vol-18/.

[Visser *et al.* 98] P.R.S. Visser, D.M. Jones, T.J.M. Bench-Capon, and M.J.R. Shave. Assessing Heterogeneity by Classifying Ontology Mismatches. In N. Guarino, editor, *Proceedings of 1st International Conference on Formal Ontologies in Information Systems, FOIS'98, Trento, Italy*, pages 148–162. IOS Press, June 1998.

[Voas & McGraw 98] J. Voas and G. McGraw. *Software Fault Injection: inoculating programs against errors.* Wiley & Sons, NY, USA, 1998. ISBN: 0-471-18381-4.

[Voas & Miller 95] J. Voas and K. Miller. Software Testability: The New Verification. *IEEE Software*, 12(3), March 1995.

[Voas 92] J. Voas. PIE: A Failure-Based Technique. *IEEE Transactions on Software Engineering*, 18(8):717–727, August 1992.

[Waterson & Preece 99] A. Waterson and A. Preece. Verifying Ontological Commitment in Knowledge-based Systems. *Knowledge-Based Systems*, 12:45–54, April 1999.

[WfMC 96] WfMC. Workflow Management Coalition: Abstract Specification. WFMC-TC 1012, WfMC, October 1996. Interoperability demonstration presented at the 1996 Business Process and Workflow Conference in Amsterdam.

[Winograd 96] T. Winograd. *Bringing Design to Software.* Addison-Wesley, 1996. ISBN: 0-201-85491-0.

[Zave & Jackson 97] P. Zave and M. Jackson. Four Dark Corners of Requirements Engineering. *ACM Transactions on Software Engineering and Methodology*, 6(1):1–30, January 1997.

[Zeil 89] S.J. Zeil. Perturbation techniques for detecting domain errors. *IEEE Transactions on Software Engineering*, 15(6):737–746, June 1989.

Appendix A

A.1 Transformations of cases A and B in chapter 4.2

We are using the following rule:

$(\mathcal{A})\ A \rightarrow C \bigwedge B \rightarrow D \Rightarrow A \wedge B \rightarrow C \wedge D$

For clarity and brevity we will not deal with quantifiers. Cases A and B are presented as follows:

(1) $\neg(error(G,E) \wedge provable(E)) \rightarrow e(G,E) \notin Es$
(2) $(error(G,E) \wedge provable(E)) \rightarrow e(G,E) \in Es$

we make the following substitutions: E with $E1$ in formula 1 which is now written as:

(3) $\neg(error(G,E1) \wedge provable(E1)) \rightarrow e(G,E1) \notin Es$

and we substitute G with $E1$ and E with $E2$ in formula 2 which is now written as:

(4) $(error(E1,E2) \wedge provable(E2)) \rightarrow e(E1,E2) \in Es$

we apply the rule $(\mathcal{A})$ in formulae 3 and 4 to obtain the **consequence A1**:

(5) $\neg(error(G,E1) \wedge provable(E1)) \wedge (error(E1,E2) \wedge provable(E2)) \rightarrow$
$e(G,E1) \notin Es \wedge e(E1,E2) \in Es$

we also make the following substitutions: G with $E1$ and E with $E2$ in formula 1 which is now written as:

(6) $\neg(error(E1,E2) \wedge provable(E2)) \rightarrow e(E1,E2) \notin Es$

and we substitute E with $E1$ in formula 2 which is now written as:

(7) $(error(G, E1) \wedge provable(E1)) \rightarrow e(G, E1) \in Es$

we apply the rule $(\mathcal{A})$ in formulae 6 and 7 to obtain the **consequence B1**:

(8) $(error(G, E1) \wedge provable(E1)) \wedge \neg\ (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, E1) \in Es \wedge e(E1, E2) \notin Es$

we also perform the following substitutions: E with $\neg E1$ in formula 1 which is now written as:

(9) $\neg(error(G, \neg E1) \wedge provable(\neg E1)) \rightarrow e(G, \neg E1) \notin Es$

we apply the rule $(\mathcal{A})$ in formulae 9 and 4 to obtain **consequence A2**:

(10) $\neg(error(G, \neg E1) \wedge provable(\neg E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, \neg E1) \notin Es \wedge e(E1, E2) \in Es$

and we apply rule $(\mathcal{A})$ in formulae 7 and 4 to obtain **consequence B2**:

(11) $(error(G, E1) \wedge provable(E1)) \wedge (error(E1, E2) \wedge provable(E2)) \rightarrow$
$e(G, E1) \in Es \wedge e(E1, E2) \in Es$

Appendix B

B.1 Error checking meta-interpreter

This is the Prolog implementation of the error checking meta-interpreter. It is explained in section 4.7. The `proofcheck/6` and `proofmember/3` predicates are used to implement a different style of testing and their usage is demonstrated in an example case in section 5.3. For brevity we do list here auxiliary predicates such as `add/2` and `report/3` used for adding an element in the head of a list and traversing the elements of a list, respectively.

```
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|(L,Body)],Level));
    (ontologicalDefinition(L,(X:-Body)), solve(Body,[X|(L,Body)],L))),
    NextLevel is Level + 1,
    proofcheck(X,_,ProofList,Body,ProofBody,Level),
    proofmember(ProofBody,ProofList,NextLevel),
    detect_errors(X,Path,NextLevel).
proofcheck(X,L,L,_,_,Level):-proof(X,without_body,Z),
                             \+ X == Z,
                             report([X],Z,Level).
proofcheck(X,M,M,B,L,_):-proof(X,with_body,M),
```

```
                ((logical_expression(B), convert2list(B,[],L));add2list(B,[],L)).
proofcheck(_,L,L,_,_,_).
proofmember(List,ProofList,Level):-\+ var(List),
                                   not-a-member(ProofList,List,[],R),
                                   report(R,List,Level).
proofmember(_,_,_).
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);
             (T=proof, write(not_in_proof_tree(L,X,C)),nl)),
             fail.
show_errors.
clear_errors:- retractall(found_ontological_error(_,_,_,_,_)).
logical_expression((_,_)).
logical_expression((_;_)).
logical_expression(\+ _).
not-a-member([],_,R,R).
not-a-member([H|T],L,A,R):-((member(H,L),NewA=A;add(H,A,NewA)),
                           not-a-member(T,L,NewA,R).
add2list(Element,_,[]):- predicate_property(Element,built_in).
add2list(Element,List,[Element|List]).
convert2list((X,Y),L,L2):-add(X,L,L1),
                          convert2list(Y,L1,L2).
convert2list((X;Y),L,L2):- convert2list(X,L,L1),
                          convert2list(Y,L1,L2).
convert2list(B,L,L1) :- (\+ B = (_,_) ; \+ B = (_;_)), add(B,L,L1).
```

Appendix C

C.1 Separated inference engine and error checking meta-interpreter

This is the Prolog implementation of the error checking mechanism which separates the inference engine(lines 1 to 14) from the error checking procedure(lines 15 to 46). In addition, it includes the "skip-one" technique(lines 31 to 43) which is explained in an example case in section 7.1. The predicates `member/2` and `union/3` are not listed here but are given as standard predicates in many Prolog environments. The predicate `logical_expression/1` is given in the previous appendix whereas the predicates `report/1` and `display_errors` which are dealing with filtering and presentation of checking results do not deserve detailed scrutiny and thus we omit them from this listing.

```
1 prove(Goal):-solve(Goal),
2              onto_solve(Goal,E),
3              report(E),
4              display_errors.
5 solve((A,B)):-solve(A),solve(B).
6 solve((A;_B)):-solve(A).
7 solve((_A;B)):-solve(B).
8 solve(\+ X):- \+ solve(X).
9 solve(X):- \+ logical_expression(X),
10           predicate_property(X,built_in),
11           X.
12 solve(X):- \+ (logical_expression(X) ; predicate_property(X,built_in)),
13            (ontologicalDefinition(_,(X:-B)) ; specification(_,(X:-B)),
14            solve(B).
15 onto_solve((X1,X2),E):-onto_solve(X1,E1),
16                        onto_solve(X2,E2),
17                        union(E1,E2,E).
18 onto_solve((X1;X2),E):-onto_solve(X1,E);
19                        onto_solve(X2,E).
20 onto_solve(\+ X,[]):- \+ solve(X).
21 onto_solve(\+ X,[]):- \+ logical_expression(X),
22                       predicate_property(X, built_in),
```

```
                       X.
onto_solve(X,E):- \+ (logical_expression(X);predicate_property(X, built_in)),
                  solve(X),
                  (ontologicalDefinition(_,(X:-B));specification(_,(X:-B))),
                  onto_solve(B,Eb),
                  error_check(X,Ex),
                  union(Eb,Ex,E).
error_check(X,S):-setof(errors_found(X,E,Es),ontological_error(X,E,Es),S).
error_check(X,[('cannot prove the negated condition:  ',E,' on goal:  ',X,
               ' proceed to check for errors in:  ',Goal)|S]):-
        \+ ontological_error(X,_,_),
        error(_,X,E),
        E = (\+ Goal),
        \+ error(_,Goal,\+ X),
        error_check(Goal,S).
error_check(X,[('cannot prove condition:  ',Goal,' on goal:  ',X)|S]):-
        \+ ontological_error(X,_,_),
        \+ (X = (\+ _)),
        error(_,X,Goal),
        \+ error(_,Goal,X),
        error_check(Goal,S).
error_check(X,[]):- \+ ontological_error(X,_,_).
ontological_error(X,E,Es):-error(_,X,E),
                           copy_term(E,E1),
                           onto_solve(E1,Es).
```

Appendix D

D.1 Transformation to conjunctive Normal Form

This is the Prolog implementation of transforming the given ontological axioms to a conjunctive Normal Form(NF)(see section 4.7.2. It works as follows: in order to transform the given axiom in NF, check whether the axiom contains subterms(line 14), and if so, apply the set of rewrite rules(lines 5 to 12), 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 an axiom does not contain subterm(s) then the `contains/4` predicate, will substitute the subterm with the given axiom in order to apply the set of rewrite rules.

```
1 :-dynamic contains/4.
2 :-dynamic rewrite_expression/2.
3 :-dynamic rewrite/2.
4 :-dynamic contains_args/4.
5 rewrite((A=:=B),((A:-B),(B:-A))).
6 rewrite((A:-B),(\+ (A,\+ B))).
7 rewrite((\+ (A;B)),(\+ A,\+ B)).
8 rewrite((A,(B;C)),((A,B);(A,C))).
9 rewrite(((A;B),C),((A,C);(A,B))).
10 rewrite(((A;B);C),(A;(B;C))).
11 rewrite(((A,B),C),(A,(B,C))).
12 rewrite((\+ \+ A),A).
13 rewrite_expression(Constraint,Normalform):-
14                  contains(Constraint,Subterm,Newconstraint,Newterm),
15                  rewrite(Subterm,Newterm),
16                  rewrite_expression(Newconstraint,Normalform).
17 rewrite_expression(Constraint,Constraint):-
18                  \+ (contains(Constraint,Subterm,_,_),
19                  rewrite(Subterm,_)).
20 contains(Term,TempTerm,New,New):- nonvar(Term),TempTerm=Term.
21 contains(Term,Subterm,Newterm,Newsubterm):- nonvar(Term),
22                  Term =..  [F|Args],
23                  contains_args(Args,Subterm,Newargs,Newsubterm),
24                  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).
```

D.2 Definite Clause Grammar(DCG) parser

This is a Prolog program used to produce the designated predicates format we use in the multi-layer architecture, namely: *specification*/2, *ontological_definition*/2, *axiom*/3, *error*/3, *proof*/3(see section 4.7.1). It employs the DCG technique to produce the required format(lines 36 to 61). For brevity we do not list the DCG for *ontological_definition*/2 which is similar to that of *specification*/2 described in lines 36 to 45. We also do not list auxiliary predicates, such as *writelist*/1 and *replaceAssertRetract*/3 used to traverse the results returned from the DCGs in the form of a list and deal with specific built-in predicates such as *assert*/1 and *retract*/1.

```
1 translate(InputSpec,InputAxEr,Output,Index,Type):- see(InputSpec),
2                   tell(Output),
3                   read(Clause1),
4                   writeDynamicDeclarations,
5                   readspec(Clause1,Index),
6                   seen,
7                   see(InputAxEr),
8                   read(Clause2),
9                   Idx is Index+1,
10                  readaxioms_errors(Clause2,Idx,Type),
11                  told,
12                  seen.
13 writeDynamicDeclarations:- write(':-dynamic found_ontological_error/5.'),nl,
14                  write(':-dynamic axiom/3.'),nl,
15                  write(':-dynamic error/3.'),nl,
16                  write(':-dynamic proof/3.'),nl,nl.
17 readspec(end_of_file,_Index).
18 readspec(Clause,Index):- spec_trans(Clause,Index),
19                  read(NewClause),
20                  readspec(NewClause,Index).
21 readaxioms_errors(end_of_file,_Index,_Type).
22 readaxioms_errors(Clause,Index,Type):-
23                  ((Type='axioms',dcg_axioms_trans(Clause,Index,Res,[]));
24                  (Type='errors',dcg_error_trans(Clause,Index,Res,[]))),
25                  writelist(Res),nl,
26                  read(NewClause),
27                  readaxioms_errors(NewClause,Index,Type).
28 spec_trans(Spec,Index):- ((Spec = (Goal:-SubGoal),
29               dcg_spec_trans((Goal,SubGoal,Index),Res,[]));
30               (isMetaLogic(Spec);dcg_spec_trans((Spec,true,Index),Res,[]))),
```

```
                writelist(Res),nl.
isMetaLogic(Spec):- ((Spec=(:-dynamic Predicate),write(':-dynamic'),
        write(' '),write(Predicate),write('.'));
        (Spec=(:-consult(File)),write(':-consult('),write(File),write(').'));
        (Spec=(:-use_module(Module)),write(':-use_module('),
        write(Module),write(').'))),nl.
dcg_spec_trans((G,SubG,Index)) --> [specification],
             ['('],
             [Index],
             [',('],
             replaceAssertRetract(G,Goal,Index),
             [Goal],
             [':-'],
             ((SubG == true,[true]);
             (replaceAssertRetract(SubG,SubGoal,Index),[SubGoal])),
             [')).'].
dcg_axioms_trans((Header,Constraints),Index) --> [axiom],
             ['('],
             [Index],
             [','],
             [Header],
             [',('],
             [Constraints],
             [')).'].
dcg_error_trans((Header,ErrorCondition),Index) --> [error],
             ['('],
             [Index],
             [','],
             [Header],
             [',('],
             [ErrorCondition],
             [')).'].
...
```

Appendix E

E.1 The *EcoLogic* case: Rabbit-grass energy flow model

This is the Prolog implementation of the Rabbit-grass energy flow model borrowed from [Robertson *et al.* 91]. It also includes the definition of the *lower_in_food_chain* concept used in section 5.3

```
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)).
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)).
```

Appendix F

F.1 The *EcoLogic* case: State Transition model

This is the Prolog implementation of the State Transition model([Robertson *et al.* 91]) used in section 5.3.

```
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)).
```

Appendix G

G.1 The $KA^2/SHOE$ ontologies mapping case

There are two programs described here: the first one, in section G.1.1 performs an analysis of mapping a concept from one ontology to a corresponding one in the other ontology. To perform this analysis it takes into account the ISA hierarchies of both ontologies and uses a meta-interpreter program, included in section G.1.2. It also checks for relations that hold over the same ontological concepts(lines 42 to 56 in G.1.1). Examples of its use are included in section 5.5.1.

G.1.1 The mapping analysis program

```
1 :-consult('mapmeta').
2 concept(Common):- (shoe((C:-X,Y)),C=..[Common|_]),
3        (ka2((C:-Z,W)),C=..[Common|_]),
4        X=..[Term1|_], Y=..[Term2|_], Z=..[Term3|_], W=..[Term4|_],
5        write('Initial relationships:  '),nl,
6        write(Common),write('('),write(Term1),write(','),
7        write(Term2),write(').'),write(' for the shoe ontology'),nl,
8        write(Common),write('('),write(Term3),write(','),
9        write(Term4),write(').'),write(' for the ka2 ontology'),nl,
10       write('Analysis of mapping:  '),nl,
11       mapping(Common,Term1,Term2,Term3,Term4).
12 mapping(Common,A,B,A,B):- write('complete mapping'),nl,
13                           write(Common),write('('),write(A),
14                           write(','),write(B),write(').').
15 mapping(Common,Term1,_,Term3,_):- map(findall(X,is_a(X,Term1),L1),shoe),
16                                  map(findall(Y,is_a(Y,Term3),L2),ka2),
17                                  termMap(Common,first,Term1,L1,L2),
18                                  fail.
19 mapping(Common,_,Term2,_,Term4):- map(findall(X,is_a(X,Term2),L1),shoe),
20                                  map(findall(Y,is_a(Y,Term4),L2),ka2),
21                                  termMap(Common,second,Term2,L1,L2),
22                                  fail.
23 mapping(_,_,_,_,_).
```

```
termMap(Common,Order,Term,_,L2):- member(Term,L2,Index),
  write(Common),write('('),write(Term),write(').'),nl,
  write('mapping of the '),write(Order),write(' term with information loss'),nl,
  write('for the ka2, '),write('lost concepts of ka2:  '),write(Index),nl.
termMap(Common,Order,_,L1,L2):- common(L1,L2,List,Index1),
  common(L2,L1,_,Index2),
  \+ List=[],
  [Term|_]=List,
  write(Common),write('('),write(Term),write(').'),nl,
  write('mapping of the '),write(Order),write(' term with information loss'),nl,
  write('for both ontologies, '),nl,
  write('lost concepts of ka2:  '),write(Index1),nl,
  write('lost concepts of shoe:  '),write(Index2),nl.
common([],_,[],_).
common([H|T],L,[H|R],I):-member(H,L,I),!,
                         common(T,L,R,I).
common([_|T],L,R,I):- common(T,L,R,I).
member(X,[X|_],1).
member(X,[_|Y],M):- member(X,Y,I), M is I+1.
relation(A,B):- ((shoe((X:-(A,B))), ka2((Y:-(A,B))));
                (shoe((X:-(B,A))), ka2((Y:-(A,B))));
                (shoe((X:-(A,B))), ka2((Y:-(B,A))));
                (shoe((X:-(B,A))), ka2((Y:-(B,A))))),
                write('common relation:  '),nl,
                write(X),write(' for SHOE, '),nl,
                write(Y),write(' for KA2 '),nl,
                write('hold upon the same concepts:  '),
                write(A),write(', '),write(B).
relationKA2(A,B):-(ka2((X:-(A,B)));
                  ka2((X:-(B,A)))),
                  write(X).
relationSHOE(A,B):-(shoe((X:-(A,B)));
                   shoe((X:-(B,A)))),
                   write(X).
immediateParentKA2(A,B,L):- findall(A,ka2((isa(A,B):-true)),L).
immediateChildrenKA2(A,B,L):- findall(B,ka2((isa(A,B):-true)),L).
immediateParentSHOE(A,B,L):- findall(A,shoe((isa(A,B):-true)),L).
immediateChildrenSHOE(A,B,L):- findall(B,shoe((isa(A,B):-true)),L).
```

G.1.2 The mapping meta-interpreter program

```
map((A,B),Ontology):- map(A,Ontology),
                      map(B,Ontology).
map((A;B),Ontology):- map(A,Ontology);
                      map(B,Ontology).
map(\+ A,Ontology):- \+ map(A,Ontology).
map(A,Ontology):- \+ logical_expression(A),
```

```
                     predicate_property(A,(meta_predicate _)),
                     map_metapred(A,Call,Ontology),!,
                     Call.
map_metapred(findall(X,Z,L),findall(X,map(Z,Ontology),L),Ontology).
map_metapred(setof(X,Z,L),setof(X,map(Z,Ontology),L),Ontology).
map(A,_):-\+ logical_expression(A),
          predicate_property(A, built_in),
          A.
map(A,shoe):-\+(logical_expression(A);predicate_property(A,built_in)),
             shoe((A :- Body)),
             map(Body,shoe).
map(A,ka2):-\+(logical_expression(A);predicate_property(A, built_in)),
            ka2((A :- Body)),
            map(Body,ka2).
logical_expression((_,_)).
logical_expression((_;_)).
logical_expression(\+ _).
```

Appendix H

H.1 The Object State Testing case

This program combines a generic state transition machine(lines 1 to 15) originally described in [Sterling & Shapiro 94] with the example case of traffic light application described in [Kung *et al.* 96]. This implementation of transitions between different states of the two traffic lights being modelled(*east* and *north*) realises the correct path: from (2,1) to (3,3) to (1,2) to (3,3) and then back to (2,1) for the pair (*east*,*north*) where 1 stands for red light, 2 for green, and 3 for yellow. However, as we described in section 5.5.2 a misinterpretation of *memberFunction* construct(lines 28 to 30) and the corresponding *applyFunction* declarations(lines 43 to 58) could lead to undesirable behaviour. Hence the ontological constraint of line 75 which is tailored to monitor the behaviour of *applyFunction* construct.

```
1 ontologicalDefinition(0,(find_finalState(State,_,[]):- finalState(State))).
2 ontologicalDefinition(0,(find_finalState(State,History,[Move|Moves]):-
3                        move(State,Move),
4                        update(State,Move,NewState),
5                        legal(NewState),
6                        \+ member(NewState,History),
7                        find_finalState(NewState,[NewState|History],Moves))).
8 ontologicalDefinition(0,(possible_moves(Moves):- initial_state(State),
9                                  find_finalState(State,[State],Moves))).
10 ontologicalDefinition(0,(move(State,Move):-memberFunction(Move),
11                                         possible_state(State),
12                                         applicable(Move,State))).
13 ontologicalDefinition(0,(update(State,Move,NewState):-
14                                    applyFunction(Move,State,NewState))).
15 ontologicalDefinition(0,(legal(NewState):- \+ illegal(NewState))).
16 specification(0,(initial_state([east(2),north(1),
17                               direction(east),nofinal]):-true)).
18 specification(0,(finalState([east(2),north(1),
19                           direction(east),final]):-true)).
20 specification(0,(possible_state([east(2),north(1),
21                                direction(east),nofinal]):-true)).
22 specification(0,(possible_state([east(1),north(2),
```

```
                               direction(north),nofinal]):-true)).
specification(0,(possible_state([east(3),north(3),
                               direction(_),nofinal]):-true)).
specification(0,(possible_state([east(2),north(1),
                               direction(east),final]):-true)).
specification(0,(memberFunction(setLight(east(1),north(2))):-true)).
specification(0,(memberFunction(setLight(east(3),north(3))):-true)).
specification(0,(memberFunction(setLight(east(2),north(1))):-true)).
specification(0,(applicable(setLight(east(1),_),State):-
                                   member(east(3),State))).
specification(0,(applicable(setLight(east(2),_),State):-
                                   member(east(3),State))).
specification(0,(applicable(setLight(east(3),_),State):-
                          member(east(2),State);member(east(1),State))).
specification(0,(applicable(setLight(_,north(1)),State):-
                          member(north(3),State))).
specification(0,(applicable(setLight(north(2),_),State):-
                          member(north(3),State))).
specification(0,(applicable(setLight(north(3),_),State):-
                          member(north(2),State);member(north(1),State))).
specification(0,(applyFunction(setLight(east(1),north(2)),State,NewState):-
          remove(east(_),State,S1),insert(east(1),S1,S2),
          remove(north(_),S2,S3),insert(north(2),S3,S4),
          remove(direction(_),S4,S5),insert(direction(north),S5,NewState))).
specification(0,(applyFunction(setLight(east(3),north(3)),State,NewState):-
          remove(east(_),State,S1),insert(east(3),S1,S2),
          remove(north(_),S2,S3),insert(north(3),S3,S4),
          ((member(direction(north),S4),
          remove(direction(_),S4,S5),insert(direction(east),S5,NewState));
          (member(direction(east),S4),
          remove(direction(_),S4,S5),insert(direction(north),S5,NewState))))).
specification(0,(applyFunction(setLight(east(2),north(1)),State,NewState):-
          remove(east(_),State,S1),insert(east(2),S1,S2),
          remove(north(_),S2,S3),insert(north(1),S3,S4),
          remove(nofinal,S4,S5),insert(final,S5,S6),
          remove(direction(_),S6,S7),insert(direction(east),S7,NewState))).
specification(0,(illegal(NewState):-member(east(2),NewState),
                                  member(north(2),NewState))).
specification(0,(member(X,[X|_]):-true)).
specification(0,(member(X,[_|Y]):- member(X,Y))).
specification(0,(insert(X,[Y|Ys],[X,Y|Ys]):- precedes(X,Y))).
specification(0,(insert(X,[Y|Ys],[Y|Zs]):-precedes(Y,X),
                                       insert(X,Ys,Zs))).
specification(0,(insert(X,[],[X]):-true)).
specification(0,(precedes(east(_),_):-true)).
specification(0,(precedes(north(_),A):- \+ A = east(_))).
specification(0,(precedes(direction(_),A):- \+(A=east(_);A=north(_)))).
specification(0,(precedes(_,final):-true)).
```

```
specification(0,(precedes(_,nofinal):-true)).
specification(0,(remove(_,[],[]):-true)).
specification(0,(remove(X,[X|T],S):- remove(X,T,S))).
specification(0,(remove(X,[Y|T],[Y|S]):- remove(X,T,S))).
error(1,applyFunction(setLight(_,_),[east(1),_,_,_],NewState),
                       member(east(2),NewState)).
```

Appendix I

I.1 TOVE project: Axioms for temporal relations

Axiom 1: "strictly equals"(for timepoints):

```
strictly_equals_tp(T1,T2):- time_point(T1,Min1,Max1),
       time_point(T2,Min2,Max2),
       Min1 == Max1,Min2 == Max2,Min1 == Min2.
```

Axiom 2: "strictly equals"(for time periods):

```
strictly_equals_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
       time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
       strictly_equals_tp(SP1,SP2),strictly_equals_tp(EP1,EP2).
```

Axiom 3: "possibly equals"(for timepoints):

```
possibly_equals_tp(T1,T2):- time_point(T1,Min1,Max1),
       time_point(T2,Min2,Max2),
       Max1 >= Min2,Min1 =< Max2.
```

Axiom 4: "possibly equals"(for time periods):

```
possibly_equals_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
       time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
       possibly_equals_tp(SP1,SP2),possibly_equals_tp(EP1,EP2).
```

Axiom 5: "strictly before"(for timepoints):

```
strictly_before_tp(T1,T2):- time_point(T1,Min1,Max1),
       time_point(T2,Min2,Max2),Max1 < Min2.
```

Axiom 6: "strictly before"(for time periods):

```
strictly_before_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
       time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
       strictly_before_tp(EP1,SP2).
```

Axiom 7: "strictly before"(for timepoint and time period):

```
strictly_before(T1,T2):- time_point(T1,Min1,Max1),
       time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
       strictly_before_tp(T1,SP2).
```

Axiom 8: "strictly before"(for time period and timepoint):

```
strictly_before(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
       time_point(T2,Min2,Max2),strictly_after_tp(T2,T1).
```

Axiom 9: "possibly before"(for timepoints):

```
possibly_before_tp(T1,T2):- time_point(T1,Min1,Max1),
       time_point(T2,Min2,Max2),Max1 >= Min2,Min1 < Max2.
```

Axiom 10: "possibly before"(for time periods):
```
possibly_before_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
      possibly_before_tp(EP1,SP2).
```
Axiom 11: "possibly before"(for timepoint and time period):
```
possibly_before(T1,T2):- time_point(T1,Min1,Max1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
      possibly_before_tp(T1,SP2).
```
Axiom 12: "possibly before"(for time period and timepoint):
```
possibly_before(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_point(T2,Min2,Max2),possibly_after_tp(T2,T1).
```
Axiom 13: "strictly after"(for timepoints):
```
strictly_after_tp(T1,T2):- time_point(T1,Min1,Max1),
      time_point(T2,Min2,Max2),strictly_before_tp(T2,T1).
```
Axiom 14: "strictly after"(for time periods):
```
strictly_after_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_before_tp(T2,T1).
```
Axiom 15: "strictly after"(for timepoint and time period):
```
strictly_after(T1,T2):- time_point(T1,Min1,Max1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_after_tp(T1,EP2).
```
Axiom 16: "strictly after"(for time period and timepoint):
```
strictly_after(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_point(T2,Min2,Max2),strictly_before_tp(T2,T1).
```
Axiom 17: "possibly after"(for timepoints):
```
possibly_after_tp(T1,T2):- possibly_before_tp(T2,T1).
```
Axiom 18: "possibly after"(for time periods):
```
possibly_after_p(T1,T2):- possibly_before_p(T2,T1).
```
Axiom 19: "possibly after"(for timepoint and time period):
```
possibly_after(T1,T2):- time_point(T1,Min1,Max1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_after_tp(T1,EP2).
```
Axiom 20: "possibly after"(for time period and timepoint):
```
possibly_after(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_point(T2,Min2,Max2),possibly_before_tp(T2,T1).
```
Axiom 21: "strictly contains"(for time periods):
```
strictly_contains_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
      strictly_before_tp(SP1,SP2),strictly_before_tp(EP2,EP1).
```
Axiom 22: "strictly contains"(for time period and timepoint):
```
strictly_contains(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_point(T2,Min2,Max2),
      strictly_before_tp(SP1,T2),stricly_after_tp(EP1,T2).
```
Axiom 23: "possibly contains"(for time periods):
```
possibly_contains_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
     ((possibly_before_tp(SP1,SP2),possibly_before_tp(EP2,EP1));
     (strictly_before_tp(SP1,SP2),possibly_before_tp(EP2,EP1));
     (possibly_before_tp(SP1,SP2),strictly_before_tp(EP2,EP1))).
```
Axiom 24: "possibly contains"(for time period and timepoint):

```
possibly_contains(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_point(T2,Min2,Max2),
      ((possibly_before_tp(SP1,T2),strictly_after_tp(EP1,T2));
      (strictly_before_tp(SP1,T2),possibly_after_tp(EP1,T2));
      (possibly_before_tp(SP1,T2),possibly_after_tp(EP1,T2))).
Axiom 25:  "strictly during"(for time periods):
strictly_during_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_contains_p(T2,T1).
Axiom 26:  "strictly during"(for timepoint and time period):
strictly_during(T1,T2):- time_point(T1,Min1,Max1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_contains(T2,T1).
Axiom 27:  "possibly during"(for time periods):
possibly_during_p(T1,T2):- possibly_contains_p(T2,T1).
Axiom 28:  "possibly during"(for timepoint and time period):
possibly_during(T1,T2):- time_point(T1,Min1,Max1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_contains(T2,T1).
Axiom 29:  "strictly meets"(for time periods):
strictly_meets_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_equals_tp(EP1,SP2).
Axiom 30:  "possibly meets"(for time periods):
possibly_meets_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_equals_tp(EP1,SP2).
Axiom 31:  "strictly met by"(for time periods):
strictly_met_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_meets_p(T2,T1).
Axiom 32:  "possibly met by"(for time periods):
possibly_met_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_meets_p(T2,T1).
Axiom 33:  "strictly overlaps"(for time periods):
strictly_overlaps_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
      strictly_before_tp(SP2,EP1),strictly_before_tp(EP1,EP2).
Axiom 34:  "possibly overlaps"(for time periods):
possibly_overlaps_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
      possibly_before_tp(SP2,EP1),possibly_before_tp(EP1,EP2).
Axiom 35:  "strictly overlapped by"(for time periods):
strictly_overlapped_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_overlaps_p(T2,T1).
Axiom 36:  "possibly overlapped by"(for time periods):
possibly_overlapped_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_overlaps(T2,T1).
Axiom 37:  "strictly starts"(for time periods):
strictly_starts_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
      time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
      strictly_equals_tp(SP1,SP2),strictly_before_tp(EP1,EP2).
Axiom 38:  "strictly starts"(for timepoint and time period):
strictly_starts(T1,T2):- time_point(T1,Min1,Max1),
```

```
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_equals_tp(T1,SP2).
```

Axiom 39: "possibly starts"(for time periods):

```
possibly_starts_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_equals_tp(SP1,SP2),
        (strictly_before_tp(EP1,EP2);possibly_before_tp(EP1,EP2)),Dmin1 < Dmax2.
```

Axiom 40: "possibly starts"(for timepoint and time period):

```
possibly_starts(T1,T2):- time_point(T1,Min1,Max1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_equals_tp(T1,SP2).
```

Axiom 41: "strictly started by"(for time periods):

```
strictly_started_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_starts_p(T2,T1).
```

Axiom 42: "strictly started by"(for time period and timepoint):

```
strictly_started_by(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_point(T2,Min2,Max2),strictly_starts(T2,T1).
```

Axiom 43: "possibly started by"(for time periods):

```
possibly_started_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_starts_p(T2,T1).
```

Axiom 44: "possibly started by"(for time period and timepoint):

```
possibly_started_by(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_point(T2,Min2,Max2),possibly_starts(T2,T1).
```

Axiom 45: "strictly ends"(for time periods):

```
strictly_ends_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),
        strictly_equals_tp(EP1,EP2),strictly_after_tp(SP1,SP2).
```

Axiom 46: "strictly ends"(for timepoint and time period):

```
strictly_ends(T1,T2):- time_point(T1,Min1,Max1),
time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_equals_tp(T1,EP2).
```

Axiom 47: "possibly ends"(for time periods):

```
possibly_ends_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_equals_tp(EP1,EP2),
        (strictly_after_tp(SP1,SP2);possibly_after_tp(SP1,SP2)),Dmin1 < Dmax2.
```

Axiom 48: "possibly ends"(for timepoint and time period):

```
possibly_ends(T1,T2):- time_point(T1,Min1,Max1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_equals_tp(T1,EP2).
```

Axiom 49: "strictly ended by"(for time periods):

```
strictly_ended_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),strictly_ends_p(T2,T1).
```

Axiom 50: "strictly ended by"(for time period and timepoint):

```
strictly_ended_by(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_point(T2,Min2,Max2),strictly_ends(T2,T1).
```

Axiom 51: "possibly ended by"(for time periods):

```
possibly_ended_by_p(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_period(T2,SP2,EP2,Dmin2,D2,Dmax2),possibly_ends_p(T2,T1).
```

Axiom 52: "possibly ended by"(for time period and timepoint):

```
possibly_ended_by(T1,T2):- time_period(T1,SP1,EP1,Dmin1,D1,Dmax1),
        time_point(T2,Min2,Max2),possibly_ends(T2,T1).
```