UNIVERSITAT POLITECNICA DE CATALUNYA

DEPARTAMENT DE LLENGUATGES Y SISTEMES INFORMATICS

PROGRAMA DE DOCTORAT EN INTEL·LIGENCIA ARTIFICIAL

TESIS DOCTORAL

Validation of Multi-Level Rule-Based Expert Systems

Abril 1992

Memoria presentada por Pedro Meseguer González para optar al título de Doctor en Informática. El trabajo contenido en esta memoria ha sido realizado en el Institut d'Investigació en Intel.ligència Artificial (Centre d'Estudis Avançats de Blanes-C.S.I.C.).

Director: Ramon López de Mántaras i Badia

Abstract

Expert systems (ESs), as any other kind of software, should be validated. ESs present some differences with respect to conventional software, because of that validation methods used in conventional software are not directly applicable to ESs. Some of these methods can be adapted to ESs, although specific methods for ESs are also required. After a wide review of previous work in this field, we analyze ES validation as a whole. We consider a number of fundamental questions (what, when, how) drawing the complete picture of the problem. We devote special attention to terminological issues, proposing a number of definitions for validation terms that include specific ES characteristics and keep the basic meanings of these terms in software engineering. They fit pretty well in a general framework for software validation, where different kinds of software coexist with a common understanding of the kernel issues.

We consider multi-level rule-based ESs performing medical diagnosis. These systems contain explicit representations of domain and control knowledge, including uncertainty management. For them, we present two new validation methods. The first one is a verification method that checks a number of properties in the KB, in order to assure its structural correctness. New verification issues appear on this ES model, caused by the presence of uncertainty and control knowledge. The verification method solves these issues using extended labels, an extension of ATMS constructs. We have implemented an incremental version of this method in the verifier IN-DEPTH II. Using it, we are verifying the expert system PNEUMON-IA with encouraging results. We have detected and corrected a number of errors that, without the verifier help, would have been missed. In addition, we can guarantee the absence of certain types of errors.

The second validation method is a refinement system to improve the ES performance. With respect to previous refinement approaches, this method provides three new contributions. First, ES performance is not measured as the raw number of errors performed by the ES but considering the relative importance of these errors for the ES task. Second, domain and control knowledge are subject to refinement. And third, a new type of error, ordering mismatch, is considered as a consequence of working with ESs providing multiple diagnoses. We have implemented this method in IMPROVER, an automatic refinement tool. Using IMPROVER on a library of 66 cases, we have refined PNEUMON-IA obtaining very good results, specially considering false negatives, the most important error type.

Keywords: validation, verification, testing, refinement, evaluation, ES life-cycle, user requirements, incremental verification, gold standard, medical ES validation.

A Romero.

Preface

Today, expert systems are not longer experimental programs. They have shown their effectiveness in solving complex problems in many different settings. In the last ten years, knowledge engineering has developed a significant number of methods and techniques for expert system construction. Now, these techniques allow to build expert systems on an industrial basis. In this context, a question appears: do really expert systems do what they intend to do? This question, that was previously occluded by other more exciting problems, is of fundamental importance for the use of expert systems on a regular basis. Users demand substantive evidences of expert system correctness prior to their acceptance and use. This demand has generated a new subfield in expert system research, expert system validation, that is the topic of this thesis.

Since I started on this topic, I have followed two drawing ideas: practical validation and realistic models. Practical validation refers to the imperious necessity to show the effectiveness of validation methods *in practice*. Without this test, validation methods cannot be considered really useful. Realistic model refers to the necessity to consider for validation all the information contained in the knowledge base. Many validation methods assume simplistic expert system models, what render them inapplicable to actual systems. These two ideas have been complemented with a third one, the use of automatic tools to support an effective validation. This is required by the high number of different situations that can occur in rule-based expert systems. For an actual application, manual checking of some properties is unfeasible.

This work is biased towards the validation of implemented expert systems. This is partially due to the ideas exposed in the last paragraph, but also by the availability of a shell, MILORD, and an application, PNEUMON-IA, that presented interesting validation problems not considered before. In addition, MILORD and PNEUMON-IA developers were accessible to discuss about these topics. I have exploited this infrequent situation, aware of its potential value. Results confirm that it was a good choice.

A substantial part of the work presented here was developed when I was working in the VALID project. Although this work is not directly bounded with the VALID results, it has been obviously benefited from all the studies and developments on validation made in VALID. The first version of the IN-DEPTH verifier was developed inside the project.

Discussions among VALID partners have also contributed to the clarification and consolidation of validation concepts.

This work is the natural continuation of a research line on expert systems in the IIIA (Institute for Research on Artificial Intelligence). Previous works have been focused on knowledge acquisition, shell construction, medical expert system development and uncertainty management.

Acknowledgements

This work has been influenced by many people. I specially thank Ramón López de Mántaras, who introduced me in research activities and who has provided guidance and support in the development of this work. Enric Plaza, CEAB project leader in the VALID project, has been the partner for innumerable and vivid discussions on validation topics. Carlos Sierra, MILORD developer, has provided extensive support explaining how and why MILORD works. Lluis Godo has collaborated in all the questions dealing with uncertainty management. I have to make a special mention to Albert Verdaguer, the physician who developed PNEUMON-IA, who has largely contributed to achieve the practical results of this work. Without his cooperation, an important part of this thesis would have not been possible.

I also thank Francesc Esteva, CEAB director, who has always provided every facility to perform research at CEAB. Last but not least, I thank Romero for her constant support during the development of this work.

Preliminary stages of this work were supported by a scholarship CSIC-La Caixa. Later, a substantive part of this work was partially funded by the ESPRIT II project VALID, under the grant #2148.

Contents

Chapter 1

Introduction

This thesis deals with validation of experts systems, a special kind of software. In general, validation of a product consists in checking that this product is fully operational and performs satisfactorily its intended function. When applied to software, validation aims at checking that programs are free of errors and satisfy the users needs. A number of different methods and techniques exists for program validation. In practice, no method assures total program correctness. Each method provides partial evidences about program validity. To achieve a reasonable degree of confidence in program correctness, a combination of validation methods providing complementary evidences is usually required. It is well known that validation cannot be delayed until program implementation. Then, the cost of error correction can be very expensive, since the required modifications can imply significant changes in the implemented program. To prevent that, validation activities are distributed during the software life-cycle, from specifications to maintenance. Nevertheless in practice, the major part of validation activities are still made in the implementation phase.

Experts systems (ESs) are problem solvers for specialized domains of competence in which effective problem solving normally requires human expertise. ESs are included in the broader class of artificial intelligence systems, in the sense that they solve problems that could previously be solved only applying human intelligence. ESs present two main differences with respect to conventional software: the type of problems considered and the type of development techniques used. Regarding the type of problems, typical ES applications are medical diagnosis, troubleshooting, credit authorization, equipment configuration, and others. These tasks are usually ill-structured and no efficient algorithmic approach is known for them. Humans solve these tasks using knowledge and previous experience. The combination of these two aspects is called *expertise*, and is very important to achieve efficiently an acceptable solution. Regarding the type of development techniques, ESs are constructed using declarative languages (rule or frame based) that are interpreted by inference engines. This kind of languages is mainly based on logic. ES programming is concerned with truth values, rule dependencies and heuristic associations, in contrast to conventional programming that deals with variables, conditionals, loops and procedures.

In its final aims ES validation does not differ from conventional software validation. However, differences between ESs and conventional software make difficult the application to ESs of validation methods successfully used in conventional software. ES peculiarities, such as the definition of acceptance in terms of human expert competence, cause new validation issues that have not been addressed before. In addition, differences in the programming languages used make inapplicable specific techniques of conventional software. Therefore, although the basic validation concepts are shared by conventional software and ESs, validation methods of conventional software are not directly applicable to ESs. Some of them can be adapted, while new methods specific for ES validation are also required. In this context, this thesis aims to be a step forward in the definition and consolidation of validation concepts for ESs, as well as in the development of methods allowing for an effective validation of actual ESs.

The structure of this introductory chapter is as follows. In section 1.1 we give some reasons about the necessity of validation in ESs. In section 1.2 we compare ESs versus conventional software, enumerating commonalties and differences. From this analysis, we extract validation issues that are specific for ESs. In section 1.3 we describe the approach to validation taken in this thesis. We consider ES validation as a part of software validation, that has to be developed without ignoring all the achievements and experience obtained in software engineering. Regarding new validation methods for ESs, our work is based on the three following guidelines: practical validation, realistic ES models and automatic tools. Finally, in section 1.4 we provide a brief overview of the thesis contents, describing its main parts and summarizing the obtained results.

1.1 Motivation

ESs are programs and programs must be validated. Even more, validation is a part of the programming activity [Adrion et al, 82]. This is the first reason justifying why validation is required for ESs. In addition to that, a number of characteristics specific for ES reinforce this necessity of validation. These characteristics are related to the kind of tasks performed by ESs and to the available techniques for ES development and implementation. Without aiming to be exhaustive, we discuss some of them in the following.

ESs perform tasks on specialized domains of competence usually performed by human experts. In any field, expertise is scarce and expensive, so ESs perform tasks that are economically relevant. In addition, some ESs are used in situations where a failure can cause consequences that are very expensive to recover. To illustrate these ideas, let us consider an ES for troubleshooting of satellite devices. Such an ES has a quite relevant economic importance, because if the ES can substitute the presence of astronauts in the satellite, it will save large amounts of money (sending a computer is cheaper than sending an astronaut!). In addition, if the ES fails and does not solve a problem in a satellite device, this may cause a satellite crash with the corresponding catastrophic consequences. This example, although extreme, indicates clearly the relevance of the tasks executed by ESs. The degree of validation that ESs have to achieve should be at least similar to the degree of validation of applications of conventional software that perform less important tasks. Currently, this is not the case since validation is more developed in conventional software than in ESs. Nevertheless, this situation reinforces the necessity of better methods for ES validation.

ESs perform tasks that until now have been made by humans. These tasks are usually ill-structured and they have to deal with incomplete, uncertain or even inconsistent information. Task boundaries are normally fuzzy. In practice, a task instance frequently does not have a single solution; on the contrary multiple solutions may be equally acceptable. In this context, a precise definition for the intended task is difficult. An incomplete definition of the ES task may cause important deviations of the implemented system with respect to the original aim. In addition, difficulties in ES task definition have direct consequences in ES evaluation, that usually has to rely on matching ES performance against human expert competence. However, human expert behavior is not always exact and coherent. Some experts exhibit prejudice against computers, others exhibit prejudice against other experts. Discrepancies among experts always exist. These issues have been studied in some ES experiences, getting answers for their specific needs (see in sections 2.6.1 and 2.6.2 the case studies of MYCIN and R1), although they still hold today when a new ES is developed. In summary, the elusive nature of typical ES tasks makes difficult their precise definition. This increases the risk of errors in ES development and hinders their detection in ES evaluation. To overcome these shortcomings sound validation methods for ESs are required.

ESs are usually constructed using rule-based languages. The knowledge engineer develops condition-action pairs, that are conceived separately (or in small groups) and joined together forming the knowledge base (KB). However, the ES behavior arises from the cooperative interaction of these rules. Given that the number of possible interactions among rules raises exponentially with the number of rules[1], for applications with hundreds or thousands of rules the number of possible interactions in the KB is very high. Then, spurious or unexpected interactions among rules may appear in ES execution, causing undesired behaviors. Preventing in advance these interactions is a difficult task for the knowledge engineer, who cannot be totally aware of all possible interactions. In addition, other factors such as a long KB development period, multiple KB builders or KB maintenance can contribute negatively to this problem. Therefore, specific validation techniques are required for rule-based ESs, in order to assure a correct KB structure and to prevent as far as possible unexpected interactions.

In summary, we conclude that ESs perform important tasks that demand a significant degree of validation. A precise definition of these tasks is difficult, what causes problems for ES development and evaluation. In addition, rule-based languages do not provide enough security in their implementation. All these points reinforce the necessity of ES validation. Currently, the degree of validation reachable in ESs is far below the validation degree reachable in conventional software. In spite of the significant successes obtained by the ES technology[2] and its potential benefits, it will not be really applicable in industrial settings if it cannot guarantee the validity of its products. In consequence, better validation methods and techniques are required, with the final aim to achieve for ESs the confidence level that is currently available for conventional software.

[1] Assuming that the number of possible rule chaining increases as the number of rules increases, that is the usual situation.

[2] An important deposit of molybdenum was discovered with the help of the PROSPECTOR expert system.

1.2 Experts Systems versus Conventional Software

What differentiates ESs from conventional software?. Are these differences so important to justify specific validation methods?. In the first part of this chapter, we have mentioned two basic differences between ESs and conventional software: the type of tasks considered and the type of development techniques used. From these two basic differences, we extract the following detailed points:

- *Solving Approach*: ESs are programs that solve problems by heuristic associations. ESs are based on models of the intended task, models that are inspired in the behavior of a human expert in the problem domain. Conversely, conventional software solves problems by means of algorithms. This is a quite fundamental difference regarding validation. Algorithms can be proved to be correct, and this provides substantive evidence of the correctness of a program (though the specific implementation can contain some errors). However, we do not dispose of an analogous formalization for ES tasks. The correctness of the underlying model is guaranteed by human experts, subject to personal opinion or prejudice. This point is, doubtless, the most critical for ES validation. It is concerned with the validation of the knowledge itself, and not about its representation. To consider this problem we need forms to describe, manipulate and evaluate knowledge in a way abstracted for its particular representation. These foms are currently not available, although new cognitive architectures may represent an step forward in this direction.

- *Available Information*: ESs have to deal with incomplete, uncertain or inconsistent information, while conventional software works with sharp and precise data. The presence of uncertain data is represented with certainty degrees in the KB. This causes specific validation problems, since typical issues like inconsistency or redundancy are no longer boolean but weighted with certainty degrees. Uncertainty may have also some impact in ES sensitivity, in the sense that small changes in the input certainty may produce quite different outputs.

- *Programming Languages*: ESs are usually implemented using declarative programming languages, while conventional software is constructed using procedural programming languages. This difference determines to a great extent the kind of specific errors that can be found in a program. Considering rule-based languages, the type of errors that may occur are inconsistency, circularity, redundancy, etc., quite different from errors of procedural languages.

- *Specifications*: as it has said before, typical ES tasks are difficult to define precisely. An experience has been made, trying to formally specify an ES for medical diagnosis [Krause et al, 91], but it seems that it can only partially be achieved. In contrast, conventional software can be formally specified, aiming at the presence of specifications for all the different software stages. Ideally, this situation allows to validate by the verification (matching the stage product against its specification) of each software development stage.

- *Life-Cycle*: ESs do not have a consolidated life-cycle. Early systems were developed just by exploratory programming. The necessity to develop more complex ESs has generated several approaches including the life-cycle concept. These approaches share many points but also present some differences (see section 2.5). Conventional software has a quite consolidated life-cycle, with a very exhaustive and detailed set of activities for each development stage. A consolidated life-cycle provides a standard and safe way of software development, and is an important element for assuring the quality of the final product.

From these points we conclude that ESs present important differences with conventional software. These differences justify to revisit the definitions of validation concepts in order to adapt them to ESs. This does not mean that ES validation is considered essentially different from conventional software validation, but that the validation concepts so far used in exclusive for conventional software have to be enlarged to include the special characteristics of ESs as a new type of software. In addition, new validation methods and techniques are required to deal with the specific features and issues that ESs present. This is not exclusive with the adaptation of conventional software methods to ESs, which can take advantage of years of experience in software validation.

Finally, and from a practical point of view, [Bobrow et al, 86] mention that,

```
Knowledge engineering is more than software engineering,
... but not much more.
```

In other words, the process of rule coding has much in common with conventional software. General-purpose software engineering guidelines regarding task decomposition, modularization, programming standards and documentation are totally advisable for knowledge engineering.

1.3 Scope and Orientation

This thesis is devoted to validation of multi-level rule-based expert systems. Rule-based languages are widely used for ES development. By a multi-level rule-based expert system, we mean an ES where its KB is formed by a hierarchy of different types of rules, each level acting on the levels below it. We also include the presence of an uncertainty management system in this model (for a precise description of the ES model, see section 4.1). This new type of ES generates a wide variety of validation problems that have not been considered before. We present here a number of solutions for these problems, solutions that have been theoretically developed, implemented and tested in practice with a real ES application.

The validation methods we have developed assume the existence of an implemented ES. This is not a major drawback for their usage, since many ES applications early develop an initial prototype that is subsequently expanded in an incremental form. This approach, based on rapid prototyping, is the most frequently proposed in ES development methodologies (see section 2.5.4)[3].

In addition to validation methods, we have made a conceptual analysis of ES validation as a whole. We have considered a number of key questions in ES validation (what, when, how), drawing the complete picture of the problem. We have devoted special attention to terminological issues in ES validation. We propose a number of definitions for validation terms that include specific ES characteristics and keep the basic meanings of these terms in software engineering. In this sense, the proposed definitions fit pretty well in a general framework for software validation, where different kinds of software coexist with a common understanding of the kernel issues.

Our approach to validation, both regarding conceptual analysis or effective methods, is based on the following guidelines:

- *Practical Validation*: current ESs demand methods and techniques to achieve validation in practice. This is a actual requirement of ES technology that cannot be ignored. In addition, we consider that validation methods should be theoretically well-founded, but their effectiveness has to be shown with real applications.

[3] However, some authors consider that it has some disadvantages for validation [Rushby 88b].

Practice is the test that any validation method has to pass before to be considered really useful and effective.

- *Realistic ES Models*: many validation methods currently available assume simplistic models for the target ES. Namely, they consider rules in propositional logic, fired forward, exhaustive firing, monotonic, without any feature to deal with uncertainty or control. This simplistic model is quite far from the advanced capabilities that current shells offer, inside the rule-based paradigm. When validation methods based on these simplistic models are applied to current ESs, their results are neither accurate nor complete, since an important part of ES contents has not been taken into account. Then, the consideration of realistic ES models is a necessary condition to achieve a true validation for ESs.

- *Automatic Tools*: as we have said in section 1.1, the number of possible interactions among rules in real ES applications is very high. Some validation issues (for instance, inconsistency) demand exhaustive checking, that is to say, each combination of rules that can potentially contain an error should be checked. In this context, manual validation is useless. We have to develop automatic validation tools able to detect errors and even to suggest modifications. From that point, the responsibility relies on the ES developers, that have to devise the best form to correct the detected errors. Automatic validation tools provide the necessary support to the mechanical task of error detection, that, as it is shown in chapter 4, can require important amounts of time.

1.4 Overview

This thesis is composed of six chapters, an appendix and an annotated bibliography. Broadly speaking, the contents of this thesis can be divided in two parts. First part is composed of the three first chapters and it is devoted to the analysis of validation in ESs. Second part, composed of the three last chapters and the appendix is devoted to new validation methods for ES validation. We have developed two new validation methods for the ES model, verification using extended labels and knowledge base refinement. We have implemented them, producing two automatic tools: IN-DEPTH II, an incremental verifier, and IMPROVER, an automatic refinement system. We have used these tools to validate PNEUMON-IA [Verdaguer 89], an ES for pneumonia diagnosis based in the MILORD shell [Sierra 89]. In these validation methods we have followed the previously mentioned guidelines, since we have performed a practical validation on PNEUMON-IA, we have followed closely the ES model defined by the MILORD shell, and we have developed automatic tools to effectively support validation. In the following, we describe the contents of the rest of the chapters (2 to 6) of this thesis.

Chapter 2 is devoted to previous work in ES validation. It considers the following topics: verification, testing, knowledge base refinement, evaluation, validation in the ES life-cycle and case studies. On each topic early approaches have been described first, following an historical perspective with the aim to make explicit the topic evolution. When adequate, references to conventional software validation methods have been made.

Chapter 3 contains a conceptual analysis of ES validation as a whole. It addresses four important questions: What is Validation?, What Should be Validated?, How to Validate?, and When to Validate?. In the answer to the first question, a detailed analysis of validation terminology is made. The proposed definitions capture the peculiarities of ES validation without losing the basic meanings of the validation terms in software engineering.

Chapter 4 considers the verification of multi-level rule-based ESs. It first describes the ES model used for validation. It is composed of facts, rules, modules and metarules. Facts and rules compose what is known as domain knowledge, while modules and metarules are explicit representations of control knowledge. A hierarchy exists, acting control knowledge on domain knowledge. In this model, four classical verification issues are analyzed: inconsistency, redundancy, circularity and useless objects. As a result of the presence of control knowledge, these issues considered previously single problems unfold and generate new verification problems not considered before. The concepts of extended labels and extended environments, extensions of ATMS concepts, are successfully used to formulate the solutions for the mentioned issues. These solutions are implemented in the tool IN-DEPTH II, an incremental verifier. The idea underlying the incrementality of IN-DEPTH II is the following. If a verified KB is modified obtaining a new KB', to verify KB' we do not need to repeat the verification process on all its components. Verification has to be performed on (i) the additions to KB, that is to say, the elements of the set KB' - KB, and (ii) on those KB objects for which previous verification results in KB are not valid in KB'. IN-DEPTH II has been used to verify PNEUMON-IA, with encouraging results. A low number of actual error causes has been found, comparatively with PNEUMON-IA size. Some of the detected errors reveal subtle mistakes in knowledge organization, that were not easy to find. Actual occurrences of detected errors are described.

Chapter 5 addresses the role that knowledge base refinement systems can play in ES validation. A KB refinement system is formed by an automatic testing system coupled with a learning mechanism. ES execution is performed (or simulated) on a set of test cases with known solutions. When an error is detected by the automatic testing system, the learning mechanism is able to (i) localize its causes, (ii) generate the KB modifications that solve the error, and (iii) select the modification that best solves the error (that is to say, it has less undesired side-effects). Refinement systems had been used before in the context of machine learning, but when used for validation, refinement has to change some criteria. Selection is not made on the basis of the number of errors solved and caused, but on the importance that these errors have for the ES task. To implement the selected modifications, they have to be accepted for the human expert responsible for ES development. Regarding the ES model, there are two other contributions: refinement of control knowledge and solution of ordering mismatches, a new type of error. Medical diagnosis is the considered ES task. In this context, IMPROVER, an automatic refinement tool, has been developed and implemented. Its use has substantially enhanced the performance level of PNEUMON-IA. Before IMPROVER, PNEUMON-IA exhibited more false negatives (the most serious error type for medical diagnosis) and more false positives that any of the five independent human experts used in PNEUMON-IA validation. After IMPROVER, PNEUMON-IA exhibited a number of false negatives smaller than that of *any* of the human experts. The number of false positives has also decreased. Conversely, the number of ordering mismatches, the less important error, has increased. These results shows clearly the capacity of a refinement system for validation. Actual occurrences of proposed modifications are described in this chapter.

Chapter 6 summarizes the conclusions of this work in five points: a single validation definition is applicable to software and knowledge engineering, ES validation is feasible, automatic tools are needed for ES validation, verification and refinement are useful methods with complementary effects, and theory and practice are both required in ES validation. Regarding points for further research, they are the following: conceptual models of ES tasks, validation of conceptual models, verification of knowledge properties, knowledge acquisition for validation, validation by construction and test set selection.

The appendix contains the results of IMPROVER on PNEUMON-IA. For the 66 test cases considered, the correct solution is described, as well as the solution provided by PNEUMON-IA before and after the use of IMPROVER. The three types of errors considered are detailed for each case and each PNEUMON-IA version. Finally, an annotated bibliography in ES validation with approximately one hundred entries is given..

Chapter 2

Previous Work

ES validation is a very young field. First papers are dated on late 70s and early 80s, while most of the research has been made after year 85. Only in the last years, given the increasing industrial demand on ES validation, the main AI conferences (ECAI, AAAI, IJCAI) have included in their programs tutorials and workshops on the field. As a consequence of its youngness, ES validation is not a well-structured field. Its different aspects are unevenly developed and there is not a common and comprehensive approach to ES validation. Historically, research on this field has been guided by the necessity to obtain validation solutions for specific ESs. This have produced many ad-hoc approaches, where the applicability into a more general setting was not the main concern. Nevertheless, the practical experience on validating specific ESs has been of an immense value to improve and push forward the state-of-the-art in the field.

The goal of this chapter is to present a representative sample of what has been done on ES validation, with special attention to those works that have meant significant improvements in the field, but without attempting to an exhaustive enumeration. We have divided previous work on ES validation in the following topics: verification, testing, refinement, evaluation, validation in the ES life-cycle and case studies. Each topic is contained in a different section, where a number of relevant approaches are briefly explained. We have followed an historical perspective, describing early approaches first with the aim to make explicit the topic evolution. When adequate, we have made references to conventional software validation methods, although without aiming to a comparative analysis of conventional versus AI software. Each part is closed with a summary, where the main ideas on the topic are exposed. Finally, a global summary containing a general evaluation of current validation approaches closes the chapter.

2.1 Verification

Verification aims at checking a number of mandatory or advisable properties in rule bases as well as specific situations on the set of possible inferences. Verification properties arise from the rule-based knowledge representation, which requires some conditions to be fulfilled to assure the correct usage of the KB. Specific situations are originated by the specific problem domain, which imposes some constraints on the mapping from inputs to outputs in the ES. A large set of verification properties can be tested (for a detailed description see [Nazareth 89]), many of them are defined below when describing specific verifiers. Most of these properties have a logical basis, given the close relationship between production rules and logic. They were initially classified regarding consistency and completeness of the logical theory associated to the KB. Now, it seems more accurate to classify them with respect to the role they play in the ES. Broadly speaking, verification issues can be classified in four groups:

- Inconsistency: a KB is inconsistent if it can produce contradictory or conflicting outputs from a valid input. Inconsistency includes logical contradiction (deducing p and $\neg$p) as well as semantical incompatibility (deducing facts not logically related but incompatible, for instance male and pregnancy). Causes of inconsistency are modeled by special declarations called integrity constraints.

- Redundancy: a KB is redundant when some elements, which are used, can be removed without affecting its deductive power (the transitive closure). Different types of redundancy exist, depending on the rule semantics of the specific ES.

- Circularity: a KB is circular if there is a cycle in the rule dependency chain that can generate malfunctions (endless loops) of the inference engine.

- Useless objects: an object is useless if it is never used, no matter the ES input.

Some verifiers also detect syntactic errors in rule writing, illegal values assigned to attributes (type checking) and other minor issues. In this classification, inconsistency and redundancy are the most difficult problems. To be completely tested, they require algorithms of exponential complexity in the worst case, raising the danger of combinatorial explosion. They have been considered only partially by some verifiers.

Undoubtedly, verification is the most developed subfield of ES validation. Quite a few verifiers have been developed in the last ten years, giving adequate response to many verification issues. These systems take as input the KB plus some extra, application-dependent information (typically integrity constraints) required for the verification tasks. They produce as output a listing with the verification issues detected. Some of the most representative verifiers are described below, following an historical perspective.

2.1.1 Early Systems

The ONCOCIN RULE CHECKER [Suwa et al, 82] could be considered as the first verifier referenced in the literature. It detects the following issues on attribute-value[1] rule bases:

- Conflict: two rules r, r' are in conflict when $lhs(r)=lhs(r')$ and $rhs(r)$ is contradictory or in conflict with $rhs(r')$.
- Redundancy: two rules r, r' are redundant when $lhs(r)=lhs(r')$ and $rhs(r)=rhs(r')$.
- Subsumption: rule r subsumes rule r' when $lhs(r)\supset lhs(r')$ and $rhs(r)=rhs(r')$.
- Missing rules: an input situation is not covered in the KB.

Rules are grouped by their concluding attribute, forming a table for each group. On each table, conflict, redundancy and subsumption are tested by exhaustive comparison of its rules. Missing rules are computed under the hypothesis that all possible combinations of values for the attributes present at the table should exist. The ESC system [Cragun & Steudel 87] follows a very similar approach, using decision tables to represent group of rules which causes some efficiency improvements.

The CHECK system [Nguyen et al, 85] [Nguyen et al, 87] tests goal-driven and data-driven rules with variables belonging to the LES shell. In addition to conflicts, redundancy and subsumption, issues defined above, it detects the following:

- Unnecessary if-conditions: two rules with the same right-hand side differ in their left-hand sides in one condition, affirmed in one rule and negated in the other.

[1]Attribute-value rules are equivalent to rules in propositional logic, in which literals have been substituted by boolean expressions relating attributes and their corresponding values.

- Circular rules: a set of rules forms a cycle.
- Unreferenced attribute-values: legal values of an attribute are not covered in the KB.
- Illegal attribute-values: an illegal value for an attribute is referenced.
- Unreachable conclusions: the right-hand side of a rule is unreachable if it matches neither a goal nor another condition in the left-hand side of another rule.
- Dead-end if-conditions and goals: an if-condition or goal is a dead-end if it is not askable and it does not match the right-hand side of another rule.

CHECK analyzes these issues in the following way. First, every clause (condition in the left-hand side or assertion in the right-hand side of a rule) is compared against every other clause of every other rule and every goal. Potential results are `same`, `different`, `conflict`, `subset` and `superset`. Results are stored in a table of clause relationships. Second, left and right-hand sides of every rule are compared against every left and right-hand sides of every other rule using the table of clause relationships. Results are stored in a table of part relationships. Using this table, conflict, redundancy, subsumption and unnecessary if-conditions are easily tested. Using also the table of clause relationships, unreachable conclusions, dead-end if-conditions and goals are computed. Circular rules are detected representing rule dependencies by a directed graph an checking cycles in it.

These early systems perform only a partial analysis of inconsistency (conflict) and redundancy because they test these issues locally, comparing statically pairs of rules and ignoring rule chaining. Inconsistencies and redundancies requiring more than two rules to occur cannot be detected. Figure 2-1 contains two set of rules which are trivially inconsistent (a) and redundant (b) that cannot be detected as erroneous by these verifiers. These examples show clearly that inconsistency and redundancy are global properties of sets of rules, and therefore, rule sets should be globally analyzed to achieve a complete analysis.

$$\begin{array}{llll@{\qquad\qquad}llll}
\text{(a)} & p \wedge q & \rightarrow & r & \text{(b)} & p \wedge q & \rightarrow & r \\
 & r & \rightarrow & u & & r & \rightarrow & u \\
 & p \wedge q & \rightarrow & \neg u & & p \wedge q & \rightarrow & u
\end{array}$$

Figure 2-1. Inconsistent (a) and redundant (b) sets of rules.

2.1.2 Inconsistency Checkers

The KB-REDUCER system [Ginsberg 88b] detects all inconsistencies and redundancies in forward-chaining, propositional rule bases. Ginsberg points out the different meanings of consistency in logic and in rule bases. A set of logic formulas is consistent if there exists an assignment of truth values to literals such that all formulas are true. A KB is consistent if no contradictions can be obtained from a valid input. The set of rules (a) in figure 2-1 is consistent in the logic interpretation (for example, when $p=r=false$) but it is not as a KB (when $p=q=true$, both u and $\neg u$ are concluded as true). An input is valid if it represents a real situation in the problem domain. To model legal input values in the problem domain, a set of integrity constraints is defined, including logical or numerical constraints (always holding), single valuation of attributes (assumed by default) and semantical constraints (restrictions on the values of certain attributes that are purely domain-dependent). Then, an input is valid if no integrity constraint is violated. A set of rules R is irredundant when no rule follows from other members of R, and every rule in R is satisfiable by some valid input. Inconsistency and redundancy are tested by computing labels and environments. The *label* for a deducible fact h, $L(h)$, is the minimal disjunctive normal form asserting h, where each disjunction consists solely of external facts and is denominated an *environment* for h, $E(h)$(terminology borrowed from ATMS [deKleer 86]). Thus, $L(h)$ represents all the minimal sets of external facts that cause h to be deduced. Given an integrity constraint (h_1, h_2), an inconsistency occurs if there exist $E_i(h_1) \in L(h_1)$, $E_j(h_2) \in L(h_2)$ such that one of the following conditions hold: (i) $E_i(h_1) \supset E_j(h_2)$, (ii) $E_j(h_2) \supset E_i(h_1)$, (iii) $E_i(h_1) \cup E_j(h_2)$ is valid (case (iii) includes (i) and (ii)). Redundancy is detected by computing, for every deducible fact h, the contribution of each rule r concluding h to $L(h)$. When the contribution of r is null or it implies the contribution of other rules, r is redundant and can be removed.

The COVADIS system [Rousset 88] follows a close (but independent) approach. It detects inconsistencies in forward-chaining, propositional rule bases. Inconsistency is defined in terms of meaningful (valid in KB-REDUCER) inputs using integrity constraints, which are defined as rules concluding the special fact *false*. COVADIS generates the specification of all inputs that can conclude *false*. This is made by an extended forward-chaining process, where contextual facts are computed. A contextual fact has the form $(ATTR_i = VAL_j)$ $[CONTX]$, meaning that the attribute $ATTR_i$ will take the value VAL_j only when the boolean expression in $CONTX$ is true. Contextual facts are propagated by rule firing, generating new ones. When an integrity constraint is fired, the contextual fact for its conclusion *false* is shown to the expert, who will judge whether it is meaningful or not. If it is meaningful, the KB is inconsistent and it should be corrected; if not, some integrity constraints are missing and they should be added.

The PENIC system [Meseguer 90] checks KBs formed by propositional rules for inconsistency. The KB is translated into a Petri net, representing each fact by a place and each rule by a transition. Facts considered true are places with one or more tokens. A token distribution is denominated a marking. Transition firing generates new markings. An inconsistency occurs when all the places representing facts involved in an integrity constraint are reachable from an initial marking representing a real input. Using the algebraic foundations of Petri nets, it is shown that this reachability problem is equivalent to solve a linear equation system in {0,1}. In this way, a problem traditionally solved by symbolic computation is transformed in an algebraic problem. Petri nets have also been used for verification in [Pipard 88] [Agarwal & Tanniru 91].

The GCE system [Beauvieux & Dague 90] performs incremental consistency checking in attribute-value rule bases. When a KB already verified is modified, GCE checks consistency without doing a complete proof again, just focusing on what has been changed. This is made by building maximal consistent set of facts, denominated base models. Logical or semantical inconsistencies are modeled by integrity constraints. Legal modifications are adding or removing a rule or an integrity constraint.

2.1.3 Verifiers with Extended Functionalities

The EVA system [Chang et al, 90] provides a wide range of verification facilities. EVA functions were initially designed to work on ART knowledge bases. The long-term goal of EVA is to build an integrated set of generic tools to verify any knowledge base written in any shell, such as ART, CLIPS, OPS5, KEE and LES. EVA is composed of the following modules: structure checker (extended), logic checker (extended), semantics checker, omission checker, rule refiner, control checker, behavior verifier, test case generator, uncertainty checker, rule satisfiability checker and model-based verifier. These modules cover the usual verification issues. EVA is written in PROLOG and represents knowledge bases in an internal format. Verification predicates are written as predicates on objects which are elements of the target knowledge representation language. For this reason, they can be considered as *metapredicates*, forming a metalanguage specific for verification. This approach offers a set of high order constructs to represent knowledge about the verification task, which can be of great interest to build verification tools. A crucial aspect of this approach is the genericity of the internal representation of knowledge bases. Some work has been done translating OPS5 into the EVA environment in [Childress & Valtorta, 91].

The COVER system [Preece & Shinghal 91] checks propositional rule bases for anomalies that are symptoms of possible errors. Deficiency, ambivalence, redundancy and circularity are the considered anomalies. A KB is deficient if there exists a permissible input for which not final hypothesis is given, because some knowledge is missing. Considered deficiencies are missing rules, unusued literals and missing values. Ambivalence occurs when the same input can generate different outputs. A rule is redundant when it can be removed without any effect in the hypothesis that can be inferred by the KB. Subsumption between two rules is a particular case of redundancy, as well as rules with unusable consequent (unreachable conclusion). All these issues are analyzed by COVER in three phases: integrity checker, rule checker and environment checker. The integrity checker solves the "easy problems": unusued literals, unusable consequents, circularity and missing values. The rule checker solves simplified versions of inconsistency and redundancy, restricted to rule pairs. The environment checker solves the general version of inconsistency, redundancy and missing rules.

2.1.4 Other Approaches

The SACCO system [Ayel 88] checks rule bases for incoherence (inconsistency) but it does not aim to be exhaustive. Coherence (integrity) constraints are added to the KB. Some of them are considered "pertinent" using specific knowledge. They are tested using labels that are selectively built using heuristic filters to prevent combinatorial explosion.

The SVEPOA system [Prakash et al, 91] is a specific verifier for OPS5 applications. It detects situations that may generate error: conflict and likely-to-activate relations, as well as dead-end and impossible rules. These situations are computed translating rules into a linear system of equalities and inequalities, and testing it for a feasible integer solution.

The AbsPS system [Evertsz 91] performs some verification tests based on the abstract interpretation of the KB. Abstract interpretation has the same meaning as symbolic execution in conventional software. AbsPS takes as input the KB and an specification of its input, and produces the specification of its output. In this process, AbsPS takes into account the procedural semantics of the rules, specifically the different criteria for conflict resolution.

Finally, [Herod & Bahill 91] have developed a verification tool to test the correctness of ESs in question sequencing (what they denominate the pregnant man problem). This tool requires a question matrix filled by the domain expert where question dependencies or incompatibilities are represented. Using this matrix the KB is updated using an heuristic procedure, to assure that no conflicting questions will occur (although this can affect the ES performance).

2.1.5 Verification Summary

This enumeration shows clearly the evolution of verifiers. Three generations of verifiers can be identified, corresponding respectively to early verifiers, inconsistency checkers and verifiers with extended functionalities. Evolution that can be analyzed under three points of view:

- Partial vs. complete checking: first generation verifiers check only partially the "hard problems", inconsistency and redundancy. These issues are completely tested in the next generation, which is focused on these problems. Third generation systems incorporate, in addition to complete checking, an extended set of functions to check different aspects of the encoded knowledge.

- Simple vs complex ES models: most of these verifiers work on a simple ES model, namely, rules underlaying propositional logic, without uncertainty, without control and assuming monotonic deduction. Some verifiers deal with rules with variables, only EVA has a facility to represent uncertainty, AbsPS includes the role of implicit control and SVEPOA deals with non-monotonicity.

- Number and type of verification functions: first generation verifiers consider a wide set of issues, set that is restricted in the next generation, more interested in complete inconsistency checking. This set becomes wider again in the third generation. Finally, the latest verifiers (svepoa and absps) offer a more restricted set of verification capabilities, however, they work on a more complex ES model than their predecessors. Also it is noticeable that new, non-logical properties as question ordering begin to be tested.

Future trends in verifiers can be centered mainly in two points. First, verifiers should consider more realistic ES models. Aspects such as uncertainty management and control (implicit conflict set resolution, or explicit metarules) currently available inside the rule-based paradigm, should be taken into account to achieve accurate and precise verifications. In addition, some of these aspects now ignored can be of great help to fight against the risk of combinatorial explosion, always present when checking inconsistency or redundancy. Second, verifiers should test new properties, more informative for the knowledge engineer or human expert. So far, most of the verification functionalities check some kind of correctness with respect to the rule-based representation but they do not consider specific verification functions for the intended ES task. This may require the addition of new knowledge to the ES to be exclusively used at the verification phase.

2.2 Testing

Testing examines the behavior of a program by its, actual or simulated, execution on sample data sets [Adrion et al, 82]. The main goal of testing is to check program correctness. To guarantee a complete correctness, testing has to be exhaustive, that is to say, every potential input should be tested. This is obviously unfeasible for real applications, so testing only analyzes the program behavior on a finite set of test data (*test set*). The selection of the test set is an essential point for the testing process. This set should be large enough to be a representative sample of the program domain and yet small enough to allow the testing process to be executed on each element of the test set consuming a reasonable amount of resources. A formal treatment of test set selection is given in [Goodenough & Gerhart 75], setting the following fundamental theorem of testing:

```
If there exists a consistent, reliable, valid and complete
criterion for test set selection for a program P and if a test
set satisfying the criterion is such that all test instances
succeed, then the program P is correct. 2
```

However, no algorithm can find consistent, reliable, valid and complete test criteria [Howden 76]. Therefore, other simpler criteria can be used for test set selection, such as random testing, functional testing, structural testing, heuristics, etc. [Howden 78]. In spite of the fact that, using these criteria testing can never guarantee correctness, testing has shown to be very effective in practice, and therefore it is unavoidable in program validation.

The testing fundamentals exposed so far have been developed considering conventional software, but they are equally applicable to ESs. In addition to the problem of test set

[2] For the precise meanings of consistent, reliable, valid and complete, see [Goodenough & Gerhart 75].

selection, testing in ESs raises some new issues, some of them are outlined in the following:

- The correctness of ES final output should be tested, as well as the correctness of ES reasoning. This second aspect is very important to gain confidence in the ES, specially from the final user. In this sense, definite experiences have been obtained in medical ESs (see MYCIN description in section 2.6.1).

- A correct pattern for ES behavior, also known as gold standard, has to be established. This is not an easy task because either (i) many correct solutions can exist, or (ii) if only one solution exists, it can be very difficult to determine accurately. Examples of the first case are ESs in configuration tasks, while the second case appears in ESs on medical diagnosis. In the selection of the gold standard for a problem there are two options:

  1. the objective correct answer to the problem, or
  2. the answer given by a group of experts to the problem, using the same information available to the ES.

  The first option is often unfeasible because the objective correct answer can be unknown and only can be obtained by very expensive or unacceptable methods (for example, autopsy in medicine). The second option is generally chosen, on the basis that the ES goal is to reproduce the skill of human experts so the ES should be compared against them. For medical ESs there is a wide consensus that ES output should be compared against human expert behavior [Chandrasekaran 83] [Gaschnig et al, 83]; otherwise, the ES would be tested against some kind of superhuman capacity, what can be misleading. Given that human experts may disagree, procedures to set expert consensus are needed. Whatever is the choice of gold standard, it should be agreed before ES construction and maintained throughout the ES development [Rushby 88b].

- Acceptable performance standards: an ES never achieves a performance level of 100% success, in the same way that human experts are not infallible. The performance level from which the system can be considered as expert should be stated. To assess this level, measures of the performance of human experts and practitioners in the ES task can be used. These standards of performance have to be realistic. The case studies of MYCIN and R1 contain interesting experiences on this topic.

Some other problems can also exist such as unavailability of experts to support validation, short number of test cases, bias in the test set or in expert judgements, etc. In addition to these issues and to the kernel problem of test set selection, usually there is a total lack on ES requirements. This causes that, in many occasions, testing cannot be made on a sound basis and becomes a matter of guesswork [Green & Keyes 87].

No uniform approach exists on ES testing. A variety of techniques have been developed, in many occasions adapted to specific problem domains. In the following, some ES testing techniques are briefly exposed.

2.2.1 Test Set Selection

From conventional software the most effective testing methods are random testing (test set is randomly selected), functional testing (test set is selected on the basis of the expected program function, described in the requirements, black box approach) and path testing (test set is selected to exercise a maximum of different paths in the program, white box approach) in this order and with complemented effects [Rushby 88b]. This testing schema can be adapted to ES testing. Random testing, where cases are selected using stratified sampling is recommended in [O'Keefe et al 87]. Used test values should be in accordance to the actual (or expected) distribution of these values in the problem-domain. However, random testing can generate meaningless combinations of values, of little interest for the testing process. Functional testing is suggested by [Green & Keyes 87]. It is performed by testing each single requirement, singular point and boundary condition. However, the usual lack of detailed requirements for ESs can difficult the practical application of this approach. To adapt path testing to ESs, the idea of path in conventional software is substituted by a rule sequence in ESs. Systematic path testing can be performed when the conditions for firing a rule sequence can be computed for a significant number of possible rule sequences. For this purpose a test-case generator (see below) seems very adequate.

2.2.2 Human Intervention

Human intervention is always required in testing. From [Gaschnig et al, 83] and [O'Keefe et al, 87], three different aspects of human intervention are described: face validation, Turing tests and field test. Face validation is a preliminary testing process involving developers, experts and potential end-users, who compare subjectively ES performance against human expert performance on a set of test cases. A Turing test compares the ES against human experts in the following way. A set of cases is provided to both the ES and a group of human experts. ES outputs and human experts' judgements are evaluated by another group of human experts, without knowing the identity of each performer. For a detailed description of Turing tests on medical diagnosis ESs see [Chandrasekaran 83]. Turing tests guarantee blindness, preventing from anti- or pro- computer bias in human evaluators. However, Turing tests can be very expensive to carry out. A field test consists on placing the ES at the work place under supervision for error detection. Users perform the testing process: when they detect malfunctions or incorrect results, they report them to the ES developers. An acceptable level of performance is achieved when users cease to report problems. This kind of testing is only applicable to non-critical applications, when a set of users is willing to accept the burden of ES testing.

Dealing with human judgements is not easy. Even elite experts exhibit the following problems [Green & Keyes 87]: prejudice, parochialism and inconsistence. Prejudice exists when some experts refuse to consider the possibility that the ES output is as good as theirs. Parochialism appears when experts in different geographical regions apply different criteria and different bodies of knowledge. Inconsistence occurs when some experts rated as unacceptable answers that are substantially identical to their own. To overcome these issues, blind testing and expert independence from ES developers is required.

2.2.3 Statistical Approaches

The following statistical techniques have been proposed in [O'Keefe et al, 87] to evaluate the goodness of testing results: paired t-tests, Hotelling's one-sample T^2 tests and simultaneous intervals. Paired t-tests can be used to evaluate the difference between ES output and human performance, represented as $D_i = X_i - Y_i$, where X_i is the ES output, Y_i is the human performance or known solutions. Given n cases, the ES response is acceptable if zero lies in the following confidence interval,

$$\overline{d} \pm t_{n-1,\alpha/2} \; S_d / \sqrt{n}$$

where $\overline{d}$ is the mean difference, S_d the standard deviation and $t_{n-1,\alpha/2}$ is the value from the t distribution with n degrees of freedom. When the ES output is composed of multiple elements, Hotelling's one-sample T^2 test should be used instead of multiple paired t-tests. If the set of possible elements for ES output has k elements, each test case can generate k differences between the ES output and the human performance. Repeating this process for every test case, k vectors of differences, one for each element, are obtained. Then, the one-sample T^2 test can be used to determine if the means of the difference vectors are significantly different from zero. Also simultaneous confidence intervals for differences of paired responses can be constructed when the ES output is composed of multiple elements.

The problem of consistency between multiple experts is also considered in [O'Leary & O'Keefe 91]. To measure consistency they suggest to use the interclass correlation coefficient when expert responses are in a continuous scale or the kappa statistic in the discrete case. Also another statistic to compare joint expert agreement with ES output is mentioned.

2.2.4 Test-case Generators

Given that test cases are difficult to find and select, some approaches have tried to generate test cases lying on the structure of the program to test. While this approach has not been very successful in conventional software [Howden 78], it seems more promising in ESs where it is easier to compute the conditions for a feasible path (sequence of rules to fire).

The SYCOJECT system [Vignollet & Ayel 90] is a test case generator for rule bases. SYCOJECT uses a conceptual model for test case building, which contains testing knowledge for an specific ES. Test cases are divided in equivalence classes, where each class contains all cases that will cause the same output. An equivalence class is characterized by the set of deduced facts plus the set of fired rules. Only cases representative of different classes will be generated. These cases can make use of the "inside/outside singular values" for an attribute, which are the legal/illegal values for that attribute considered semantically relevant for the expert. Test cases are checked for their structural and semantical validity, using information on the allowed values for the considered attributes.

2.2.5 Heuristic Testing

Another approach is heuristic testing [Miller 90] for attribute-value rule bases. The basic idea underlying this approach consists in classifying all possible ES failures into categories based on their consequences. Failures are ordered and tested by decreasing risk for system competency. This approach is denominated heuristic testing since it is a reasonable strategy to eliminate the worst problems first. Faults are classified by decreasing importance in the following classes: basic safety, system integrity, essential function, robustness-failure, secondary function, incorrect input/output, user-interface, error -metric, resource consumption and other. Faults are solved when they are detected, in a fix-as-you-go basis. The heuristic testing procedure is composed of the following steps: (i) selection of the fault classes to be considered, (ii) selection of test cases for each considered fault class, (iii) preparation of the testing plan, and (iv) execution and evaluation of the selected tests. Testing can consider only one rule or a chain of rules. Test cases are generated using the generic testing method, which computes the adequate values for the attributes of a rule (or rule chain) to be satisfied with three levels of reliability, regarding the vicinity of the constant appearing as value for each attribute. Test cases are specifically generated for each fault class, requiring human intervention to discriminate among the fault classes and to evaluate the results of test outputs. When an error is fixed performing some change in the KB, regression testing (repetition of the testing process on previously considered cases) is required to check that this modification does not cause previously considered errors.

2.2.6 Testing Summary

From the previous description, it is apparent that ES testing is a quite complex topic on which there are many relevant factors. A large number of different aspects have been studied and several testing techniques have been developed, although a uniform and integrated approach of ES testing including all different aspects is currently missing.

Essentially, ES testing presents two problems: test set selection and ES performance evaluation. The test set should be reasonably small but representative of the problem domain. Sometimes, test case representativity is not easily assessed since typical ES applications consider ill-structured domains with fuzzy boundaries. Independent experts can assess the representativeness of the test set. ES performance evaluation depends largely on the existence of an absolute and correct standard of performance (also known as "gold standard") commonly accepted by the experts in the problem-domain, for all the cases in the test set. When no external gold standard exists, ES performance should be compared against human performance using experts' judgements. However, experts' judgements can be biased, inconsistent or even erroneous. To overcome these issues, statistical techniques can evaluate the degree of agreement among human experts themselves, which can be compared with the degree of agreement of the ES against the experts' judgements. Blind testing and independent expert assistance seem to be necessary to guarantee as much as possible an objective testing.

2.3 Knowledge Base Refinement[3]

Knowledge base refinement considers the improvement of performance of an ES by learning from a set of known cases $\boldsymbol{C}$ that fall into the problem domain. A refinement problem for an ES exists when, presenting a reasonable performance degree, a number of cases in the problem domain are treated incorrectly. Refinement is focused on the declarative part, the knowledge base, assuming that the procedural part of the ES is correct. It is assumed that only minor changes are required to lead the knowledge base to a satisfactory state, so the learning process will respect as much as possible its original structure and vocabulary. The knowledge base is usually formed by rules in propositional logic (Horn clauses), although some systems include rules with certainty factors. Knowledge base facts are divided in two disjoint sets, O the set of observables and H the set of hypothesis (also known as the sets of external and deducible facts), representing respectively the sets of inputs and deductions of the ES. Taking as input a subset of valued observables, the ES is executed obtaining as result the subset of hypothesis that it assigns for this particular input. The closed world assumption usually holds: if an hypothesis is not deduced for a particular case, its negation is assumed. A case $C_i \in \boldsymbol{C}$ is composed of two sets $< O_i , H_i^* >$, such that $O \supset O_i$ and $H \supset H_i^*$ [4]. The sets O_i and H_i^* respectively represent the subset of observables and the subset of hypothesis that effectively hold for the case C_i.

The refinement process consists of the following four phases:

- *Identification*: for each case C_i, the set of hypothesis H_i that the ES assigns to C_i is computed and compared with the correct set of hypothesis H_i^*. From this comparison, hypothesis $h \in H$ are classified in:

  - True positives: $h \in H_i \cap H_i^*$.
  - True negatives: $h \notin H_i \cup H_i^*$.
  - False positives: $h \in H_i$, $h \notin H_i^*$.

[3]The terms "knowledge base refinement", "theory refinement" and "theory revision" are used with very close meanings in the literature. The first one has been selected because it is the most specific in the context of expert systems. In the description of specific systems we have followed each author's terminology.

[4]H_i^* usually contains a small subset of the total set of hypothesis H: the final goals of the ES. However, this discussion can be applied to any intermediate hypothesis without losing generality.

- False negatives: $h \notin H_i$, $h \in H_i^*$.

The existence of false positives means that the knowledge base is too general and it should be specialized. Conversely, the existence of false negatives means that the knowledge base is too specific and it should be generalized. In both cases, the knowledge base should be refined to achieve all the correct hypothesis, and only those, for each C_i.

- *Localization*: once an error has been identified, the part of the theory responsible for it should be localized. This task requires the analysis of the knowledge base, normally using explanation-based learning techniques. The way the knowledge base is used by the ES should be taken into account.

- *Generation*: in this phase potential refinements to solve an error are generated. The generation process is driven by the error nature (specialization/generalization) to select refinement operators. Typical specialization/generalization operators are adding conditions or deleting rules/removing conditions or creating new rules. Also changes in the certainty values associated to the rules are considered. Refinement operators will be applied on the knowledge base part localized as responsible for the error. When new conditions or rules are added, they are usually computed by inductive learning methods over the set of cases.

- *Selection*: from the set of generated refinements one is selected and applied on the knowledge base. One principle drives the selection process: refinements should not generate new errors in the ES performance. Following this principle, the selection process has to know how a refinement can affect the ES functionality, in order to maintain its correct behavior on those cases in which no error has been detected. This can require either simulation or real execution of the modified knowledge base over the set of cases. The set of generated refinements is ordered following rankings or heuristics. Refinements are tested in this order, and when a refinement leads the modified ES to a satisfactory performance it is selected and added to the knowledge base as a consolidated change.

The identification phase is performed at the beginning of the process, to determine the cases that are correctly treated by the ES and those cases that show errors. Next phases, localization, generation and selection, are usually performed sequentially once for each identified error. To evaluate the power of the refinement process, the set of available cases is divided in two: the training set and the test set. The initial *KB* is refined using the training set and generating *KB'*. These knowledge bases, *KB* and *KB'*, are executed over the test set and their results are compared. It is expected that *KB'* will show a better performance than *KB*, assuming the representativity of the training set over the problem domain and the homogeneity in both training and test sets.

In the following some refinement systems are described. They have been selected for their capabilities to refine knowledge bases from both generalization and specialization issues, with empirical results of their performance. Anyway, this enumeration is not exhaustive.

2.3.1 Empirical approaches

The SEEK system [Politakis 85] suggests possible refinements for tabular knowledge bases developed with the shell EXPERT. A table consists of (i) a description of major and minor findings for a hypothesis, and (ii) a set of rules concluding this hypothesis. A rule contains a conjunction of conditions (involving numbers of major and minor findings) and a conclusion confidence degree (no confidence propagation exists). Rules with higher confidence degrees have priority. Performance is evaluated by matching the ES top ranked conclusion with the expert's conclusion in each case (no multiple diagnosis is considered). When a discrepancy exists, all fired rules with wrong conclusions and higher confidence degree than the expert's conclusion are marked for specialization, while the unsatisfied rule closest to the expert's conclusion is marked for generalization. Specialization operators weaken or remove rule conditions while generalization operators strengthen or add rule conditions. Increasing rule confidence is seen as a generalization because a conclusion overrides previous conclusions with lower confidence. Conversely, decreasing rule confidence is seen as an specialization. Changes are limited to existing rules (adding or removing rules is not allowed). Generalization/specialization statistics are gathered for marked rules. Heuristics use these statistics and specific knowledge about conditions to generate refinements (denominated "experiments"), which are proposed to the user in a ranked list. The user can interact with SEEK and evaluate the exact effect of each experiment, and eventually accept one.

Based on SEEK, the SEEK2 system [Ginsberg 88a] enhances the capabilities of its predecessor in three aspects. First, it extends the SEEK tabular knowledge representation, since SEEK2 works on any knowledge base developed with the shell EXPERT. Second, it has an "automatic pilot capability", able to perform all the refinement phases without human interaction and to generate a refined knowledge base (that is, SEEK2 automatizes the selection phase). This capability is based on a hill-climbing strategy. Suggested experiments are ordered by their potential net gain. Every proposed experiment for every final diagnostic is attempted in the knowledge base, assuming that heuristics have generated a small fraction of all admissible experiments. Out of all these experiments SEEK2 accepts only the one that provides the highest net gain in ES performance for all the final diagnostics. And third, a metalanguage for knowledge base refinement, RM, has been developed in which knowledge about the refinement process (metaknowledge) can be naturally expressed, separated from implementation details and available to the refinement user. SEEK2 has been rewritten in RM.

The KRUST system [Craw & Sleeman, 90][Craw & Sleeman, 91] refines propositional, backward chaining rule bases taking into account rule priority. Rules have a priority factor used to solve the conflict set. When a case is misclassified, rules are divided in error-causing, potential error-causing and target. The error-causing rule is the fired rule generating the erroneous conclusion. Potential error-causing rules are those rules that have not been fired because they had lower priority than the error-causing rule, but they are satisfied and their conclusion is erroneous. Target rules are those with the correct conclusion. Target rules are further subclassified considering why they have not been fired, because unsatisfied or low priority. Refinement operators are adding or removing conditions to rules, changing rule priorities, changing conclusions and adding new rules. All possible minimal refinements are generated. These refinements are filtered using statistical, heuristic and consistency criteria. The remaining refinements are actually implemented and evaluated on a set of test cases. A subset of cases denominated "chestnuts" are used to remove those refined knowledge bases that do not provide the correct answer. Remaining knowledge bases are ranked by discrepancies in the set of cases.

2.3.2 Operationalization

Ginsberg in [Ginsberg 88c] describes a revision method for propositional logic theories restricted to rules (Horn clauses) with three stages: theory reduction, theory revision and retranslation. The first stage, theory reduction, translates the original theory into the reduced theory, a more amenable form for revision. The reduced theory is formed by the set of labels for all the hypothesis in the theory. Theory reduction is computed by KB-REDUCER [Ginsberg 88b] (see section 2.1.2). The second step, theory revision, is the refinement process of the reduced theory. A case C_i poses a generalization problem for hypothesis h when $h \in H_i^*$ but there is no environment in $L(h)$ satisfied by (contained in) O_i. Conversely, C_i poses a specialization problem for h when $h \notin H_i^*$ but there are environments in $L(h)$ satisfying O_i. Refinement generation and selection is made in five steps: massive label generalization and specialization, focussed label generalization and specialization, and correction of remaining errors. The first two steps try to solve problems spread out in the theory, while the three last steps consider specific problems. Refinement operators are add/remove observables to/from environments and, in the last case, add/remove environments to/from labels (minor changes are always preferred). The evaluation of refinement effects in the reduced theory is just testing environment (set) inclusion. Theory revision is performed by the RTLS system [Ginsberg 88c]. The third stage, theory retranslation, considers the translation of the revised reduced theory into rule terms. The goal is to obtain a new theory that will perform over C at least as good as the revised reduced theory, although its reduction could not be exactly identical to the revised reduced theory (relaxed retranslation). This is made by top-down retranslation, starting from the final hypothesis and helped by the rule structure of the original theory [Ginsberg 90].

2.3.3 One-concept theories

The EITHER system [Ourston & Mooney, 90] refines propositional Horn clause theories handling multiple faults of both generalization/specialization issues. The theory describes one concept, so cases are just positive or negative examples. Identification is straightforward: a problem exists when a positive example fails to be proved or a negative example is proved. Refinement is performed separately for both types of issues, since the generalization algorithm will not cause new specialization problems and vice versa. Refinement operators are remove conditions or add rules for generalization, and remove rules or add conditions for specialization. To generalize the theory, the minimum set of assumptions that will classify correctly all the unproved positive examples are computed using partial proof trees. Each rule containing one of these assumptions is considered in turn. Assumptions in the current rule are removed if no negative example becomes provable. Otherwise, one or more rules are learned by induction over the set of positive and negative examples misclassified for the current rule. To specialize the theory, the minimum set of retractions that will not prove all the proved negative examples are computed. Each rule containing one of these retractions is considered in turn. The current rule is removed if all positive examples remain probable. Otherwise, conditions are added to the current rule, learned by induction over the set of positive and negative examples misclassified for this rule.

Similar to EITHER is the DUCTOR system [Cain 91]. It revises one-concept propositional theories using a set of positive and negative examples. DUCTOR generates refinements for each generalization/specialization problem in turn. For each unproved positive example, an explanation is constructed assuming literals. If a non-operational literal has been assumed, a rule is learned for this literal. Otherwise, DUCTOR removes the assumed literals from the rules in the explanation if no negative example becomes provable. For each proved negative example, DUCTOR tries to remove a rule from its explanation. If this rule is required for positive examples, DUCTOR adds conditions to it if no positive example becomes unprovable. As inductive learning method, DUCTOR computes the most specific cover of positive examples that do not cover any negative example (EITHER computes the most general one). The revision process loops until no modification is performed on the knowledge base.

2.3.4 Knowledge Base Refinement Summary

Significant similarities can be found among the previous knowledge base refinement systems. Regarding the knowledge representation on which they work, all of them consider propositional rule bases. Two systems, SEEK and SEEK2, include uncertainty in rules, and only one, KRUST, considers the control role in the refinement process. All of them assume that the solution is to select 1 class from one set of predetermined classes. These points are summarized in table 2-1.

Regarding the refinement process, all the described systems share the basic sequence composed of identification, localization, generation and selection (with the extra stages of theory reduction and retranslation for RTLS). In the identification phase, all of them execute the knowledge base on the set of known cases, except RTLS, which can anticipate the knowledge base results by label analysis. All the systems perform this phase on all the available cases, except KRUST, that does it only for one case. In consequence, KRUST will introduce refinements to solve just this case. This simplifies the localization and generation phases but requires an exhaustive testing in the selection phase. More diversity exists in the localization phase, where statistics, rule analysis, environment tests and explanation analysis coexist. In the generation phase all the systems generate many refinements trying to solve all the identified problems, but KRUST generates all the refinements to solve one problem. Of these refinements, only 1 is selected in SEEK and SEEK2, a few are selected in

KRUST, and many can be selected in RTLS, EITHER and DUCTOR. In table 2-2 all these aspects are summarized, including the basic cycle for each system denoted by the initials of the four considered phases and where a parenthesis indicates that a loop exists in the contained sequence.

	Rules	Uncertainty	Control	#Classes in sol	# Classes
SEEK	yes (tabular)	yes	no	1	N
SEEK2	yes	yes	no	1	N
KRUST	yes	no	yes	1	N
RTLS	yes	no	no	1	N
EITHER	yes	no	no	1	2
DUCTOR	yes	no	no	1	2

Table 2-1. Comparison of refinement systems regarding the assumed knowledge representation.

	Identification (method/#cases)	**Localization** (method)	**Generation** #refinements	**Selection** (#refinements/method)	**Cycle**
SEEK	execution/all	statistics	many	1/execution	(I-L-G-S)
SEEK2	execution/all	statistics	many	1/execution	(I-L-G-S)
KRUST	execution/one	rule analysis	all	a few/filters, execution	I-L-G-S
RTLS	label analysis/all	environment tests	many	many/env test	I-L-(G-S)
EITHER	execution/all	explanation analysis	many	many/execution	I-L-(G-S)
DUCTOR	execution/all	explanation analysis	many	many/execution	(I-(L-G-S))

Table 2-2. Comparison of refinement systems regarding their phases and cycle.

2.4 Evaluation

Evaluation considers the assessment of ES characteristics. Many ES characteristics can be considered, from those that are of general applicability to very specific or application-dependent ones. General characteristics are correctness, efficiency, complexity, utility and quality. Except for ES correctness, assessed using testing methods (see section 2.2), little attention has been devoted to this field [Liebowitz 86]. However, ES characteristics can play a significant role in ES acceptability [Buchanan & Shortliffe 84].

Four evaluation principles are provided in [Gaschnig et al, 83]. They are the following:

- Complex objects or processes cannot be evaluated by a single criterion or number.
- The larger number of evaluation criteria, the more information is gathered.
- People will disagree about the relative significance of evaluation criteria according with their respective interests.
- Anything can be measured experimentally, as long as how to take the measurements is exactly defined.

The generality of these principles is illustrative of the looseness of the topic. In the following, some scattered work on it is briefly described.

2.4.1 Complexity

Several attempts have been made to evaluate ES complexity. Some simple metrics on basic parameters of a KB, such as the number of rules or the vocabulary size are suggested in [Buchanan 87]. The vocabulary is assessed as the sum of different objects, attributes and legal values present in the KB. Other metrics include the size of the solution space, computed as the set of potential outputs (10^9 in MYCIN), and the complexity of the solution space defined as b^d, where d is the average depth of search and b is the average branching factor ($5.5^4 = 1000$ in MYCIN). How well these metrics assess ES complexity is not clearly known, since they are not supported by a sound experimental work.

More formally, [Shapiro 84] has defined some complexity measures for logic programs, based on the dimensions of proof tree. Let R be a proof tree. The *length* of R is defined as the number of nodes in R. The *depth* of R is the depth of the tree. The *goal-size* of R is the maximum size of any node of the proof tree, where the size of a node is the number of symbols in its textual representation. A logic program P is of goal-complexity $G(n)$ if for any goal A in $I(P)$ of size n there is a proof of goal-size $\leq G(n)$, where $I(P)$ is an interpretation for P, a set of variable-free goals. Similar definitions are given for depth and length complexity.

2.4.2 Utility and Quality

A number of utility and quality characteristics for ESs have been defined, based in occasions on equivalent aspects in conventional software [Boehm et al, 78] [Lopez et al, 90]. Among them, the following are briefly discussed: efficiency, accuracy, sensitivity, robustness, understandability, usability and transferability.

Efficiency in ESs can be seen as execution efficiency or as task efficiency. Execution efficiency considers the cost in computing resources of executing an abstract inference structure (a bunch of rules), looking for less costly ways to achieve the same result. Execution efficiency of rule structures has been analyzed in [Tao et al, 87], using a Petri net model. Task efficiency considers the optimum use of knowledge encoded in the KB to achieve correct results with less cost. Task efficiency addresses points such as opportunistic scheduling and complex control hierarchies, allowing for the required flexibility to obtain the best usage of knowledge.

Accuracy is the capacity of an ES to produce the most exact response for the input data. Neither extra nor missing information is given in the ES output. A related characteristic is sensitivity, that considers how variations in the input have consequences in the output. ESs have been criticized for having very sensitive, that is to say, small variations in the input can cause exaggerated changes in the output. Robustness is the capacity of an ES to give meaningful outputs (or at least, to keep operative) when non-meaningful inputs are provided.

Man-machine communication is usually performed through a computer terminal. Understandability consider the quality of ES questions and messages. On a more global setting, usability addresses the quality of man-machine interface, how easy the user can interact with the ES and its potential misuse. Transferability considers the ES capacity to interact or to be integrated with other software. The usage of knowledge representation standards as well as machine or shell dependency is also addressed.

2.4.3 Quantitative Evaluation

In [Liebowitz 86] a method to obtain quantitative measurements of different evaluation criteria is described. This method is based on the Analytic Hierarchy Process, which breaks down a problem into its smaller consistent parts and then calls for pairwise comparison judgements to develop priorities among them. This method allows for multiple criteria to be used and it can quantify the user's relative significance of various criteria, in accordance with the principles of [Gaschnig et al, 83] described above. This method has been implemented in a software tool called Expert Choice. The READ system, a prototype ES for determining software functional requirements, has been evaluated using this tool. Seven criteria have been evaluated: ability to update, easy to use, hardware, cost-effectiveness, discourse, quality of decisions and design time. Three experts gave their opinions on READ behavior on these criteria. Experts gave their opinions as preferred alternatives, without numerical guesses. Numerical values for each criterion are internally computed and given as evaluation result.

2.4.4 Evaluation Summary

There are dozens of characteristics to be assessed in ESs since, as stated in [Gaschnig et al, 83], complex systems as they are cannot be evaluated with a criterion or number. Some of these characteristics have been presented, but no technique to compute them has been given. Only one case study, the READ evaluation, has been provided. With the exception of performance evaluation that is assessed using testing methods (see section 2.2), evaluation is poorly developed. Characteristics are measured on ad-hoc basis, depending on evaluators preferences and without standard techniques. Transference from conventional software, where some similar issues have been found, can help greatly to develop a sound basis for ES evaluation.

2.5 Validation in the ES Life-Cycle

Several methodologies for ES development have been proposed in the last ten years. All of them contain, to more or less extent, elements of verification and testing. Only recently, methodologies including a comprehensive treatment of validation in the ES life-cycle have been proposed.

2.5.1 Buchanan´s approach

One of the first comprehensive methodologies for ES development is proposed in [Buchanan et al, 83]. This methodology is composed of the following five stages:

- *Identification*: this phase consists in determining clearly the problem to be solved (including the ES final purpose) and establishing the participants and their roles in the ES development, together with the necessary allocation of resources.
- *Conceptualization*: the key concepts and relations of the problem are made explicit. These concepts refer to the structure of the problem domain, its decomposition in subproblems, hierarchies and causal relations among facts, strategies and heuristics used, etc. Intensive interaction with human experts is required.
- *Formalization*: the key concepts and relations identified at the previous stage are mapped into more formal representations based on knowledge representation schemas. The selection of a shell is a major goal that can require the evaluation of different shells and their adequacy to the problem at hand.
- *Implementation*: the formalized knowledge is coded using the shell selected at the previous phase. A prototype is developed to show the adequacy of the formalization and the feasibility of the ES approach in this specific problem. This prototype can be used as an starting point for further versions.
- *Testing*: the implemented ES is executed on a set of test cases with known solutions. The ES behavior is analyzed in different aspects, including: ES input/output adequacy, rule errors identified from wrong ES outputs and issues in

the control strategy. The selection of a representative set of test cases is essential to achieve an accurate testing.

These phases are performed sequentially. Depending on the testing results, the ES development process can go back to previous phases resulting in an iterative process that will terminate after achieving a satisfactory performance at the testing phase. Concerning validation, the main drawback of this methodology is that testing is delayed to the last phase. Errors can be detected only when the ES (prototype or operational system) has been implemented and their correction can be very expensive, specially when error repair implies major changes in the formalization or conceptualization stages.

To avoid this weakness [Radwan et al, 89] suggest to adapt an independent verification schema extracted from classical software engineering [Fujii 77], with the goal of adding verification checkpoints after each development phase. The initial schema of [Buchanan et al, 83] is modified, specifying the development product obtained after each phase and adding verification activities to be performed after the completion of each phase. The results of these verification activities have effect on previous phases. Seven independent verification activities are proposed: (1)literature base critique (literature review), (2) paper-based methodology analysis, (3) heuristic approach analysis, (4) program logic component integrity verification, (5) example application of methodology conformance analysis, (6) literature base methodology conformance analysis, and (7) problem domain definition conformance. The experience of using this verification framework to develop an ES in traffic engineering is described in [Radwan et al, 89], although its applicability to other ES domains is not considered.

2.5.2 Rapid Prototyping

Rapid prototyping is in the kernel of several ES methodologies. The basic idea of rapid prototyping is that software development can be guided by incrementally expanding a product until reaching an operational state. An executable prototype constitutes an operational environment in which the user can experiment and determine further improvements, at relatively low cost. The prototype is used as an starting point for the specifications of next developments. In ESs, rapid prototyping gives response to two important issues when a problem is going to be solved by ES means: the ES feasibility in this specific problem is shown by developing an early prototype, and rapid prototyping allows to circumvent the usual absence of detailed requirements at the beginning of an ES development, absence that makes inapplicable the conventional software life-cycle.

Following this methodology, the ES development is divided in four stages: [Geissman & Schultz, 88]

- *Problem Determination*: the major objective is to ensure that the intended ES will satisfy a real need and it is technically feasible. This involves analysis of existing successful ESs and shells and determine knowledge base requirements (not very detailed at this point).

- *Initial Prototype*: the goal is to show the economical and technical feasibility of the intended ES through an executable prototype, which is concerned with a subpart of the problem. The initial prototype is the source of requirements for further developments.

- *Expanded Prototype*: the target is to achieve an ES with a full range of capacities to solve the problem, either by expansion of the initial prototype or by rewriting it from scratch. As in the previous stage, the prototype is the source of requirements for the next stage.

- *Delivery System*: the goal is to develop an operational system suitable for an specific hardware, optimizing critical resources and with a user-friendly interface. Requirements identified at previous stages are used to verify the delivery system.

In this model the following validation guidelines are proposed:

- Develop initial prototype resulting in testable requirements.
- Design in terms of formal paradigms.
- Certify inference engines.
- Design for verification.
- Verify the knowledge base.
- Perform formal validation.

These recommendations are all quite reasonable, but they are not detailed enough to be considered as a comprehensive methodology for ES development. Following the rapid prototyping approach, [Weitzel & Kerschberg, 89] suggest a more detailed scheme for building a prototype. Their proposal (KBSDLC) consists of a modification of the sequential phases for conventional software development. The KBSDLC is composed of a list of processes that can be activated, deactivated and reactivated. Initially a process only can be activated after the previous process has been activated. An active process can reactivate any previously activated process, so processes can run in parallel. The ordered list of processes is:

- Identify problem.
- Definition/Feasibility.
- Identify subproblems.
- Identify concepts.
- Conceptual design.
- Detailed design.
- Code.
- Test reasoning.
- Test knowledge.
- Validation.
- Convert, maintain/enhance.

This approach has been used to develop successfully an ES in medical insurance. However, the process reactivation capability arises some questions that are not solved in the proposal. For instance: Does exist any synchronization between the reactivating and the reactivated processes, once the second has been reactivated?. How can a process reactivation affect other active processes that are below it and different from the reactivating process?. Is termination guaranteed?.

Another approach inside the rapid prototyping paradigm is the waterfall life-cycle [Miller 89]. This approach combines the waterfall development process in conventional software engineering [Royce 70] as a fixed sequence of development steps driven by the system requirements, with the advantages of prototype building. There are five steps that are performed sequentially: (1) requirements, (2) knowledge acquisition, (3) design, (4) development and (5) evaluation. This sequence is performed twice, for prototype development and for building the baseline system. In addition, there are some steps devoted to the delivery system. Validation procedure consists in checking the system performance against system requirements. Validation is performed at each evaluation step, with more emphasis in the delivery system. Validation phases, documents and reviews are defined and located in the ES life-cycle.

2.5.3 Spiral Model

The spiral model [Boehm 88] is a software process model based on risk analysis that can integrate previous models such as the waterfall model [Royce 70] or the evolutionary model [McCracken & Jackson, 82]. The nuclear idea of this model is the following: software development is driven by risk analysis and risk resolution. If in a portion of software product, performance or user-interface risks dominate program development or internal interface-control risks, an evolutionary development can be selected. On the contrary, if program development or interface-control risks dominate, a waterfall approach can be selected. Thus, this model can accommodate different approaches to software development at different stages.

Based on the evolutionary model and on the spiral model, [O'Keefe & Lee, 90] propose an integrative model of ES verification and validation. Each spiral cycle begins with a requirement analysis, followed by knowledge acquisition and resulting in setting the Acceptable Level of Performance (ALP). Next phases include developing a prototype (at the corresponding stage) and verifying and testing it with respect to the established ALP. In particular, verification and validation activities are concentrated in the following phases:

- *Requirements analysis*: requirements involve the fit between the intended ES and the receiving organization, the problem determination and identification of potential users. ES functionalities and constraints are also defined. The final step consists in verifying the collected requirements, and it can be performed manually.

- *Knowledge acquisition*: the knowledge acquisition (KA) process is a major task in ES development, and its validation is very important. This can be made by diagrams made by knowledge engineers summarizing the collected knowledge and verified by the experts. Verification is also possible in automated KA tools, specially to check conflicts.

- *Prototypes*: prototypes are developed following the evolutionary (rapid prototyping) approach. Prototypes can be validated by mechanical verification, case testing, field testing and Turing tests. When the prototype evolves to a production system, a control group experiment can be made.

The main advantage of this approach is that validation is incorporated into all the stages of ES development. In spite of the fact that is based on the spiral model, risk analysis is not present in the proposal.

2.5.4 Validation in the ES Life-Cycle Summary

Significant similarities can be extracted from a comparison among the previous approaches. All of them include the concept of prototype as a basic concept that drives (or in [Buchanan et al, 83] can drive) the development process. To develop a prototype they provide a sequence of steps (except [Geissman & Schultz, 88]). Comparing these steps among the different approaches shows that they they match quite well. This matching is shown in the table 2-3.

Buchanan et al, 83	Weitzel & Kershberg 89	Miller 89	O'Keefe & Lee 90
Identification	Identify problem Definition/Feasibility	Requirements	Requirement analysis (includes verification)
Conceptualization	Identify subproblems Identify concepts	Knowledge Acquisition	Knowledge Acquisition (includes verification)
Formalization	Conceptual design Detail design	Design	Prototype
Implementation	Code	Development	Prototype
Testing	Test reasoning Test knowledge Validation	Evaluation	Prototype (verification, case test, field test, control group)

Table 2-3 Matching development steps.

The five steps of [Buchanan et al, 83] have an almost direct correspondence in the most detailed methodology of [Weitzel & Kershberg, 89]. Only the last step in Weitzel's methodology, Convert/Maintain/Enhance, is not considered (maintenance is also mentioned in Buchanan's approach but it is not included in the basic stages). However, there is a major difference in the step sequencing, it is iterative from the testing step to previous steps in Buchanan's approach while Weitzel allows steps to be executed in parallel. There is also a pretty direct correspondence with the steps proposed in [Miller 89], but it should be taken into account that Miller proposes an specialization of these steps, first for the initial prototype and later for the baseline system. Also the final stage in Miller's approach (integration stage) is not included. In spite of the differences there is a high degree of matching among these three proposals. The approach of [O'Keefe & Lee, 90] is more distant because it includes verification activities in the initial steps of the ES development, and prototype design and coding are not separated.

The following points summarize the different approaches to ES life-cycle:

- Prototype building is in the kernel of ES methodologies.
- Several step sequences are proposed to develop a prototype. There exists a significant degree of matching among steps belonging to different approaches.
- Early approaches [Buchanan et al, 83] only consider testing after implementation. This is identified as a weakness that later approaches try to solve including validation activities after each development step.
- Some proposals, [Radwan et al, 89] [Weitzel & Kershberg, 89], have been developed to implement an specific ES but their applicability as global ES methodologies is unclear.
- Regarding validation, the most promising approaches are those that include a comprehensive treatment of validation in the whole ES life-cycle [Miller 89] [O'Keefe & Lee, 90]. No practical case studies of these proposals has been reported.

2.6 Case Studies

Most of what has been learned on ES validation has been obtained from the practical experience on validating ESs, when these systems were more software experiments than mature programs with interesting capabilities. Two of the most widely cited ESs, MYCIN and R1, have been chosen to summarize their well-documented experience in validation. The description of these systems, implemented 10-15 years ago, is completed with a survey of the state-of-the-practice in ES validation in USA.

2.6.1 MYCIN

The MYCIN system provides therapy advice for infectious diseases. It is probably the best documented ES in history due to the book of [Buchanan & Shortliffe 84]. Chapters 30, 31 and 36 of this book contain significant information on MYCIN validation. Among the lessons learned with MYCIN, it is mentioned a clear separation between the performance level achieved by an ES and those factors affecting its utility and acceptance by its end-users. It is concluded that high performance is a necessary but not sufficient condition for ES usage, and it deserves separate evaluation.

MYCIN performance was assessed using three evaluation studies. The first study was made in early 70s, using 15 test cases on which MYCIN had offered therapy advice. Five physicians of Stanford reviewed these cases, giving an approval rating of 75%. In mid 70s, the second study was made using the same 15 cases with the MYCIN recommendations. Ten physicians (five from Stanford, five from other centers) acted as evaluators. Although MYCIN knowledge had been considerably refined from the first study, the approval ration was again about 75%. The third study was made in a different form. Ten "diagnostically challenging" cases were selected from a county hospital. Ten prescriptions were compiled for each case: the one actually given by the treating physicians at the hospital, the recommendation made by MYCIN and the recommendations of seven Stanford physicians and a medical student. These 100 prescriptions were given to eight non-Stanford experts in the field, who acted as evaluators without knowing the prescribers' identity (Turing test). Results showed that the evaluators disagreed with MYCIN recommendations no more often than with the recommendations of Stanford experts. In other words, MYCIN performance was at the same level than human experts. The MYCIN experience has shown the real utility of Turing tests. However, they are expensive and require a considerable effort by researchers and experts in the problem domain.

Regarding utility and acceptance factors for ESs, the MYCIN project does not provide an study as detailed as the performance one. It was assumed that for ES acceptance, it was enough with the existence of a demonstrated need coupled with a high performance of the ES for that need. Retrospectively, this point of view was naive. Acceptability is different from high performance. To assess physicians' opinions about ESs in medicine, a survey was made. Three aspects were considered: acceptability in medicine, effect expectations and performance demands. This survey collected opinions of 146 physicians, of which 85 had followed a tutorial on medical computing and 61 had not had prior relation with computing. From the survey results, it is clear that explanations are necessary for acceptance. If an ES is unable to explain its line of reasoning it will not gain the confidence of the clinicians. Most physicians accepted ESs as aids to clinical practice but rejected them as a way to automate clinical activities. They could see ESs as a threat to the traditional clinical function. As tools, ESs are accepted when they provide assistance to physicians, but rejected if they provide a dogmatical advice, no matter how good is their performance.

2.6.2 R1

The R1 system provides help in configuring computer systems. In practice, R1 testing was made in two phases: a formal testing plus a field test at the working place. Formal testing consisted on configuring 50 orders on which no selection was made; they were just the last orders to come through. Twelve people involved in the configuration task, six of them were actual experts, acted as evaluators working together. They examined carefully the R1 outputs. Significant disagreement among evaluators about the right way to do configurations was reported, although agreement about the difference between acceptable and unacceptable configurations was pretty good [Gaschnig et al, 83]. In the examination of the produced configurations, twelve were found to contain errors. All but two of these mistakes were at a level of detail below that at which humans work out configurations. The team of experts was very impressed of R1 performance, and after correcting the rules responsible for these errors, R1 was formally accepted in December 1979. After formal acceptance, R1 was put in operation on a regular basis, causing a real field test. Users reviewed R1 outputs, reporting errors to the review committee. According to the R1 performance results given in [Bachant & McDermott 84], during the first year of use, 1980, R1 was a quite unskilled configurer with more than 50% of errors in the processed orders (339 problem instances in a total of 591 orders). However, R1 performance improved drastically until achieving 10.8% of errors in more than 20,000 orders in December 1983. Most of these errors, 9.8%, were due to incorrect or missing update of the data base, while knowledge in rules was responsible for only the 0.4%.

A number of lessons were learned from R1 experience. The test set of the formal testing procedure was judged too short and not representative by his developer, from a retrospective point of view [McDermott 81]. Rushby points out that the test criterion was naive, without a clear gold standard, confusing development and acceptance purposes, and without enough involvement of end-users [Rushby 88b]. Acceptance testing cannot be considered complete until the ES is actually employed routinely for the intended task [Gaschnig et al, 83]. Regarding the high performance expectations (90% hits expected during the first year) [Bachant & McDermott 84] qualify this kind of expectations for ESs in complex tasks as a serious mistake. They also consider that to keep the ES away from regular use until completing its knowledge is a poor idea. R1 experience has shown the capacity of field test and the important role that end-users can play to achieve a fully operational ES.

It should be noticed that R1 was greatly expanded during these four years. At December 1979, R1 was composed of 777 rules and 420 parts in the database, and was able to configure one computer model. At December 1983, R1 was composed of 3,383 rules with a database of 5,481 elements, with a capacity of configuring ten different computer models. The problem of maintainability of R1 has been addressed in [Soloway et al, 87].

2.6.3 State-of-the-Practice

The results of a survey assessing the state-of-the-practice in verification and validation of ESs are reported [Hamilton et al, 91]. The survey was composed of two parts: a questionnaire and an interview. The questionnaire asked for the kind of application, expertise information, ES development, evaluation and performance information. Seventy people responded to the survey, most of them (93%) were developers. The majority of considered ESs were operational (70%) with predominance on diagnosis in the aeroespace field (73%). Regarding performance, most of them considered their ESs to be less accurate than expected and also less accurate than the expert. No requirements were documented, using the expert as oracle for correct ES behavior. The most frequently (40%) used life-cycle model was the evolutionary model (rapid prototyping), while 22% did not use any model. Most common validation activities were functional testing (66%) and structural testing (44%), commonly performed by the expert (59%) and requiring the 24% of development effort. The validation issues most often cited were test coverage determination (63%), knowledge validation (60%) and problem complexity (40%). There was a wide range of techniques mentioned.

Interviews were used to gather additional information and clarify survey responses. Structural testing was generally performed on each element as an independent piece, without considering rule interactions. The measure of test coverage was not generalized. Experts were heavily involved in the development and evaluation of the knowledge base, to the point that some of them were the only developers. Writing requirements were associated to software development models not adequate for ESs, detecting a clear reluctance to them. Also the distinction between prototype and operational system was not clear.

From these information, the survey authors concluded that the current state-of-the-practice could be improved, suggesting direct and inferred recommendations. Direct recommendations included: (i) developing requirements for ES validation, (ii) addressing common issues to ES validation (test coverage, knowledge validation, real-time performance and problem complexity) and (iii) recommending a life-cycle for ES development. As inferred recommendations, they suggest to address the following issues: (i) readability and modularity, (ii) configuration management , (iii) criteria to classify ESs by their intended use, and (iv) applicability of analysis tools.

2.6.4 Case Studies Summary

The experiences of MYCIN and R1 illustrate the random path followed to achieve an accurate evaluation of ES performance. Both failed in the first testing approach (unblind testing in MYCIN, unstructured testing in R1) and succeeded in the subsequent (Turing test in MYCIN, field test in R1). The difference between performance and acceptance is clear, together with the importance of end-users and actual task execution on acceptability criteria. These experiences have required lots of effort by developers, experts and users, and it seems reasonable to get benefits from them. The survey shows that current ES developers devote 1/4 of development effort to validation (what could be reasonable) but it is also clear that validation is made without plan or methodology. The wide variety of validation methods, together with the incomplete testing (rules are tested independently, without considering interactions), suggest that ES validation does not achieve the degree of depth and exhaustivity required to guarantee a high level of quality in ES applications. On a more global basis, ESs are developed informally, without written requirements and leaving the expert alone in many cases. Survey recommendations attack these pitfalls aiming to a more formal development methodology combined with the usage of available techniques that can largely improve ES quality.

2.7 Summary

Analyzing each of the considered topics, it is clear that verification is the most developed part of ES validation. This is due, to a large extent, to the logical foundations of the knowledge representation based on rules. Verification is now evolving, leaving the assumption that rules are considered as pure logical entities, without taking into account the way they are actually used. More realistic (and therefore more complex) ES models (including uncertainty, control and non-monotonicity) begin to be considered. Also, no logically-based verification properties are checked.

Regarding testing, a diversity of partial approaches is currently available. However, a comprehensive testing approach including the fundamental issues of test set selection and performance evaluation is missing. Testing is a key aspect of ES validation that can be solved on an ad-hoc basis with substantial effort (see section 2.6 on Case Studies). Regarding its importance, testing can be seen as a weak point in ES validation. Some of these lacks can be solved using knowledge base refinement techniques. Refinement, a topic traditionally included into machine learning, can be of great help for ES validation. It aims at automatically improving the contents of the knowledge base by inductive learning from a library of cases with known solutions. The knowledge base is considered as the base theory that is imperfect and should be refined. Refinement tools can automatize the testing process to a certain extent and under human supervision. They can automatically compute the performance level as the number of the cases correctly solved in the case library. This can have a significant impact in acceptance procedures and in ES maintenance, since the case library can be maintained and updated during the ES existence.

Evaluation is a part of ES validation very unevenly developed. Except for evaluating ES performance, where testing techniques are used, no methods exist to assess ES characteristics. Some theoretical studies have been made, but confirmation from practical experience is missing. It is reasonable to expect some further developments on this topic in the next years, as a result of the implantation of ES applications on a regular basis.

Regarding the ES life-cycle, an almost general agreement exists about the fact that validation should be present in all stages of ES development. Early validation is recommended because the later an error is detected the more expensive it is to correct. Some experiences have been made to include validation in all stages of ES development, modifying proposed ES methodologies. Also, new proposals of ES life-cycle with special attention to validation have been made. Special attention should be deserved to the existence of written requirements for ESs, because of the key role they play in validation. Testing and verification depend heavily on the existence of clear and detailed requirements for the intended task. ES methodologies have to address the effective development of ES requirements.

The Case Studies section contains the description of two validation experiences (MYCIN and R1), from which many practical conclusions can be extracted, specially about the testing aspects. In addition, the interesting survey of the state-of-the-practice suggests, to some extent, a certain divorce between ES research and ES industry in the validation field. For instance, verification is not mentioned as an important validation technique, in spite of being the most developed part and of the existence of a large number of automatic verifiers available for different shells. Conversely, testing is rightly mentioned as one of the first validation problems, but ES research does not offer a comprehensive solution to it. In spite of the general agreement on the convenience of using ES life-cycle models, the survey reveals little usage of these models in practice as well as a clear absence of written requirements. This divorce can be explained by a double hypothesis: ES research might have been more attracted to solve feasible problems than to cope with real issues; conversely ES industry might not have been receptive to research results, considered too theoretical.

There is a clear evolution of ES validation work regarding its relation with conventional software validation. Early work on ES validation included almost no references to conventional software. As the field matured, the presence of conventional software increased, specially regarding validation concepts and methods that can be successfully adapted to ES peculiarities.

Finally, it is apparent that a lot of work has been devoted to ES validation in the last years. This work is now producing results, which are improving rapidly the state-of-the-art on the field. In consequence, it seems to be a quite reasonable forecast to expect better ES validation methods in the near future.

Chapter 3

The Validation Issue

In this chapter, we address a number of fundamental questions around ES validation. The final aim is to get a conceptual framework for ES validation, where the meaning and main components of ES validation are defined and understood in an integrated way. This framework also consider those ES aspects that are suitable for some kind of validation, as well as specific techniques to achieve validation in practice. Developing a conceptual framework is a difficult aim because of the immaturity of the field. Different validation aspects are unevenly developed, appearing clearer those parts to which more work has been devoted. Fundamental definitions such as, for instance, the meaning of ES validation, are still not consolidated. Many validation aspects are today elements for debate in the research community. Therefore, the work we present here reflects the lacks and defects of the current state. Nevertheless, drawing the whole validation picture is a useful exercise that, at least, will allow us to identify the most obscure parts.

We analyze ES validation from four points of view. First, we discuss the meaning of validation in ES. This raises some terminological questions about the precise meaning of terms like verification, validation, evaluation or testing, commonly used in the literature. We propose precise definitions for these terms using the concepts of totally and partially formalizable requirements. In addition, these definitions are in compliance with the established meanings for these terms in software engineering. Second, we consider elements and processes on which some kind of validation can be performed. We consider the following validation targets: user requirements, knowledge acquisition, expert system architecture, knowledge base structure and contents, inference engine and expert system behavior. Third, we identify specific techniques for each one of these validation targets, aiming for an effective validation. We devote special attention to those aspects that are specific of ESs, although there are a number of points in common with software engineering. And fourth, we locate these validation targets in an ES life-cycle. Every step in the life-cycle has associated one or several validation activities. Then, the whole validation process is the agregattion of all these validation activities.

The structure of this chapter reflects these four points of view. Section headings are questions on the meaning, contents, methods and time of validation. Section 3.1 raises the question *What is Validation?*, analyzing validation in ESs. This section contains a terminological discussion about validation terms in ESs, considering the meaning of these terms in software engineering. Section 3.2 addresses the question *What Should be Validated?*, considering the previously mentioned validation targets. We analyze the relevance and impact of their validation in the knowledge engineering process. Section 3.3 investigates the question *How to Validate?*, considering how to perform validation on these validation targets. Section 3.4 addresses the question *When to Validate?*, looking for the right time in the ES life-cycle for validation activities. Finally, section 3.5 summarizes the chapter.

3.1 What is Validation?

Validation is not a new topic in computer science. In the context of software engineering, [Adrion et al, 82] define software validation as,

```
Determination of the correctness of a program with respect to
the user needs and requirements.
```

This is an open, non-constructive definition. No single validation method can assure a complete validation. On the contrary, a large variety of validation methods for conventional software exist with complemented effects.

Differences between ESs and conventional software, specially regarding the kind of problems addressed and the type of programming languages used in both fields, have caused that some authors consider ES validation as somehow different from conventional software validation. We do not share this point of view. An ES is a piece of software that performs some tasks aiming to satisfy the needs of potential users. These needs have to be explicited to allow for an effective definition of the ES purposes. Therefore, the previous definition for validation is perfectly applicable to ESs. From now on, we assume this definition as the right one for ES validation.

The applicability of the conceptual definition does not imply the direct applicability of many validation methods for conventional software to ESs. The existing differences between both fields (see section 1.2) justify and demand the existence of specific validation methods for ESs. However, the accumulated experience in conventional software validation cannot be ignored in ES validation. This experience has to be applied considering the essential issues to be solved (shared to a significant extent by both conventional software and ESs) regardless of specific techniques employed that are usually bounded to the style of the used programming languages. A study of conventional validation methods and its potential application to ESs is found in [Rushby 88b].

In the current state of ES validation, terms like *validation*, *verification*, *testing* or *evaluation* are very frequently used. However, their exact definitions and the ways to achieve them in practice are still unclear. In the following we make a tour on these terms, trying to elucidate their meanings and relations. We consider *validation* as a global term, that embodies all others terms as specific aspects.

3.1.1 Requirements and Specifications

According to the previous definition of validation in software engineering, a program cannot be validated without a set of user requirements. These requirements can be clear or ill-defined, they can be communicated by speech or by written means, or even they can be implicit. But when somebody emits a judgement evaluating a program, these requirements are involved. User requirements are usually expressed in natural language. To enforce their understanding, user requirements are translated into a formal language, generating what is called user requirement specifications, or simply specifications. A specification is a non-ambiguous expression defining some kind of program characteristic or property. A specification must be verifiable and testable with respect to the specified program. Specifications can exist for all the software development steps.

Regarding ESs, user requirements play the same role that in software engineering[1]. However, the complete translation of user requirements into specifications, theoretically

[1] However, a recent survey on the state-of-the-practice reveals a strong reluctance to record written requirements for ESs [Hamilton et al, 91].

assumed in conventional software, does not hold for ESs. There are user requirements that, at the current state of the ES technology, cannot be expressed in formal terms using an specification language[2]. This conclusion, extracted from our own experience [Hoppe & Meseguer, 91], is also shared by [Krause et al, 91] in the context of medical ESs. They used the formal specification language Z to specify the intended behavior of a medical ES using available medical protocols. They conclude that the protocol acts as a specification of the default behavior, but an explicit representation of the circumstances in which this behavior is valid, is missing. So, it seems that the ill-defined nature of ES tasks is the main reason to prevent an accurate and complete specification of the intended ES.

3.1.1.1 Totally and Partially Formalizable Requirements

Although a complete translation of user requirements into specifications seems unfeasible, a partial translation is reachable. This suggests us to divide the user requirements in two classes: *totally formalizable and partially formalizable*[3]. A requirement is totally formalizable if it can be completely translated into specifications. Conversely, a requirement is partially formalizable if it cannot be completely translated into specifications. An example of totally formalizable requirement is:

"The KB should be *structurally correct*"

The concept of structural correctness for KBs is well-defined, so this requirement can be translated into the following specifications (assuming the rule-based paradigm):

1. The KB objects should be syntactically correct.
2. The KB should be contradiction-free.
3. The KB should not contain redundancies.
4. The KB should be cycle-free.
5. All the KB objects should be potentially usable.

These specifications are expressed in natural language for readability reasons, but they can be clearly expressed in a formal language. Considering medical ESs, an example of partially formalizable requirement is:

[2] We stress that a specification must be verifiable and testable against the ES. Therefore, it should be expressed in terms of actual elements of the ES design and implementation.

[3] These concepts correspond to formalizable and informal specifications introduced in [Hoppe & Meseguer 91]. However, after reading an earlier version of [Laurent 92], I realized that a specification is always formal, so the concept of informal specification was somehow contradictory. I also realized that the insights underlying these concepts (and the examples provided) were aiming to the user requirements. These are the reasons for this terminological, but important, change.

"The ES has to provide an *acceptable* diagnosis for all the *typical* cases"

In many medical domains there is not a sharp definition of what is an acceptable diagnosis for a case. In addition, the concept of typicality is not well-defined and, in occasions, it is a matter of personal preference. Due to the lack of a precise definition for these concepts, this requirement cannot be *completely* translated into specifications. However, it can be partially translated into specifications, which capture to a certain extent the meaning of the requirement. For instance, in the context of pneumonia diagnosis, the previous requirement about correct diagnosis for typical cases can generate the following specification:

"If rusty sputum is present, the pneumococal etiology has to be considered"

that is expressed in formal terms saying that the ES cannot stop without having pursued the pneumococal etiology as a goal. The translation is performed by human experts, who interpret the requirement and provide some specifications. In occassions, these specifications are tentative and they have to be confirmed or discarded by experimentation. Nevertheless, specifications obtained from partially formalizable requirements are completely formal. This is pointed out in [Laurent 92], who calls them pseudo-specifications but stressing their formal nature.

3.1.1.2 Service and Competence Requirements

Rushby points out the necessity of requirements for ESs [Rushby 88a]. He divides ES requirements into *service* and *competence* requirements. Service requirements consider the operation of the ES as a software tool: input-output formats, processing rate, operational conditions, and so on. Competence requirements are concerned with the definition of the intended task for an ES, in the sense that it is a typical human task requiring "knowledge". Competence requirements are further subdivided into *desired* and *minimum* requirements. Desired competence requirements describe *how well* the ES is expected to perform. Minimum competence requirements define *how badly* the ES is allowed to perform: they state the threshold for ES acceptance. Rushby admits that desired competence requirements can be vague or incomplete, because they are usually made in comparison with human performance. However, he states that service requirements and minimum competence requirements should be precisely defined, specially for safety critical applications.

Rushby classifies requirements guided by their role in defining the ES task. Our classification of requirements (totally formalizable vs. partially formalizable) is guided by a more syntactical criterion: their complete translation into specifications. A clear parallelism exists between both classifications. Service and minimum competence requirements are totally formalizable requirements, since they generate "specifications that should be traceable, verifiable and testable just like those of conventional software" [Rushby 88a]. Desired competence requirements can be vague or incomplete, so they fall into the class of partially formalizable requirements, to an extent depending on the specific application.

3.1.1.3 User Requirements at the Knowledge Level

Why do partially formalizable requirements exist?. Why are not all the requirements totally formalizable?. One may think that is a purely technological matter, that the development of better specification languages would solve. Although this point is important (specially for practical reasons), we think on deeper causes to explain the nature of partially formalizable requirements. These causes are intimately related with the question of knowledge and its representation, addressed by Allen Newell in [Newell 82].

Newell differentiates between the *knowledge level* and the *symbol level* in AI systems. At the knowledge level, a system is viewed as an idealized rational agent, who posseesses knowledge, goals and actions. This agent is not subject to computational limitations and it behaves according to the principle of rationality:

```
If an agent has knowledge that one of its actions will lead
to one of its goals, then the agent will select that action.
```

At the symbol level, a system is viewed as a collection of of computational entities (facts, rules, frames, etc) that interact according to some predetermined operations (matching, assignement, etc.). The symbol level is the computational realization of the knowledge level. The relation between knowledge and symbol levels is not univoque. On the contrary, a radical incompleteness characterizes the knowledge level that cannot predict the system behavior in many cases. Mapping a knowledge level description into an adequate symbol representation is an open problem, currently considered by a number of cognitive architectures [Chandrasekaran 87] [McDermott 88] [Steels 90].

The distintion bewteen knowledge and symbol level in ESs is very relevant to us. When users express their requirements, specially the functional ones, they are usually considering the ES at the knowledge level. An expert views the ES as another colleague and expresses the requirements assuming that it posseses the background to understand them. However, specifications are clearly at the symbol level. They deal with well-defined computational objects such as facts, certainty degrees, pursued goals or termination conditions. In this view, the difficulties to express user requirements into specifications are explained by the following causes:

- The knowledge level is an approximation of the actual ES behavior in degree as well as in scope. When user requirements at the knowledge level are translated into specifications at the symbol level, their approximate nature causes difficulties in their complete translation.

- The knowledge level is incomplete since it does not determine completely the ES behavior. A user requirement at the knowledge level may generate many possible and alternative specifications at the symbol level. In addition, Newell affirms that knowledge level descriptions may be completed with precise descriptions at the symbol level (mixed systems). These precise descriptions can be seen as the requirements that are totally formalizable.

- The absence of precise and complete mappings between the knowledge and symbol levels is another cause for the partial translation of user requirements into specifications. The development of cognitive architectures at the knowledge level can represent a significant step towards the accurate expression of user requirements.

3.1.2 Verification

We define *ES verification* as the process of checking an ES against the specifications generated by its formalizable requirements. The verification process produces accurate responses about the satisfaction of each specification. Given that specifications completely contain the meaning of totally formalizable requirements, the verification process allows for a complete checking of them.

Usually, specifications establish conditions that should (or should not) hold for all the possible ES executions. To test these specifications, verification has to be exhaustive, that is to say, it has to analyze all the possible situations where a specification may be violated. Frequently, specifications contain terms involving the dynamic execution of the ES, such as termination, goal pursuing or fact deduction. In order to check accurately these specifications, the verification process has to reproduce faithfully the execution conditions of the specific ES, that is too say, verification has to consider the operational semantics of KB objects [Evertsz 91]. Otherwise, specifications will not be totally checked and the verification results may be questionable.

Totally formalizable requirements can be domain-dependent or domain-independent. An example of domain-dependent requirement in pneumonia diagnosis is that the concepts of typical bacterial pneumonia and atypical pneumonia cannot be considered both with a high degree of certainty for the same patient (see section 4.7.2.1 for a medical explanation). This domain-dependent requirement is modeled as an integrity constraint and it is tested as a potential inconsistency. An example of domain-independent requirement is the common requirement about the structural correctness of a KB, that can be decomposed into (1) correct syntax, (2) contradiction-free, (3) no redundant, (4) cycle-free and (5) without useless objects. Some specifications from domain-independent requirements remain somehow implicit. For instance, the trivial specification of correct syntax is often not explicited. A KB containing objects with syntax errors can cause serious problems in the ES function (an ES crash or an erratic behavior). Therefore, the decomposition of requirements into specifications should be careful and detailed. Some work on ES specifications can be found in [Slage et al, 90] [Batarekh et al, 91].

So far, verification has been mainly focused on checking properties coming from the knowledge representation language used in the KB (see section 2.1), corresponding to domain-independent requirements. The reason for this predominance relies on the fact that knowledge representation languages offer a higher level of formalization than the knowledge coded using them. In consequence, extracting formal properties of a representation language is more direct than obtaining formal properties of the involved knowledge. Properties of the representation language can only assure some kind formal correctness in the knowledge expression, but they cannot deal with the knowledge organization and semantics. To check properties more related to knowledge, some extra knowledge with respect to the KB contents is usually required. The reason for this extra knowledge relies on the fact that the KB usually contains enough knowledge to achieve correct conclusions, but this knowledge is not enough to check the correctness of these conclusions. For instance, if we want to check the following property regarding pneumonias,

When *ecthyma-gangrenosum* is present, the main symptoms and signs supporting atypical pneumonias should not be considered.

we need to describe what are the main symptoms and signs[4] for atypical pneumonias. This information is not explicitely recorded in the KB, since a classification of symptom importance is not contained in it. This information is not explicitely recorded in the KB because it is not needed for deduction. The KB has been constructed only considering deduction, without including support for verification. To a minor extent, another example of extra knowledge can be seen in section 4.7.1, where incompatible values for multivalued facts and integrity constraints on the input are added to the KB contents for verification purposes. This necessity of specific knowledge for verification, and in general for validation, has to be considered at early stages in ES development in order to collect it using knowledge acquisition techniques.

3.1.3 Evaluation

We define *ES evaluation* as the process of checking an ES against its partially formalizable requirements. This definition is very close to the one given in [Laurent 92]. Given that no complete translation of partially formalizable requirements into specifications is achieved, the evaluation process always contains a subjective element of interpretation.

In ES evaluation we can differentiate three steps: (i) *requirement analysis*, (ii) *specification checking* and (iii) *result interpretation*. The requirement analysis step consists in expressing the partially formalizable requirements into specifications. This step can be tentative or experimental in many cases, given the vagueness and imprecision of the requirements. The specification checking step considers the satisfaction of the specifications for the ES, in the same way they are considered in verification. The result interpretation step consists in, from the specification checking results, assessing to which extent the original requirements are satisfied. The subjective element of interpretation is present in steps (i) and (iii).

An example may help to understand these steps. Let us consider the following partially formalizable requirement for ESs in medical diagnosis:

"The ES has to provide an *acceptable* diagnosis for all the *typical* cases"

The first step, requirements analysis, has to specify what is understood by *acceptable* diagnosis and by *typical* cases. A possible interpretation of acceptable diagnosis is the

[4] A symptom is what the patient feels (headache, sickness, etc) while a sign is what the physician objectively obtain from the patient state (temperature, number of leukocytes, etc).

ability to explain all the important symptoms and signs encountered in the patient. To define typical case, a subset of all possible combinations of the considered symptoms and signs has to be specified. Assuming these definitions as specifications, the second step checks them in the ES. This can be made either (i) analyzing the KB, inferring the ES function for the typical cases and showing for which typical cases all the important symptoms and signs are explained, or (ii) selecting a sample of typical cases and executing the ES on them. The third step interprets the ES results in terms of the original requirement. Those typical cases for which no acceptable diagnosis has been obtained, are studied to assess (a) the importance of the non explained symptoms, and (b) their degree of typicality. If specification checking has been made by test sample, the degree of sample representativity has also to be considered.

So far, evaluation has been focused almost exclusively on assessing ES performance using testing methods. It is known that high performance is not synonymous of user acceptability for ESs [Buchanan & Shortliffe, 84]. Therefore, other ES characteristics have to be evaluated to achieve a global ES validation. In this sense, the development of metrics assessing ES characteristics numerically can be of significant help. ES characteristics can be quantitative or qualitative, while metrics are always quantitative, so metric results have to be interpreted in the context of the specific ES in order to obtain their real meaning. No one-to-one relation exists between characteristics and metrics, for each characteristic several metrics can be considered.

In the three step process for ES evaluation described above, the use of metrics to assess user requirements is limited to step (ii). Step (i) considers the expression of requirements in terms of metrics, while step (iii) interprets metric results. A variety of ES metrics is needed to assess ES characteristics, since according to [Gasching et al, 83], complex objects like ESs cannot be evaluated by a single number. The final goal of ES metrics is to summarize information to allow for an effective checking of the different types of user requirements, ranging from well-defined structural properties to other aspects of imprecise definition like utility or understandability.

3.1.4 Testing

We define *ES testing* as the process of examinating ES behavior by its execution on sample cases. According to this definition, ES testing has the same meaning that conventional program testing [Adrion et al, 82]. In practice, ES testing is a very important technique to evaluate ES correctness. Testing has been used as a fundamental criterion for formal acceptance of ESs (see in section 2.6 the MYCIN and R1 experiences).

A central issue in testing, shared for both conventional software and ESs, is the selection of the test set. Random, structural and functional approaches have been developed for conventional software. ES random testing can be used in ESs to check robustness, although it seems to be of little value to check correctness, since a random input will probably be meaningless in the ES domain. ES structural testing is performed using test-case generators (see section 2.2). ES functional testing relies completely on the selection of test cases by human experts. Historical files on the domain of the ES task are usually taken as source of potential test cases. The quality of the test set depends largely on the extension of these files and on the accuracy of the recorded data.

Specific issues of ES testing are: (i) comparing the ES performance against human expert competence and (ii) judging the adequacy of the deduction path followed, as well as the correctness of the ES output. To assess the ES performance, the presence of human experts as evaluators is needed. This raises new issues, since human experts often disagree and can exhibit problems like prejudice, parochialism and inconsistence. To prevent prejudice in human evaluators, Turing tests have been successfully used (see section 2.6.1, the MYCIN case). To measure the consistency degree among several experts some statistical approaches have been proposed. Considering the adequacy of the deduction path followed by the ES, this aspect has been traditionally evaluated in relation to its explanation capability. This aspect has great importance for final ES acceptance, so it has to be considered in isolation. Some work related to ES deduction path adequacy can be found in [Lopez 91].

3.1.5 The Global Picture

The validation definition given at the beginning of this section establishes that validation has to be determined with respect to the user requirements. We have classified user requirements into totally formalizable and partially formalizable, depending on their total or partial translation into specifications. Verification is the process of checking an ES against its totally formalizable requirements, while evaluation considers ES compliance with respect to its partially formalizable requirements. Combining these definitions, validation appears composed of two parts: verification and evaluation [Laurent 92]. This separation is induced by the existence of two types of requirements, differentiated by a syntactical criterion. In this way, properties that can be objectively checked (by verification) are differentiated from the other aspects that require subjective assessment (by evaluation)[5]. With respect to testing, it is currently the most important technique for assessing ES performance.

In practice, the difference between totally and partially formalizable requirements, or what is the same, between verification and evaluation processes is not always totally definite. With the exception of domain-independent requirements, that are always perfectly defined in formal terms since they are originated by properties of the knowledge representation language, a few user requirements are totally formalizable. Some requirements are of a clear formalizable nature, but some experimentation is required to obtain the right specifications. For instance, the domain-dependent, totally formalizable requirement explained in section 3.1.2,

> "Typical bacterial pneumonia and atypical pneumonia cannot be considered with high evidence for the same patient"

generates four alternative specifications for a MILORD-based ES,

1. (`bact` ≤ *quite-possible*) and (`atip` ≤ *quite-possible*)
2. (`bact` ≤ *very-possible*) and (`atip` ≤ *quite-possible*)
3. (`bact` ≤ *quite-possible*) and (`atip` ≤ *very-possible*)
4. (`bact` ≤ *very-possible*) and (`atip` ≤ *very-possible*)

where the facts `bact` and `atip` represent the concepts of typical bacterial pneumonia and atypical pneumonia respectively. These specifications are not independent, since some are included in others. Then, what is the right specification set for the previous requirement?. This problem occurred in the verification of PNEUMON-IA. We tested the four specifications, computing the situations in which these specifications were violated. From these situations, we realized that the specification (1) was too restrictive while (4) was too loose. Specifications (2) and (3) reflected properly the desired behavior, so we selected them as the right specification set for the mentioned requirement. This example shows that boundaries between requirements in practice are not as crisp as they can be conceived in theory. Some interpretation is almost always required when checking requirements.

In the mid-term future it seems reasonable that partially formalizable requirements will evolve into a more formalizable ones. This is based on the short history of ES verification.

---

[5] Other authors, like [O'Keefe et al, 87] and [Grogono et al, 92], assign different meanings to these terms. They consider evaluation as the most global term, that includes validation as checking the correctness of ES behavior. These terminological differences are due to the early development stage of the field.

Early verifiers saw ESs as classical logical systems, where only logical issues were tested. Currently, verifiers consider ESs as complex software programs that have to be analyzed taking into account the operational semantics of the knowledge representation language, which is only partially based on logic. Verification issues do not only consider properties of the knowledge representation, but also include properties of knowledge. This evolution will continue in the future, allowing for a more accurate and exhaustive checking of requirements.

Finally, we can view ES validation into the wider framework of software quality assurance (SQA). SQA is concerned not only with assuring the compliance of a program with the user requirements at the present, but also with those aspects that influence how well the software will continue to satisfy the user needs in the future [Rushby 88b]. Software quality is analyzed in [Boehm et al, 78]. A hierarchy of software attributes is proposed in [Adrion et al, 82], including the concepts of reliability, adequacy, testability, understandability, measurability, usability, efficiency, transportability and maintainability. Very little work has been made on these topics considering ESs [Barrett 90]. It seems reasonable to think that these topics will be addressed in the future for ESs, as a result of the consolidation of ES technology.

3.1.6 Relation with Software Engineering

Previous definitions for validation, verification, testing and evaluation are in compliance with the essential meaning for these terms in software engineering. Definitions of validation and testing are the same for ESs and for conventional software. The definition of verification for conventional software as "the demonstration of the consistency, completeness and correctness of the software at each stage and between each stage of the development life cycle" [Adrion et al, 82], is based on the notion of specification, that determines the correct pattern at each stage. This definition fits pretty well the one we give for ES verification as the checking the compliance of an ES against the specifications coming from its formalizable requirements. The definition of evaluation in software engineering has always an aspect of subjective assessment. This aspect fits the subjective interpretation of the satisfaction of partially formalizable requirements, involved in the definition of evaluation for ESs.

Regarding validation methods, a major difference exists between software engineering and ESs. In software engineering it is assumed, at least theoretically, that validation can be obtained by verification of successive software development stages [Adrion et al, 82]. This approach does not hold for ESs, because of the impossibility of a complete translation of user requirements into specifications [Hoppe & Meseguer 91].

In summary, previous definitions are in compliance with corresponding terms in software engineering, capturing their essential meanings. The stated difference about the complete achievement of validation by verification, is caused by the type of user requirements for ESs, but this difference is not fundamental. In this sense, the previous definitions fit pretty well in a general framework of software validation, where different kinds of software coexist with a common understanding of the kernel issues.

3.2 What Should be Validated?

The practical application of ES validation can be seen from two points of view:

1. An ES is composed of several parts (KB, IE, etc.). ES validation consists in the validation of ES components plus the validation the relationships among these components.

2. An ES is the result of a knowledge engineering process with several phases. A set of intermediate results are produced along these phases. The validation of an ES consists in the validation of each phase, by validating the intermediate results produced.

Both approaches are complementary and non-exclusive. Approach (1) is effective in the sense that decomposes the validation of a complex object into the validation of its parts. Approach (2) includes the idea of validation by construction. Validating intermediate results demands the existence of requirements for the knowledge engineering phases. These requirements would be originated from the decomposition of high-level user requirements. Given the early development stage of user requirements in ESs, this decomposition is only partial. In addition, the kind of products generated at each phase is not yet consolidated, as a result of the evolution of knowledge engineering techniques. Nevertheless, validation cannot be delayed until the final ES stage. The cost of correcting an error escalates as the development advances[6], so correcting an error at the very end of the development can be very expensive. Therefore, checking a number of properties enforcing validation on these intermediate results is almost mandatory, in order to assure the effective validation of the final system.

[6] Just as it happens in software engineering.

With these ideas in mind, we identifiy the following set of processes and ES parts on which validation has to be present: user requirements, knowledge acquisition, ES architecture, KB structure and contents, inference engine and ES behavior. This list does not aim to be an exhaustive enumeration of all the potential validation aspects, but a set of validation targets common to any ES. Regarding the different procedural elements composing the ES, all of them can be validated using software engineering techniques and they are not analyzed here. As an exception, we consider the inference engine because of its key role in the ES function. In the following, we discuss each one of these validation targets.

3.2.1 User Requirements

Given that ES validation is performed against user requirements, it seems quite reasonable to validate them prior to any use, assessing their internal consistency and completeness. Service and competence requirements have to be clearly differentiated, as well as minimum and desirable competence requirements. Totally formalizable requirements have to be completely decomposed into specifications, while for partially formalizable requirements this decomposition is only partial and in some cases tentative. The subjective aspects of partially formalizable requirements have to be clearly identified.

3.2.2 Knowledge Acquisition

Knowledge acquisition (KA) is defined as the elicitation and analysis of data on expertise with the aim of building an ES [Breuker & Wielinga 87]. KA activities are developed during the initial stages of ES development, aiming to obtain a conceptual picture of the task to be performed by the ES. This picture has to be detailed and complete, since it will be the basis for ES implementation. The elusive nature of human expertise causes many difficulties to capture it, to the extent that KA has been considered as a bottleneck for knowledge engineering (for an analysis of human expertise see [Gaines 87]). Many different KA techniques exists, all of them sharing an intense interaction with a human expert in order to extract his/her expertise. Among these techniques, we can mention focussed interviews, structured interviews, instrospection, self-report, expert-user dialoges and reviews. See [Hart 86] for an overview of these techniques. A number of tools supporting automated KA exist, see [Buchanan et al, 83] [Anjewierden 87] for further details.

Some authors consider knowledge engineering as a modelling activity. In this view, an ES is not a container filled with extracted knowledge, but a model that behaves in an specific way analogous to a system in the real world. To build this model we have to set a mapping from the data extracted from the expert into a given representation formalism. But in addition to this *transformation*, the assessment of data relevance and data structuration has to be made by *abstraction* [Breuker & Wielinga 87]. The knowledge acquisition process is very related to abstraction, since it is a fundamental method to conceptualize and structure complex domains. In this view, an important result of knowledge acquisition is a conceptual model of the intended task for the ES, expressed in some language.

KA is at the basis of knowledge engineering. KA results have a determinant impact in the further stages of ES development. In addition, the conceptual model of the ES task will be used for validation purposes on the implemented ES. Therefore, validation of KA appears as a very important aspect in the global set of validation activities. Validation of KA has two main parts: assuring the quality of KA techniques used in the knowledge engineering process, and checking the adequacy and completeness of the conceptual model conceived. Adequacy is concerned with the correction, grain-size description, homogeneity and integrity of the conceptual model. Completeness considers to which extent teh conceptual model captures the whole intended task.

A related point is the acquisition of knowledge for validation, mentioned in section 3.1.2. It is increasingly apparent that verification of knowledge properties demands more knowledge than the minimum required to perform correct deductions. This extra knowledge has to be obtained in the KA phase, requesting the expert for justifications or causal explanation of his/her behavior. The knowledge for validation has to be included in the conceptual model of the ES, completing and enhancing the conceptual model required for pure deduction.

3.2.3 Expert System Architecture

An ES is a complex piece of software that contains different parts such as the knowledge base, the inference engine, the user-interface, explanation capabilities, input-output facilities, and others that are application-dependent. ES validation requires the validation of each part in isolation plus the validation of the interrelations among these parts. This problem exists in software engineering when a complex program composed of different modules is validated. Except the KB, all the mentioned ES parts are conventional procedures and their relations can be validated using software engineering techniques. In the following, we will concentrate on the validation of the KB architecture.

A previous step to the validation of KB architecture is the evaluation of the representation capabilities offered by the knowledge representation language with respect to the problem to be solved. If the language does not provide facilities to represent the different aspects of the domain knowledge, the resulting KB is unlikely to be adequate. As an example of this kind of inadequacy, consider a problem dealing with uncertain or imprecise information and a representation language without uncertainty management.

A KB is no longer a bunch of rules without structure. On the contrary, a KB has an internal structure induced by the characteristics of the problem domain. Different types of knowledge are expressed by different elements of the representation. The variety of KB objects (facts, goals, rules, rule sets, metarules and others) and its internal organization are the actual expression of the KB architecture (see section 4.1 for a specific example). The KB has to be designed before rule coding, to assure that its organization is adequate for the problem to be solved and not the result of an unordered addition of KB objects. Assuring the adequacy of the KB architecture is a main concern in the validation of the overall ES architecture. The conceptual model of the ES task obtained during the knowledge acquisition phase acts as the basic support for the validation of the KB architecture.

3.2.4 Knowledge Base Structure and Contents

The KB is a central part of the ES It contains the knowledge of the system coded in some representation language (frequently production rules). The KB is formed by a set of declarations that are interpreted by a fixed procedure, called inference engine, that carries out the operational semantics of the representation language. The KB plays a fundamental role in the ES function since all the actions performed by the system have their origin in the interpretation of the KB contents. Therefore, the validation of KB is a essential step in the global ES validation.

Regarding KB validation, two main aspects exist: validation of KB structure and validation of KB contents. Validation of KB structure is concerned with checking a set of properties of the knowledge representation language on the KB objects. These properties have a double aim. First, to guarantee that no ES malfunctions will exist during the interpretation process performed by the inference engine. And second, to detect those KB objects that can appear as anomalous, although they do not cause any malfunction. So far, validation of KB structure is the most developed subfield of ES validation, with a significant a number of techniques available.

Validation of KB contents considers the adequacy of the KB objects with respect to the knowledge they are supposed to represent. Adequacy includes correctness, consistency and completeness, and it requires checking knowledge properties. Validation of KB contents can be two-fold. First, validation of each single KB object, in the sense that it is a proper representation of a piece of knowledge. This validation is relatively easy to perform, since it implies the analysis of simple elements (typically condition-action pairs). Second, validation of those sets of KB objects that can interact in a ES execution, producing a global effect. The number of these sets is very high (consider for instance the potential number of rule chains in a KB of some hundreds of rules). For this reason, this second aspect of validation is quite difficult in practice.

Validation of KB contents cannot be performed in a complete form without a conceptual model of the knowledge involved in the KB. This conceptual model acts as a framework that supports, integrates and gives sense to the different validation activities, that can be performed regarding the KB contents. Without this model, different aspects can be validated in isolation but a global validation of KB contents will not be reached. The role of models in ES validation is analyzed in [Bellman 90]. She mentions that ES task models act as the framework supporting ES design and implementation, so they should be used to support ES validation. Several models can be used for different aspects of the ES task (they are called minimodels). In this sense [Rushby 88b] points out that models bring internal coherence to collections of rules. He considers the explicit construction and scrutiny of models as an essential aspect for trustworthy ESs.

3.2.5 Inference Engine

The inference engine (IE) is the procedure that interprets the KB objects and executes the actions stated in them according to their operational semantics. Most of the effort in ES validation has been put on the declarative part, i.e. the KB, while the IE was usually assumed correct by default. However, the role of the IE is so important in the ES function that an ES cannot be considered validated without a prior validation of its IE.

Validation of the IE aims at checking that it implements correctly the operational semantics of its knowledge representation language. To perform IE validation, a precise and formal definition of the operational semantics of the considered knowledge representation language is required. This formal definition acts as the specification for the desired behavior of the IE. However, most of the knowledge representation languages do not have a formal definition for their operational semantics. This represents a serious drawback to validate an IE since, without a formal definition, IE validation can only be made by informal means. The absence of formal validation increases the probability of errors in the IE code. In addition, many IEs allows the user to program some IE aspects such as the conflict-set resolution strategy. This means that the user can modify the operational semantics of KB objects. If this occurs, the IE has to be revalidated as well as those KB objects that have been created assuming a different sematics.

The absence of formal definitions for the operational semantics of knowledge representation languages has other consequences for ES validation. Automatic verifiers analyze KB objects considering their operational semantics, in order to predict the ES behavior in a number of situations. If this semantics is not well-defined, the simulation intended by the verifier will be only approximate. This may cause that erroneous situations will be missed, at expenses of the verifier accuracy.

3.2.6 Expert System Behavior

Validation of the ES behavior consists in assessing to which extent the ES fulfils the end-user requirements for the intended task. While in previous sections, validation was focused on specific ES parts, now validation considers the ES as a whole. User requirements are again the key point for a true validation. User requirements are clearly application-dependent but they commonly refer to operation conditions, performance level, user interaction, explanation capabilities and acceptance criteria.

Operation conditions establish the requirements for a fully operational ES. Basic hardware and software for the ES, loading time, maximum requirements of time and memory for an execution, etc. are examples of these requirements. For safety-critical applications the operation conditions have to be carefully detailed and exhaustively checked, since a failure in the ES operation may lead to situations very expensive to recover (and this may be caused by a simple syntax error).

The performance level of an ES plays an important role in user requirements, since the ES is expected to perform at a human expert competence level. An accurate assessment of human expert competence is a difficult task that requires statistical measures and consensus functions among experts' opinions (see sections 2.2 and 2.6). For this reason, performance evaluation is a complex task always requiring human assistance. Performance evaluation can also include the assessment of a lower performance bound, corresponding to the minimal competence requirements described in section 3.1.1.2.

User interaction involves all the ways in which the user can interact with and use the ES. The communication with the ES is performed through the user interface, that has to be easy-to-use and adaptable to the operational environment. An important point is the quality of ES questions and answers, that should be precise and prevent possible misunderstandings. The ES execution can admit several options: providing data manually or from a file, storing the dialog for later examination, recording the fired rules for archival purposes, etc. A flexible set of execution options can enhance the ES usability. The explanation capability is an aspect of user interaction of particular importance in ESs. Users will not gain confidence in the system without a clear explanation of its reasoning line. This fact conveys a great relevance to the explanation quality.

Acceptance criteria are a number of conditions that the ES must satisfy. In the selection of acceptance criteria, we have to consider that no technique provides a complete validation. On the contrary, partial validation evidence is obtained from a variety of different techniques with complemented effects. Therefore, the acceptance criteria have to involve a representative combination of validation techniques focusing on different ES parts. The critical conditions for the ES function have also to be present in this selection.

3.3 How to Validate?

A set of techniques for ES validation have been developed in the last years (see chapter 2 for a detailed description). This set is far from being complete, since there are many validation aspects without an appropriate answer. However, these techniques can validate significant portions of current ESs and they can be of great help for knowledge engineers. Validation methods developed for software engineering can be imported into knowledge engineering, after their adaptation to ES peculiarities. AI techniques, and specially the ones coming from machine learning like KB refinement, can also be of great help for ES validation. In the following, the validation aspects considered in section 3.2 are revisited, identifying some of the available techniques that are suitable for them.

3.3.1 User Requirements

No specific technique has been developed to validate user requirements for ES. This problem exists in software engineering, so it seems reasonable to adapt software engineering techniques to the ES case. Reviews involving end-users, human experts and knowledge engineers can be the basic element for validation of these requirements.

3.3.2 Knowledge Acquisition

Little work has been devoted to develop validation techniques for KA. The work of [Benbasat & Dhaliwal, 89] provides an initial framework for KA validation. They define validation of KA as the degree of homomorphism between the representation system, i.e. the ES, and the system that it is supposed to represent, i.e. the expert. They differentiate three stages in the development of a KB: conceptual KB, elicited KB and implemented KB. They consider four types of validation: conceptual, elicitation, implementation and representational. Conceptual validation considers the quality of the modelling process that has generated the conceptual KB from the human expert. Elicitation validation addresses the completeness and correctness of the process of translating the conceptual KB into the elicited KB. Implementation validation considers the quality of the process that transforms the elicited KB into the implemented KB. Representational validation aims at matching the characteristics of the implemented ES with the human expert. For each validation type, they give a number of tests to assess different aspects with respect to the intended homomorphism. Regarding conceptual and elicitation validation, they suggest as specific validation techniques different forms of KB inspections and structured walkthroughs, involving source experts, independent experts, knowledge engineers and final users.

The ES development methodology determines, to a great extent, the style of the KA process and the kind of validation we can perform on it. Broadly speaking, we can differentiate two types of ES development methodologies: bottom-up and top-down. A bottom-up methodology develops an implemented system as soon as some domain data are structured and understood. Rapid prototyping is the paradigm of bottom-up methodologies. In this approach, the conceptual structuration and abstraction of the data extracted from the expert is relatively low and a global conceptual model is missing. KA validation is limited to assuring the quality of KA techniques, such as expert interviews, and an adequate implementation of the recorded knowledge. This second aspect is not easy to fulfil, given that available techniques such as KB inspections are of difficult applicability when a significant amount of knowledge is involved.

Top-down methodologies, like KADS methodology [Breuker & Wielinga 87], include the development of a conceptual model of the intended task prior to any implementation. This conceptual model is expressed in a modelling language, that provides a vocabulary in which the expertise can be described in a coherent way. In this approach, validation can be performed not only assuring the quality of expert interviews, but also and what is more important, on the conceptual model. Getting a true validation of the conceptual model would mean an step of immense value in knowledge engineering. In this case, the most ellusive validation issues, those involving deep knowledge aspects that are not properly captured by its representation, would be solved. Some preliminary results on validation of acquired knowledge using specific tools, as well as on validation of conceptual models can be found in [Shadbolt 91].

3.3.3 Expert System Architecture

As stated in section 3.2.3, validation of ES architecture can be done for all the ES components (except the KB) by using standard techniques of software engineering. Validating the internal KB architecture involves experts and knowledge engineers. Experts have to evaluate how well the KB design fits the overall structure of the problem domain. Knowledge engineers act as interfaces between experts and specific constructs of the KB. The conceptual model of the ES task is an important element to support this kind of validation. This model acts as a reference point for both experts and knowledge engineers. Without a conceptual model, differences of vocabulary and points of view between experts and knowledge engineers may cause a superficial validation of the KB architecture. In this case, undetected errors will appear later in the ES development, with a higher correction cost.

The expressive capacity of KB objects has a significant impact in this kind of validation. Advanced knowledge architectures, like generic tasks [Chandrasekaran 87] or component of expertise [Steels 90], provide high level constructs that are closer to the conceptual model expression and make easier its scrutiny and validation. On the other hand, modularity features in the knowledge representation language facilitate a correct KB design and allow to define some hierarchical relationships among KB objects that can be exploited for validation purposes [Sierra et al, 91].

3.3.4 Knowledge Base Structure and Contents

Validation of KB structure is performed by checking those requirements related with the knowledge representation language used. These requirements are usually formalizable and domain-independent, so validation of the KB structure is totally achieved by verification. Validating the KB structure demands exhaustive checking, only reachable by automatic verifiers. A significant number of verifiers is currently available for different ES models (see section 2.1), to the extent that this is the most developed subfield of ES validation. A verifier of the KB structure has to analyze KB objects considering their operational semantics [Evertsz 91].

Validation of KB contents implies checking knowledge properties, to assess the knowledge correctness, consistency and completeness. The set of knowledge properties to be tested has to convey a significant evidence about the adequacy of the KB contents. Selection of the knowledge properties has to be supported by the conceptual model of the intended ES task. Otherwise, only isolated knowledge properties will be tested without achieving an integrated revision of the encoded knowledge. The existence of a conceptual model is specially relevant when checking the KB completeness, in the sense that all the knowledge required for the intended task is contained in the KB. In the elicitation of the knowledge properties to be tested, new aspects of the knowledge contained in the KB are usually required. These aspects have not been explicited because they are not required for deduction, but they are needed for validation purposes. These new aspects have to be encoded in auxiliary representations (like integrity constraints). Techniques used in automatic verifiers can be adapted to check those knowledge properties that assess the validity degree of KB contents.

Validation of individual KB objects can be made by inspection of the KB using an independent expert. It provides evidence on the correctness of single KB objects, but does not give any proof of the correctness of the KB as a whole. Validation of sets of interacting KB objects always requires computer-supported tools, since manual checking is unfeasible due to the large number of sets involved. Inconsistency is a good example of this kind of validation, that has been extensibly considered in the literature (see section 2.1). Inconsistency is a serious error that demands important computational efforts for effective checking. On the other hand, procedures for checking inconsistency are relatively simple. Integrity constraints declaring those facts that are incompatible are the only extra knowledge required for inconsistency checking. Facilities for inconsistency checking are currently available in most of the existing automatic verifiers.

Validation of KB contents can be obtained by means of ES testing. Validation through testing is indirect, partial and incomplete. Validation is indirect because it can only be induced from the results of ES executions. Validation is partial because there is no guarantee that all the KB components have been used in the testing executions. Validation is incomplete because no specific knowledge properties are checked in ES executions, except for the functionality of some KB objects in some test cases. In spite of these drawbacks, testing is the most frequently used method to assess the validity of KB contents. This is due to the following causes: (i) testing is much easier and direct than using complex verifiers for knowledge properties (verifiers that have to be built specifically for a shell), (ii) conceptual models supporting the validation of KB contents are infrequently built, and (iii) developers have an strong inclination to validate KB contents by ES execution. This inclination is reinforced by the usual lack of written requirements for the ES, so the only way to check the correctness of its constituting parts is by comparing ES outputs with the opinion of human experts.

Validation of KB contents can be made using KB refinement techniques. KB refinement aims at improving the KB contents from a set of cases with known solutions. When some cases are treated incorrectly by the ES, the KB is modified to achieve a correct treatment in all cases. It is assumed that only minor KB changes are required. KB refinement is founded on ES testing, so it shares the testing drawbacks. However, automatic refinement tools able to detect errors and to suggest substantiated changes in the KB, represent a significant help for the practical assessment of KB validity.

3.3.5 Inference Engine

An IE is a conventional interpreter of a knowledge representation language. It can be validated using standard techniques of software engineering. Knowledge representation languages do not often provide a formal definition of their operational semantics (see section 3.2.5). This is an important difficulty to achieve a true IE validation. Facilities to link user programs to the IE also difficult its validation.

As any other interpreter, the IE has to be defined in terms of a set of elementary operations that actually implement its functionality. Interpreters of standard programming languages are defined using memory cells, stacks, assignments, increments, conditionals and branchings. Elementary operations for an IE can be condition satisfaction, logical operators, truth value assignment and others. No consolidated set of elementary operations to deal with knowledge representation languages exists. This lack is an extra difficulty since, prior to any IE validation, the set of elementary operations has to be defined. Methods and techniques used to validate interpreters of standard programming languages can be adapted for IE validation.

3.3.6 Expert System Behavior

Validation of ES behavior is usually made by testing, although in case of critical requirements other techniques such as exhaustive verification can be used. The testing process has to allow for an effective checking of user requirements. Simulated test cases can be used to check specific conditions, while real test cases are needed to assess ES performance. Test set composition and comparison with human behavior are two of the key issues in ES testing (see section 2.2 for further details).

Two testing techniques have an special relevance in ESs, Turing tests and field tests. A Turing test consists in the evaluation of the ES output mixed with recommendations of human experts for a set of given cases. A set of independent experts acts as evaluators without knowing which is the system and who are the humans. This test is very suitable for performance evaluation. A field test consists in the regular use of the ES in its target working environment. ES errors and users complains are recorded and solved. The ES is considered tested when user complains have ceased. This test is suitable for assessing aspects of the user interaction, overall utility, and regular performance. Both types of testing require important amounts of human effort.

3.4 When to Validate?

Paraphrasing [Adrion et al, 82], knowledge engineering is an exercise in problem solving. As with any problem-solving activity, determination of the validity of the solution is part of the process. Therefore, validation has to be explicitely included in knowledge engineering, that will not be considered finished until the produced ES has been validated.

Validation cannot be delayed until the final phases of ES development. At this stage, ES errors could be very expensive to correct. As we have stated in section 3.2, the cost of correction of a hidden error escalates as the development advances. Therefore, early validation is always recommended. A number of validation activities can be performed during the ES development. The position of the validation activities in the ES life-cycle is discussed in the following.

3.4.1 Validation in the Life-Cycle

Following the comparative study of ES life-cycle contained in the section 2.5, we propose an ES life-cycle composed of the following steps: requirements, knowledge acquisition, design, implementation and maintenance. The first four steps fit exactly with the first four steps of the development cycles proposed by [Buchanan et al, 83] or [Miller 89]. We add a fifth step, maintenance, that includes all the operations made on the ES after its release. Maintenance operations can be frequent in some applications (it has been said that a KB is never complete because new knowledge is always ready to be added[7]). Therefore, this step is needed for a comprehensive treatment of the different stages in the ES life-cycle. No specific step devoted to testing or evaluation exists; validation activities are included in each development step. The whole process is represented in figure 3-1.

We want to stress the importance of validation activities in the ES life-cycle in a two-fold way. First, the products originated in each development step are validated by specific activities. This enhances the quality of intermediate products and increases the validity of the final ES. Second, the presence of validation in each step reinforces the design for validation. If knowledge engineers are aware that validation is a part of their job, they will acquire knowledge for validation and they will include facilities for validation in the design and implementation phases. The final product should be *valid* and this has to be kept in mind throughout the ES development.

We notice that the validation of an ES is more that the validation of its KB. If in the implementation step procedural parts are developed, they should be verified following standard techniques of software engineering. In particular, this holds for the inference engine that should be validated, at least informally, if it is not certified. Changes in the maintenance phase can affect to any previous development step. To validate these changes, the validation activities corresponding to the affected steps should be repeated.

The proposed life-cycle does not imply necessarily a waterfall or rapid prototyping methodology. It can be used to develop a prototype, that can be later refined and expanded

[7] We share this opinion. After the release of PNEUMON-IA, a new etiological diagnosis called *Chlamidia Pneumoniae* was identified. To include this new knowledge, we have reorganized an existing module and we have created a new one.

Life-cycle step	Step Products	Validation Activity
Requirements	Service/Competence Totally/Partially Formalizable Specifications	Validation of User Requirements
Knowledge Acquisition	Conceptual Model	Validation of Knowledge Acquisition
Design	ES architecture	Validation of ES architecture
Implementation	Operational ES	Validation of KB Struct & Contents Validation of Procedural Parts Validation of ES Behavior
Maintenance	New operational ES	Any of the previous

Figure 3-1. Proposed ES life-cycle and corresponding validation activities.

to achieve a final system, like [Miller 89]. In this case, all the life-cycle steps have to be repeated (requirements, knowledge acquisition, etc.), although the implementation step will not start from scratch but from the previously developed prototype.

3.5 Summary

This chapter contains several ideas about the meaning of validation for ESs and on the forms to achieve it in practice. Some of these ideas are supported by the experience, others are proposals to improve the current state of validation. In the following, the main ideas exposed in this chapter are summarized:

- *Validation Terminology*: we define validation in terms of user requirements, just like in software engineering. User requirements for ESs are totally or partially formalizable, depending on their complete or partial decomposition in specifications. Validation is composed of two main parts, verification and evaluation, depending on the type of user requirements respectively checked. Verification aims at objectively checking ES properties, while evaluation considers those aspects that require subjective assessment. Testing is viewed as the main technique to assess ES performance. These definitions are in compliance with the corresponding ones in software engineering, forming a general framework of sofware validation.

- *Complete Validation*: KB validation is not synonymous for ES validation. To achieve a complete validation, all the ES parts have to be validated. Procedural parts can be validated using standard software engineering techniques. Special emphasis has to be devoted to the inference engine, because of its central role in the ES function. To validate declarative parts specific techniques for ESs are required.

- *Validation throughout the ES life-cycle*: an ES life-cycle is proposed, including maintenance as a separate step. Validation, as an element of knowledge engineering, has to be present in all the steps of the ES development. . Different validation activities can be performed at each step. This presence in the whole life cycle enforces validation by construction, improving the quality of the intermediate products. In addition, it stresses the design and implementation for validation.

- *Knowledge Validation*: all the knowledge extracted from the expert has to be validated. To effectively perform this validation, the knowledge has to be organized forming a conceptual model. This model has to contain all the dependencies and relations among the intended task elements, as well as the heuristics and processes used by the expert to perform this task. The accuracy and completeness of this model is a crucial point, since it acts as the supporting framework to validate the ES implementation. In order to validate knowledge properties, some extra knowledge is required. This extra knowledge is not used for deduction, it is just for validation. It is acquired and represented in the usual ways, and it is also subject to validation. Early validation activities in the ES life-cycle enforce the acquisition of this kind of knowledge.

Chapter 4

Verification using Extended Labels

As we have stated in the chapter 3, verification consists in checking the ES against the specifications obtained from its formalizable requirements. From now on, we assume the existence of these specifications and we concentrate our efforts on checking four classical verification issues: inconsistency, redundancy, circularity and useless objects. The first one can be considered domain-dependent since it depends on the integrity constraints to be tested, while the other three can be considered as domain-independent. This chapter considers the effective checking of these four verification issues in an ES model including uncertainty and control.

We cannot talk about verification or validation techniques in precise terms without a previous definition of the ES model on which these techniques will be applied. We have selected an ES model that includes explicit and implicit control representation and uncertainty management. This model is a faithful representation of the MILORD shell [Sierra 89], on which the medical applications PNEUMON-IA [Verdaguer 89] and RENOIR [Belmonte 90] have been developed. The motivation for this choice is a very practical one: the absence of techniques applicable to verify these applications. Available verification techniques usually assume a quite simple ES model, which does not include essential aspects of MILORD-based applications such as uncertainty or the explicit representation of control knowledge. If these techniques were applied, they would have a very partial and incomplete representation of the target application and the results would be neither accurate nor complete. As Evertsz states [Evertsz 91], ES verification should be made considering the procedural semantics of the knowledge representation language used, that is to say, considering the way in which KB objects will be used in practice. In consequence, the model assumed in verification has to be a faithful representation of the target ES.

On this ES model we analyze the four classical verification issues mentioned above: inconsistency, redundancy, circularity and useless objects. From this analysis we obtain new verification issues, caused by the splitting of the original issues when considered in an ES model with several layers of knowledge organized hierarchically. For instance, inconsistency does not only occur between rule chains concluding incompatible facts (classical case). Inconsistency also occurs between metarules concluding opposite control actions (inconsistency in the control knowledge), or between rules and metarules (inconsistency relating control and domain knowledge). The inclusion of uncertainty also modifies the definitions of these issues, that have been conceived in a boolean framework.

To solve these verification issues, we use ATMS constructs. Taking the work of Ginsberg in KB-REDUCER [Ginsberg 88] as starting point, we extend the concepts of labels and environments to incorporate more information, defining the concepts of extended-labels and extended environments. Using them, we formulate a constructive way to test the verification issues. The verifier IN-DEPTH II implements this verification method in an incremental way. It accepts a KB and a set of modified objects as input and performs the minimal verification tests to achieve a global correctness of the KB with respect to the modifications.

Using IN-DEPTH II we are verifying PNEUMON-IA, an ES for etiological diagnosis of community acquired pneumonias in adults during the first days of infection. PNEUMON-IA is composed of 600 rules, 100 metarules and 24 modules, covering 22 different etiologies. For its size and complexity, PNEUMON-IA is an ideal application to assess the practical effectivity of a verifier. In addition, the expert who developed it participates very actively in the verification, what is very important to interpret and correct the defects detected by the verifier. So far we have verified the 50% of PNEUMON-IA rules, finding a low number of errors. That could be expected, because PNEUMON-IA had been previously validated showing a performance level comparable to human experts [Verdaguer 89]. IN-DEPTH II has allowed us to identify some errors in the organization of knowledge, errors that would be overlooked without the usage of a verifier. After the verification process, the confidence level in PNEUMON-IA has increased by two reasons: (i) the low number of actual errors detected, what is indicative of a good KB structure, and (ii) the improvement of the KB by correcting the detected errors. We expect to complete the verification of PNEUMON-IA during the next months.

The structure of this chapter is as follows. Section 4.1 contains a description of the ES model considered for verification and refinement (chapter 5). Section 4.2 includes the definition of the four verification issues in the ES model. Section 4.3 comprises the concepts, operations and formal construction of the extended labels and extended environments. Section 4.4 contains the tests for the verification issues in terms of extended labels and extended environments. Section 4.5 presents an example to help in understanding these ideas. Section 4.6 includes the incremental verification approach and a brief description of the algorithms used in the verifier IN-DEPTH II. Section 4.7 describes our practical experience on verifying PNEUMON-IA. Section 4.8 encloses some conclusions, supported by our practical experience on verification.

4.1 The ES Model

We assume the following ES model: (i) based on rules, underlying propositional logic, (ii) with uncertainty management, (iii) with implicit and explicit control, and (iv) monotonic. A description of this model follows, detailing its objects and how they interact. Regarding control, we have made some choices to allow for an effective computation. The results obtained from this model remain essentially applicable to any other ES with a similar structure.

4.1.1 KB Objects

A *KB* is composed of a 5-tuple <*F*, *R*, *M*, *MR*, *IC*> where *F* is a set of facts, *R* is a set of rules, *M* is a set of modules, *MR* is a set of metarules and *IC* is a set of integrity constraints. A fact $f \in F$ represents an attribute of the problem domain. Facts are divided into *deducible* (or hypotheses) and *external* (or findings) depending on whether their values can be deduced or they must be provided as an input, forming the sets *DF* and *EF*. A syntactic criterion discriminates them: a deducible fact appears in the right-hand side of some rule, while an external fact only appears in the left-hand side of rules. Special facts called *goals* drive the deduction process.

Two kind of rules exist, *concluding rules* and *up-down rules*, forming the disjoint sets R_{cl} and R_{ud}, such that $R = R_{cl} \cup R_{ud}$. A concluding rule $r \in R_{cl}$ is formed by a conjunction of conditions on facts in its left-hand side (*lhs*), an assertion about the value of one fact in its right-hand side (*rhs*) and a *cv*. When *r* is fired, the concluding fact is asserted with a *cv* computed from the *cv*s of *lhs*(*r*) and *r*. An up-down rule $r \in R_{ud}$ is formed by a conjunction of conditions on facts in *lhs*(*r*) and an action to increase or decrease the *cv* of a fact *f* in *rhs*(*r*). Up-down rules concerning the fact *f* are always fired after rules concluding *f*. Rules are fired backwards. Each rule belongs to one module. A module $m \in M$ contains a collection of rules and one or several goals. Rules belonging to a module are intended to be enough to conclude the module goals, although rules can use facts deduced in modules different from their own module. A metarule $mr \in MR$ is formed by a conjunction of conditions on facts in its left-hand side, and an action in the the right-hand side. Two different types of actions are allowed: actions on modules and actions on the whole ES. The set *MR* is divided in two disjoint sets *MRM* and *MRS*, formed by metarules acting on modules and metarules acting on the whole ES, respectively. Metarules are fired forward as soon as their conditions are fulfilled.

An integrity constraint $ic \in IC$ is an expression involving certainty values of one or several facts, which should be satisfied by every deduction in order to avoid inconsistencies. Integrity constraints have no role in the ES execution, since no testing process is performed in real time to check whether they are satisfied. However, they contain essential information for consistency checking.

4.1.2 Uncertainty Management

The uncertainty management system assigns one certainty value *cv* to each fact, representing always positive evidence. A fact (*f*) and its opposite ($\neg f$) are represented separately and each has its own certainty value. Concluding rules and metarules have also certainty values associated. The certainty value of a concluding rule *r* represents the expert evidence for the fact asserted in *rhs*(*r*), assuming *lhs*(*r*) as true. The certainty value of a metarule *mr* represents the strength for the control action in *rhs*(*mr*), assuming *lhs*(*mr*) as true. When a concluding rule or metarule *x* is fired, the *cv* of the concluding fact in *rhs*(*x*) is computed from the *cv*s of facts in *lhs*(*x*) and the *cv* of *x*., using the functions *cv-conjunction* and *cv-modus-ponens*. The function *cv-conjunction* computes the certainty value of a conjunction of facts from the certainty value of each fact. It is used to obtain the certainty value of the whole left-hand side part of a concluding rule or metarule when it is going to be fired. The function *cv-modus-ponens* computes the certainty value associated to the consequent of a logical implication from the certainty values corresponding to the antecedent and the implication. It is used to obtain the certainty value of the right-hand side of a concluding rule or metarule when it is fired. There exists a *certainty threshold* τ, and all deductions with a *cv* less than τ are eliminated.

When a fact *f* can be concluded by more than one rule, all the possible deductions are tried. Then a parallel certainty combination occurs, combining the different certainty values obtained for *f* using the *cv-disjunction* function. The resulting value is assigned as the final *cv* for *f*.

4.1.3 Control Representation

The control is divided in implicit and explicit. Implicit control is coded as the conflict-set resolution criteria, embedded in the inference engine. Three criteria of decreasing importance have been considered: (i) select the most specific rule (ii) select the rule with highest *cv*, and (iii) select the first rule. These criteria establish a total order in R. A rule r is more specific than r' if $rhs(r) = rhs(r')$ and $lhs(r') \subset lhs(r)$. The most specificity criterion induces a mutual exclusion relation between those rules among which a relation of specificity holds. Let $r,r' \in R$, such that r is most specific than r'. If r has been fired, there is no point in firing r' because all the information contained in r' has already been used in r. So r' should never be fired. Conversely, if r' has been fired, it means that r had been tried but failed. Under the monotonicity assumption r cannot be fired later. Therefore, if r is more specific than r' (or vice versa), r and r' are mutually exclusive and cannot occur in the same deduction.

Explicit control is coded in metarules acting on modules (*MRM*) or on the whole ES (*MRS*). On modules two actions can be performed, *add m* or *remove m*, meaning that the module *m* is activated or inhibited for deduction. Activated/inhibited modules are added/removed to/from the active module list. A module can be added several times to the active module list, but once it has been removed, it cannot be entered again. At each time, the active module list contains those modules considered more adequate to contribute to the final solution. On the whole ES only the *stop* action can be made. Stopping metarules have priority for firing over metarules acting on modules, which have priority for firing over rules.

4.1.4 ES Function and Structure

When the ES starts, a metarule builds up an initial active module list. Then the following cycle starts. A module is selected from the active module list and it becomes the current module. Their goals are pursued using the rules contained in it. As soon as new facts are deduced, metarules are tested for firing. If a metarule acting on modules is fired, the active module list is updated. Metarules adding modules to the active module list can have a certainty value associated, as a measure of the metarule strength. Certainty values have a significant role in selecting the current module. When every goal in the current module has been tried (no matter whether it has been effectively deduced or it remains unknown), a new current module is selected and the cycle restarts. The ES stops when no more modules are available in the active module list, or a metarule stopping the ES is fired.

From the previous description it is clear that there is a knowledge hierarchy in the ES, composed by the control knowledge and the domain knowledge, acting the former on the latter. This knowledge hierarchy is translated into a KB object hierarchy, according to their capabilities in the ES model. The KB object hierarchy is formed by

Level 4: metarules acting on the ES
Level 3: metarules acting on modules
Level 2: modules containing goals and rules
Level 1: rules acting on facts
Level 0: facts

4.1.5 Verification Assumptions

In the verification work described in this chapter, we have made two simplifying assumptions on this ES model: we have neither considered up-down rules nor the parallel certainty combination performed by the *cf-disjunction* function. These simplifications allow us to decrease significantly the computational complexity of the verification process.

Verification is exhaustive, that is to say, all the possible rule chains are explored. Parallel certainty combination has a determinant impact in the number of rule chains. Let assume that there are n different rules concluding the fact f. Without parallel certainty combination, a lower bound of the number of rule chains to consider is n (in the trivial case when each rule is involved in a single rule chain). With parallel certainty combination, we have to consider all the combinations of these n rules, in groups of one, two, three, until n rules. That is to say,

$$\sum_{m=1}^{n} \frac{n!}{m!\,(n-m)!} = 2^n - 1$$

so the lower bound for rule chains to consider is 2^n - 1. This increment in the computational complexity is extremely high, specially if compared with the accuracy gain that would be obtained. The parallel certainty combination produces small changes in the certainty value of f (or not changes at all) depending on the certainty values to be aggregated.

The impact of up-down rules in complexity is not as important as the caused by the parallel certainty combination. Let us consider n concluding rules and k up-down rules for f. Assuming that the lower bound for rule chains caused by concluding rules is n, the inclusion of up-down rules increases this lower bound to nk, since for each rule chain there are k up-down rules potentially applicable. This cost is expensive for the accuracy gain obtained, since up-down rules perform small changes in the certainty of deduced facts. Typically they have a tuning effect, including minor elements (that by themselves would have neither caused nor prevented the deduction of a fact) into the final certainty for this fact.

Therefore, we have preferred to loose some accuracy exploring rule chains, that could require very high computational resources, for an easier and more efficient verification implementation. Given the practical usage of the verification results (see section 4.7, where we report the practical experience) we estimate that these simplifications have a minimum impact in the overall correctness of the verification method. In this chapter, given that no up-down rules are mentioned, concluding rules are simply called rules.

4.2 Verification Issues

We consider four classic verification issues: inconsistency, redundancy, circularity, and useless objects. A conceptual definition for each of them follows, definitions that become operational after their application on the assumed ES model. Following the hierarchy of levels present in the ES model, we analyze each issue in a twofold way: *intra-level*, considering the KB objects contained at each level in isolation, and *inter-level*, considering the KB objects contained in a level together with all the objects belonging to upper levels.

4.2.1 Inconsistency

A KB is inconsistent if *conflicting* situations among KB objects can be obtained from a *valid* set of input data. Typically, only situations in which an integrity constraint was violated were considered as conflicting and therefore causing inconsistency. However, in the multi-level ES model, new conflicting situations can occur besides integrity constraint violation. These new causes are (i) potential conflicts can happen at different levels, and (ii) conflicts can occur between levels, specifically between rules and metarules. A set of input data is valid when it reflects some situation that happens in the real world. Assuming that the integrity constraints contained in *IC* are a good model of the real world, a set of input data will be considered valid if it is consistent. Conflicting situations are the following:

<u>Intra-level</u>

I-1. At level 3: if after an action remove m, an action add m is performed, on the same module *m*.

I-2. At level 0: if two or more facts violating an integrity constraint are deduced together from a valid set of input data.

<u>Inter-level</u>

I-3. Between levels 3 and 1: if the firing of a metarule adding the module *m* prevents a rule contained in *m* to be fired.

I-4. Between levels 3 and 1: if the firing of a metarule removing the module *m* would cause a rule contained in *m* to be fired.

Clearly, these four types of conflicting situations focus on different aspects of the ES function. Type I-2 involves facts that can be provided to the user as output, so it can generate erroneous answers. Types I-1, I-3 and I-4 are more related with the internal structural consistency of the ES. Type I-1 reveals a conflict in the control knowledge guiding the solution search process through the modules, this means that the right solution can be missed. Types I-3 and I-4 show that a conflict exists between control knowledge and domain knowledge, situation that can affect the ES response.

4.2.2 Redundancy

A KB is redundant if it contains repeated or duplicated knowledge. Redundancy can either have no effect on the ES functionality (just affecting the computational efficiency) or influence some deductions especially when they are weighted with certainty values. In any case, repeated knowledge should be identified and removed from the KB. Redundancy between KB objects can occur in the following cases:

Intra-level

R-1. At level 4, 3 or 1: let $x, x' \in R$ or $x, x' \in MR$. Then, x' is redundant with x if (i) $rhs(x) = rhs(x')$, and (ii) whenever x' is fireable, x is also fireable with identical results.

R-2. At level 0: let $f, f' \in DF$, f' is redundant with f if in all the situations where f' is deduced, f is also deduced with the same certainty value.

Inter-level

R-3. Between levels 3 and 1: let $m \in M$, $mr \in MRM$, and $r \in R$, such that $r \in m$ and $rhs(mr) = add\ m$. Then, r is redundant with mr if whenever mr is fireable, r is also fireable.

In the R-1 case, the redundant object can be removed from the KB without losing information. In the R-2 case, the role of the redundant object in further deductions has to be analyzed, to see whether f' can be substituted by f. The R-3 case is not properly redundant, since the role of rules and metarules is different and they cannot supersede each other. We consider it as an anomaly that can be useful to detect. In this case, rule r can neither be removed (a rule chain could be broken), nor modified without considering the metarules adding m (r could be non-redundant with other metarules, different from mr, adding m).

4.2.3 Circularity

A KB is circular if it contains a cycle. A cycle exists if an object depends on itself. Different kind of cycles can exist, as a function of the dependencies among KB objects. Three different dependency relationships exist in the ES model. First, a rule (or metarule) r depends on facts f such that $f \in lhs(r)$. If $lhs(r)$ is not satisfied, r will not be fired. This dependency is represented by $r \leftarrow f$, reading *r requires f*. This dependency also exists between the fact $f' = rhs(r)$ and r, represented by $f' \leftarrow r$. Second, a rule r depends on the module m to which it belongs. If m is not activated, r will not be considered for firing. This dependency is represented by $r \Leftarrow m$, reading *r belongs to m*. And third, a module m depends on the metarules activating it. If no metarule mr such that $rhs(mr) = add\ m$ is fired, m will not be considered for deduction. This dependency is represented by $m \Lleftarrow mr$, reading *m is activated by mr*. Potential cycles are the following:

<u>Intra-level</u>

C-1. A fact f_1 depends on itself through a chain of rules and facts.

$$f_1 \leftarrow r_1 \leftarrow f_2 \leftarrow r_2 \leftarrow \dots \leftarrow r_n \leftarrow f_n = f_1 \qquad f_1, f_2, \dots, f_n \in DF \quad r_1, r_2, \dots, r_n \in R$$

<u>Inter-level</u>

C-2. A fact f depends on rules contained in different modules, and a metarule adding one of these modules depends on f.

$$f \leftarrow r \leftarrow f' \leftarrow r' \Leftarrow m' \Lleftarrow mr' \leftarrow f \qquad f, f' \in DF;\ r, r' \in R;\ m' \in M;\ mr' \in MRM$$

C-3. Two metarules mr, mr' adding respectively the modules m and m', depend on facts f' and f, deduced by rules contained in m' and m.

$$f \leftarrow r \Leftarrow m \Lleftarrow mr \leftarrow f' \leftarrow r' \Leftarrow m' \Lleftarrow mr' \leftarrow f \qquad f, f' \in DF;\ r, r' \in R;\ m, m' \in M;\ mr, mr' \in MRM$$

Only in case C-1 the ES can effectively loop because only rules are involved in the cycle. Cases C-2 and C-3, including dependencies $\Leftarrow$ and $\Lleftarrow$, do not cause the ES to loop because these dependencies do not generate any chaining. These cycles can prevent some KB parts to be considered by the inference engine, what can seriously affect the correctness of the ES output.

4.2.4 Useless Objects

A KB object is useless if it can never be used. Useless objects include non-fireable objects (rules or metarules), unreachable objects and shadowed objects. An object is non-fireable when it is supported by a non-valid set of input data. An object is unreachable when there exists a gap in the dependency graph linking the object with the inputs. An object is shadowed if there exist other objects that prevent it to be used. Potential cases for useless objects are the following:

Intra-level

U-1. At level 4, 3 or 1: a non-fireable metarule or rule.

U-2. At level 0: let $f \in F$, $r \in R$, f is unreachable if every r, such that $f \in rhs(r)$, is non-fireable.

Inter-level

U-3. Between levels 4 and 3: let $mr \in MRS$, $mr' \in MRM$, mr' is shadowed by mr if mr is always fired before mr'.

U-4. Between levels 4 and 1: let $mr \in MRS$, $r \in R$, r is shadowed by mr if mr is always fired before r.

U-5. Between levels 3 and 2: let $m \in M$, $mr \in MRM$, m is unreachable if every metarule mr, such that $rhs(mr)$ = *add m*, is either non-fireable or shadowed.

U-6. Between levels 2 and 1: let $m \in M$, $r \in R$, $r \in m$, r is unreachable if m is unreachable.

U-7. Between levels 1 and 0: let $f \in F$, $r \in R$, f is unreachable if every r, such that $f \in rhs(r)$, is either non-fireable, shadowed or unreachable.

4.3 Extended Labels and Extended Environments

deKleer introduced the concepts of label and environment for a deducible fact f in the ATMS context [deKleer 86]. An environment for f, $E_i(f)$, is a minimal conjunction of external facts supporting f. The label for f, $L(f)$, is the minimal disjunctive normal form of external facts supporting f. Clearly, $L(f)$ includes all the $E_i(f)$ as disjunctions. Ginsberg has used successfully labels and environments for verification purposes (inconsistency and redundancy checking [Ginsberg 88]). However, they do not contain all the required information to verify ESs with uncertainty and control features. Additional information is needed about (i) the *cv* range in which a fact can be deduced and (ii) the control actions required for a fact to be deduced.

To apply these concepts for verification purposes on our ES model, we have extended their definition in the following way. An *extended environment* (e-environment, for short) for a deducible fact f, $EE_i(f)$, is a triplet $< SS_i(f), RCV_i(f), RS_i(f) >$. $SS_i(f)$ is a minimal set of external facts supporting f, that is to say, an environment in deKleer or Ginsberg sense. $RCV_i(f)$ is the range of certainty values in which f can be deduced from $SS_i(f)$. $RCV_i(f)$ is represented by the interval $[LCV_i(f), UCV_i(f)]$, where $LCV_i(f)$ and $UCV_i(f)$ are respectively the lower and upper bounds. $RS_i(f)$ is the rule sequence connecting $SS_i(f)$ with f, and it is recorded for control reasons: (a) to identify e-environments that are incompatible with this one because of mutual exclusion between their corresponding rule sequences, and (b) to identify the set of metarules that have been eventually fired to activate the rules contained in the rule sequence. Two rule sequences RS_i, RS_j are *mutually exclusive* (m-exclusive for short) if there exist $r \in RS_i$ and $r' \in RS_j$ such that r and r' are mutually exclusive. Two rules are mutually exclusive if one is more specific than the other. The *extended label* (e-label for short) for a deducible fact f, $EL(f)$, is the minimal collection of e-environments for f.

The concepts of e-environment and e-label are not only applicable to deducible facts, but also to rules and metarules. An e-environment for x, $EE_j(x)$, $x \in R \cup MR$, is formed by the same components $< SS_j(x), RCV_j(x), RS_j(x) >$ with the same meanings: $SS_j(x)$ is the set of external facts causing r to be fired, $RCV_j(x)$ is the allowed range for the *cv* of the *rhs*(x), and $RS_j(x)$ is the rule sequence required for x to be fired. Similarly, the e-label for x, $EL(x)$ is the minimal collection of e-environments. E-environments and e-labels for m,

$m \in M$, are defined in terms of e-environments for metarules that introduce m in the active module list. Thus, $EL(m) = \cup EL(mr)$, $mr \in MR$, such that $rhs(mr) = add\ m$.

A number of relations hold between e-labels and e-environments. Let $\mathfrak{E}\mathfrak{E}$ be the set of e-environments, $EE_i, EE_j \in \mathfrak{E}\mathfrak{E}$. EE_i is *compatible* with EE_j iff (a) $SS_i \cup SS_j$ is consistent, and (b) RS_i is not m-exclusive with RS_j. EE_i *subsumes* EE_j iff (a) SS_i contains SS_j and (b) RS_i is not m-exclusive with RS_j. EE_i *includes* EE_j iff (a) EE_i subsumes EE_j and (b) RCV_i is contained in RCV_j. Let $\mathfrak{E}\mathfrak{L}$ be the set of e-labels, $EL, EL' \in \mathfrak{E}\mathfrak{L}$. EL is *compatible* with EL' iff there exists $EE_i \in EL$, $EE_j \in EL'$ such that EE_i is compatible with EE_j. EL is *fully compatible* with EL' iff for all $EE_i \in EL$ there exists a $EE_j \in EL'$ such that EE_i is compatible with EE_j. EL *partially subsumes* EL' iff there exist $EE_i \in EL$, $EE_j \in EL'$, such that EE_i subsumes EE_j. EL *totally subsumes* EL' iff for all $EE_i \in EL$ there exists a $EE_j \in EL'$, such that EE_i subsumes EE_j. EL *totally includes* EL' iff for all $EE_i \in EL$ there exists a $EE_j \in EL'$, such that EE_i includes EE_j.[1]

4.3.1 Operations

To effectively compute e-labels and e-environments for KB objects, we need the following operations: (1) a disjunction $\vee$ between e-labels, (2) a conjunction-1 $\wedge_1$ between e-labels or between e-environments, (3) a modus-ponens $\otimes$ between rules or metarules and e-labels or e-environments, and (4) a conjunction-2 $\wedge_2$ between e-labels or between e-environments. These operations allow to represent all potential steps that the ES can perform in terms of e-labels and e-environments. The definition for each operation follows. The same symbol is used to indicate the same operation on e-labels or e-environments.

1. *Disjunction*: models the different ways to conclude a fact. It is defined by,

   $\vee: \mathfrak{E}\mathfrak{L} \times \mathfrak{E}\mathfrak{L} \rightarrow \mathfrak{E}\mathfrak{L}$ $\qquad EL \vee EL' = \{EE_i \mid EE_i \in EL \text{ or } EE_i \in EL'\}$

   The disjunction between two e-labels is the union of the sets of e-environments belonging to each e-label.

2. *Conjunction-1*: models the computations performed at the left-hand side of a rule or metarule when it is going to be fired. It is defined by,

   $\wedge_1: \mathfrak{E}\mathfrak{L} \times \mathfrak{E}\mathfrak{L} \rightarrow \mathfrak{E}\mathfrak{L}$ $\qquad EL \wedge_1 EL' = \{EE_i \wedge_1 EE_j \mid EE_i \in EL, EE_j \in EL\}$

[1]This terminology slightly differs from [Meseguer 91] to better fit the intuitive meaning of inclusion.

$\wedge_1: \mathfrak{EE} \times \mathfrak{EE} \rightarrow \mathfrak{EE}$ $EE_i \wedge_1 EE_j = EE_k$, defined by

SS_k	$= SS_i \cup SS_j$	if consistent
	empty	otherwise
RCV_k	$= [LCV_k, UCV_k]$	if $UCV_k > \tau$
	empty	otherwise
RS_k	$= RS_i \cup RS_j$	if not m-exclusive
	empty	otherwise

where $LCV_k = \textit{cv-conjunction}\ (LCV_i, LCV_j)$ and $UCV_k = \textit{cv-conjunction}\ (UCV_i, UCV_j)$. When any of the parts is *empty*, the resulting environment is empty. Assuming that the the left-hand side of a rule r is the conjunction of the facts x and y, $EE_i\ (x) \wedge_1 EE_j(y)$ models the computation performed to check if $lhs(r)$ is satisfied: (a) if the union of their support sets is consistent, (b) if the *cv-conjunction* of their certainty values is greater than the threshold τ (only upper bounds are considered), and (c) if their respective rule sequences are not mutually exclusive.

3. *Modus ponens*: models the computations performed when a rule or metarule is fired, assuming its left-hand side is satisfied. It is defined by,

$\otimes: \mathfrak{R} \times \mathfrak{EL} \rightarrow \mathfrak{EL}$ $x \otimes EL = \{ x \otimes EE_i \mid EE_i \in EL \}$

$\otimes: \mathfrak{R} \times \mathfrak{EE} \rightarrow \mathfrak{EE}$ $x \otimes EE_i = EE_k$, defined by

SS_k	$= SS_i$		
RCV_k	$= [LCV_k, UCV_k]$		
RS_k	$= RS_i$	$x \in MR$	
	$RS_i \cup \{x\}$	$x \in R;\ x, RS_i$	not m-exclusive
	empty	$x \in R;\ x, RS_i$	m-exclusive

where $\mathfrak{R} = R \cup MR$, and $LCV_k = \textit{cv-modus-ponens}(LCV_i, cv(x))$, $UCV_k = \textit{cv-modus-ponens}(UCV_i, cv(x))$. Assuming that EE_i is an e-environment satisfying $lhs(x)$ (x stands for a rule or a metarule), $x \otimes EE_i$ models the action of firing x: (a) if x is a rule it should be not mutually exclusive with the rule sequence required to satisfy its left-hand side, and (b) if it is fired, the certainty value of $rhs(x)$ is obtained from the certainty values of $lhs(x)$ and x itself, using the function *cv-modus-ponens*.

4. *Conjunction-2*: models the relations that should exist between a rule *r* belonging to a module *m* and a metarule *mr* activating *m*, to allow *r* to be fired. It is defined by,

$\wedge_2: \mathbb{EL} \times \mathbb{EL} \rightarrow \mathbb{EL}$ $\qquad EL \wedge_2 EL' = \{EE_i \wedge_2 EE_j \mid EE_i \in EL,\ EE_j \in EL\}$

$\wedge_2: \mathbb{EE} \times \mathbb{EE} \rightarrow \mathbb{EE}$ $\qquad EE_i \wedge_2 EE_j = EE_k$, defined by

$$\begin{array}{lll} SS_k & = SS_i \cup SS_j & \text{if consistent} \\ & \quad empty & \text{otherwise} \\ RCV_k & = RCV_i & \\ RS_k & = RS_i \cup RS_j & \text{if not m-exclusive} \\ & \quad empty & \text{otherwise} \end{array}$$

Assuming *r* and *mr* as above, $EE_i(r) \wedge_2 EE_j(mr)$ models the conditions for *r* to be fired: (a) the union of their support sets should be consistent and (b) their rule sequences should be not mutually exclusive. The metarule *mr* has no action on the certainty value range of the rule. Obviously $\wedge_2$ is not conmutative, e-environments for rules must appear on its left, while e-environments for metarules must appear on its right.

As an example of the previous definitions, let *EL* and *EL'* be e-labels defined by,

$$\begin{array}{ll} EL = \{EE_1, EE_2\} & EL' = \{EE_3, EE_4\} \\ EE_1 = \{(a\ b)[0,1]\ (r1\ r2\ )\} & EE_2 = \{(c\ d)[0,0.4]\ (r3)\} \\ EE_3 = \{(e\ f)[0,0.4]\ (r4)\} & EE_4 = \{(\neg a\ c)[0,0.6]\ (r5)\} \end{array}$$

with numeric certainty values, from 0 to 1, *cv-conjunction* and *cv-modus-ponens* are the product of reals, threshold $\tau = 0.2$, and rules *r3* and *r5* are mutually exclusive. Thus,

$$EL \vee EL' = \{EE_1, EE_2, EE_3, EE_4\}$$

$$\begin{aligned} EL \wedge_1 EL' &= \{EE_1 \wedge_1 EE_3,\ EE_1 \wedge_1 EE_4,\ EE_2 \wedge_1 EE_3,\ EE_2 \wedge_1 EE_4\} = \\ &= \{EE_1 \wedge_1 EE_3\} = \{(a\ b\ e\ f)[0,0.4]\ (r1\ r2\ r4\ )\ \} \end{aligned}$$

$$\begin{aligned} EL \wedge_2 EL' &= \{EE_1 \wedge_2 EE_3,\ EE_1 \wedge_2 EE_4,\ EE_2 \wedge_2 EE_3,\ EE_2 \wedge_2 EE_4\} = \\ &= \{EE_1 \wedge_2 EE_3,\ EE_2 \wedge_2 EE_3\} = \\ &= \{(a\ b\ e\ f)[0,1](r1\ r2\ r4\ ),\quad (c\ d\ e\ f)[0,0.4]\,(r3\ r4)\} \end{aligned}$$

$$r5 \otimes EL = \{r5 \otimes EE_1,\ r5 \otimes EE_2\ \} = \{(a\ b)[0,1]\ (r1\ r2\ r5\ )\}$$

In $EL \wedge_1 EL'$ only $EE_1 \wedge_1 EE_3$ is not empty. The rest of e-environments are empty because of the following reasons: in $EE_1 \wedge_1 EE_4$ the union of support sets is not consistent, in $EE_2 \wedge_1 EE_3$ the threshold is not surpassed, and in $EE_2 \wedge_1 EE_4$ the rule sequences are mutually exclusive. Similar situations occur in $EL \wedge_2 EL'$, except for the e-environment $EE_2 \wedge_2 EE_3$ which is not empty because there is no condition on certainty in $\wedge_2$. Finally, in $r5 \otimes EL$ the e-environment $r5 \otimes EE_2$ is empty because $r3$ and $r5$ are mutually exclusive.

4.3.2 Computing e-labels

Using the operations defined above, the e-labels for the KB objects are the following:

$f \in EF$ $\quad EL(f) = \{EE_0(f)\}$ (1)

$f \in DF$ $\quad EL(f) = \bigvee_{r \in R,\, f \leftarrow r} EL(r)$ (2)

$r \in R,\ r \Leftarrow m$ $\quad EL(r) = r \otimes \left(\bigwedge_{1\ f \in F,\, r \leftarrow f} EL(f) \right) \wedge_2 EL(m)$ (3)

$mr \in MR$ $\quad EL(mr) = mr \otimes \left(\bigwedge_{1\ f \in F,\, mr \leftarrow f} EL(f) \right)$ (4)

$m \in M$ $\quad EL(m) = \bigvee_{mr \in MR,\, m \Leftarrow\equiv mr} EL(mr)$ (5)

The meaning of these expressions is straightforward. In (1) the e-label of an external fact f is composed by only one e-environment $EE_0(f)$, the terminal environment, trivially built. In (2) the e-label of a deducible fact f is the disjunction of the e-labels of the rules concluding f. In (3) the e-label of a rule r is obtained in three steps: (a) conjunction-1 of the e-labels of facts appearing in the left-hand side of r, (b) modus ponens with r, and (c) conjunction-2 with the e-label of m, the module to which r belongs. The justification of each step is clear: step (a) requires the left-hand side of r to be satisfied, computing the certainty value of the whole premise (that should be greater than the threshold τ), and checking that no mutually exclusive rules have been used to deduce facts in the left-hand side, step (b) adds the rule r in the computation, including its certainty value and checking again r for mutual exclusion with those rules previously used, and step (c) establishes that r can only be fired when the explicit control has activated the module containing r. In expression (4) the e-label of a metarule is computed like the e-label of rule that is not included in any module. Finally, in (5) the e-label of a module is the disjunction of the e-labels of the metarules activating m. Expressions (1) to (5) allow the e-label computation for any KB object.

The information contained in an e-environment *EE(f)* has been generated by the action of (a) the explicit control knowledge and (b) the domain knowledge. These actions are disjoint, as they are the constructs representing them (rules vs metarules). So we can distinguish, for each piece of information contained in *EE(f)*, whether it has been generated by the action of domain or control knowledge. According to this *EE(f)* can be decomposed into an e-environment restricted to the domain $EE(f)|_d$, and an e-environment restricted to the control $EE(f)|_c$, related by the operator $\wedge_2$,

$$EE(f) = EE(f)|_d \wedge_2 EE(f)|_c \tag{6}$$

The main motivation for computing these components is because they are adequate to check a number of verification issues, specially those in which a rule is tested against the metarules that add or remove its module. To compute each of these components, $EE(f_i)$ is expressed in terms of terminal environments. Using (2) and (3)

$$EE(f_i) = r_i \otimes \{ \underset{r_i \leftarrow f_j}{\wedge_1} r_j \otimes [\underset{r_j \leftarrow f_k}{\wedge_1} r_k \otimes \ldots r_m \otimes (\underset{r_m \leftarrow f_n}{\wedge_1} EE_0(f_n))$$

$$\wedge_2 EE(m_m) \ldots \wedge_2 EE(m_k)] \wedge_2 EE(m_j)\} \wedge_2 EE(m_i)$$

where, for all i, it is understood that r_i is the rule concluding f_i, and m_i is the module containing r_i. It is easy to see that these operations verify the following properties,

$$\underset{i}{\wedge_1}(EE_i \wedge_2 EE_i') = (\underset{i}{\wedge_1} EE_i) \wedge_2 (\underset{i}{\wedge_2} EE_i')$$

$$r \otimes (EE \wedge_2 EE') = (r \otimes EE) \wedge_2 EE' \tag{7}$$

Using (7) all the module e-environments can be shifted to the right, and

$$EE(f_i) = < r_i \otimes \{ \underset{r_i \leftarrow f_j}{\wedge_1} r_j \otimes [\underset{r_j \leftarrow f_k}{\wedge_1} r_k \otimes \ldots r_m \otimes (\underset{r_m \leftarrow f_n}{\wedge_1} EE0(f_n))] \} > \wedge_2$$

$$< \{ \underset{r_i \leftarrow f_j}{\wedge_2} [\underset{r_j \leftarrow f_k}{\wedge_2} \ldots (\underset{r_m \leftarrow f_n}{\wedge_2} EE(m_m)) \ldots EE(m_k)] \, EE(m_j) \} \wedge_2 EE(m_i) >$$

expression in which all the information corresponding to the domain knowledge appears in the first part, while the second part contains the e-environments corresponding to modules.

Therefore, they constitute the expressions for the restricted environments $EE(f_i)|_d$ $EE(f_i)|_c$, expressions that can be rewritten as,

$$EE(f_i)|_d = r_i \otimes \underset{r_i <- f_j}{\wedge_1} EE(f_j)|_d \tag{8}$$

$$EE(f_i)|_c = \underset{r_i \in RS(f_i)|_d, r_i <= m_i}{\wedge_2} EE(m_i) \tag{9}$$

From expression (8) e-labels and e-environments restricted to domain knowledge can be defined for any ES object as follows,

$$f \in EF \qquad EL(f)|_d = \{EE_0(f)\} \tag{10}$$

$$f \in DF \qquad EL(f)|_d = \underset{r \in R,\, f <- r}{\vee} EL(r)|_d \tag{11}$$

$$r \in R, r \in m \qquad EL(r)|_d = r \otimes (\underset{f \in F,\, r <- f}{\wedge_1} EL(f)|_d) \tag{12}$$

$$mr \in MR \qquad EL(mr)|_d = mr \otimes (\underset{f \in F,\, mr <- f}{\wedge_1} EL(f)|_d) \tag{13}$$

$$m \in M \qquad EL(m)|_d = \underset{mr \in MR,\, m <\equiv mr}{\vee} EL(mr)|_d \tag{14}$$

The e-environments restricted to the control knowledge depend on the modules to which belong the rules contained in the rule sequence of each e-environment. This set of modules can vary among e-environments belonging to the same e-label. For this reason is not possible to define e-labels restricted to control knowledge, and the most general expression for e-environments restricted to control knowledge is the following,

$$EE(x)|_c = \underset{r_i \in RS(x)|_d, r_i <= m_i}{\wedge_2} EE(m_i), \qquad x \in F \cup R \cup MR \cup M \tag{15}$$

It is worth noticing that, to compute all e-labels, is enough to compute all the e-labels restricted to domain knowledge. In (15) e-environments restricted to control knowledge depend on e-environments of modules. These e-environments can also be decomposed into domain and control parts, and control parts are subsequently decomposed until modules with empty control parts (not depending on other modules) are reached.

4.4 Solving Verification Issues

From the definition of the verification issues and e-labels, we can easily express each issue in terms of e-label relations. Using the constructive definitions of e-labels, we obtain a effective method to check the verification issues. The precise checking for each issue is as follows:

Inconsistency

I-1. Let $m \in M$ and mr, $mr' \in MRM$, such that $rhs(mr) = add\ m$ and $rhs(mr') = remove\ m$. There exists inconsistency if $EL(mr')$ partially subsumes $EL(mr)$.

I-2. Let $f, f' \in F$, $ic \in IC$, such that $f, f' \in ic$. There exists inconsistency if there exist $EE_i(f) \in EL(f)$ and $EE_j(f') \in EL(f')$ such that they are compatible and ic is violated in $RCV_i(f)$ and $RCV_j(f')$.

I-3. Let $m \in M$, $mr \in MRM$ and $r \in R$, such that $rhs(mr) = add\ m$ and $r \in m$. There exists inconsistency if $EL(mr)$ is not fully compatible with $EL(r)$.

I-4. Let $m \in M$, $mr \in MRM$ and $r \in R$, such that $rhs(mr) = remove\ m$ and $r \in m$. There exists inconsistency if $EL(r)|_d$ partially subsumes $EL(mr)$.

Redundancy

R-1. Let $x, x' \in R$ or $x, x' \in MR$, such that $rhs(x) = rhs(x')$. Then, x' is redundant with x if $EL(x')$ totally includes $EL(x)$.

R-2. Let $f, f' \in DF$. Then, f' is redundant with f if $EL(f')$ totally includes $EL(f)$.

R-3. Let $m \in M$, $mr \in MRM$ and $r \in R$, such that $r \in m$ and $rhs(mr) = add\ m$. Then, r is redundant with mr if $EL(mr)$ totally subsumes $EL(r)|_d$.

In addition to these tests, we found convenient to test situations that can be close to redundancy. One of these situations occurs between two rule chains deducing the same fact f, when the conditions of one rule chain are a subset of the conditions of the other. No restrictions are imposed on the certainty values of f reachable by both rule chains. We have called this situation R-4, but it is not an actual error pattern, since correct deductions can satisfy it. In terms of e-labels this situation is as follows, R-4. Let $f \in DF$, a potential redundancy may occurs when there exist $EE_i(f)$, $EE_j(f) \in EL(f)$ such that $EE_i(f)$ subsumes $EE_j(f)$.

Circularity

C-1. Let $r \in R$ and $m \in M$, such that $r \in m$. There is a cycle if there exists $EE_i(r)|_d \in EL(r)|_d$, such that $r \in RS_i(r)|_d$.

C-2. Let $r \in R$ and $m \in M$ such that $r \in m$. Let $EE_i(r)|_d \in EL(r)|_d$. Let $m_1,...,m_k$ be the modules to which rules in $RS_i(r)|_d$ belong. There is a cycle if there exists $EE_l(m_j)|_d \in EL(m_j)|_d$ such that $r \in RS_l(m_j)|_d$, $j = 1,...,k$.

C-3. Let $m \in M$ and $EE_i(m)|_d \in EL(m)|_d$. Let $m_1,...,m_k$ be the modules to which rules in $RS_i(m)|_d$ belong. There is a cycle if there exists $EE_l(m_j)|_d \in EL(m_j)|_d$ and $r \in m$, such that $r \in RS_l(m_j)|_d$, $j = 1,...,k$.

Useless objects

U-1. Let $x \in R \cup MR$. Then, x is non-fireable if $EL(x) = \emptyset$.

U-2. Let $f \in F$. Then, f is unreachable if $EL(f) = \emptyset$.

U-3. Let $mr \in MRS$ and $mr' \in MRM$. Then, mr' is shadowed by mr if $EL(mr')$ totally subsumes $EL(mr)$.

U-4. Let $mr \in MRS$ and $r \in R$. Then, r is shadowed by mr if if $EL(r)$ totally subsumes $EL(mr)$.

U-5. Let $m \in M$ and $mr \in MRM$, such that $rhs(mr) = add\ m$. Then, m is unreachable if every metarule mr is either non-fireable or shadowed.

U-6. Let $m \in M$ and $r \in R$, such that $r \in m$. Then, r is unreachable if m is unreachable.

U-7. Let $f \in F$ and $r \in R$, such that $f \in rhs(r)$. Then, f is unreachable if every r is either non-fireable, shadowed or unreachable.

From these expressions we can design a verification procedure with the following steps: (i) for all KB objects compute their e-labels restricted to the domain knowledge, detecting all circularities of type 1 (otherwise the procedure could loop itself), (ii) check circularities cases 2 and 3, (iii) detect all useless KB objects, removing them for further processing, (iv) check inconsistency, and (v) check redundancy.

The computational complexity of this method is, in the worst case, exponential with respect to the set of external facts. By worst case it is understood that all the possible combinations of external facts with all their possible values are present as different e-environments. But this is quite far from reality, given that a theory of these characteristics will lack the required degree of compactness to be learned by anyone in the first place [Ginsberg 88]. So, it is reasonable to expect an average case complexity that allows one to effectively check a rule base of several hundreds of rules (< 1000). The practical results obtained using IN-DEPTH II to verify PNEUMON-IA confirm this estimation (see section 4.7.3).

4.5 An Example

Let us consider the following rule base,

$$
\begin{aligned}
MRS &= \{mr0\} \\
MRM &= \{mr1, mr2, mr3, mr4, mr5, mr6, mr7, mr8\} \\
M &= \{m1, m2, m3\} \\
R &= \{r1, r2, r3, r4, r5, r6, r7, r8, r9, r10, r11, r12\} \\
EF &= \{a, b, c, d, e, f, g\} \\
DF &= \{p, q, r, s, t, v, w, x, y, z\} \\
IC &= \{\, cv(q) + cv(w) \leq 1\}
\end{aligned}
$$

$mr0)\ a, x, e \rightarrow stop$	$mr3)\ y \rightarrow add\ m2$	$mr6)\ b \rightarrow add\ m3$
$mr1)\ a \rightarrow add\ m1$	$mr4)\ w \rightarrow add\ m2$	$mr7)\ q \rightarrow add\ m3$
$mr2)\ \neg e \rightarrow add\ m1$	$mr5)\ a, e \rightarrow remove\ m2$	$mr8)\ x, f \rightarrow remove\ m3$
<u>*m1:*</u>	<u>*m2:*</u>	<u>*m3:*</u>
$r1)\ c, d \rightarrow x;\ cv{=}1$	$r5)\ x \rightarrow t;\ cv{=}1$	$r10)\ \neg a, y \rightarrow s;\ cv{=}1$
$r2)\ x, e \rightarrow y;\ cv{=}0.8$	$r6)\ p \rightarrow t;\ cv{=}1$	$r11)\ d, f \rightarrow v;\ cv{=}1$
$r3)\ c, d, e \rightarrow y;\ cv{=}0.8$	$r7)\ t \rightarrow p;\ cv{=}1$	$r12)\ f, r \rightarrow w;\ cv{=}0.8$
$r4)\ y \rightarrow z;\ cv{=}1$	$r8)\ t, v \rightarrow q;\ cv = 0.7$	
	$r9)\ z, \neg g \rightarrow r;\ cv = 0.9$	

External facts can take boolean values only. Certainty values are numeric, from 0 to 1, with the product as *cv-conjunction* and *cv-modus-ponens* functions, and threshold = 0.2. Computing e-labels restricted to domain knowledge a circularity C-1 is found, between *p* and *t*, involving the rules *r6* and *r7*. Removing these rules for further processing, the e-labels restricted to domain knowledge are, for deducible facts,

$EL(p)\vert_{d} =$	$\emptyset$			
$EL(q)\vert_{d} = \{$	$(c\ d\ f)$	[0, 0.7]	$(r1\ r5\ r8\ r11)$	}
$EL(r)\vert_{d} = \{$	$(c\ d\ e\ \neg g)$	[0, 0.72]	$(r1\ r2\ r4\ r9)$	
	$(c\ d\ e\ \neg g)$	[0, 0.72]	$(r3\ r4\ r9)$	}
$EL(s)\vert_{d} = \{$	$(\neg a\ c\ d\ e)$	[0, 0.8]	$(r1\ r2\ r10)$	}
	$(\neg a\ c\ d\ e)$	[0, 0.8]	$(r3\ r10)$	}
$EL(t)\vert_{d} = \{$	$(c\ d)$	[0, 1]	$(r1\ r5)$	}
$EL(v)\vert_{d} = \{$	$(d\ f)$	[0, 1]	$(r11)$	}
$EL(w)\vert_{d} = \{$	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r1\ r2\ r4\ r9\ r12)$	
	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r3\ r4\ r9\ r12)$	}
$EL(x)\vert_{d} = \{$	$(c\ d)$	[0, 1]	$(r1)$	}
$EL(y)\vert_{d} = \{$	$(c\ d\ e)$	[0, 0.8]	$(r1\ r2)$	
	$(c\ d\ e)$	[0, 0.8]	$(r3)$	}
$EL(z)\vert_{d} = \{$	$(c\ d\ e)$	[0, 0.8]	$(r1\ r2\ r4)$	
	$(c\ d\ e)$	[0, 0.8]	$(r3\ r4)$	}

for rules,

$EL(r1)\vert_{d} = \{$	$(c\ d)$	[0, 1]	$\emptyset$	}
$EL(r2)\vert_{d} = \{$	$(c\ d\ e)$	[0, 0.8]	$(r1)$	}
$EL(r3)\vert_{d} = \{$	$(c\ d\ e)$	[0, 0.8]	$\emptyset$	}
$EL(r4)\vert_{d} = \{$	$(c\ d\ e)$	[0, 0.8]	$(r1\ r2)$	
	$(c\ d\ e)$	[0, 0.8]	$(r3)$	}
$EL(r5)\vert_{d} = \{$	$(c\ d)$	[0, 1]	$(r1)$	}
$EL(r8)\vert_{d} = \{$	$(c\ d\ f)$	[0, 0.7]	$(r1\ r5\ r11)$	}
$EL(r9)\vert_{d} = \{$	$(c\ d\ e\ \neg g)$	[0, 0.72]	$(r1\ r2\ r4)$	
	$(c\ d\ e\ \neg g)$	[0, 0.72]	$(r3\ r4)$	}
$EL(r10)\vert_{d} = \{$	$(\neg a\ c\ d\ e)$	[0, 0.8]	$(r1\ r2)$	
	$(\neg a\ c\ d\ e)$	[0, 0.8]	$(r3)$	}
$EL(r11)\vert_{d} = \{$	$(d\ f)$	[0, 1]	$\emptyset$	}

$EL(r12)\|_d$ ={	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r1\ r2\ r4\ r9)$	
	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r3\ r4\ r9)$	}

for metarules,

$EL(mr0)\|_d$ ={	$(a\ c\ d\ e)$	[0, 1]	$(r1)$	}
$EL(mr1)\|_d$ ={	(a)	[0, 1]	ø	}
$EL(mr2)\|_d$ ={	$(\neg e)$	[0, 1]	ø	}
$EL(mr3)\|_d$ ={	$(c\ d\ e)$	[0, 0.8]	$(r1\ r2)$	
	$(c\ d\ e)$	[0, 0.8]	$(r3)$	}
$EL(mr4)\|_d$ ={	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r1\ r2\ r4\ r9\ r12)$	
	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r3\ r4\ r9\ r12)$	}
$EL(mr5)\|_d$ ={	$(a\ e)$	[0, 1]	ø	}
$EL(mr6)\|_d$ ={	(b)	[0, 1]	ø	}
$EL(mr7)\|_d$ ={	$(c\ d\ f)$	[0, 0.7]	$(r1\ r5\ r8\ r11)$	}
$EL(mr8)\|_d$ ={	$(c\ d\ f)$	[0, 1]	$(r1)$	}

for modules,

$EL(m1)\|_d$ ={	(a)	[0, 1]	ø	
	$(\neg e)$	[0, 1]	ø	}
$EL(m2)\|_d$ ={	$(c\ d\ e)$	[0, 0.8]	$(r1\ r2)$	
	$(c\ d\ e)$	[0, 0.8]	$(r3)$	
	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r1\ r2\ r4\ r9\ r12)$	
	$(c\ d\ e\ f\ \neg g)$	[0, 0.5]	$(r3\ r4\ r9\ r12)$	}
$EL(m3)\|_d$ ={	(b)	[0, 1]	ø	
	$(c\ d\ f)$	[0, 0.7]	$(r1\ r5\ r8\ r11)$	}

From these labels is easy to see that circularities of type C-2 and C-3 exist. A circularity of type C-2 is formed by *q*, *r8*, *v*, *r11*, *mr7*, *q*. This is detected considering the rule sequence for *r8*, (*r1 r5 r11*), which includes rules belonging to *m1*, *m2* and *m3*, and checking that *r8* appears in one e-environment for one of these modules, specifically for *m3*. A circularity of type C-3 is formed by *w*, *r12*, *mr7*, *q*, *r8*, *mr4* , *w*. This is detected considering one of the rule sequences in the e-label for *m2*, for instance (*r3 r4 r9 r12*), containing rules belonging to *m1* and *m3*. But there is an e-environment for *m3* including a rule belonging to *m2*, specifically the rule *r8*. To allow for an effective computation of e-labels, metarules *mr4* and *mr7* are removed for further processing. So, without *r6*, *r7*, *mr4* and *mr7*, the full e-labels for deducible facts are,

$EL(p) =$	$\emptyset$			
$EL(q) = \{$	$(a\ b\ c\ d\ e\ f)$	$[0,\ 0.7]$	$(r1\ r2\ r5\ r8\ r11)$	
	$(a\ b\ c\ d\ e\ f)$	$[0,\ 0.7]$	$(r1\ r3\ r5\ r8\ r11)$	$\}$
$EL(r) = \{$	$(a\ c\ d\ e\ \neg g)$	$[0,\ 0.72]$	$(r1\ r2\ r4\ r9)$	
	$(a\ c\ d\ e\ \neg g)$	$[0,\ 0.72]$	$(r3\ r4\ r9)$	$\}$
$EL(s) =$	$\emptyset$			
$EL(t) = \{$	$(a\ c\ d\ e)$	$[0,\ 1]$	$(r1\ r2\ r5)$	
	$(a\ c\ d\ e)$	$[0,\ 1]$	$(r1\ r3\ r5)$	$\}$
$EL(v) = \{$	$(b\ d\ f)$	$[0,\ 1]$	$(r11)$	$\}$
$EL(w) = \{$	$(a\ b\ c\ d\ e\ f\ \neg g)$	$[0,\ 0.5]$	$(r1\ r2\ r4\ r9\ r12)$	
	$(a\ b\ c\ d\ e\ f\ \neg g)$	$[0,\ 0.5]$	$(r3\ r4\ r9\ r12)$	$\}$
$EL(x) = \{$	$(a\ c\ d)$	$[0,\ 1]$	$(r1)$	
	$(\neg e\ c\ d)$	$[0,\ 1]$	$(r1)$	$\}$
$EL(y) = \{$	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r1\ r2)$	
	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r3)$	$\}$
$EL(z) = \{$	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r1\ r2\ r4)$	
	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r3\ r4)$	$\}$

for rules,

$EL(r1) = \{$	$(a\ c\ d)$	$[0,\ 1]$	$\emptyset$	
	$(\neg e\ c\ d)$	$[0,\ 1]$	$\emptyset$	$\}$
$EL(r2) = \{$	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r1)$	$\}$
$EL(r3) = \{$	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$\emptyset$	$\}$
$EL(r4) = \{$	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r1\ r2)$	
	$(a\ c\ d\ e)$	$[0,\ 0.8]$	$(r3)$	$\}$
$EL(r5) = \{$	$(a\ c\ d\ e)$	$[0,\ 1]$	$(r1\ r2)$	
	$(a\ c\ d\ e)$	$[0,\ 1]$	$(r1\ r3)$	$\}$
$EL(r8) = \{$	$(a\ b\ c\ d\ e\ f)$	$[0,\ 0.7]$	$(r1\ r2\ r5\ r11)$	
	$(a\ b\ c\ d\ e\ f)$	$[0,\ 0.7]$	$(r1\ r3\ r5\ r11)$	$\}$
$EL(r9) = \{$	$(a\ c\ d\ e\ \neg g)$	$[0,\ 0.72]$	$(r1\ r2\ r4)$	
	$(a\ c\ d\ e\ \neg g)$	$[0,\ 0.72]$	$(r3\ r4)$	$\}$
$EL(r10) =$	$\emptyset$			
$EL(r11) = \{$	$(b\ d\ f)$	$[0,\ 1]$	$\emptyset$	$\}$
$EL(r12) = \{$	$(a\ b\ c\ d\ e\ f\ \neg g)$	$[0,\ 0.5]$	$(r1\ r2\ r4\ r9)$	
	$(a\ b\ c\ d\ e\ f\ \neg g)$	$[0,\ 0.5]$	$(r3\ r4\ r9)$	$\}$

for metarules,

EL(mr0) ={	*(a c d e)*	[0, 1]	*(r1)*	}
EL(mr1) ={	*(a)*	[0, 1]	ø	}
EL(mr2) ={	*(¬e)*	[0, 1]	ø	}
EL(mr3) ={	*(a c d e)*	[0, 0.8]	*(r1 r2)*	
	(a c d e)	[0, 0.8]	*(r3)*	}
EL(mr5) ={	*(a e)*	[0, 1]	ø	}
EL(mr6) ={	*(b)*	[0, 1]	ø	}
EL(mr8) ={	*(a c d f)*	[0, 1]	*(r1)*	
	(¬e c d f)	[0, 1]	*(r1)*	}

for modules,

EL(m1) ={	*(a)*	[0, 1]	ø	
	(¬e)	[0, 1]	ø	}
EL(m2) ={	*(a c d e)*	[0, 0.8]	*(r1 r2)*	
	(a c d e)	[0, 0.8]	*(r3)*	}
EL(m3) ={	(b)	[0, 1]	ø	}

Using these results, the verification procedure identifies the following errors:

I-1. Metarule *mr3* is inconsistent with *mr5* because an action *add* can be performed on *m2* after an action *remove* (*EL(mr3)* partially subsumes *EL(mr5)*).

I-2. Integrity constraint $cv(q) + cv(w) \leq 1$ is violated by the input set $(a\ b\ c\ d\ e\ f$ $\neg g)$ which can deduce q and w with maximum certainty values of 0.72 and 0.5 respectively.

I-3. Metarule *mr2* is inconsistent with *r2* because if *mr2* is fired (adding the module *m1* to which *r2* belongs), *r2* cannot be fired (*EL(mr2)* is not fully compatible with $EL(r2)|_d$).

I-4. Metarule *mr8* is inconsistent with *r11* because when *mr8* is fired (removing the module to which *r11* belongs), *r11* would have satisfied its conditions for firing (*EL(mr8)* partially subsumes $EL(r11)|_d$).

R-1. There exists redundancy between *r2* and *r3*, because they require the same conditions to be fired with identical results (*EL*(*r2*) totally includes *EL*(*r3*) and vice versa).

R-2. There exists redundancy between *y* and *z*, because whenever *z* is deduced *y* is also deduced with the same *cv* (*EL*(*y*) totally includes *EL*(*z*) and vice versa).

R-3. There exists redundancy between *mr3* and *r5*, because when *mr3* is fired (adding the module *m2* to which *r5* belongs), all the conditions of *r5* are already satisfied (*EL*(*mr3*) totally subsumes $EL(r5)|_d$).

U-1. Rule *r10* is non-fireable because there is no consistent input set firing *r10* (*EL*(*r10*) = ∅).

U-2. Fact *s* is unreachable because there is no consistent input set supporting *s* (*EL*(*s*) = ∅).

U-3. Metarule *mr3* is shadowed by *mr0* because they have the same support set (*a c d e*) but *mr0* has priority for firing because it is an stopping metarule. Therefore, *mr0* is always fired before *mr3* (*EL*(*mr3*) totally subsumes *EL*(*mr0*)).

U-4. Rules *r2*, *r3*, *r4*, *r5*, *r8*, *r9*, *r12*, are shadowed by *mr0* because *mr0* is always fired before any of them (the e-label of each one of these rules totally subsumes *EL*(*mr0*)).

U-5. Module *m2* is unreachable because *mr3*, the unique metarule adding *m2*, is shadowed.

U-6. Every rule belonging to *m2* is unreachable because *m2* is unreachable.

U-7. Facts *y*, *z*, *t* , *q*, *r*, *w*, are unreachable because their concluding rules are shadowed or unreachable.

4.6 Incremental Verification

We implemented the verification method in a verifier called IN-DEPTH. This verifier was able to verify KBs of small-to-medium size only (<300 rules) because it required large computational resources (CPU time and memory). The reason was the brute-force approach being used: the whole KB was verified without considering the relations between its different parts nor the necessity for specific verification tests with respect to previous verifications. Looking for an effective verification of larger KBs, we developed an incremental approach of the verification method. We have implemented this approach in a new verifier, called IN-DEPTH II. This section describes the incremental verification approach and the IN-DEPTH II implementation.

4.6.1 Conditions for Incremental Verification

Incrementality of a process P on a set S is based on the following facts. First, the same result is obtained when P is executed on the whole set S as when performing a sequence of executions of P on a collection of subsets S_i, such that $\cup S_i = S$. And second, if subsets S_i are chosen incrementally, that is to say $S_i \subset S_{i+1}$, then when executing P on S_{i+1} the results of a previous execution of P on S_i can be reused, simplifying the computation. Incremental processes are of great interest, because they can provide important savings of computational resources. Candidates for incrementality of special interest are processes of exponential complexity, because from several executions on input sets of small size the same result of a single execution on a large set can be obtained (execution that could be unfeasible because of computational resource constraints). Also good candidates for incrementality are processes which should be repeated on sets with high change rate but where the changed part is relatively small with respect to the set size.

Expert system verification shares both conditions: exponential complexity and high change rate in rule bases, resulting from previous error correction. Therefore, it is a very suitable candidate for incrementality. An incremental verification process can be formulated in the following way. Let KB_0 be a verified rule base on which a change operator Θ is applied on an object o, generating a new rule base KB_1.

$$KB_1 = KB_0 + \Theta(o)$$

Then, what kind of tests should be performed and on which elements of KB_1 to verify the new rule base with a minimum effort?. Legal change operators are *ADD*, *MODIFY* and *REMOVE*,[2] acting on rules, modules or metarules. Verification of KB_1 is composed of two independent steps:

[2] They are different of *add* and *remove* actions performed by metarules on modules.

1. If KB_1 contains new objects not appearing in KB_0, these objects should be verified.

2. Changes in KB_1 can affect the verification results on objects in $KB_1 \cup KB_0$ (objects already verified in KB_0). Those verification tests and those objects for which verification results obtained at KB_0 do not hold in KB_1, should be repeated.

An example may clarify these steps. Let $KB_1 = KB_0 + ADD\ (m)$, $m \in M$. Step 1 consists in verifying the new module, for instance testing that it does not contains inconsistencies or checking that its rules are not redundant with other existing rules. Step 2 consists in determining how m affects the results of verification tests performed at KB_0, and to which extent these results are maintained in KB_1. This step only considers those verification tests which, performed in KB_0, should be repeated in KB_1 *on the same objects*. For instance, if m contains a new way to deduce a fact f existing in KB_0, all verification tests involving f should be repeated since they are incomplete in KB_1. Step 2 is of key importance for incrementality, and the remainder of this section is devoted to it.

The verification method is based on computing e-labels for objects in KB and testing that some relations hold among these e-labels. Those objects in $KB_1 \cap KB_0$ for whose e-labels do not change from KB_0 to KB_1, the results of verification tests in KB_0 are still valid at KB_1, since they cannot change. On the contrary, those objects for which their e-labels change from KB_0 to KB_1 should be verified again, because their results can no longer be guaranteed. The set of objects in $KB_1 \cap KB_0$ sharing e-labels in KB_0 and KB_1 is called KB_{inv}. In the following, we investigate the composition of KB_{inv} in function of legal change operators.

4.6.1.1 Operator *ADD*

Proposition 1: Let $KB_1 = KB_0 + ADD\ (x)$, $x \in R \cup M \cup MRM$. The set KB_{inv} is composed of those objects in KB_0 that do not depend on x.

Proof. The proof is based on the form of the equations (1)-(5) for e-label computation in section 4.3.2. Let $x \in R$. If facts in KB_{inv} do not require x to be deduced, equation (2) will be applied in KB_1 to the same rules than in KB_0. If rules and metarules of KB_{inv} do not require facts deduced by x, equations (3) and (4) will be applied in KB_1 to the same facts than in KB_0. If modules in KB_{inv} are not activated by metarules requiring x, equation (5) will be applied in KB_1 to the same metarules than in KB_0. Let $x \in M$. Adding a module is reduced to adding rules: it simply consists of a sequence of rule additions. Let $x \in MRM$. If modules in KB_{inv} are not activated by x, equation (5) will be performed on KB_1 on the same metarules than on KB_0. If rules in KB_{inv} do not depend on x, equation (3) will be applied in KB_1 to the same objects than in KB_0. Analogous reasoning can be applied to facts and metarules, equations (2) and (4). Therefore, for objects not depending on x their e-labels are not modified by ADD (x).♦

Proposition 2: Let $KB_1 = KB_0 + \text{ADD}\ (mrs)$, $mrs \in MRS$. Then, $KB_{inv} = KB_0$.

Proof. Metarules MRS do not appear in the computation of e-labels, equations (1)-(5).♦

4.6.1.2 Operator REMOVE

Proposition 3: Let $KB_1 = KB_0 + \text{REMOVE}\ (x)$, $x \in R \cup M \cup MRM$. Set KB_{inv} is composed of those objects in KB_0 which do not depend on x.

Proof. Same as that of proposition 1.♦

Proposition 4: Let $KB_1 = KB_0 + \text{REMOVE}\ (mrs)$, $mrs \in MRS$. Then, $KB_{inv} = KB_0$.

Proof. Same as that of proposition 2.♦

Finally, to show that not all verification tests should be repeated in $KB_1 - KB_{inv}$ after an operator REMOVE, we need the following proposition.

Proposition 5: Let $KB_1 = KB_0 + \text{REMOVE}\ (x)$, $x \in R \cup M \cup MRM$. Then, for $y \in F_1 \cup R_1 \cup M_1 \cup MR_1$, $EL_1(y) \subseteq EL_0(y)$.

Proof. Disjuntions in equations (2) and (5) are performed in KB_1 on a subset of the set of objects on which they were performed at KB_0. Therefore, $EL_1(y) \subseteq EL_0(y)$ for $y \in F_1 \cup M_1$. Using the equations (3) and (4) this condition is extended to $y \in R_1 \cup MR_1$.♦

Corollary 1: Let $KB_1 = KB_0 + \text{REMOVE}\ (x)$, $x \in R \cup M \cup MRM$. The verification tests to be repeated in $KB_1 - KB_{inv}$ are the following: I-3, R-1, R-2, R-3, U-1, U-2, U-3, U-4, U-5, U-6, U-7.

Proof. Verification tests I-1, I-2, I-4, R-4, C-1, C-2, C-3 generate errors if there exists a pair of e-environments in the e-labels of two objects satisfying some condition. All these tests have failed in KB_0, that is to say, it does not exist a pair of e-environments in the e-labels of objects in KB_0 satisfying this condition. Proposition 5 shows that e-labels of objects in KB_1 are subsets of their corresponding e-labels in KB_0. Therefore, these verification tests will also fail in KB_1.♦

4.6.1.3 Operator MODIFY

Proposition 6: Let $KB_1 = KB_0$ + MODIFY (x), $x \in R \cup M \cup MRM$. The set KB_{inv} consists of those objects in KB_0 that do not depend on x.

Proof. Direct from propositions 1 and 3, since MODIFY (x) = REMOVE (x) + ADD (x). ♦

4.6.2 IN-DEPTH II: An Incremental Verifier

The IN-DEPTH II verifier can perform specific verification tests on specific KB parts, allowing for a flexible verification process. IN-DEPTH II implements the verification method in an incremental way. It accepts a KB and the list of modifications as input and performs the minimum set of tests to achieve a global correctness. Obviously, it can also work in a non-incremental mode. Incrementality is based on the computation of KB_{inv}. Specific algorithms used in IN-DEPTH II follow.

4.6.2.1 Computation of KB_{inv}

Given $KB_1 = KB_0 + \Theta(x)$, from propositions 1, 3 and 6 it is clear that KB_{inv} is the set of objects of KB_0 which do not depend on x. To compute KB_{inv} we only need to build a directed dependency graph G representing all the dependencies in KB_{max}, where KB_{max} = KB_1 if Θ = ADD, MODIFY and $KB_{max} = KB_0$ if Θ = REMOVE. G is formed by (N, E) where N is a set of nodes and E is a set of directed arcs. Each fact, rule, module or metarule of KB_{max} is represented by a different node in N. Each node is labelled by the object that it represents. There is an arc from node n_i to n_j when the object labelling n_i depends on the object labelling n_j by the three possible dependencies: <- , <= , <≡ . The transitive closure of G can be computed using the Warshall's algorithm [Sedgewick 88], giving the set of nodes connected with any other node in G. In particular, those nodes reachable from x represent all objects in KB_{max} depending on x. The complement set, nodes not reachable from x, represents those objects in the set KB_{inv}.

4.6.2.2 Incremental Algorithm

The incremental algorithm consists of the two steps mentioned at the beginning of section 4.6.2. In step 1, objects added to the KB are verified by themselves and in their relation with other existing objects. This step is described in figure 4-1. If a stop metarule is added it is tested for useless objects only, because this type of metarules is not involved in any other verification issue. Before executing step 2, the directed dependency graph G is built to compute the set KB_{inv}. In step 2, objects in KB_1 - KB_{inv} are verified. If an object is removed, only the following verification tests should be made (corollary 1): I-3, R-1, R-2, R-3, U-1, U-2, U-3, U-4, U-5, U-6 y U-7. This step is described in figure 4-2.

```
procedure step-1 ($\Theta, x$)
        if $\Theta$ = ADD, MODIFY then
                case $x$
                        rule, module, metarule on module
                        execute        U-1, U-2, U-3, U-4, U-5, U-6, U-7,
                                       C-1, C-2, C-3,
                                       I-1, I-2, I-3, I-4,
                                       R-1, R-2, R-3, R-4
                                       on $x$
                        stop metarule
                        execute        U-1, U-2, U-3, U-4, U-5, U-6, U-7
                                       on $x$
                endcase
        endif
endprocedure
```

Figure 4-1. Step 1 procedure.

```
procedure step-2 ($KB_1$, $KB_{inv}$,$\Theta$)
        case $\Theta$
                ADD, MODIFY
                execute        U-1, U-2, U-3, U-4, U-5, U-6, U-7,
                               C-1, C-2, C-3,
                               I-1, I-2, I-3, I-4,
                               R-1, R-2, R-3, R-4
                               on $KB_1$ - $KB_{inv}$
                REMOVE
                execute        U-1, U-2, U-3, U-4, U-5, U-6, U-7
                               I-3, R-1, R-2, R-3
                               on $KB_1$ - $KB_{inv}$
        endcase
endprocedure
```

Figure 4-2. Step 2 procedure.

4.6.2.3 Multiple Operators

The previous algorithms assume that KB_1 is obtained from KB_0 from the action of a single change operator. This situation can be generalized to multiple change operators, when

$$KB_n = KB_0 + \Theta_1(o_1) + \ldots + \Theta_n(o_n)$$

Procedures of section 4.6.2.2 can be applied to the sequence of operators, considering each time one $\Theta_i(o_i)$ and obtaining a new verified rule base KB_i. This formulation considers $KB_i = KB_{i-1} + \Theta_i(o_i)$ where KB_{i-1} includes operators $\Theta_1(o_1) + \ldots + \Theta_{i-1}(o_{i-1})$. However, this approach is not optimal in the common case when many change operators act on the same module *m* of KB_0. At the execution of the `step-2` procedure for $\Theta_i(o_i)$, module *m* will be verified, operation that will be repeated again at the next operator acting on *m*. Therefore, it seems more efficient to group all the change operators in the sequence, performing the `step-2` procedure only once. This modifies the computation of KB_{inv} in the following way: the dependency graph *G* is built from $KB_n \cup KB_0$, and KB_{inv} is computed as the subset of unreachable nodes from any object o_i present in the change operator sequence. The multiple change procedure is described in figure 4-3.

```
procedure multiple-change
    For every $\Theta_i(o_i)$ do
        step-1 ($\Theta_i$, $o_i$)
    endfor
    compute $KB_{inv}$ from $\Theta_1(o_1) + \ldots + \Theta_n(o_n)$
    if any $\Theta_i = REMOVE$ then $\Theta = REMOVE$ else $\Theta = ADD$ endif
    step-2 ($KB_n$, $KB_{inv}$, $\Theta$)
endprocedure
```

Figure 4-3. Multiple-change procedure.

4.7 Verifying PNEUMON-IA

We have used the IN-DEPTH II verifier to verify the expert system PNEUMON-IA [Verdaguer 89], which was developed with the shell MILORD [Sierra 89]. This system performs etiological diagnosis of community acquired pneumonias in adults. It is composed of 500 facts, 600 rules and 100 metarules distributed in 25 modules considering 22 different etiologies. PNEUMON-IA was validated after its construction by a team of 5 independent experts in the field using a test set composed of 66 cases. Validation results showed that the performance level of PNEUMON-IA was comparable to that of human experts. In addition, a field test was performed using PNEUMON-IA in a hospital for a three months period.

In spite of the quality of PNEUMON-IA recommendations, its verification was challenging. We started executing IN-DEPTH II on the whole KB and testing all the verification issues. That generated long listings of results difficult to read and manipulate, so we proceeded in a different way. We concentrated our verification efforts on specific parts of the KB and considered specific issues. Then we started to get results, as non-fireable rules, redundancies or potential inconsistencies. For the expert developer of PNEUMON-IA that was a innovative way to analyze the system. He studied IN-DEPTH II outputs to discriminate actual errors from situations that appeared to be anomalous but they had some kind of justification. Correcting actual errors caused further modifications in the KB improving its structure, robustness and safety. So far, we have verified the 50% of the KB contents. The number of actual errors detected has been low in comparison with the KB size. The verification process has increased our confidence on PNEUMON-IA, specially because it has certified the absence of certain error types.

Verification sessions were made weekly. Each session lasted two or three hours, where the developer of IN-DEPTH II and the expert worked together analyzing verification results and considering potential updates. Sessions were devoted to a small KB part, one or two modules, and considering specific problems. After each session the KB was updated and IN-DEPTH II was executed on the KB parts under study. The incremental procedure has been of great help for an effective verification at a reasonable computational cost. These sessions also caused improvements to IN-DEPTH II. In addition to a few bugs fixed, some extra functionalities were conceived and added to the verifier, as the redundancy R-4.

4.7.1 Adding Extra Knowledge

The first problem we faced when started to use IN-DEPTH II for verification was the large amount of unrealistic situations considered by the verifier. This was specially relevant for inconsistency checking, where situations are combined in unexpected ways. We had to add extra knowledge for verification purposes, to define those inputs that were meaningless in the problem domain. This was made in two ways: (i) by defining incompatible values on multivalued facts and (ii) by adding a set of integrity constraints on the input.

A multivalued fact represents an attribute in the problem domain that can take several values. In occasions, not all the possible values can be taken simultaneously because some of them are incompatible. For example, there are several types of sputum, i.e, watery sputum, rusty sputum, purulent sputum, etc. A watery sputum can also be rusty (in the beginning of the disease), but it cannot be purulent. In PNEUMON-IA there are 19 multivalued facts. Their possible values were analyzed by the expert for incompatibilities. He found 54 sets of incompatible values affecting to 10 multivalued facts. These incompatible values were recorded in a file that is read by IN-DEPTH II before any testing. When the verifier compares two e-environments with a common multivalued fact, it checks if their values are not incompatible.

An integrity constraint on the input is a conjunction of external facts that represent an incompatible input in the problem domain. There are thousand of incompatible inputs in a real application, and declaring all of them as integrity constraints would be unfeasible. (in PNEUMON-IA there are 400 external facts, many of them with a wide range of possible values). We declare as integrity constraints on the input the minimal set of incompatible inputs that guarantee that no inconsistency of type I-2 is reached. Given that these inconsistencies are tested based on integrity constraints on the PNEUMON-IA output (representing constraints in the problem domain), the set of integrity constraints on the input is developed according to the required degree of consistency on the system output. So far, the number of these integrity constraints is reasonably low. The set of integrity constraints on the input is recorded in a file. IN-DEPTH II reads this file before any test and takes into account their contents to prevent the generation of meaningless input situations.

4.7.2 Interpretation and Correction of Error Occurrences

IN-DEPTH II output is a collection of occurrences of KB objects satisfying some predefined pattern considered as erroneous. An error occurrence is different from the error causing it since an error can cause many error occurrences. For instance, an incorrect condition in a rule can cause many occurrences of inconsistency errors. In the verification process we have found quite often this one-to-many relation between error causes and error occurrences. This implies that the number of error occurrences is not an accurate estimation of the quality level of an ES.

After detecting an error occurrence, we have to determine its causes, consequences and the best way to correct it. This can be made at two levels: at a purely syntactical (structural) level, or at a semantic (knowledge) level. At a syntactic level, objects involved in an error occurrence are seen in abstract, without interpreting the knowledge they contain with respect to the problem domain. At a semantic level, those objects are analyzed taking into account the knowledge they represent and the role that this knowledge plays in the intended ES task. Our experience indicates that analyzing error occurrences at a syntactical level leads to a partial description of their causes and consequences, and provides a poor advice for their correction. Only when these occurrences are considered from a semantical point of view their actual causes and consequences appear, with a full repertoire of correction possibilities. This semantical analysis directly involves the human expert in the verification process, in order to provide their adequate interpretation in terms of domain knowledge. Examples of this semantical interpretation of error occurrences and their global correction are given in section 4.7.2.

Regarding the error relevance for the intended ES task, we divide error occurrences in two types: ignorable and actual. An error occurrence is *ignorable* when, in spite of being a defect or anomaly in the KB, it neither reveals incorrect knowledge nor causes any ES malfunction. Ignorable error occurrences do not have to be corrected. The presence of ignorable error occurrences is explained in two ways. First, the verifier can have too restrictive error patterns for the considered application. Second, some tricks in the knowledge organization can be made to simulate specific behaviors, what can be detected as erroneous. In PNEUMON-IA we consider the error types I-3 and R-3 as ignorable, because of the specific conditions of the application (see sections 4.7.2.1 and 4.7.2.2 for further details). On the other hand, an error occurrence is *actual* when it reveals incorrect knowledge or it can cause ES malfunctions. Obviously, actual error occurrences must be corrected.

The correction process of an error occurrence has two steps. First, we have to fix its actual causes. This can generate several modifications in the KB. After these modifications the error occurrence is corrected and if the verifier were executed again, it would not detect it. Second, this specific error occurrence can suggest further modifications of other KB objects not connected with the objects involved in it. These modifications are induced by the existence of analogies or similarities among KB parts. Frequently, objects involved in this second step do not cause any verification error in their initial form. However, after the induced modification the KB is better than before, in the sense that it can be more accurate, safe or robust.

A detailed description of each type of detected error in PNEUMON-IA follows. Circularity errors do not appear because they have not been detected. Each error type is illustrated with a real occurrence plus the solution we gave to it.

4.7.2.1 Inconsistency

We found only two types of inconsistency errors, types I-2 and I-3. Inconsistency of type I-2 is the classical concept of inconsistency as the violation of an integrity constraint on deducible facts for some input. Several occurrences of I-2 type appeared, caused by (i) an non-valid input and (ii) an inadequate knowledge organization and structure. However, the knowledge itself was not incorrect. We solved them by (i) adding more integrity constraints on the input facts and (ii) developing a better knowledge structure and organization. The expert has a central role when checking this type of inconsistency. He provides the integrity constraints to be tested and analyzes the potential inputs violating them. Integrity constraints often have different degrees of incompatibility, so their analyses should be made on a one-to-one basis. Frequently, inputs leading to inconsistent situations are in the application boundaries of the expert system. This can force the expert to give a more precise definition of the applicability range of the system, making explicit those assumptions that had never been elicitated during the system development phase. Inconsistency of type I-3 relates metarules adding a module and the rules contained in it. The expert considered it as an ignorable error. Examples of actual occurrences of inconsistencies of types I-2 and I-3 are given in the following.

I-2. To test PNEUMON-IA for inconsistency of type I-2 a number of integrity constraints on the system output are needed. These constraints depend on the semantics of the problem domain. Concerning pneumonia clinical manifestation, pneumonias are classically divided into two main groups, bacterial and atypical. Some pneumonias are clearly typical bacterial pneumonias while others are atypical pneumonias. Some pneumonias share clinical features of both classes. It is assumed that if there is high evidence of a pneumonia for one class, the evidence for the other class should be low. Then, we conceived as integrity constraint a situation where both bacterial and atypical are considered with high evidence. In PNEUMON-IA, these classes are represented by the facts `bacterial` and `atypical`, so we tested the following integrity constraint:

$$\texttt{bacterial} \geq \textit{quite-possible} \text{ and } \texttt{atypical} \geq \textit{quite-possible}$$

Around 50 different input situations were generated as potential inputs that would violate this constraint. They were carefully analyzed by the expert, causing either changes in the KB or the addition of integrity constraints on the input when the computed input was meaningless. In the following, two inconsistency occurrences with respect to this constraint are explained.

Consider the rules in figure 4-4. Rule `r03004` states that from two conditions about blood analysis, the number of leukocytes and its composition, `bacterial` can be concluded with high evidence. Rule `r03026` establishes a set of conditions from which `atypical` can be concluded with high evidence. Joining the left-hand sides of both rules we get the conditions to violate the considered constraint. The expert noticed that these conditions were not compatible. In particular, the conditions appearing in the left-hand side of `r03004` are incompatible with `patient-status ≠ serious or very-serious`, condition of `r03026`. This incompatibility was formulated as an integrity constraint on the input, and no modification on the KB was made.

```
r03004:   if    leukocytes > 15000
                left-shift ≥ quite-possible  then   bacterial quite-possible

r03026:   if    not comma
                patient-status ≠ serious or very-serious
                headache
                unknown sputum
                relative-bradycardia      then   atypical quite-possible
```

Figure 4-4. Occurrence of inconsistency of type I-2.

As another example, consider the rules described in figure 4-5. Rule `r03001` indicates that `herpes-labialis` provides high evidence for `bacterial`, while rule `r03026` says that the absence of `predisposing-factors` and the existence of `interstitial-pattern` provides high evidence for `atypical`. A patient satisfying

```
r03001:   if    herpes-labialis           then   bacterial very-possible

r03026:   if    not predisposing-factors
                interstitial-pattern      then   atypical quite-possible
```

Figure 4-5. Occurrence of inconsistency of type I-2.

the three conditions would get high evidence for both `bacterial` and `atypical`. From the expert opinion, no incompatibility exists between `herpes-labialis` (an illness in the lips) the absence of `predisposing-factors` (a guard for the rule) and `interstitial-pattern` (an X-ray film pattern). After a detailed study of the role of each symptom, the expert concludes that `herpes-labialis` does not provide evidence for bacterial pneumonia, but for a subclass of bacterial pneumonias. This situation is solved removing `herpes-labialis` as a condition for `bacterial` and adding it to the specific etiologic agents, subclasses of bacterial pneumonias, which are related to it.

I-3. We found many inconsistencies of type I-3 but the expert considered this type of error as ignorable. An inconsistency of type I-3 exists when the conditions required to fire a metarule *mr* adding a module *m* are incompatible with the conditions of a rule *r* inside *m*. This does not mean that rule *r* is non-fireable since module *m* can be activated by other metarules different from *mr*. An occurrence of inconsistency of type I-3 is shown in figure 4-6. If the module `pneum` is accessed through the metarule `r25888`, the rule `r04001` will not be fired. However, this is not an error because there are other ways to access this module enabling this rule to fire, as the metarule `r25901`.

```
pneum module

  r04001:   if    bacterial
                  predisposing-factors
                  chronic-obstructive-pulmonary-disease
                                                then  pneumococus possible

metarules

  r25888:   if    not predisposing-factors  then  add bact-atip, pneum, ...

  r25901:   if    predisposing-factors
                  chronic-obstructive-pulmonary-disease
                                                then  add bact-atip, pneum, ...
```

Figure 4-6. Occurrence of inconsistency of type I-3.

4.7.2.2 Redundancy

We have found occurrences of the four types of redundancy. Each type of redundancy has been interpreted in a different way, generating different correction patterns. In general, the level of redundancy detected in PNEUMON-IA has been low. In the following, an actual occurrence of every redundancy issue is explained.

R-1. Consider the rules described in figure 4-7. The rule `r03020` is redundant with the rule `r0302a`, since both conclude the same object but `r0302a` requires less conditions than `r03020`. Let assume that all the conditions in the left-hand side of `r03020` are satisfied. In this case, the metarule `r25705` will be fired (the fact `pleural-fluid-sample` is true) activating the module `pleural`. When visiting `pleural`, the rule `r26aa2` will be fired because its left-hand side is satisfied, concluding the fact empyema with certainty *almost-sure*. Later, when visiting the module `bact-atip` the rule `r0302a` will be fired, jointly with `r03020` and with identical results, since both conclude the same object with the same certainty, computed through the fired rules. The fact `pleural-fluid-sample` is obtained in the module `data-collection`. Regarding the order in which modules are visited `data-collection` is always the first one, `pleural` is next (when adequate), and `bact-atip` is the next one. This redundancy involved the rule r03020 versus the sequence (`r25705`, `r26aa2`, `r0302a`). It was solved by removing the rule r03020. No other modifications were made.

```
bact-atip module

  r03020:   if    pleural-effusion
                  pleural-fluid-sample
                  pleural-exudate
                  pleural-polynuclears-% > 50
                  pleural-purulent          then   bacterial almost-sure

  r0302a:   if    empyema                   then   bacterial almost-sure
---------------------------------------------------------------------------
pleural module

  r26aa2:   if    pleural-exudate
                  pleural-polynuclears-% > 50
                  pleural-purulent          then   empyema almost-sure
---------------------------------------------------------------------------
metarules

  r25705:   if    pleural-fluid-sample      then   add pleural
```

Figure 4-7. Occurrence of redundancy of type R-1.

R-2. We found some redundancies of type R-2 but we considered them as ignorable errors because the specific facts involved. In MILORD there are two kind of rules, the concluding rules that assert a fact appearing in their right-hand side, and the up-down rules that modify the certainty degree of a deduced fact. IN-DEPTH II only considers concluding rules and it does not analyze up-down rules. This caused that some correct situations were considered as redundant by the verifier.

In PNEUMON-IA, up-down rules are used to tune the final certainty degree of a goal in the following way. Two different facts are used to represent a goal. The first fact, `g-d`, collects all the evidence for the goal using the concluding rules. The second fact, `g`, is deduced from `g-d` with its same certainty degree. The up-down rules act on `g`, which contains the final certainty for the goal. The usage of two facts is a pure matter of convention in rule writing. With this structure in the knowledge base, IN-DEPTH II detected as redundant the facts `g-d` and `g` for every goal. Given that these facts were not really redundant, no modification was made on these cases. No other occurrence of type 2 redundancy was detected.

R-3. We found some redundancies of type R-3, but the expert considered this type of error as ignorable. A redundancy of type R-3 exists when a metarule adding a module subsumes one rule of this module. This type of redundancy assumes that either some facts used in metarules (control purpose) are not used in rules (deduction purpose), or that rules should be more specific than metarules. In pneumonia diagnosis these two conditions are not satisfied. Facts are used interchangeably for control and deduction purposes, and it is not possible to assure that rules will be more specific than metarules. In consequence, these occurrences were revised by the expert but they did not cause any change.

R-4. We found a large number of redundancies of type R-4. As it was stated in section 4.4, this kind of error does not always represent an actual error. It is provided for expert analysis of those situations that could be close to redundancy.

An example of detected redundancy of type R-4 that caused modifications in PNEUMON-IA is contained in figure 4-8. The rule sequences (`r03010a`, `r09001a`) and (`r09011b`, `r09013a`) deduced the same fact `anaerobes`, from the same set of external conditions (`foul smell sputum` and `alcoholic`) but with different certainty degrees (*moderately-possible* versus *quite-possible*). This error occurrence revealed an inadequate usage of `foul smell sputum` as a condition for `bacterial` in the rule `r03010a`. It was corrected by removing the rule `r03010a`.

`r03010a:`	**if**	`foul smell sputum`	**then**	`bacterial` *almost-sure*
`r09001a:`	**if**	`alcoholic` `bacterial`	**then**	`anaerobes` *moderately-possible*

`r09011b:`	**if**	`alcoholic`	**then**	`anaerobes-risk` *almost-sure*

`r09013a:`	**if**	`foul smell sputum`
		`anaerobes-risk`
			then	`anaerobes` *quite-possible*

Figure 4-8. Occurrence of redundancy of type R-4.

4.7.2.3 Useless Objects

In PNEUMON-IA we have found a low number of useless objects, involving only two error types. In type U-1, rules are non-fireable because their e-label is empty (no consistent input exists for them). In type U-2, facts are unreachable because their concluding rules are non-fireable. In the following, an actual occurrence of these useless object types is explained.

U-1. Consider the rules described in figure 4-9. The rule `r01062` is useless because its e-label is empty. The fact `interstitial-pattern` is external, while `alveolar-pattern` is a deducible fact that can be concluded by the rules `r01058`, `r01059` and `r01059a`. In order to satisfy the condition `alveolar-pattern` ≥ *possible*, the fact `alveolar-pattern` can only be deduced by `r01058`, since the other two rules do not deduce it with enough certainty (*moderately-possible* < *possible* < *almost-sure*). However, `air-bronchogram` and `interstitial-pattern` are incompatible features for a X-ray film (in the context of an acute pneumonia). Therefore, rule `r01062` is non-fireable.

`r01058:`	**if**	`air-bronchogram`	**then**	`alveolar-pattern` *almost-sure*
`r01059:`	**if**	`fluffy-margins`	**then**	`alveolar-pattern` *moderately-possible*
`r01059a:`	**if**	`early-coalescence`	**then**	`alveolar-pattern` *moderately-possible*
`r01062:`	**if**	`alveolar-pattern` ≥ *possible* `interstitial-pattern`	**then**	`alveol-inter-pattern` *almost-sure*

Figure 4-9. Occurence of an useless object of type U-1.

This issue was solved by changing in r01062 the condition alveolar-pattern ≥ *possible* by alveolar-pattern ≥ *moderately-possible*. In this way, the rules r01059 and r01059a are useful for firing the rule r01062, since the values fluffy-margins and interstitial-pattern are compatible, as well as early-coalescence and interstitial-pattern. This issue reveals as erroneous the threshold of a condition. Without the addition of integrity constraints discussed in section 4.7.1, this error occurrence would have not been discovered.

U-2. Consider the rule r01072 in figure 4-10. The fact clinical-xr-dissociation is unreachable because r01072 is the only rule concluding it and it is non-fireable. This rule is non-fireable because the fact early-coalescence is always deduced with a certainty degree higher or equal than *possible*, so its condition will never be satisfied. This error occurrence was solved by changing the condition for early-coalescence, to allow for its effective satisfaction accordingly with the rules concluding it.

```
r01072:   if     alveolar-pattern ≥ possible
                 early-coalescence ≤ possible
                 then   clinical-xr-dissociation almost-sure
```

Figure 4-10. Occurrence of an useless object of type U-2.

4.7.3 Verifier Usage

Verifying PNEUMON-IA, we have performed many IN-DEPTH II executions. Some of these executions involved the whole KB, testing all the verification issues. Some others involved just a few modules testing specific problems. This variety originates many verification reports that are quite difficult to summarize. As a representative sample of the IN-DEPTH II output, we have selected a verification report considering the whole KB and testing all the issues. The results of this report as well as the CPU time and memory required for its execution are described in the following.

4.7.3.1 Collected Error Occurrences

The number and type of the error occurrences detected when executing IN-DEPTH II on an early version of PNEUMON-IA are summarized in table 4-1. The first row contains the error

M	I1	I3	I4	R1	R2	R3	C1	C2	C3	U1	U2	U3	U4	U5	U6	U7	Tot
1	0	0	0	35	6	0	0	0	0	2	1	0	0	0	0	0	44
2	0	0	0	0	2	0	0	0	0	0	0	0	0	0	0	0	2
3	0	6	0	0	4	2	0	0	0	0	0	0	0	0	0	0	12
4	0	18	0	0	1	0	0	0	0	1	0	0	0	0	0	0	20
5	0	11	0	0	1	0	0	0	0	*1	0	0	0	0	0	0	13
6	0	20	0	0	1	1	0	0	0	*1	0	0	0	0	0	0	23
7	0	14	0	0	0	1	0	0	0	*1	0	0	0	0	0	0	16
8	0	5	0	0	1	0	0	0	0	*1	0	0	0	0	0	0	7
9	0	13	0	1	1	0	0	0	0	*1	0	0	0	0	0	0	16
10	0	4	0	0	1	0	0	0	0	*1	0	0	0	0	0	0	6
11	0	2	0	1	3	0	0	0	0	*1	0	0	0	0	0	0	7
12	0	18	0	0	4	1	0	0	0	2	1	0	0	0	0	0	26
13	0	5	0	0	1	0	0	0	0	0	0	0	0	0	0	0	6
14	0	4	0	0	1	0	0	0	0	0	0	0	0	0	0	0	5
15	0	22	0	0	3	1	0	0	0	0	0	0	0	0	0	0	26
16	0	0	0	0	0	0	0	0	0	4	0	0	0	0	0	0	4
17	0	27	0	3	3	2	0	0	0	*1	0	0	0	0	0	0	36
18	0	10	0	0	1	0	0	0	0	*1	0	0	0	0	0	0	12
19	0	10	0	0	0	0	0	0	0	*1	0	0	0	0	0	0	11
20	0	2	0	0	1	0	0	0	0	*1	0	0	0	0	0	0	4
21	0	34	0	0	2	0	0	0	0	*1	0	0	0	0	0	0	37
22	0	1	0	0	5	3	0	0	0	*1	0	0	0	0	0	0	10
23	0	0	0	0	2	0	0	0	0	*1	0	0	0	0	0	0	3
24	0	1	0	0	1	0	0	0	0	0	0	0	0	0	0	0	2
Tot	0	209	0	40	45	11	0	0	0	9	2	0	0	0	0	0	316

Table 4-1. Summary of error occurrences detected by IN-DEPTH II.

type while the first column identifies the module number. The verification has been made on a modulé basis, so error occurrences are grouped in the module they have been detected. Redundancies of type R-4 are not considered because they have been added to IN-DEPTH II after this specific execution. Inconsistencies of type I-2 are not detailed in table 4-1 because they cannot be associated to one module and they depend on the set of integrity constraints tested. These two error types are detailed later.

This table offers some direct information. Concerning inconsistencies, the only detected error type is I-3. This error is not a serious one, and the expert considered it as ignorable. The absence of inconsistencies of types I-1 and I-4 indicates a good structure in the control knowledge itself. Redundancies have occurrences of the three types. In particular, the 35 redundancies of type R-1 in module 1 do not represent actual errors. They are caused by the specific structure of module 1. This module is purely devoted to acquire patient's data in an order that appears natural for a physician, but no real deduction is made. Some tricks have been made to simulate this natural order, tricks that have caused these redundancies. About circularities, there is a neat absence of this serious error. Specific redundancies of type R-2 have not been considered actual errors as is explained in section 4.7.2, and redundancies of type R-3 have been considered as ignorable. Regarding useless objects only two types of errors, U-1 and U-2, have been detected. About useless objects of type U-1, nine different rules or metarules have been detected as unfireable. An asterisk (*) preceding some numbers in the U-1 column means that the specific useless object (a metarule in this case) has been already identified in a previous module (specifically in module 4). For this reason, these occurrences are not added in the total for U-1 objects. About useless objects of type U-2, two unreachable facts have been detected as a consequence of unfireable rules. The absence of useless objects of types U-3 to U-7 provides evidence for the correctness of stopping metarules. In summary, from 316 different error occurrences detected, only 16 occurrences represented actual errors, that is the 0.5% of the total error occurrences detected.

Concerning redundancy of type R-4, some tests have been made on 10 modules. Results are variable, ranging from 0 occurrences (in modules 4, 11 and 12) to 55 (in module 17). This type of redundancy was added to allow for a less restrictive, more open test. It detects situations that are good candidates to contain some kind of redundancy. These situations have to be analyzed by the expert who judges about their potential redundancy with respect to the domain knowledge.

Concerning inconsistency of type I-2, we have tested one dozen of integrity constraints involving the goals of 6 modules (modules 1, 2, 3, 4, 12 and 17). Verifier outputs indicated that some constraints were violated from 20 to 60 different inputs, although a few constraints were not violated by any input. After studying all the potential inconsistent situations for a constraint, we discovered that most of them were generated by one error in one specific rule. This error was amplified by the action of rule chaining, causing that many inputs could lead to inconsistent situations. This is an example of a single error cause and many error occurrences.

4.7.3.2 Time and Memory Requirements

The verification of the whole KB is performed in an incremental way, as stated in section 4.6. The verification process is performed on an single module, obtaining a small verified KB. Then, a new module and its associated metarules are added to this small KB already verified, obtaining a new KB on which the verification is executed again. This process is repeated 24 times, corresponding to the 24 modules of PNEUMON-IA. The CPU time of the verification process on a SUN-4/260 is given in figure 4-11. Verification after adding a new module (30 rules and 4 metarules approximately) requires 5.3 CPU hours on the average. This performance is adequate for a knowledge engineering workbench,

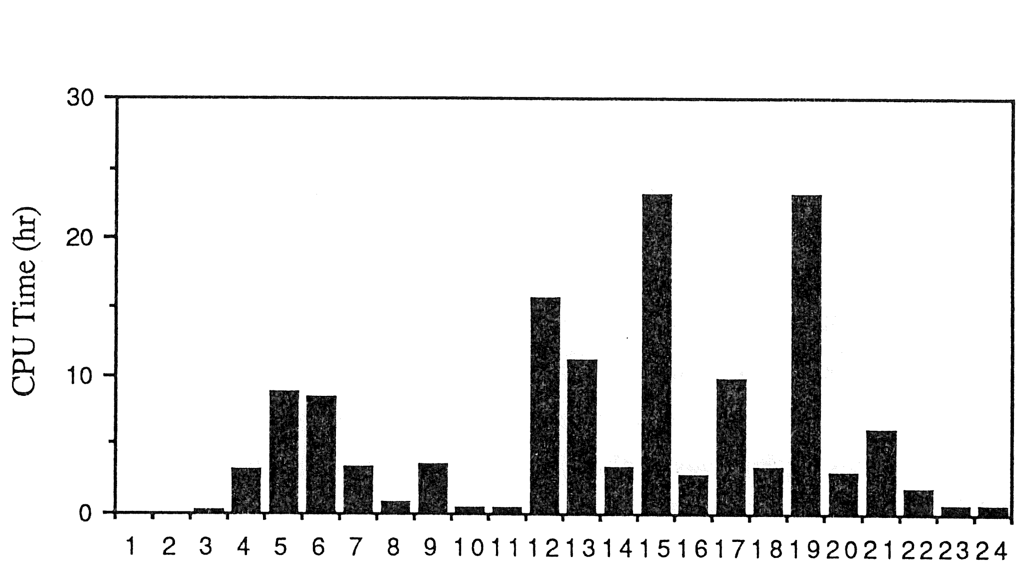

Figure 4-11. CPU time required for the verifier when adding each time a new module.

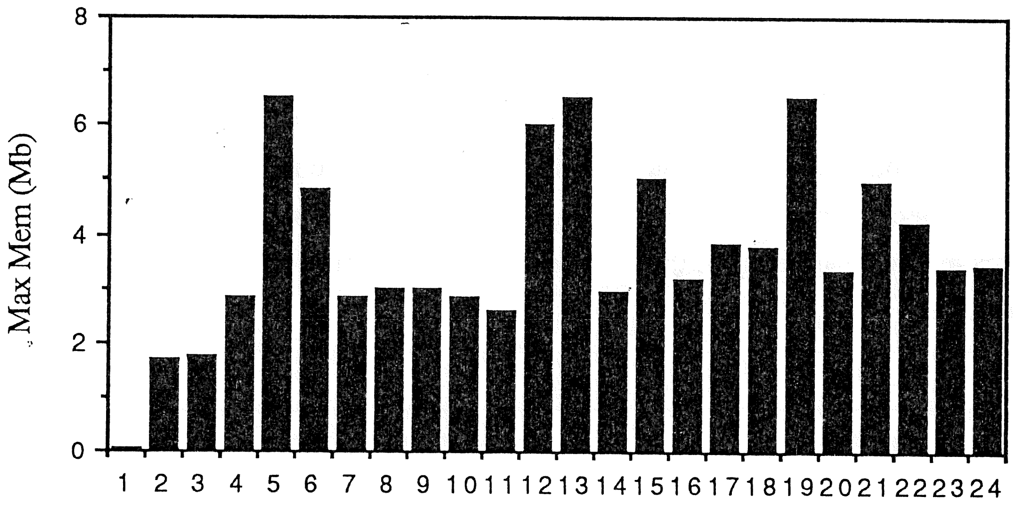

Figure 4-12. Maximum memory consumed by verified module.

running during the night the verification process to check the changes that have been made during the day. Maximum memory requirements for each execution are shown in figure 4-12. The memory used in the verification of a module has been 3.6 Mbytes on the average. The maximum memory requirement has been of 6.7 Mbytes, which is easily available in current workstations.

4.7.4 Verification Impact

The impact of the verification that we have performed on PNEUMON-IA using the IN-DEPTH II verifier can be seen in a twofold way. First, we have detected a number of defects, error occurrences and anomalies in the KB. The existence of these issues suggested several changes in the knowledge structure of the KB. These changes not only correct the error occurrences, but also improve the KB quality level, in the sense that now the KB is better structured and more robust and accurate. Second, as the verification process goes we are gaining confidence in PNEUMON-IA, because we learn that several error types are not present. This means that the KB was well developed. Absence of errors can be assured by the exhaustive verification performed by IN-DEPTH II.

So far we have verified around 50% of the KB contents. In this process we have made 42 changes, that are summarized in table 4-2. The first row in this table indicates whether the change was directly suggested by the verifier output or was decided after the expert analysis of this output. The second row shows the type of error that originated the changes: I, R and U stand for inconsistency, redundancy and useless objects respectively. We can see that most of redundancy and useless objects changes were directly suggested by IN-DEPTH II output. Considering inconsistencies, the reverse is true: most changes were made after expert analysis. This is a quite reasonable result, since redundancies or useless objects are less dependent on the domain semantics than inconsistencies. The impact of these changes in the KB in terms of the knowledge of pneumonia diagnosis is explained in the following.

VERIFIER			VERIFIER + EXPERT			TOTAL
I	R	U	I	R	U	
3	4	7	24	2	2	42

Table 4-2. Changes performed on PNEUMON-IA.

4.7.4.1 Major Changes in Knowledge Expression

The major changes in the knowledge contained in PNEUMON-IA have been originated when solving inconsistency error occurrences. To achieve their global correction, we had to modify not only the specific objects involved in them but also other semantically related objects in the problem domain.

Conceptually, the first goal in the overall PNEUMON-IA task is to classify a patient's pneumonia in the classes *bacterial* or *atypical*. This is made using the rules contained in the module `bact-atip`. This classification is also made by the physician at the beginning of the diagnostic process, when little information is available. From this first syndromic classification, the physician evaluates the different etiologies that can be included in these two classes. In the same way, PNEUMON-IA uses the conclusions obtained in the module `bact-atip` to decide the strategies that give the list of modules to visit, corresponding to specific etiologies (*pneumococus*, *legionella*, *mycoplasma*, etc.).

Checking a number of potential inconsistencies (I-2 type) provided by the expert in charge of KB development, we have detected a set of error occurrences caused by the excessive generalization of facts. Rules that had to be included in modules of specific etiologies were erroneously located in the more general-purpose module `bact-atip`. This error has been caused by the fact that, years ago, the prototype of typical bacterial pneumonia was the pneumococal pneumonia, and these terms were considered practically synonyms. When the features (clinical, radiological, etc.) corresponding to the different agents causing the pneumonias considered as typical bacterial pneumonias were discriminated, the concept of typical bacterial pneumonia was no longer synonymous of pneumococal pneumonia. Now, the clinical features attributed to typical bacterial pneumonia should be attributed (when possible) to the specific etiologic agents. Only those facts that are common to all the bacterial pneumonias should be included in the typical bacterial pneumonia syndrome. This structuration that usually is not explicited in the clinical practice should be clearly explicited in the KB to prevent errors due to excessive generalization.

We can see two examples of this excessive generalization. The first one involves herpes labialis and typical bacterial pneumonia, and the second involves relative bradycardia and atypical pneumonia. The presence of herpes labialis in a patient suffering from pneumonia indicates a high evidence of pneumococal pneumonia. The expert built a rule deducing typical bacterial pneumonia from herpes labialis, including it in the `bact-atip` module. Later, typical bacterial pneumonia was used to deduce the *pneumococus* etiology, but it was also used to deduce other etiologies such as *haemophilus* or *gramnegative bacili*, but this was incorrect. This error was detected by the verification process as an occurrence of inconsistency of type I-2. Its interpretation allowed us to correct it, removing herpes labialis as a condition for typical bacterial pneumonia and including it as a condition for pneumococal pneumonia.

PNEUMON-IA deduces the concept of relative bradycardia from the pulse rate and the temperature of a patient, with a certainty degree depending on the values of these facts. It is well known by the physicians that atypical pneumonias may occur with relative bradycardia. The expert gave a rule deducing atypical pneumonia from relative bradycardia, including it in the `bact-atip` module. However, in the consulted medical documentation, relative bradycardia appears as a symptom for only two of the etiologies included in atypical pneumonia: *chlamydia psittaci* and *virus*. For *mycoplasma* the relation is not so clear. In addition, relative bradycardia could also be applicable to *legionella*, included in typical bacterial pneumonia. Removing this error prevents the existence of false positives (*mycoplasma*) and false negatives (*legionella*). This error was also detected as an occurrence of inconsistency of type I-2, and it has been corrected in the same way as the first one.

4.7.4.2 Minor Changes in Knowledge Expression

We have made just minor changes in the knowledge structure of PNEUMON-IA to solve some error occurrences. Typical errors solved by minor changes are some redundancy types and specially useless objects. We have frequently employed two modification patterns. First, slight modification of the certainty degrees associated to rules, to allow for its conclusion to be deduced with a higher certainty degree. Second, the threshold of some conditions have been modified to allow to be satisfied in the adequate situations. Examples of minor changes when solving verification errors can be seen in section 4.7.2.3, solving useless objects.

4.8 Conclusions

The practical experience of PNEUMON-IA verification has been very fruitful. The expert has seen the power of automated verification, able to perform thousands of tests among rule chains to detect an error. This capacity, unreachable by manual testing, can be of great help for ES development and maintenance. On the other hand, the knowledge engineer developer of IN-DEPTH II has learnt that verification as a blind search for predetermined erroneous patterns has to be complemented by the domain knowledge provided by the expert. From this experience verifying PNEUMON-IA we extract the following conclusions:

- *Verification Utility*: verification really improves the quality of expert systems. PNEUMON-IA, an expert system that had been previously validated, is being improved by a systematic verification.

- *Error Causes and Error Ocurrences*: a verifier detects error occurrences that are different from error causes. An error occurrence has to be interpreted in terms of the domain knowledge to identify its real causes and obtain the best correction for it. It this task, the domain expert plays a fundamental role.

- *Faithful Representation*: the verification method has to include a faithful representation of the target expert system. Otherwise its results will be neither reliable nor complete, and the expert will lose confidence in it.

- *Extra Knowledge*: verification often requires knowledge that is not included in the KB. This extra knowledge has to be added as integrity constraints or by other means.

- *Adequate Functionality*: the verifier has to offer an adequate functionality for the verification task, allowing the user to check specific parts of the KB or perform specific tests. The incremental verification feature of IN-DEPTH II has been very useful in practice.

- *Verification after Updating*: each time the KB is modified, the verification process should be repeated. A small KB change may include a serious error able to generate ES malfunctions.

We have worked on an already developed ES, so our practical experience does not include verification during the development process. Nevertheless, we are convinced of the verification utility, so we include the following points regarding the verification role during ES development:

- *Early verification*: early verification allows for the development of structurally correct KB parts, since they can be tested as soon as they are developed.

- *Design for Verification*: including verification in ES development enforces the design for verification. In particular, the extra knowledge required for verification purposes can be collected and included in the KB during its construction.

In conclusion, verification is a very useful technique for ES validation, allowing for the detection of hidden errors in the KB. So far, exhaustive verification is the only technique assuring the absence of certain types of errors, what can be of great interest for critical applications. Verification has also some limits. It deals with formalizable requirements only, with particular emphasis on the structural correctness of the KB. Other techniques can be used to assess the degree of correctness in ES performance and functionality. Given that a KB free of structural errors is a common prerequisite for these techniques, verification appears as the first step to obtain a global validation for implemented ESs.

Chapter 5

Performance Improvement by Knowledge Base Refinement

As we have stated in the chapter 3, evaluation consists in checking the ES against its non-formalizable requirements. Usually, non-formalizable requirements are not very detailed, and their checking demands subjective interpretation and, sometimes, tentative experimentation. In one way or another, testing appears as an unavoidable technique in ES evaluation specially when considering the assessment of ES performance. In this chapter we propose the use of KB refinement techniques to evaluate and improve ES performance.

A common requirement for ESs consists in checking when the ES performance is at a human expert competence level. This non-formalizable requirement involves the assessment and comparison of ES performance and human expert competence. This is usually made by ES testing on known cases, and matching ES outputs against experts' opinions. When significant discrepancies for the initial requirement exist, the ES is manually updated and testing is repeated again (regression testing). This testing-and-update cycle is performed until a satisfactory performance is achieved.

Theory refinement considers the improvement of an approximate (inconsistent, incomplete or incorrect) domain theory from a library of cases with known solution. A refinement problem exists when some of these cases are treated incorrectly in the theory. Using machine learning techniques, the theory can be modified to achieve a correct treatment on all cases. It is assumed that the theory requires only minor changes to reach a satisfactory state, so the refinement process will maintain as much as possible the structure and vocabulary of the original theory. The set of cases should be a representative sample in the problem domain.

The testing-update cycle performed manually in evaluating ES performance can be automated using theory refinement techniques. In this case, the theory is the ES knowledge base (KB) and the process is known as KB refinement. This approach has been considered by several authors (see section 2.3) developing automatic KB refinement tools. These works have been mainly focused on developing different learning strategies to improve the KB, but the validity of the refined ES has not been considered in detail. Performance errors (false positives, false negatives) are considered of equal importance regardless of their impact in the ES task. These tools assume a relatively simple ES model and consider that a case solution consists of a single element.

Our approach considers KB refinement as an important technique in ES evaluation. It can automatically measure the ES performance on the case library, and more important, it can provide those modifications that improve the ES performance. In this sense, this work shares concepts with previous KB refinement tools, although the emphasis is put on the validity of the final ES instead on the learning capabilities of the refinement process. Since the main concern is to improve ES validity, performance errors are no longer considered of equal importance but they are weighted by their relative impact on the ES task. This impact depends on the type of error or on the elements involved in this error, aspects that are completely application-dependent. For an specific ES task, medical diagnosis, we consider three kind of errors: *false negatives*, *false positives* and *ordering mismatches*. A false negative occurs when a diagnostic is not present in the ES solution but it should. A false positive occurs when a diagnostic is present in the ES solution but it should not. An ordering mismatch occurs when the ranking order of diagnostics composing the ES solution is not correct. These errors have a different importance in the ES task. To maximize performance improvement the refinement process is guided by error importance, solving the most important errors first. When correcting an error, new errors of lower importance may be generated, but always assuring a neat performance gain.

Based on these ideas we have developed IMPROVER, a KB refinement tool working on the ES model described in section 4.1. Refinement operators have new effects when applied on this ES model. For instance, both false negatives and false positives may be solved by specialization and generalization operations. We are using IMPROVER to evaluate and improve the performance of PNEUMON-IA, using a library of 66 cases. The right solution for each case, the gold standard, is computed as a consensus among the opinions of five independent experts. Refinements suggested by IMPROVER are not automatically added to the KB but they are given to the expert, who accepts, rejects or modifies them. In this way, the knowledge integrity is assured. IMPROVER has shown to be quite effective in detecting and solving minor defects and anomalies contained in the rules, that would have been very difficult to detect by manual inspection and testing.

The structure of this chapter is as follows. Section 5.1 addresses the relation of KB refinement and validation, stressing that performance is not only measured by the number of errors, but also by their significance. Section 5.2 describes an specific ES task, medical diagnosis, identifying the types of errors and their relative importance. Section 5.3 addresses the problems of gold standard construction and solution matching, two questions prior to refinement generation. Section 5.4 contains strategies to effectively solve the three types of errors considered: false negatives, false positives and ordering mismatches. Section 5.5 describes how we have implemented these strategies on IMPROVER, an automated refinement tool. Section 5.6 considers the practical use of IMPROVER to refine PNEUMON-IA. Finally, section 5.7 encloses some conclusions about our practical experience of using refinement to validate and improve ESs.

5.1 Validation and Refinement

In a pure machine learning context, the goal of KB refinement is to decrease the number of errors detected executing the ES on the case library, using empirical learning techniques. When KB refinement is used for validation purposes, the goal is to improve ES performance, that is to say, the ES has to provide better recommendations after implementing each single refinement in the KB. This goal slightly differs from the machine learning one, since *better performance* is not equivalent to *performing a lower number of errors*. Different errors can have a different impact on the overall ES performance, so decreasing the number of errors does not always mean producing a better ES. To increase ES performance, KB refinement should be guided by error importance with respect to the ES task. Most serious errors has to be solved first, even by causing a larger number of less important errors but always assuring a neat gain in ES performance. A classification of error importance with respect to the ES task is needed. This classification is application dependent and may be based on the error type (different types of errors have a different importance for the ES task) and on the elements involved in an error (different elements in the problem domain have different importance, which is maintained when an specific element is involved in an error).

Considering the medical diagnosis domain, a false negative (a diagnosis that does not appear in the ES output but it should) is a more serious error than a false positive (a diagnosis that appears in the ES output but it should not). A false negative may cause that the actual origin of an illness be ignored. A false positive introduces an extra diagnosis (what can be seen as noise in the ES output) but does not cause missing the right one. Difference in error importance comes from the impact of each error on the ES task. In the same way, a false positive involving a very dangerous diagnosis is a more serious error than a false positive involving a diagnosis without risk for the patient life. The first false positive introduces more noise in the ES output than the second, because doctors devote more attention to diagnoses with higher risk for the patient life. Difference in error importance comes from the way the ES is used (ES pragmatics).

The usage of KB refinement for validation also differs from KB refinement in machine learning. Every modification introduced in the KB by automatic refinement has to be evaluated by the human expert responsible for the ES development. Detailed justification of the proposed modifications should be provided, detailing the solved errors as well as the potential new errors introduced. After expert's evaluation, the modification can be accepted, rejected or modified to include the expert's opinion without changing performance. Refinement failures have to be reported to the human expert with all the available information, since they leave unsolved errors that will be treated by hand. The knowledge engineer and the human expert may also be interested in the intermediate results of each refinement phase.

A KB refinement system can play a significant role in the validation process during the whole ES life-cycle. The two main functions of a KB refinement system, namely, the automatic testing facility and the learning capability, can be of great help at different development stages. As soon as a prototype has been built up, the automatic testing facility can establish the prototype performance level as the subset of cases correctly solved in the case library. At further stages, automatic testing will show how performance varies, and the learning capability can improve the updated prototype. On the final version, automatic testing will compute whether the achieved performance level complies with the user needs. At the maintenance stage, when modifications are made on the KB, automatic testing can check the impact of these modifications on the performance level, activating the learning capability if this level has decreased. Conversely, when new cases are known in the problem domain they are added to the case library. By automatic testing, a new performance level is computed and eventually enhanced by learning mechanisms. In this way, an accurate measure of the achieved ES performance level is always available as the subset of cases correctly solved and the complementary subset of cases incorrectly treated. This measure is close to the way in which humans are evaluated in tests or examinations: by measuring their ability to solve a set of problems. Obviously, a KB refinement system does not release the human expert or the knowledge engineer who are in charge of ES development from their responsibilities. They should control all the steps in the refinement process (case selection, refinement acceptance, KB updating, etc.) and they should solve the remaining unsolved issues. A KB refinement system automatizes a set of activities that would require lots of effort if they were performed manually, providing modifications that objectively improve the ES performance according to the classification of error importance considered for the specific ES task.

5.2 The ES Task

We assume the ES model described in section 4.1 (without any simplification). The ES output in a case C_i is H_i, a set of deduced hypotheses ordered by their certainty values. To test performance, H_i should be matched against the correct ordered set H_i*, with the following potential errors:

- false negative: $h \notin H_i$, $h \in H_i^*$
- false positive: $h \in H_i$, $h \notin H_i^*$
- ordering mismatch: $h \in H_i$, $h \in H_i^*$, $position\ (h, H_i) \neq position\ (h, H_i^*)$.

While false negatives and false positives have been extensively studied, ordering mismatch is a new type of error not considered previously in the literature. It is caused by the ordered structure of ES output.

The ES task we are going to analyze is medical diagnosis, specifically diagnosis of community acquired pneumonias. The ES goal is to obtain, from the patient's data, the subset of microorganisms that are more likely to cause the infection. In a very few cases this subset is formed by a single element, because usual symptoms do not discriminate enough to isolate a single cause. The classification of error importance is as follows:

false negative > false positive > ordering mismatch

We have already mentioned that a false negative is a more important error than a false positive in medical diagnosis, because a false negative may cause to miss the actual illness cause while a false positive can be seen as adding noise to the ES output. Obviously, both false negatives and false positives are more important than ordering mismatch, which simply indicates discrepancy in diagnostic ranking. No further classification is made based on specific elements for simplicity reasons, although some correspondence exists between microorganisms and potential patient risk. In general, typical bacterial pneumonias may be more serious than atypical pneumonias, although the evaluation of the relative importance for the above mentioned errors when considering specific etiologies is case dependent.

5.3 Refinement on the ES Model

When we apply refinement to our ES model (section 4.1), we find a number of practical issues. According to the previous description of KB refinement (section 2.3), a refinement process is composed of four phases: *identification*, *localization*, *generation* and *selection*. The ES model has no influence in the identification phase, but it has a great influence in the other three phases.

The localization phase aims at identifying those KB parts that could be responsible for an error. In the multi-level ES model considered, this implies that both control and domain knowledge have to be analyzed for error responsibility. In addition, this analysis has to follow the hierarchical relations existing between both types of knowledge. The generation phase has to build a number of adequate refinements to solve an error. In this phase, some risk of combinatorial explosion exists because of the large number of refinements that can be potentially adequate. This risk is specially high in the considered ES model. The presence of control knowledge (activating/inhibiting modules) and the existence of up-down rules combined with the cut-off action of the certainty threshold τ, provide many different possibilities for refinement generation. The selection phase has to choose those refinements considered as acceptable. Acceptability depends on the impact that a given refinement has on the ES task, in terms of the number and importance of solved errors with respect to the number and importance of new errors generated. In the following, we discuss in detail all these issues.

5.3.1 Refining Control and Domain Knowledge

In order to identify all the possible causes for an error, we have to analyze the whole KB. This implies that we have to consider the rules/metarules that have been fired in the ES execution causing the current error, as well as those rules/metarules that have not been fired but they would have prevented the error to occur. In this analysis, we have to consider the different actions performed by the considered KB objects.

Control and domain knowledge perform different actions on the ES. Control knowledge is responsible for the sequence of visited modules and the termination conditions, while domain knowledge is responsible for deducing new facts and assigning certainty degrees. In the localization phase, we can perform separate analyses for control and domain knowledge, according to their actions and the type of error considered. Both types of knowledge may be responsible for false negatives and false positives, while domain knowledge is the only responsible for ordering mismatches. Let *d* be a diagnostic contained in the module *m*. If *d* is a false negative, we have to check whether *m* has been visited or not during the ES execution. If not, metarules adding/removing *m* and metarules stopping the ES are potentially responsible for this error. Besides, we have to analyze the contents of module *m*, to check whether *d* would have been deduced if module *m* had been visited. Conversely, if *d* is a false positive, we have to analyze the metarules adding/removing *m*, that could be responsible for this error. In addition, the rules contained in *m* are directly responsible for deducing *d*, so they have to be analyzed. If *d* is an ordering mismatch, we do not have to analyze control knowledge dealing with *m* (it has been correctly visited), and we only consider the rules involved in the deduction of *d*.

Classically, false negatives have been solved by generalization operators while false positives have been solved by specialization operators. This is no longer true in the ES model, where both generalization and specialization operators may solve both types of errors. If *d* is false negative caused by a metarule *mr* removing *m* that is too general (it has been fired when it should not), the solving operator is a specialization of *mr*. Conversely, if *d* is a false positive caused by a metarule *mr* removing *m* that is too specific (it has not been fired when it should), the solving operator is a generalization of *mr*.

5.3.2 Heuristics to Control Refinement Generation

Considering refinement as a search process (where the goal is to find the best refinement for the current error), the generation phase determines the next points in the search space to be analyzed. In the ES model, the search space is very large. To prevent a blind and erratic search, we control refinement generation using some heuristics. They are based on the following assumption: the KB state is close to a correct state. Therefore, refinement aims at causing only minor changes in the KB structure and contents, maintaining as much as possible the original KB. Considering the assumed ES model, this initial assumption means that the KB is considered basically correct in the sense that single erroneous rules as wrong associations of conditions and conclusion do not exist. Errors are caused by KB incompleteness (missing conditions) or by small rule defects (incorrect *cvs*, conditions too specific or too general, etc.), that can affect both control and domain knowledge. The KB is assumed consistent, since consistency and other structural properties can be achieved using verifiers (see chapter 4). The refinement process pursues a tuning effect on the KB, by testing and learning on the case library. The heuristics controlling refinement generation are the following:

- *Minimal change*: those refinements proposing minimal changes with respect to the original KB state are preferred. This conservative criterion is supported by the initial purpose of performing small changes in the original KB. In addition, there is a limit on the elementary changes contained in an acceptable refinement. An elementary change is a change in a condition in the left-hand side of a rule or metarule, a change in the certainty value of a rule or metarule, or a change in the right-hand side of an up-down rule or metarule. The rationale for this limit is to prevent the generation of refinements causing major changes in the KB.

- *One object only*: we consider refinements causing modifications in one object only (rule or metarule). This heuristic is based on the fact that rules/metarules contain elementary associations of knowledge. When a refinement involves a single rule, the expert can easily provide an opinion. However, when a refinement is scattered in a rule chain, the expert finds more difficult to judge it. This heuristic simplifies the refinement generation, although it may cause to miss the right refinement.

- *Closeness*: refinements involving rules that are close in the deductive chain to the ES goal causing the error are preferred. In the assumed ES model this implies that refinement generation is, in some cases, limited to the rules contained in the module of the considered goal. This heuristic is based on the fact that refining a rule close to the target goal will cause little effect on other goals, minimizing undesired side-effects.

- *No deletion*: refinements cannot remove KB objects. This heuristic is based on the initial assumption of correctness for the KB rules/metarules. This assumption is quite reasonable for ES developed under some methodology, with the support of knowledge engineers and with available verification tools.

All the refinements satisfying these heuristics are generated. This is equivalent to making an exhaustive search on the vicinity of the current KB state. If no solution is found in this exhaustive search, we can enlarge the searching space by increasing the limit of elementary changes allowed and repeat the process. If no acceptable refinement exists, the problem is considered too difficult for refinement and it is passed to the knowledge engineer.

5.3.3 Legal Refinement Operators

According to the previous assumption of avoiding deletion operators in the refinement process, we restrict the set of legal refinements operators to the followings:

OP-1. Generalize/specialize conditions in the left-hand side of rules/metarules. A condition is generalized from a set of examples for which it is not satisfied but it should. The minimum change enabling the condition to be satisfied in all the considered examples is made. Conversely, a condition is specialized from a set of examples for which it is satisfied but it should not. The minimum change disabling the condition to be satisfied in any considered example is made.

OP-2. Modify the certainty degree of rules/metarules. In case of rules, the new certainty degree is computed to deduce the rule conclusion with a higher or lower certainty degree . The target certainty degree is determined by the refinement goal: surpass the threshold τ or satisfy a condition of another rule involving a certainty degree. In case of metarules, the new certainty degree is tentatively set to increase or decrease the possibility of a given module to be visited.

OP-3. Modify the certainty degree in conclusions of up-down rules. The new certainty degree decreases in one degree[1] the change in certainty performed by the original up-down rule.

OP-4. Modify the right-hand side of a metarule, adding or removing modules of the *add* part. No action is performed on the *remove* part.

OP-5. Add conditions to the left-hand side of rules/metarules. Conditions are added in case of specialization of a rule/metarule, from positive and negative examples. New conditions should be satisfied by all the positive examples but not by any negative example. The conjunction between the most specific discriminant between positive and negative examples and the left-hand side of the considered rule/metarule is computed, removing redundancies if they occur.

OP-6. Add new rules/metarules to the KB. New rules/metarules are learned from a set of positive and negative examples. These new rules/metarules should be satisfied by all the positive examples but not by any negative example. The most specific discriminant between positive and negative examples constitutes the left-hand side of a new rule/metarule. The right-hand side is provided by the refinement process.

Operators OP-2 and OP-3 dealing with *cv* change are used opportunistically, to adapt the ES behavior to a given pattern. Operators OP-2 used on metarules and OP-4 are always tentative. These operators try to change the set of modules visited in an execution. This set is computed dynamically and it is very difficult to calculate in advance the impact of a single change. Operators OP-5 and OP-6 cause the most drastic changes in the KB, and we use them as the last resort to solve an error.

5.3.4 Acceptance Criteria

Proposed refinements to solve an error may cause new errors. According to the classification of error importance with respect to the ES task (section 5.2),

false negative > false positive > ordering mismatch

[1] We consider a discrete set of possible certainty values.

we obtain the following acceptance criteria for a refinement in the selection phase:

AC-1. A modification solving m false negatives but causing n false positives is acceptable when $m \geq n$.

AC-2. A modification solving n false positives but causing m false negatives is acceptable when $n \geq m/\alpha$, where α is a constant relating the importance of a false negative with a false positive.

AC-3. A modification solving m ordering mismatches is acceptable when (i) it neither generates new false negatives nor false positives, and (ii) when it causes n new ordering mismatches, $m > n$.

These criteria determine when a refinement is considered acceptable, but they do not determine when a refinement has to be performed on the KB. Acceptable refinements are provided to the expert, who performs the formal acceptance, rejection or modification of the proposed refinement.

5.4 IMPROVER: A Tool for Knowledge Refinement

IMPROVER is a KB refinement tool designed to increase ES validity after each implemented refinement. Based on the classification for error importance (section 5.2), IMPROVER is composed of three stages: *solving-false-negatives*, *solving-false-positives* and *solving-ordering-mismatches*. Each stage is devoted to solve an specific type of error. These stages should be invoked sequentially, because solving an error may generate other errors of lower importance. Before each stage the error identification process is executed to determine precisely the errors remaining in the ES. After each stage, the human expert has to accept, reject or modify the changes proposed by the refinement process. The whole process is depicted in figure 5-1.

In the following the basic components of IMPROVER are detailed. Specific procedures used at each refinement stage are explained in terms of the legal set of operators {OP-1, OP-2, OP-3, OP-4, OP-5 OP-6} described in section 5.3.3.

#Sequence	Preprocess	Refinement Stage	Expert Action
1	Error Identification	Solving False Negatives	Modification acceptance
2	Error Identification	Solving False Positives	Modification acceptance
3	Error Identification	Solving Ordering Mismatches	Modification acceptance

Figure 5-1. IMPROVER refinement stages.

5.4.1 Error Identification

KB refinement is based on the existence of cases that are incorrectly treated by an ES. The identification of these cases is the first question for a refinement process. In order to perform this identification, we need to establish: (i) the correct solution (gold standard) for a case and (ii) how to match the ES output against the gold standard. IMPROVER addresses these two points in the following form.

5.4.1.1 Gold Standard

The first issue in error identification is the definition and selection of the right solution for a case, what is known as *gold standard* in the literature. The definition problem was considered in [Gasching et al, 83] providing two definitions: the gold standard for a case is either (i) the objective correct answer, or (ii) what a human expert (or a group) says is the correct answer, using the same information that is available to the ES. They adopted the second definition because the first one is not applicable in many occasions. We also take this approach, given the peculiarities of the problem domain. In medical diagnosis, the exact illness cause is often unknown, and in occasions can only be obtained by aggressive tests or by autopsy. Physicians usually diagnose without using these tests, so it seems quite reasonable for the ES to perform the diagnosis task in the same way. When objective correct answers are known, these can be used not as gold standards but as oracles in specific testing. Their usage in KB refinement is unclear because they have been obtained by methods unavailable to the ES.

To compute a gold standard the opinion of a group of experts is required for each case. These experts should be totally independent from ES developers in order to prevent bias. Experts usually disagree, so we need a consensus function to achieve a gold standard from their recommendations.

5.4.1.2 Solution Matching

Error identification is performed by matching the ES output against the gold standard for every case. ES output can be obtained by actual ES execution or by ES simulation. IMPROVER uses the second option for practical reasons. Data coming from ES execution and from KB refinement are treated in an integrated way. An independent analysis of domain and control knowledge is made, identifying defects that would be occluded by other defects in actual ES execution. All the satisfied rules/metarules are identified even if they are not used in the deduction. This information is recorded in data structures that will be used later, causing important savings of computational effort specially at the selection phase. Specifically, domain knowledge is represented by an *and/or tree*, where *and* nodes represent rules and *or* nodes represent facts in rule conclusions. Rule priority is recorded by the position of *and* nodes in an *or* node, decreasing from left to right. Control knowledge is represented in a separated *and/or tree*, where *and* nodes represent metarules and *or* nodes represent modules. Control knowledge imposes some dependencies on domain knowledge, but separate structures facilitates the analysis. Deduction details for all the deduced facts in all cases are recorded, summarizing errors in a table indexed by cases/diagnoses. These data structures allow refinement to focus on specific KB parts. Control knowledge can be refined independently from domain knowledge and refinement can concentrate on specific goals, always considering the whole case library.

5.4.2 Solving False Negatives

The first refinement step of IMPROVER is the solving-false-negatives stage, devoted to solve the most serious error in the ES task. A refinement solving false negatives is considered acceptable when it satisfies the criterion AC-1, that is to say, when the number of false negatives solved is higher or equal than the number of false positives caused.

Anyway, the final decision about refinement acceptance and its actual implementation on the KB depends on the expert responsible for the ES development, who guarantees the integrity of KB contents. In the following, we describe the different phases of this stage.

5.4.2.1 Localization

When IMPROVER detects a false negative diagnosis *d*, it analyzes both control and domain knowledge to locate the error causing part in the KB. Control knowledge is responsible for the false negative when the module *m* containing *d* has not been visited for deduction. This can happen by one of the following causes:

FN-1. No metarule adding *m* has been satisfied.
FN-2. A metarule removing *m* has been satisfied and fired.
FN-3. Execution has terminated before *m* has been visited.

Domain knowledge is responsible for the false negative when, assuming that *m* has been visited, *d* has not been deduced. This can happen by one of the following causes:

FN-4. One or several rules required to deduce *d* have not been satisfied.
FN-5. Threshold τ has cut the deduction for *d*.
FN-6. An up-down rule has decreased *d* certainty below τ.

Causes related to control or domain knowledge can coexist. For causes requiring generalization, FN-1 and FN-4, the partial proof trees for the unsatisfied rules/metarules are computed and ordered by the number of assumptions required to complete the proof (that is to say, to satisfy the corresponding rules/metarules). For each specific cause, the minimum number of assumptions required to complete the partial proof trees are selected and filtered, eliminating inconsistent or redundant proof trees. The remaining partial proof trees are passed to the generation/selection phases. For the rest of cause types, the responsible rules/metarules are located and passed to the next generation/selection phases. When all false negatives for *d* have been processed, they are grouped by cause type. Inside each type, specific refinements are ordered by their number of occurrences in the set of causes.

5.4.2.2 Generation/Selection

These phases process the list of causes generated in the localization phase. Each cause type is treated in the following way:

FN-1. Unsatisfied conditions in the partial proof trees are tentatively generalized using the operator OP-1. If the generalization satisfies the criterion AC-1 (see section 5.3.4), it is accepted. Otherwise, one or several metarules adding *m* are inductively learned using the operator OP-6, taking as positive examples the false negatives that are being solved and as negative examples the false positives caused by the previous unsuccessful generalization.

FN-2. Conditions in metarule *mr* removing *m* are specialized using the operator OP-1. If the specialization satisfies the criterion AC-1, it is accepted. Otherwise, conditions are added to *mr* using the operator OP-5, taking as positive examples the true negatives in which *mr* has been fired, and as negative examples the false negatives that are being solved.

FN-3. Refinement aims at *m* to be visited in the ES execution. Refinement tries to locate *m* in a better position in the active module list (the list of modules that have been activated by metarule action, from which the current module is selected, see sections 4.1.3 and 4.1.4). The following tentative modifications are performed of the fired metarules in the ES execution. The certainty degree of a metarule adding *m* is increased by the operator OP-2. If module *m* appears in the *add* part of a metarule, it is located at the beginning of the *add* part by the operator OP-4. If the module *m* does not appear in the *add* part of a metarule, it is located at the end of the *add* part by the operator OP-4.

FN-4. Unsatisfied conditions in the partial proof trees are tentatively generalized using the operator OP-1. If the generalization satisfies the criterion AC-1, it is accepted. Otherwise, one or several rules concluding *d* are inductively learned using the operator OP-6, taking as positive examples the false negatives that are being solved and as negative examples the false positives caused by the previous unsuccessful generalization.

FN-5. The certainty value of the fact *f* responsible for threshold action is increased. This can be made by either increasing the *cv* of rules concluding *f* using the operator OP-2, or increasing the *cv* of facts supporting *f*, by the operators OP-2 and OP-3. Fact *f* is identified as the deducible fact with lowest *cv* in *lhs*(*r*), being *r* the rule where the deduction has been cut. These changes are evaluated under the criterion AC-1 and accepted in case of positive answer.

FN-6. When an up-down rule *r* is responsible for a false negative, three refinement actions are tried. First, *r* is smoothed making smaller the certainty subtraction using the operator OP-3. Second, the *cv* of *d* before *r* is applied is increased, following the FN-5 procedure. And third, *r* is specialized following the FN-2 procedure.

IMPROVER treats all causes of false negatives, generates all the refinements satisfying the conditions in section 5.3.2 and selects those refinements considered acceptable. The solving-false-negatives stage terminates producing as output the set of acceptable refinements.

5.4.3 Solving False Positives

The second refinement step of IMPROVER is the solving-false-positives stage, devoted to solve the second error in importance for the ES task. A refinement solving false positives is considered acceptable when it satisfies the criterion AC-2, that is to say, when the number of false positives solved compensates the number of false negatives caused. This depends on the factor α, that is application-dependent. Again, the final decision about refinement acceptance depends on the expert responsible for the ES development. In the following, we describe the different phases of this stage.

5.4.3.1 Localization

When IMPROVER detects a false positive diagnosis *d*, it analyzes both control and domain knowledge to locate the error causing part in the KB. Contrary to the false negative case, there are not definite but tentative causes since there is no evidence of what rule/metarule should not be fired to prevent *d* to be deduced. Control knowledge is responsible for the false positive when the module *m* containing *d* has been visited but it should not. This can happen by the following causes:

FP-1. A metarule adding *m* has been fired but it should not.

FP-2. No metarule removing *m* has been satisfied.

FP-3. Execution has terminated after *m* has been visited.

Domain knowledge is responsible for the false positive when, assuming that *m* has been correctly visited, *d* should not be deduced. This can happen due to the following causes:

FP-4. Rules required to deduce *d* have been satisfied.
FP-5. The certainty degrees of rules used to deduce *d* is too high to be cut by τ.
FP-6. An up-down rule has increased *d* certainty.

Given that no definite evidence of the actual cause exists (except for FP-6), all the possible causes for a false positive are considered. For the cause FP-2 requiring generalization, the partial proof trees requiring a minimum number of assumptions are computed and passed to next phases. For the rest of causes, responsible rules are located and passed to the next phases. After all false positives are processed, causes are grouped by cause type. Inside each type specific refinements are ordered by their number of occurrences in the set of causes.

5.4.3.2 Generation/Selection

These phases receive a list of causes for all false positives of a diagnosis, grouped by cause type. Each type is treated in the following way:

FP-1. Conditions in metarules *mr* adding *m* are specialized by the operator OP-1. If the result satisfies the criterion AC-2, it is accepted. Otherwise, each metarule *mr* is specialized by the operator OP-4, taking as positive examples the true positives in which *mr* has been fired, and as negative examples the false positives that are being solved.

FP-2. Unsatisfied conditions in the partial proof trees are tentatively generalized using the operator OP-1. If the generalization satisfies the criterion AC-2, it is accepted. Otherwise, one or several metarules removing *m* are inductively learned using the operator OP-5, taking as positive examples the false positives that are being solved and as negative examples the true positives caused by the previous unsuccessful generalization.

FP-3. Refinement aims at avoiding to visit *m* during the ES execution. Refinement tries to locate *m* in a worse position in the active module list (sections 4.1.3 and 4.1.4). The following tentative modifications are performed of the fired metarules adding *m* during the ES execution. The certainty degree of a metarule adding *m* is decreased by the operator OP-2. If module *m* appears at the beginning of the *add* part of a metarule, it is located at the end of the *add* part by the operator OP-4. If the module *m* does not appear at the beginning of the *add* part of a metarule, it is removed of the *add* part by the operator OP-4.

FP-4. Conditions in rules *r* deducing *d* are specialized by the operator OP-1. If the result satisfies the criterion AC-2, it is accepted. Otherwise, each rule *r* is specialized by the operator OP-4, taking as positive examples the true positives in which *r* has been fired, and as negative examples the false positives that are being solved.

FP-5. The certainty degrees of rules involved in the deduction are decreased by the operator OP-2. These changes are accepted if criterion AC-2 is satisfied.

FP-6. When an up-down rule *r* has increased the *cv* of a false positive, it is specialized following the procedure described in FP-1.

IMPROVER treats all causes of false positives, generates all the refinements satisfying the conditions in section 5.3.2 and selects those refinements considered acceptable. The solving-false-positives stage terminates producing as output the set of acceptable refinements.

5.4.4 Solving Ordering Mismatches

The third and last refinement step of IMPROVER is the solving-ordering-mismatches stage, devoted to solve the lowest serious error for the ES task. A refinement solving ordering mismatches is considered acceptable when it satisfies the criterion AC-3, that is to say, when it solves some ordering mismatches without causing new false negatives nor positives. The final decision about refinement acceptance depends on the expert responsible for the ES development. In the following, we describe the different phases of this stage.

5.4.4.1 Localization

An ordering mismatch for a diagnosis *d* means that *d* is misplaced in the ES output with respect to the gold standard. Since ES output is ordered by *cvs* of diagnoses, an ordering mismatch for *d* implies that its *cv* is too high/too low and it should be modified accordingly. When an ordering mismatch is detected, only domain knowledge is analyzed because control knowledge has no effect on *cvs* of diagnoses. An ordering mismatch can happen by one of the following causes:

OM-1. The certainty degree of rules used to deduce *d* are too high/too low.
OM-2. An up-down rule has incorrectly increased/decreased *d* certainty.

For each cause, responsible rules are located and passed to the next phases.

5.4.4.2 Generation/Selection

These phases receive a list of causes for all ordering mismatches of a diagnosis, grouped by cause type. Each type is treated in the following way:

OM-1. The certainty degree of rules involved in the deduction of *d* are decreased/increased using the operator OP-2.

OM-2. When an up-down rule *r* has incorrectly increased/decreased the *cv* of *d*, two refinement actions are tried. First, *r* is smoothed decreasing the change in certainty performed by the up-down, by the operator OP-3. Second, the *cv* of *d* before *r* is applied is decreased/increased, following the OM-1 procedure.

IMPROVER treats all causes of ordering mismatches, generates all the refinements satisfying the conditions in section 5.3.2 and selects those refinements considered acceptable. The solving-ordering-mismatches stage terminates producing as output the set of acceptable refinements.

5.5 Refining PNEUMON-IA

We have used IMPROVER to enhance the performance level of the expert system PNEUMON-IA [Verdaguer 89]. As it has been said before, this system performs etiological diagnosis of community acquired pneumonias in adults. It is composed of 500 facts, 600 rules and 100 metarules distributed in 25 modules considering 22 different etiologies. PNEUMON-IA was validated after its construction by a team of 5 independent experts in the field using a test set composed of 66 cases, showing a performance level comparable to that of human experts.

The available information and the state of PNEUMON-IA were quite adequate to use a refinement tool. On one hand, we disposed of a library of 66 cases with the recommendations of five independent experts. Using this valuable information, we computed the gold standard for each case as a consensus among the experts' recommendations. On the other hand, the PNEUMON-IA state was assumed to be close to a correct one. PNEUMON-IA had been manually validated with good results, but the addition of new knowledge (a new etiological agent *Chlamidia Pneumoniae* was discovered) and the use of the IN-DEPTH II verifier had caused a number of modifications in the KB. The impact of these modifications in the performance of PNEUMON-IA was unknown. A refinement tool like IMPROVER was adequate to evaluate and improve as far as possible the performance level with respect to the gold standard.

IMPROVER has followed the heuristics to control refinement generation exposed in section 5.3.2. The limit of elementary changes in an acceptable refinement was set to 1. That is to say, to solve an error, IMPROVER generates refinements causing one elementary change in one rule or metarule only. This is the *minimum possible change* in the KB. This restrictive heuristic has been very appropriate to limit the number of refinements, and regarding IMPROVER results, it has showed an acceptable effectiveness in detecting causes to solve errors.

5.5.1 Gold Standard

The initial step in the refinement of PNEUMON-IA was to select the gold standard for the 66 cases contained in the library, from the recommendations of the five experts. The consensus function, mentioned in section 5.4.1.1, takes as input the five recommendations for a given case, providing as output the gold standard for that case. An expert recommendation is a list of diagnoses, qualified with certainty degrees and ranked by decreasing certainty. A first issue for the consensus function is the treatment of certainty degrees provided by the experts. There is no evidence for a consistent use of certainty degrees among them. On the contrary, some experts have systematically used high certainty degrees (*very-possible*, *quite-possible*) to qualify their diagnoses, while others have used a homogeneous distribution (see section 7.3.5 in [Verdaguer 89]). This fact suggested us to ignore the specific value of the certainty degree used to qualify a diagnosis, but keeping the order in which an expert has ranked their diagnoses for a case by decreasing certainty. This approximation may seem too drastic, since the relative qualification of diagnoses is lost. However, taking into account that the consensus function groups experts recommendations for *the same case*, grouping diagnoses by their relative position is, in fact, quite reasonable.

Then, an expert recommendation is just a ranked list of diagnoses and the consensus function will produce another ranked list as the gold standard for each case. No preference exists among experts; all of them are considered as equally good. The consensus function works as follows. Each diagnosis in the gold standard is assigned to the position obtained by computing the position mean value of the diagnosis in the experts' recommendations. Diagnoses with very close positions are grouped in a class, assigned to the mean value of the corresponding positions. The last class in the consensus is eliminated, because it corresponds to diagnoses only mentioned by one or two experts in a low position.

5.5.2 False Negatives

Solving false negatives is the major goal of IMPROVER when refining PNEUMON-IA, given that they are considered as the most serious errors. The acceptance of a refinement depends on the satisfaction of the AC-1 criterion (the number of false negatives solved has to be higher than the number of false positives generated), plus the acceptance of the expert developer of PNEUMON-IA, who guarantees the integrity of the knowledge in the KB.

At the beginning of the refinement process, PNEUMON-IA showed 85 false negatives and 87 false positives in the whole case library. After the solving-false-negatives phase, PNEUMON-IA exhibited 43 false negatives and 95 false positives, that is to say, solving 42 false negatives we have caused only 8 new false positives. This change means a decrease of 49% in false negatives, and a small increase of 9% in false positives. Clearly, solving almost half of the total number of false negatives by causing less than ten percent of increase in false positives represents an important gain in PNEUMON-IA performance. In the following, specific occurrences of accepted refinements solving false negatives are explained.

5.5.2.1 False Negatives: Refining Control Knowledge

The expert has accepted 7 refinements dealing with control knowledge to solve false negatives. All of them enlarge the list of adding modules in metarules. In the following we explain an actual occurrence of the accepted refinements.

Consider the metarule `r25900` acting on modules described in figure 5-2. It establishes the list of modules to be visited when the patient is `alcoholic`. In this list, the module `hemoph` is not included, and for this reason the `haemophilus` etiology is not considered in a number of cases when it should. The sign `alcoholic` suggests to visit the `haemophilus` module, although it does not provide a definitive evidence for it. IMPROVER added `hemoph` to `r25900`, realizing that the number of false negatives solved was higher than the number of false positives created (satisfying criterion AC-1). The modification was accepted by the expert.

```
r25900:   if   alcoholic   then  all   bact-atip  pneum  enterobact
                                       legion  anaerob  tbc  staph
                                       str-pio
```

Figure 5-2. Metarule on modules causing false negatives due to a missing module.

5.5.2.2 False Negatives: Refining Domain Knowledge

The expert has accepted 32 refinements in the domain knowledge to solve false negatives. These refinements are of three types: (i) generalization of a condition in the left-hand side of a rule, (ii) increase the certainty degree of a rule, and (iii) making smaller the decrease in certainty of an up-down rule. An example of each refinement type is explained in the following.

Consider the rule `r05005c` described in figure 5-3. It deduces `legionella` from `cough` and **not** `expectoration`, when `patient-status` is `serious` or `very-serious`. However, in the case library there are several cases with `cough`, **not** `expectoration` and `patient-status = moderate`, for which the gold standard includes `legionella`. IMPROVER included `moderate` as another legal value for `patient-status` in `r05005c`, solving eight false negatives and causing two new false positives, so criterion AC-1 was satisfied. The expert accepted the modification.

```
r05005c:  if   patient-status = serious or very-serious
               cough
               not expectoration  then   legionella  moderately-possible
```

Figure 5-3. Rule causing false negatives due to the condition on `patient-status`.

Consider the rules r14015 and r14024 described in figure 5-4. Rule r14015 concludes tuberculosis with *slightly-possible* as maximum certainty degree. When tuberculosis is concluded and Ziehl-Nilssen-stain is negative, the up-down rule r14024 decreases the certainty of tuberculosis in one degree. In the case library, a case including tuberculosis in its gold standard satisfied both rules. For that case, the combined effect of rules r14015 and r14024 made that tuberculosis was concluded with a certainty of *very-small-chance*, and it was cut off by the action of the threshold[2]. IMPROVER generated several refinements to solve this case, and the only one satisfying criterion AC-1 was to change the certainty degree of r14015 to *moderately-possible*. Then, after firing r14024 tuberculosis was concluded with a certainty of *slightly-possible*, over the threshold action. The expert accepted the modification.

```
r14015:  if   bacterial
              chronic-obstructive-pulmonary-disease
              silicosis          then  tuberculosis  slightly-possible

r14024:  if   tuberculosis
              Ziehl-Nilssen-stain = negative
                                 then  decrease tuberculosis 1
```

Figure 5-4. Rule pair causing a false negative because of r14015 certainty.

Consider the up-down rule r03029y described in figure 5-5. It decreases in two degrees the certainty of bacterial when the number of leukocytes is higher than 4000 but lower than 9000. IMPROVER realized that this change was too drastic, causing four false negatives in etiologies requiring bacterial as condition. IMPROVER suggested to decrease in one degree the change in certainty of this rule, solving these four false negatives and causing one false positive (satisfying criterion AC-1). The expert accepted the modification.

```
r03029y: if   bacterial
              leukocytes > 4000
              leukocytes < 9000       then   decrease bacterial 2
```

Figure 5-5. Rule causing false negatives because of its change in certainty.

[2] The threshold in PNEUMON-IA is *very-small-chance*.

5.5.3 False Positives

Solving false positives is the second goal in importance of IMPROVER, given that they are considered as a less serious error than false negatives. The acceptance of a refinement depends on the satisfaction of the AC-2 criterion, that relates the importance of false negatives and false positives with a constant α. We considered $\alpha = 2$, that is to say, a refinement is acceptable when the number of false positives solved is al least twice the number of new false negatives caused. Again, the final acceptance depends on the expert developer of PNEUMON-IA.

At the beginning of the solving-false-positives stage, PNEUMON-IA showed 43 false negatives and 95 false positives in the whole case library. After the solving-false-positives stage, PNEUMON-IA exhibited 43 false negatives and 78 false positives, that is to say, we have solved 17 false positives without causing any new false negative. This change means a decrease of 18% in the total amount of false positives. This result represents a neat, although modest, gain in PNEUMON-IA performance. In the following, specific occurrences of accepted refinements solving false positives are explained.

5.5.3.1 False Positives: Refining Control Knowledge

The expert has accepted a single refinement dealing with control knowledge to solve false positives. It shortens the list of adding modules in a metarule, and is explained in the following.

Consider the metarule `r25882a` described in figure 5-6. It establishes the list of modules to be visited when `bacterial`, the typical bacterial pneumonia syndrome, has been deduced. In this list, the module `clam` is included, and for this reason the *chlamidia* etiology is considered in a number of cases when it should not. The sign `bacterial` does not provides enough evidence to consider *chlamidia* as etiology. In addition, the metarule `r25883a` establishes the list of modules to be visited when `atypical`, the atypical pneumonia syndrome, is present. This list includes `clam`, and when these two metarules are fired in the same session, the module `clam` appears with too priority. IMPROVER removed `clam` from `r25882a`, decreasing in three the number of false positives without creating any false negative (satisfying criterion AC-2). The modification was accepted by the expert.

```
r25882a:  if   bacterial   then  add   pneum legion clam enterobact
                                       anaerob hemoph staph str-pio
                                       meningo branha tbc

r25883a:  if   atypical    then  add   myco viruses clam fq
```

Figure 5-6. Metarule on modules causing false positives due to an extra module.

5.5.3.2 False Positives: Refining Domain Knowledge

The expert has accepted 8 refinements in the domain knowledge to solve false positives. These refinements are of three types: (i) specialization of a condition in the left-hand side of a rule, (ii) decrease the certainty degree of a rule, and (iii) making smaller the positive change in certainty of an up-down rule. An example of each refinement type is explained in the following.

Consider the rule `r18006` described in figure 5-7. It deduces `chlamidia` from `atypical` and `dispnea`, when `patient-status` is not `slight`. However, in the case library there are two cases satisfying the left-hand side of `r18006`, for which the gold standard does not include `chlamidia`. IMPROVER realized that the `patient-status` value for these cases was `moderate`. Including moderate as another forbidden value in `r18006`, these two false positives were solved without causing any new false negative, satisfying criterion AC-2. The expert accepted the modification.

```
r18006:  if   patient-status ≠ slight
              atypical
              dispnea      then   chlamidia    moderately-possible
```

Figure 5-7. Rule causing false positives due to the condition on `patient-status`.

Consider the rule `r13003` described in figure 5-8. It concludes `streptococcus-pyogenes` with certainty *slightly-possible.* In the case library there is a case that satisfies this rule and `streptococus-pyogenes` is not included in its gold standard. IMPROVER suggested to change the rule certainty to *very-small-chance,* the threshold

```
r13003:  if    bacterial
               chronic-obstructive-pulmonary-disease
               sputum = blood stained
               bronchopneumonic-inflitrates
                          then   streptococcus-pyogenes slightly-possible
```

Figure 5-8. Rule causing false positives due to its certainty.

value. This change solved the false positive without causing new false negatives, satisfying criterion AC-2. The expert accepted the modification.

This change does not make useless the rule `r13003`, since if it is fired when other rules concluding `streptococcus-pyogenes` with certainty *very-small-chance* are also fired, the certainty of `streptococcus-pyogenes` will increase by the action of *cv-disjunction* function (section 4.1.2) and it will pass the threshold. Only when `r13003` is the only rule fired concluding `streptococcus-pyogenes`, the propagation of this diagnosis as a final one will be cut by the threshold action.

Finally, consider the rule `r14019` described in figure 5-9. It increases in two degrees the certainty of `tuberculosis` when `xr-old-tuberculosis-pattern` are present. However, this increase in certainty was too high, causing a false positive in the case library. IMPROVER suggested to limit the certainty increment to one degree, solving the false positive without creating new false negatives, satisfying criterion AC-2. The expert accepted the modification.

```
r14019:  if    tuberculosis
               xr-old-tuberculosis-pattern
                            then   increase tuberculosis 2
```

Figure 5-9. Rule causing false positives due to its change in certainty.

5.5.4 Ordering Mismatches

We have defined an ordering mismatch for a diagnosis *d* as the misplacement of *d* in the ES output with respect to the gold standard. This definition assumes that the diagnosis ranking of the gold standard is perfect. In PNEUMON-IA, we cannot assure that the ranking of the computed gold standard is perfect. While we can consider the gold standard as a good estimation of the correct solution for a case in terms of false negatives and false positives, we cannot guarantee the ranking to the extent of assuring that diagnosis *d* should be in the i-th position in the gold standard. The reason for that relies on the inexact nature of the experts recommendations, from which the gold standard is computed (see section 5.5.1).

Although we cannot assure that the *absolute* positions of diagnoses in the gold standard are totally correct, we think that their *relative* positions are correct. In other words, if *chlamidia* appears in the second position of the gold standard for a given case, we cannot flag as erroneous when *chlamidia* appears in the third position of the PNEUMON-IA output for that case. However, if *mycoplasma* appears above *chlamidia* in the gold standard but below *chlamidia* in the PNEUMON-IA output, this is an error that should be corrected. This situation, called *inversion*, occurs when the relative position of two diagnoses in the gold standard and in the PNEUMON-IA output is the opposite. We have tried to solve only inversions at the solving-ordering-mismatches stage. From now on, ordering mismatches will be restricted to inversions.

The expert has accepted 6 refinements to solve ordering mismatches. All of them modify the certainty degree of a rule, considered too high or too low. In the following, an example of these modifications is provided.

Consider the rule `03010` described in figure 5-10. It concludes `bacterial` with *possible* as certainty degree, when `sputum` is `purulent`. The certainty of this rule is too low, and it causes an ordering mismatch between the diagnoses *pneumococus* and *mycoplasma* in a case (`bacterial` is a condition for *pneumococus*). IMPROVER suggested to change the certainty degree to *quite-possible*, solving this inversion and without causing any other error (satisfying criterion AC-3). The expert accepted the modification.

```
r03010:   if    sputum = purulent       then   bacterial    possible
```

Figure 5-10. Rule causing an ordering mismatch due to its certainty.

Solving-ordering-mismatches has solved 1 false negative, 2 false positives and 3 ordering mismatches. False negatives and false positives have been solved by the effect that changing rule certainty degrees has had on firing stopping metarules. Changing rule certainty has caused modifications in the certainty degree of final diagnoses. Some stopping metarules contain conditions on the certainty of final diagnoses. So, changes in rule certainty degrees have originated that some stopping metarules have been early fired, solving two false positives. In the same way, another stopping metarule has not been fired in a case, allowing PNEUMON-IA to visit more modules and solving a false negative.

5.5.5 Performance Improvement

The action of IMPROVER has enhanced the performance of PNEUMON-IA over the case library. The performance gain is measured as the number of false negatives, false positives and ordering mismatches that have been solved by IMPROVER modifications. In the following, we discuss the performance gain from two points of view: (i) assessing the relative importance of the different IMPROVER stages on this gain, and (ii) comparing the final performance of PNEUMON-IA against the performance of the five human experts that have been used to compute the gold standard.

5.5.5.1 Performance Improvement of IMPROVER Stages

Before the use of IMPROVER, the performance of PNEUMON-IA on the case library was the following: 85 false negatives, 87 false positives and 13 ordering mismatches. PNEUMON-IA provided a total of 222 diagnoses for the whole case library, while the gold standard gave 218 diagnoses. Since the case library is composed of 66 cases, both PNEUMON-IA and the gold standard gave approximately 3.3 diagnoses for a case on the average.

The results of the three stages of IMPROVER are contained in table 5-1, where percents are rounded to next integer. The solving-false-negatives stage decreases by 49% the number of false negatives, increasing by 9% the number of false positives. Ordering mismatches also increases by 77%. The global evaluation of this stage is clearly very positive, since the number of false negatives solved is considerably higher than the number of new false positives created (42 versus 7). In addition, the percent of false negatives solved (49%) is quite important. The solving-false-positives stage does not cause any change in false negatives, and it decreases by 18% the number of false positives. Ordering mismatches decreases by 9%. This stage also provides a neat performance gain, solving

	PNEUMON-IA (initial)	PNEUMON-IA (after S-F-N)	PNEUMON-IA (after S-F-P)	PNEUMON-IA (after S-O-M)
#FN	85	43 (-49%)	43 (0%)	42 (-2%)
#FP	87	95 (+9%)	78 (-18%)	76 (-3%)
#OM	13	23 (+77%)	21 (-9%)	18 (-14%)

Table 5-1. Change in performance of PNEUMON-IA caused by IMPROVER stages.

17 false positives without causing any false negative. The solving-ordering-mismatches stage solves 1 false negative, 2 false positives and decreases in 3 the number of ordering mismatches (-14%).

Comparing the initial and final versions of PNEUMON-IA, the overall improvement in performance is clear. After the use of IMPROVER, the number of false negatives has decreased by 51%, the number of false positives has decreased by 13%, while the number of ordering mismatches has increased by 38%. This gain is quite important for an automatic procedure, specially if working on an ES providing multiple diagnoses.

5.5.5.2 Performance Improvement with Respect to Human Experts

To accurately evaluate the performance level of PNEUMON-IA, we have to compare it against human competence. To make this comparison, we express the recommendations of the five independent human experts for every case in terms of false negatives, false positives and ordering mismatches with respect to the gold standard. We remind that the gold standard has been computed as a consensus among the opinions of these five experts. These data and the performance of PNEUMON-IA, before and after the use of IMPROVER are recorded in table 5-2, where #FN, #FP, #OM and #DIAG stand for the number of false negatives, false positives, ordering mismatches and the total number of diagnoses respectively.

	Expert 1	Expert 2	Expert 3	Expert 4	Expert 5	PNEUMON-IA (before)	PNEUMON-IA (after)
#FN	69	60	57	118	67	85	42
#FP	38	32	45	18	31	87	76
#OM	25	31	23	11	20	13	18
#DIAG	187	190	206	118	182	222	252

Table 5-2. Comparison of performance among five human experts and PNEUMON-IA.

Regarding false negatives, PNEUMON-IA (before) surpasses in performance to a human expert only (expert 4), while the other four experts exhibit a better performance. However, PNEUMON-IA (after) surpasses in performance *to all the human experts*. This surprising result shows clearly the usefulness and power of IMPROVER, as well as the quality of the knowledge contained in PNEUMON-IA. We have to stress that all the modifications performed have been previously accepted by the expert who developed PNEUMON-IA, who did not accept all the proposed changes satisfying the criterion AC-1. In fact, IMPROVER proposed changes that decreased the number of false negatives to 37, but some of them were not accepted because they had no meaning from the medical point of view.

Regarding false positives, all the experts surpass in performance to both PNEUMON-IA (before) and PNEUMON-IA (after). In spite of the decreasing 11 false positives, this is not enough to obtain a better performance than human experts. According to these data, experts are more focused on the final diagnosis than PNEUMON-IA, they consider less potential diagnoses than the system. This strategy allows them to have a relatively low number of false positives, at the expense of a higher number of false negatives. PNEUMON-IA follows a more conservative approach, considering all the diagnoses for which some initial evidence exists.

Regarding ordering mismatches, both PNEUMON-IA (before) and PNEUMON-IA (after) surpass in performance to all the experts except for expert 4. This expert has a very low number of ordering mismatches because he provided a low number of diagnoses, 118 versus 218 diagnoses in the gold standard. Taking into account that PNEUMON-IA (after) gives the highest number of diagnosis, surpassing in performance to all except one expert is a very good result. In addition, the ratio of ordering mismatches versus total number of diagnoses of PNEUMON-IA (after) is lower than this ratio for all the experts. This means that the ranking of PNEUMON-IA outputs has *the highest degree of agreement* with the ranking provided by the gold standard.

In summary, we can say that PNEUMON-IA (after) offers a performance that is clearly comparable to human expert competence with respect to the gold standard. In some aspects, PNEUMON-IA offers better performance than human experts. The action of IMPROVER has been essential to achieve these good results, confirming the important role that automatic refinement tools can play in the ES validation process.

5.5.6 Evaluation of the Refinement Process

Achieving good results in terms of performance is not enough to guarantee the correctness of a refinement process. We have to analyze a number of aspects in the process in order to check that the performance gain is actually true with respect to the problem domain, and it is not a consequence of parameter manipulation. In the following, we discuss three aspects of the refinement process to assess its degree of quality and reliability: case library extension, gold standard quality and analysis of IMPROVER behavior.

5.5.6.1 Case Library Extension

The case library is composed of 66 cases that were collected in the files of four hospitals in the neighborhood of Barcelona [Verdaguer 89]. To select a case, it had to be a community acquired pneumonia[3] on an adult patient, with an adequate documentation including an X-ray film of the patient's chest. No other condition was required. The selected cases consider a wide spectrum of patients, without an appreciable bias in age, sex, patient status or other characteristics. They can be considered as a representative sample of the population of community acquired pneumonias in adults in the Barcelona area [Verdaguer 89].

To which extent does this case library test the contents of PNEUMON-IA? Regarding the number of cases, 66, it is higher than the number of cases used to test MYCIN (10) and R1

[3] A pneumonia is acquired in the community when it has been contracted out of a hospital. Later, the patient may enter into a hospital for medical care. This is what happened for the 66 cases of the library.

(50) (see section 2.6). Regarding the number of diagnoses effectively tested, the gold standard gives 13 different diagnoses in the whole case library while PNEUMON-IA (before) gives also 13 but differing in one with the gold standard. Given that PNEUMON-IA consider 22 different etiologies, this means that 8 etiologies have not been tested[4]. These 8 etiologies are quite infrequent, and their absence does not question to a significant extent the overall validity of the refinement process. On the contrary, the main pneumonia causes, *pneumococus*, *mycoplasma*, *legionella* and *virus* [5], have been extensively tested. Therefore, we conclude that this case library tests to a reasonable extent the PNEUMON-IA contents, although it would be desirable to have a total coverage of all the considered etiologies.

5.5.6.2 Gold Standard Quality

The gold standard has been obtained from the recommendations of five independent human experts on pneumonia diagnosis, who have used the same information that was provided to the ES. It has been computed by a consensus function that produces a ranked list of diagnoses (section 5.5.1). No preference among experts exists, to prevent bias.

In the case library there are 8 cases for which the actual diagnosis is known [Verdaguer 89]. We have used this cases to assess the gold standard quality, checking whether the actual diagnosis was present in the gold standard or not. For the 8 cases, the gold standard included the actual diagnosis. This test provides positive evidence about the gold standard quality. In addition, we have made the same test with PNEUMON-IA (before) and PNEUMON-IA (after), obtaining that PNEUMON-IA (before) contained the actual diagnosis in 7 of the 8 cases, while PNEUMON-IA (after) contained the actual diagnosis in the 8 cases. This confirms the actual improvement of PNEUMON-IA by using refinement techniques.

5.5.6.3 Analysis of IMPROVER Behavior

This section aims at discussing different aspects of the IMPROVER behavior, to see whether this behavior has some explanation or, on the contrary, it has no justification. In the following, we discuss the performance gain in the different IMPROVER stages, the refinement of domain and control knowledge and the type of accepted refinements.

[4] In the sense that there are not true positives for these etiologies in the case library. An indirect evidence of the correctness for the knowledge supporting these etiologies is that they do not appear as false positives in any execution for the whole case library.

[5] They are mentioned in the medical literature as *the big four*, considering that from 60% to 70% of the total number of pneumonias is caused by them.

Regarding the performance gain caused by the IMPROVER stages, the greater gain in performance has been produced by solving-false-negatives, followed by solving-false-positives and solving-ordering-mismatches (see table 5-1). This result is quite reasonable, since refinement stages were prioritized in this order by the stage sequencing and by the acceptance criteria AC-1, AC-2 and AC-3.

Regarding the refinement of control and domain knowledge, we can say that domain knowledge has been extensively refined in the three stages. Control knowledge has been extensively refined in solving-false-negatives, including in the conclusion of metarules those modules that have not been visited when they should. Refinement of control knowledge in solving-false-positives has been less extensive: only one modification has been accepted. This is partially due to the complex form of computing the active module list, based on combinations of the certainty degrees of metarules responsible for module activation. The position of a module in the active module list appears as a cooperative effect of its activating metarules. Given that IMPROVER modifies a single element of the KB, refinements on metarules are not guaranteed to achieve its final goal. This is true for both solving-false-negatives and solving-false-positives. In the PNEUMON-IA case, the addition of a module to a metarule has been more effective for solving-false-negatives than the retraction of a module in a metarule for solving-false-positives. The only unexpected interaction between refinement of domain and control knowledge has happened in solving-ordering-mismatches, when modifications in the certainty degree of some rules has caused changes in the firing time of stopping metarules (section 5.5.4). This has had a positive effect, since 1 false negative and 2 false positives have been solved. Nevertheless, this unexpected behavior indicates the interdependency between domain and control knowledge, and reinforces the necessity of an accurate computation of the consequences of each suggested refinement.

Regarding the kind of accepted refinements, they have been generated by the action of the following operators: OP-1, OP-2, OP-3 and OP-4. The action of operators OP-5 and OP-6 has been aborted by setting to 1 the limit in the elementary changes in an acceptable refinement. These operators add new conditions or new rules to the KB, that are computed from the most specific discriminant between positive and negative examples. The most specific discriminant usually involves more than one single element. Therefore, the limit of elementary changes has prevented the practical application of these operators.

In summary, we conclude that IMPROVER behavior is clearly explainable and justifiable. This conclusion, grouped with the analysis of the case library and the gold standard quality, provides positive evidence of the quality of the refinement process and reinforces the validity of the refined system, PNEUMON-IA (after).

5.5.7 Time and Memory Requirements

IMPROVER does not require significant amounts of time and memory for its execution. In figures 5-11, 5-12 and 5-13 we give the CPU time on a SUN-4/260 for the execution of IMPROVER stages: solving-false-negatives, solving-false-positives and solving-ordering-mismatches, respectively. In these figures, the horizontal axis indicates the different diagnoses that have been involved in false negatives, false positives or ordering mismatches when executing PNEUMON-IA on the whole case library. No relation exists among diagnoses positions across these figures.

When applied to a diagnosis, solving-false-negatives requires 20 CPU minutes on the average, while solving-false positives and solving-ordering-mismatches require 90 and 70 CPU minutes respectively, on the same conditions. Solving-false-negatives requires less execution time because it performs a informed search, guided by the points where a deductive chain is actually broken. On the contrary, solving-false-positives and solving-ordering-mismatches perform a relatively uninformed search. Solving-false-positives acts on all the points of a deductive chain where this chain can be broken with a single modification. Solving-ordering-mismatches follows the same approach to increase/decrease the certainty of a diagnosis. Therefore, solving-false-negatives generates

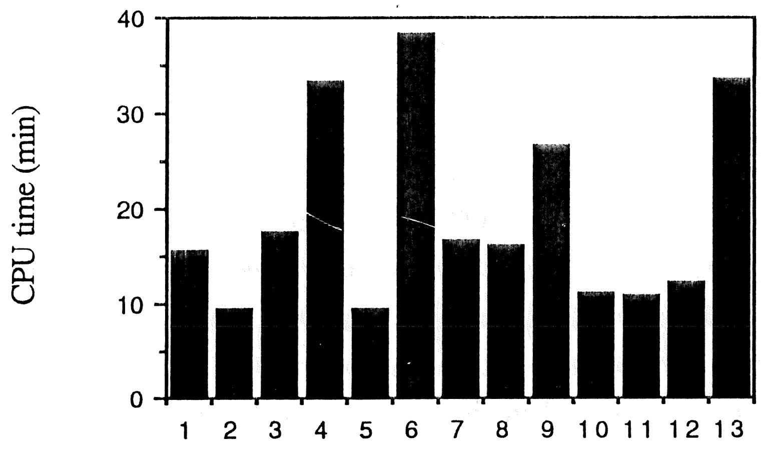

Figure 5-11. CPU time required for the solving-false-negatives stage, considering 13 different diagnoses.

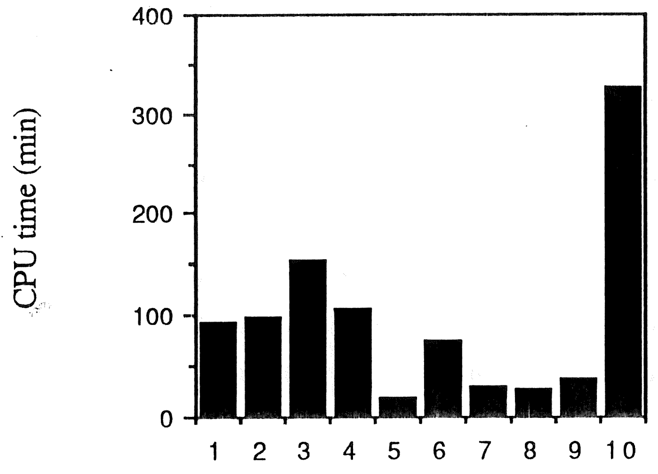

Figure 5-12. CPU time requred for the solving-false-positives stage, considering 10 different diagnoses.

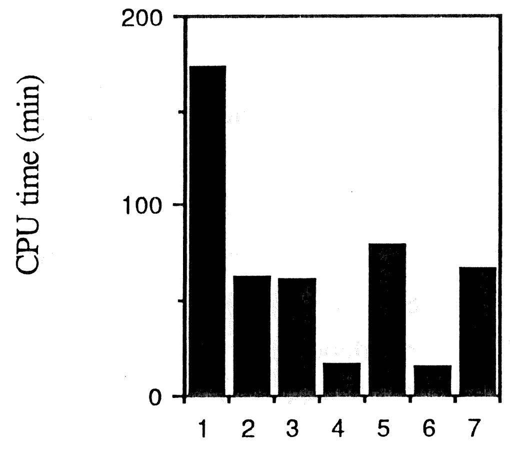

Figure 5-13. CPU time required for the solving-ordering-mismatches stage, considering 7 different diagnoses.

less refinements and with higher chance to satisfy the corresponding acceptance criterion than solving-false-positives and solving-ordering-mismatches. This is the cause for the differences in execution time among the three stages.

Regarding computational complexity, it has been proved [Valtorta 91] that the general problem of knowledge base refinement is exponential in the worst case[6]. IMPROVER does not suffer from this exponentiality due to the heuristics that control the combinatorial

[6] For a knowledge base composed of rules with certainty degrees and without metarules.

explosion in refinement generation (section 5.3.2). Despite of the restrictive approach used in IMPROVER to refine PNEUMON-IA (only refinements with an elementary change on a single rule or metarule) this approach exhibits a good effectiveness/cost ratio, since IMPROVER has obtained quite good results with a small computational cost.

Regarding memory requirements, they are proportional to the KB size and to the number of considered cases. The three IMPROVER stages demand the same amount of memory, determined by the and/or graphs representing control and domain knowledge and internal tables to record refinement results. In the PNEUMON-IA case, IMPROVER has used a maximum of 2 Mbytes for execution, what is easily available in current workstations.

5.6 Conclusions

From the practical use of IMPROVER refining PNEUMON-IA we extract the following conclusions:

- *Refinement Relevance*: KB refinement techniques play a significant role in ES validation. They automatically perform a number of activities that are very useful for validation when considering implemented ESs, and that would be very costly to perform manually. First, refinement carries out an automatic testing of the ES over the case library, evaluating ES performance in terms of false negatives, false positives and ordering mismatches. Second, refinement identifies those KB objects that are responsible for ES errors. Identification is performed considering the dynamic behavior of KB objects, what is very valuable for the ES developers who may not be aware of all possible interactions and effects that a single rule or metarule can cause. And third, refinement provides those KB modifications that cause a neat performance gain, computing in advance the impact of these modification on the ES. This facilitates to a great extent the selection of adequate modifications by the expert.

- *Error Importance*: the number of solved errors is no longer the criterion to accept a refinement, but the relevance of these errors with respect to the ES task. Most important errors have to be solved first. A refinement solving an error can cause other errors of lower importance as long as there is a neat performance gain.

- *Expert Acceptance*: a refinement system always suggests some modifications that have no meaning when considered from the problem domain. In spite of the fact that they produce some performance gain, these modifications should not be accepted. To prevent their inclusion in the KB, the expert responsible for ES development has to inspect all the proposed modifications, accepting only those that are meaningful. The expert guarantees the integrity of the knowledge contained in the KB.

- *Control and Domain Knowledge*: all types of knowledge contained in the KB are subject to potential refinement. Refinement has to consider each type of KB object regarding the way it is used in practice (operational semantics). In PNEUMON-IA, both control and domain knowledge have been refined, producing each one specific improvements in performance.

In a KB refinement process, there is a number of points that are very important for achieving a true performance gain. These points are the following:

- *KB State*: an initial assumption for KB refinement is that the KB state is close to a correct state. Otherwise, a refinement system will never be able to improve the ES performance. The good results obtained by IMPROVER would have not been achieved if the knowledge contained in PNEUMON-IA had been incorrect, incomplete or of poor quality.

- *Case Library*: a refinement system is based on ES testing over the case library. If the case library is not a representative sample of problem occurrences in the considered domain, the refinement system will test the KB only partially. The improvement in performance will be biased towards the kind of cases that have been used to support the refinement process.

- *Gold Standard*: the overall effect of a KB refinement system is to move the ES output towards the gold standard. If the gold standard is not the correct one, the refinement system is moving the ES in the wrong direction, making it worse instead of improving it. The gold standard is used in the identification phase, to detect errors in the ES output. All the subsequent processes, error localization, refinement generation and selection, are based on this initial identification. The gold standard plays a fundamental role in refinement, acting as the driver of the whole process. Therefore, its quality and accuracy should be guaranteed as far as possible.

- *Biases*: as any other learning system, a refinement system includes a number of choices with respect to the practical form of learning. These choices are denominated generically as biases of the learning method. In the case of IMPROVER, the heuristics to control refinement generation, the legal set of refinement operators and the acceptance criteria are the biases of the system. They have an important impact in the refinement process and determine to a great extent its improvement in performance. Therefore, the biases of a refinement system has to be initially justified, and its effectivity has to be confirmed in practice.

Chapter 6

Conclusions and Further Research

In previous chapters we have analyzed the problem of validation in ESs and we have proposed some solutions. We have made a tour on the different aspects of validation in ESs, devoting special attention to the conceptual and terminological issues of this field. We have seen that the validation definition in software engineering is perfectly applicable to ESs. This does not imply that techniques used in conventional software validation are directly applicable to ES validation. Some of these techniques can be adapted to ESs, while new techniques specific for ES validation are also required.

We have focused on rule-based ESs with a multi-level architecture performing medical diagnosis. For these systems we have presented two new validation methods. The first one is a verification method that checks a number of properties in the KB, in order to assure its structural correctness. New verification issues exist in the considered ES model, caused by the presence of uncertainty and by the action of control knowledge. The verification method solves these new issues using extended labels, an extension of ATMS constructs. We have implemented an incremental version of this method in the verifier IN-DEPTH II. Using IN-DEPTH II, we are verifying the expert system PNEUMON-IA with encouraging results. We have detected and corrected a number of errors that, without the verifier help, would have been missed. Some of these errors were not just simple mistakes in rule coding, but they were caused by an erroneous knowledge organization. In addition, we can guarantee the absence of certain types of errors, what conveys a substantive confidence gain in the ES.

The second validation method consists in a refinement system to improve ES performance. With respect to previous refinement systems, this method provides three new contributions. First, ES performance is not measured as the raw number of errors performed by the system but takes into account the relative importance of these errors on the ES task. Second, domain and control knowledge are subject to refinement. And third, a new type of error, ordering mismatch, is considered as a consequence of working with ESs providing multiple diagnoses. We have implemented this method in IMPROVER, an automatic refinement tool. Using IMPROVER on a library of 66 cases, we have refined PNEUMON-IA obtaining very good results, specially considering the solution of false negatives, the most important error in medical diagnosis. From all the modifications suggested by IMPROVER, only those accepted by the expert developer of PNEUMON-IA have been implemented. They can be considered as minor changes in the knowledge expression.

In this short chapter, we give the conclusions of our work as well as some lines for further research in validation. Regarding conclusions, we have already given specific conclusions about the verification and refinement processes, based on our practical experience validating PNEUMON-IA with IN-DEPTH II and IMPROVER. Here we give some conclusions in a more general setting, with the aim of being applicable to any ES. On the other hand, this work is just a step forward to get a better ES validation. Many validation aspects are not developed yet, and they are needed to improve ES quality. Some of these aspects are briefly outlined in the further research section.

6.1 Conclusions

From this work, we extract the following general conclusions:

- *Validation in Software and Knowledge Engineering*: a single validation definition exists and it is applicable to knowledge and software engineering. Validation is defined in terms of user requirements, that are divided into totally of partially formalizable. Validation is composed of two main parts, verification and evaluation. These definitions are in compliance with the corresponding ones in software engineering, forming a general framework for software validation.

- *Validation is Feasible*: ES validation is a difficult but not impossible task. We have identified a number of validation activities to be performed during the ES life-cycle, that will improve to a large extent the quality and validity of the final system. Considering implemented ESs, we have developed and implemented two different methods that have allowed to detect and correct a number of errors. Using these methods we guarantee the absence of some types of errors, and we improve ES performance aiming at human expert competence. Reported results indicate the level of validity currently reachable in ES applications.

- *Automatic Tools*: rule-based ESs are composed of hundreds or thousands of condition-action pairs, that are conceived separately but that act in a cooperative form. In these systems, manual validation is practically impossible due to the extremely large amount of possible combinations to be checked. Therefore, the necessity of automatic tools to perform validation activities on implemented ESs is clear. Nevertheless, automatic validation tools do not release ES developers from their responsibilities. Tools can detect some errors, suggest modifications to correct them and even anticipate what will happen in the ES behavior if a modification is accepted. But the actual correction of an error is the responsibility of ES developers, who have to devise the best way to correct it.

- *Verification and Refinement*: a direct conclusion of this work is to stress the usefulness of verification and refinement methods for ES validation. The current state of the art offers a significant number of available techniques for verification and refinement, that can be applied to different types of ESs. Nevertheless, the the state-of-the-practice survey suggests that verification and refinement are not yet considered as practical methods for ES validation (see section 2.6.3).

- *Theory and Practice*: theory and practice are indissolubly joined in ES validation. Theory is needed to develop sound and complete validation methods. These methods have to be applied in practice to show their actual effectiveness. In addition, validation in practice is a useful exercise for the developer of validation methods. Then, he/she discovers that some error types are ignorable, realizes the importance of a flexible validation environment, or rejects a modification as meaningless, no matter how many errors it solves. Practice is the test that any validation method or tool has to pass prior to be considered actually useful.

6.2 Further Research

As elements for further research,we identify the following points:

- *Conceptual Models*: we have stressed the importance of conceptual models of the ES task in the validation process, as the basic framework to support validation activities. So far, no practical experience about the use of models in validation of actual ESs has been reported. Probably this is due to the absence of written models for actual applications. Nevertheless, the use of detailed and explicit conceptual models is the only way to achieve a true validation independently from an active participation of human experts in the field. Otherwise, validation will always depend on human experts who suggest the points to be checked and who evaluate validation results, accepting or rejecting proposed modifications as an oracle for ES correctness.

- *Validation of Conceptual Models*: if a conceptual model of the ES task is available, validation has to be made first on the model itself. Once the model is considered acceptable or correct, validation of the implemented ES is much simpler, since it has just to assure that the implemented system is a faithful representation of the model. In addition, validation of conceptual models allows to identify strengths and weaknesses of these models, indicating those points on which further validation activities should be concentrated.

- *Verification of Knowledge Properties*: so far, verification has been focused on checking properties very related with the representation of knowledge but not with knowledge itself. Next step in verification consists in checking knowledge properties, independently of the representation language used. These properties can be extracted from the conceptual model of the ES task. Verification of knowledge properties will produce a substantial increase of confidence in ES applications.

- *Knowledge Acquisition for Validation*: verification of knowledge properties requires the acquisition of some knowledge just for validation purposes. This knowledge has to be acquired in the usual ways, in early stages of ES development. Early presence of validation in ES development reinforces the design and implementation for validation.

- *Validation by Construction*: new developments in knowledge representation languages and in cognitive architectures provide constructs that have more expressive power than just rules. These architectures intend to be closer to the knowledge level for the target task. It seems reasonable to expect that these architectures will facilitate the validation process, in a two-fold way. First, they can make unnecessary a number of validation checks currently performed on rules, because the knowledge representation language already guarantees their correct form (in the same way that a programmer using a high-level programming language is not worried about issues that occur in an assembler language). Second, they may include validation facilities in the language, that can facilitate the expression of validation conditions and their automatic checking at compilation time.

- *Test Set Selection*: testing is a validation technique of great effectivity in knowledge and software engineering, currently considered as indispensable. A central issue in testing is the selection of the test set. In knowledge engineering, no consolidated methodology exists for test set selection. Previous work and experience in software engineering on this point can be adapted to develop sound methods to obtain adequate test sets for ES validation.

Appendix

Refinement Results

In the following we present the results of refining PNEUMON-IA using IMPROVER. For each case of the library, we provide the gold standard, the output of PNEUMON-IA before IMPROVER use, and the output of PNEUMON-IA after IMPROVER use. We identify false negatives, false positives, and ordering mismatches for each case and version of PNEUMON-IA.

The solution for a case is a ranked list of diagnoses, ordered by decreasing certainty. When the value NIL appears in a position, it means that no diagnosis has been assigned to this position. Several diagnoses may appear in the same position, meaning that all share the same certainty value. When *m* diagnoses share the same certainty, and the next position in the ranking is the *i*-th, we assign the value NIL to the next $m + i - 1$ positions in the ranking, assigning the *m* diagnoses to the position $m + i$.

```
CASE: R01V1

GOLD-STANDARD                    PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------       ---------------------------        --------------------------
(PNEUMOCOC)                      (PNEUMOCOC)                        (PNEUMOCOC)
(ENTEROBACTERIES)                NIL                                NIL
(MYCOPLASMA)                     (ENTEROBACTERIES ANA)              (ENTEROBACTERIES ANA)
NIL                              NIL                                NIL
NIL                              NIL                                NIL
NIL                              (HEMOPHILUS MYCOPLASMA VIRUS)      (HEMOPHILUS MYCOPLASMA)

                                 FN: NIL                            FN: NIL
                                 FP: (VIRUS HEMOPHILUS ANA)         FP: (HEMOPHILUS ANA)
                                 OM: NIL                            OM: NIL

-----------------------------------------------------------------------------------------------

CASE: R02V1

GOLD-STANDARD                    PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------       ---------------------------        --------------------------
(PNEUMOCOC)                      (SA)                               NIL
NIL                              (PNEUMOCOC)                        (SA PNEUMOCOC)
(ENTEROBACTERIES SA)             (ENTEROBACTERIES)                  (ENTEROBACTERIES)

                                 FN: NIL                            FN: NIL
                                 FP: NIL                            FP: NIL
                                 OM: (SA PNEUMOCOC)                 OM: NIL

-----------------------------------------------------------------------------------------------

CASE: R03V1

GOLD-STANDARD                    PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------       ---------------------------        --------------------------
(PNEUMOCOC)                      (PNEUMOCOC)                        (PNEUMOCOC)
(CLAMIDIA)                       NIL                                (LEGIONELLA)
(LEGIONELLA)                     NIL                                NIL

                                 FN: (LEGIONELLA CLAMIDIA)          FN: (CLAMIDIA)
                                 FP: NIL                            FP: NIL
                                 OM: NIL                            OM: NIL

-----------------------------------------------------------------------------------------------

CASE: R04V1

GOLD-STANDARD                    PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------       ---------------------------        --------------------------
(PNEUMOCOC)                      (MYCOPLASMA)                       NIL
(MYCOPLASMA)                     NIL                                (MYCOPLASMA PNEUMOCOC)
NIL                              (VIRUS PNEUMOCOC)                  NIL

                                 FN: NIL                            FN: NIL
                                 FP: (VIRUS)                        FP: NIL
                                 OM: (PNEUMOCOC MYCOPLASMA)         OM: NIL

-----------------------------------------------------------------------------------------------

CASE: R05V1

GOLD-STANDARD                    PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------       ---------------------------        --------------------------
(PNEUMOCOC)                      (PNEUMOCOC)                        (PNEUMOCOC)
NIL                              (ENTEROBACTERIES)                  NIL
(LEGIONELLA ENTEROBACTERIES)     NIL                                (ENTEROBACTERIES HEMOPHILUS)

                                 FN: (LEGIONELLA)                   FN: (LEGIONELLA)
                                 FP: NIL                            FP: (HEMOPHILUS)
                                 OM: NIL                            OM: NIL

-----------------------------------------------------------------------------------------------
```

```
CASE: R06V1

GOLD-STANDARD                       PNEUMON-IA (before)                     PNEUMON-IA (after)
--------------------------          ---------------------------             --------------------------
(MYCOPLASMA)                        (PNEUMOCOC)                             (PNEUMOCOC)
(PNEUMOCOC)                         NIL                                     NIL
NIL                                 (CLAMIDIA MYCOPLASMA)                   NIL
NIL                                 NIL                                     (CLAMIDIA MYCOPLASMA LEGIONELLA)
NIL                                 NIL                                     NIL
(LEGIONELLA CLAMIDIA HEMOPHILUS VIRUS)NIL                                   NIL

                                    FN: (VIRUS HEMOPHILUS LEGIONELLA)       FN: (VIRUS HEMOPHILUS)
                                    FP: NIL                                 FP: NIL
                                    OM: (PNEUMOCOC MYCOPLASMA)              OM: (PNEUMOCOC MYCOPLASMA)

-------------------------------------------------------------------------------------------------------------

CASE: R07V1

GOLD-STANDARD                       PNEUMON-IA (before)                     PNEUMON-IA (after)
--------------------------          ---------------------------             --------------------------
(PNEUMOCOC)                         (PNEUMOCOC)                             (PNEUMOCOC)
NIL                                 NIL                                     NIL
(ENTEROBACTERIES LEGIONELLA)        (ENTEROBACTERIES LEGIONELLA)            (ENTEROBACTERIES LEGIONELLA)
(HEMOPHILUS)                        NIL                                     (HEMOPHILUS)

                                    FN: (HEMOPHILUS)                        FN: NIL
                                    FP: NIL                                 FP: NIL
                                    OM: NIL                                 OM: NIL

-------------------------------------------------------------------------------------------------------------

CASE: R08V1

GOLD-STANDARD                       PNEUMON-IA (before)                     PNEUMON-IA (after)
--------------------------          ---------------------------             --------------------------
(PNEUMOCOC)                         (PNEUMOCOC)                             (PNEUMOCOC)
NIL                                 NIL                                     NIL
(ENTEROBACTERIES HEMOPHILUS)        (ENTEROBACTERIES LEGIONELLA)            (ENTEROBACTERIES LEGIONELLA)
NIL                                 NIL                                     (HEMOPHILUS)
(ANA LEGIONELLA)                    NIL                                     NIL

                                    FN: (ANA HEMOPHILUS)                    FN: (ANA)
                                    FP: NIL                                 FP: NIL
                                    OM: NIL                                 OM: (HEMOPHILUS LEGIONELLA)

-------------------------------------------------------------------------------------------------------------

CASE: R09V1

GOLD-STANDARD                       PNEUMON-IA (before)                     PNEUMON-IA (after)
--------------------------          ---------------------------             --------------------------
(PNEUMOCOC)                         (PNEUMOCOC)                             (PNEUMOCOC)
(LEGIONELLA)                        NIL                                     NIL
NIL                                 (CLAMIDIA MYCOPLASMA)                   (CLAMIDIA MYCOPLASMA LEGIONELLA)
NIL                                 (VIRUS)                                 (VIRUS)

                                    FN: (LEGIONELLA)                        FN: NIL
                                    FP: (VIRUS CLAMIDIA MYCOPLASMA)         FP:(VIRUS CLAMIDIA MYCOPLASMA)
                                    OM: NIL                                 OM: NIL

-------------------------------------------------------------------------------------------------------------

CASE: R10V1

GOLD-STANDARD                       PNEUMON-IA (before)                     PNEUMON-IA (after)
--------------------------          ---------------------------             --------------------------
(PNEUMOCOC)                         (PNEUMOCOC)                             NIL
(HEMOPHILUS)                        (ENTEROBACTERIES)                       (PNEUMOCOC LEGIONELLA)
(ENTEROBACTERIES)                   (CLAMIDIA)                              (ENTEROBACTERIES)
NIL                                 NIL                                     (HEMOPHILUS)

                                    FN: (HEMOPHILUS)                        FN: NIL
                                    FP: (CLAMIDIA)                          FP: (LEGIONELLA)
                                    OM: NIL                                 OM: (ENTEROBACTERIES HEMOPHILUS)

-------------------------------------------------------------------------------------------------------------
```

```
CASE: R11V1

GOLD-STANDARD                 PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------    ---------------------------      --------------------------
(TUBERCULOSI)                 (PNEUMOCOC)                      (PNEUMOCOC)
NIL                           (ENTEROBACTERIES)                NIL
(PNEUMOCOC LEGIONELLA)        NIL                              (ENTEROBACTERIES HEMOPHILUS)

                              FN: (LEGIONELLA TUBERCULOSI)     FN: (LEGIONELLA TUBERCULOSI)
                              FP: (ENTEROBACTERIES)            FP: (ENTEROBACTERIES HEMOPHILUS)
                              OM: NIL                          OM: NIL
----------------------------------------------------------------------------------------------------
CASE: R12V1

GOLD-STANDARD                 PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------    ---------------------------      --------------------------
(ANA)                         (PNEUMOCOC)                      NIL
(ENTEROBACTERIES)             (TUBERCULOSI)                    (PNEUMOCOC ANA)
(TUBERCULOSI)                 (LEGIONELLA)                     (TUBERCULOSI)
NIL                           NIL                              NIL
NIL                           NIL                              (HEMOPHILUS ENTEROBACTERIES)
NIL                           NIL                              (LEGIONELLA)

                              FN: (ANA ENTEROBACTERIES)        FN: NIL
                              FP: (LEGIONELLA PNEUMOCOC)       FP: (LEGIONELLA PNEUMOCOC
                                                                    HEMOPHILUS)
                              OM: NIL                          OM: (ENTEROBACTERIES TUBERCULOSI)
----------------------------------------------------------------------------------------------------
CASE: R13V1

GOLD-STANDARD                 PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------    ---------------------------      --------------------------
(PNEUMOCOC)                   (PNEUMOCOC)                      (PNEUMOCOC)
(HEMOPHILUS)                  (ENTEROBACTERIES)                NIL
(ENTEROBACTERIES)             NIL                              (ENTEROBACTERIES HEMOPHILUS)

                              FN: (HEMOPHILUS)                 FN: NIL
                              FP: NIL                          FP: NIL
                              OM: NIL                          OM: NIL
----------------------------------------------------------------------------------------------------
CASE: R14V1

GOLD-STANDARD                 PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------    ---------------------------      --------------------------
(PNEUMOCOC)                   (PNEUMOCOC)                      (PNEUMOCOC)
(HEMOPHILUS)                  (HEMOPHILUS)                     (HEMOPHILUS)
NIL                           NIL                              NIL
NIL                           (ENTEROBACTERIES ANA)            (ENTEROBACTERIES ANA)

                              FN: NIL                          FN: NIL
                              FP: (ANA ENTEROBACTERIES)        FP: (ANA ENTEROBACTERIES)
                              OM: NIL                          OM: NIL
----------------------------------------------------------------------------------------------------
CASE: M01V2

GOLD-STANDARD                 PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------    ---------------------------      --------------------------
(TUBERCULOSI)                 (ANA)                            (ANA)
(ANA)                         NIL                              (LEGIONELLA)
(HEMOPHILUS)                  NIL                              NIL
NIL                           NIL                              (TUBERCULOSI ENTEROBACTERIES)
NIL                           (CLAMIDIA TUBERCULOSI            (HEMOPHILUS)
                               ENTEROBACTERIES LEGIONELLA)

                              FN: (HEMOPHILUS)                 FN: NIL
                              FP: (ENTEROBACTERIES             FP: (ENTEROBACTERIES LEGIONELLA)
                                   LEGIONELLA CLAMIDIA)
                              OM: (ANA TUBERCULOSI)            OM: (ANA TUBERCULOSI)
----------------------------------------------------------------------------------------------------
```

```
CASE: M02V1

GOLD-STANDARD                      PNEUMON-IA (before)                   PNEUMON-IA (after)
---------------------------        ---------------------------           ---------------------------
(PNEUMOCOC)                        (PNEUMOCOC)                           (PNEUMOCOC)
(LEGIONELLA)                       (LEGIONELLA)                          NIL
(HEMOPHILUS)                       NIL                                   (LEGIONELLA ENTEROBACTERIES)
(ENTEROBACTERIES)                  NIL                                   (HEMOPHILUS)

                                   FN: (ENTEROBACTERIES HEMOPHILUS)      FN: NIL
                                   FP: NIL                               FP: NIL
                                   OM: NIL                               OM:(HEMOPHILUS ENTEROBACTERIES)
------------------------------------------------------------------------------------------------------------

CASE: M03V2

GOLD-STANDARD                      PNEUMON-IA (before)                   PNEUMON-IA (after)
---------------------------        ---------------------------           ---------------------------
(ENTEROBACTERIES)                  (ENTEROBACTERIES)                     NIL
NIL                                (PNEUMOCOC)                           (PNEUMOCOC ENTEROBACTERIES)
(HEMOPHILUS PNEUMOCOC)             (LEGIONELLA)                          NIL
(LEGIONELLA)                       (STR-A)                               (LEGIONELLA HEMOPHILUS)

                                   FN: (HEMOPHILUS)                      FN: NIL
                                   FP: (STR-A)                           FP: NIL
                                   OM: NIL                               OM: NIL
------------------------------------------------------------------------------------------------------------

CASE: M04V1

GOLD-STANDARD                      PNEUMON-IA (before)                   PNEUMON-IA (after)
---------------------------        ---------------------------           ---------------------------
(PNEUMOCOC)                        (ANA)                                 (PNEUMOCOC)
(ANA)                              (PNEUMOCOC)                           (ANA)
(HEMOPHILUS)                       (ENTEROBACTERIES)                     (ENTEROBACTERIES)

                                   FN: (HEMOPHILUS)                      FN: (HEMOPHILUS)
                                   FP: (ENTEROBACTERIES)                 FP: (ENTEROBACTERIES)
                                   OM: (ANA PNEUMOCOC)                   OM: NIL
------------------------------------------------------------------------------------------------------------

CASE: M05V1

GOLD-STANDARD                      PNEUMON-IA (before)                   PNEUMON-IA (after)
---------------------------        ---------------------------           ---------------------------
(PNEUMOCOC)                        (PNEUMOCOC)                           (PNEUMOCOC)
(ANA)                              NIL                                   NIL
NIL                                (ENTEROBACTERIES ANA)                 (ENTEROBACTERIES ANA)
(ENTEROBACTERIES HEMOPHILUS)       (HEMOPHILUS)                          (HEMOPHILUS)

                                   FN: NIL                               FN: NIL
                                   FP: NIL                               FP: NIL
                                   OM: NIL                               OM: NIL
------------------------------------------------------------------------------------------------------------

CASE: M06V6

GOLD-STANDARD                      PNEUMON-IA (before)                   PNEUMON-IA (after)
---------------------------        ---------------------------           ---------------------------
(ENTEROBACTERIES)                  (ENTEROBACTERIES)                     NIL
(ANA)                              NIL                                   (ENTEROBACTERIES LEGIONELLA)
NIL                                NIL                                   (ASPERGILOSI-INV)
(TUBERCULOSI ASPERGILOSI-INV) NIL  NIL

                                   FN: (ASPERGILOSI-INV TUBERCULOSI ANA) FN: (TUBERCULOSI ANA)
                                   FP: NIL                               FP: (LEGIONELLA)
                                   OM: NIL                               OM: NIL
------------------------------------------------------------------------------------------------------------
```

```
CASE: M07V1

GOLD-STANDARD                      PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------         ---------------------------            --------------------------
NIL                                (PNEUMOCOC)                            (PNEUMOCOC)
(LEGIONELLA PNEUMOCOC)             (LEGIONELLA)                           (LEGIONELLA)
(ENTEROBACTERIES)                  (ENTEROBACTERIES)                      (ENTEROBACTERIES)
NIL                                NIL                                    (HEMOPHILUS)

                                   FN: NIL                                FN: NIL
                                   FP: NIL                                FP: (HEMOPHILUS)
                                   OM: NIL                                OM: NIL

----------------------------------------------------------------------------------------------------------

CASE: M08V3

GOLD-STANDARD                      PNEUMON-IA (before)                    PNEUMON-IA (after)
-------------------------          ---------------------------            --------------------------
(PNEUMOCOC)                        (PNEUMOCOC)                            (PNEUMOCOC)
NIL                                NIL                                    NIL
(STR-A TUBERCULOSI)                NIL                                    (LEGIONELLA MYCOPLASMA)
NIL                                NIL                                    NIL
NIL                                (LEGIONELLA MYCOPLASMA CLAMIDIA)       (VIRUS CLAMIDIA)
(SA LEGIONELLA VIRUS)              (VIRUS)                                NIL

                                   FN: (SA STR-A TUBERCULOSI)             FN: (SA STR-A TUBERCULOSI)
                                   FP: (CLAMIDIA MYCOPLASMA)              FP: (CLAMIDIA MYCOPLASMA)
                                   OM: NIL                                OM: NIL

----------------------------------------------------------------------------------------------------------

CASE: B01V1

GOLD-STANDARD                      PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------         ---------------------------            --------------------------
(PNEUMOCOC)                        (PNEUMOCOC)                            (PNEUMOCOC)
NIL                                (ANA)                                  (ANA)
NIL                                (ENTEROBACTERIES)                      (ENTEROBACTERIES)

                                   FN: NIL                                FN: NIL
                                   FP: (ENTEROBACTERIES ANA)              FP: (ENTEROBACTERIES ANA)
                                   OM: NIL                                OM: NIL

----------------------------------------------------------------------------------------------------------

CASE: B02V1

GOLD-STANDARD                      PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------         ---------------------------            --------------------------
(PNEUMOCOC)                        (VIRUS)                                NIL
NIL                                NIL                                    (VIRUS PNEUMOCOC)
(HEMOPHILUS LEGIONELLA)            (MYCOPLASMA CLAMIDIA)                  NIL
(ENTEROBACTERIES)                  NIL                                    (MYCOPLASMA HEMOPHILUS)
NIL                                NIL                                    (ENTEROBACTERIES)

                                   FN: (ENTEROBACTERIES PNEUMOCOC         FN: (LEGIONELLA)
                                         HEMOPHILUS LEGIONELLA)
                                   FP: (CLAMIDIA MYCOPLASMA VIRUS)        FP: (MYCOPLASMA VIRUS)
                                   OM: NIL                                OM: NIL

----------------------------------------------------------------------------------------------------------

CASE: B03V1

GOLD-STANDARD                      PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------         ---------------------------            --------------------------
(PNEUMOCOC)                        NIL                                    NIL
(ENTEROBACTERIES)                  (ANA PNEUMOCOC)                        (ANA PNEUMOCOC)
NIL                                (ENTEROBACTERIES)                      (ENTEROBACTERIES)
NIL                                NIL                                    (LEGIONELLA)

                                   FN: NIL                                FN: NIL
                                   FP: (ANA)                              FP: (LEGIONELLA ANA)
                                   OM: NIL                                OM: NIL

----------------------------------------------------------------------------------------------------------
```

```
CASE: B04V1

GOLD-STANDARD                       PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------          ---------------------------            --------------------------
(PNEUMOCOC)                         (PNEUMOCOC)                            (PNEUMOCOC)
NIL                                 NIL                                    NIL
NIL                                 NIL                                    (ENTEROBACTERIES HEMOPHILUS)
(HEMOPHILUS MYCOPLASMA CLAMIDIA)    NIL                                    NIL

                                    FN: (CLAMIDIA MYCOPLASMA HEMOPHILUS)   FN: (CLAMIDIA MYCOPLASMA)
                                    FP: NIL                                FP: (ENTEROBACTERIES)
                                    OM: NIL                                OM: NIL

-------------------------------------------------------------------------------------------------------------

CASE: B05V1

GOLD-STANDARD                       PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------          ---------------------------            --------------------------
(MYCOPLASMA)                        NIL                                    NIL
NIL                                 (MYCOPLASMA VIRUS)                     NIL
NIL                                 NIL                                    (MYCOPLASMA VIRUS HEMOPHILUS)
NIL                                 NIL                                    NIL
(CLAMIDIA FEBRE-Q VIRUS HEMOPHILUS) NIL                                    NIL

                                    FN: (HEMOPHILUS FEBRE-Q CLAMIDIA)      FN: (HEMOPHILUS CLAMIDIA)
                                    FP: NIL                                FP: NIL
                                    OM: NIL                                OM: NIL

-------------------------------------------------------------------------------------------------------------

CASE: B06V1

GOLD-STANDARD                       PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------          ---------------------------            --------------------------
NIL                                 (PNEUMOCOC)                            (PNEUMOCOC)
(PNEUMOCOC ENTEROBACTERIES)         NIL                                    (HEMOPHILUS)
(ANA)                               (ENTEROBACTERIES HEMOPHILUS)           (ENTEROBACTERIES)

                                    FN: (ANA)                              FN: (ANA)
                                    FP: (HEMOPHILUS)                       FP: (HEMOPHILUS)
                                    OM: NIL                                OM: NIL

-------------------------------------------------------------------------------------------------------------

CASE: B07V1

GOLD-STANDARD                       PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------          ---------------------------            --------------------------
(PNEUMOCOC)                         (VIRUS)                                NIL
NIL                                 NIL                                    (PNEUMOCOC VIRUS)
NIL                                 (PNEUMOCOC ANA)                        NIL
(MYCOPLASMA LEGIONELLA ANA)         NIL                                    (ANA HEMOPHILUS)
NIL                                 NIL                                    NIL
NIL                                 (CLAMIDIA ENTEROBACTERIES MYCOPLASMA)  (ENTEROBACTERIES MYCOPLASMA)


                                    FN: (LEGIONELLA)                       FN: (LEGIONELLA)
                                    FP: (ENTEROBACTERIES CLAMIDIA VIRUS)   FP: (ENTEROBACTERIES HEMOPHILUS
                                                                                VIRUS)
                                    OM: NIL                                OM: NIL

-------------------------------------------------------------------------------------------------------------

CASE: B08V1

GOLD-STANDARD                       PNEUMON-IA (before)                    PNEUMON-IA (after)
--------------------------          ---------------------------            --------------------------
NIL                                 (MYCOPLASMA)                           (PNEUMOCOC)
(TUBERCULOSI PNEUMOCOC)             NIL                                    (MYCOPLASMA)
NIL                                 NIL                                    NIL
NIL                                 NIL                                    NIL
(ANA ENTEROBACTERIES MYCOPLASMA)    NIL                                    NIL
NIL                                 (PNEUMOCOC ENTEROBACTERIES             (ENTEROBACTERIES ANA
                                     VIRUS CLAMIDIA)                        VIRUS CLAMIDIA)
NIL                                 NIL                                    (LEGIONELLA)

                                    FN: (ANA TUBERCULOSI)                  FN: (TUBERCULOSI)
                                    FP: (VIRUS CLAMIDIA)                   FP: (LEGIONELLA VIRUS CLAMIDIA)
                                    OM: (MYCOPLASMA PNEUMOCOC)             OM: NIL

-------------------------------------------------------------------------------------------------------------
```

```
CASE: B09V1

GOLD-STANDARD                   PNEUMON-IA (before)                  PNEUMON-IA (after)
--------------------------      ---------------------------          --------------------------
(PNEUMOCOC)                     (PNEUMOCOC)                          (PNEUMOCOC)
(LEGIONELLA)                    (CLAMIDIA)                           NIL
NIL                             (MYCOPLASMA)                         (MYCOPLASMA LEGIONELLA)
NIL                             (VIRUS)                              (VIRUS)

                                FN: (LEGIONELLA)                     FN: NIL
                                FP: (VIRUS CLAMIDIA MYCOPLASMA)      FP: (VIRUS MYCOPLASMA)
                                OM: NIL                              OM: NIL

----------------------------------------------------------------------------------------------------

CASE: B10V1

GOLD-STANDARD                   PNEUMON-IA (before)                  PNEUMON-IA (after)
--------------------------      ---------------------------          --------------------------
(ENTEROBACTERIES)               NIL                                  NIL
(PNEUMOCOC)                     (VIRUS PNEUMOCOC)                    (VIRUS PNEUMOCOC)
(SA)                            NIL                                  (ANA)
NIL                             (MYCOPLASMA CLAMIDIA)                NIL
NIL                             (LEGIONELLA)                         (HEMOPHILUS MYCOPLASMA)
NIL                             NIL                                  NIL
NIL                             NIL                                  (ENTEROBACTERIES LEGIONELLA)

                                FN: (SA ENTEROBACTERIES)             FN: (SA)
                                FP: (LEGIONELLA VIRUS                FP: (LEGIONELLA VIRUS ANA
                                    MYCOPLASMA CLAMIDIA)                  MYCOPLASMA HEMOPHILUS)
                                OM: NIL                              OM: (ENTEROBACTERIES PNEUMOCOC)

----------------------------------------------------------------------------------------------------

CASE: B11V1

GOLD-STANDARD                   PNEUMON-IA (before)                  PNEUMON-IA (after)
-------------------------       ----------------------------         -------------------------
(PNEUMOCOC)                     (PNEUMOCOC)                          (PNEUMOCOC)
NIL                             (CLAMIDIA)                           NIL
NIL                             NIL                                  (CLAMIDIA MYCOPLASMA)
NIL                             (MYCOPLASMA VIRUS)                   NIL

                                FN: NIL                              FN: NIL
                                FP: (VIRUS MYCOPLASMA CLAMIDIA)      FP: (MYCOPLASMA CLAMIDIA)
                                OM: NIL                              OM: NIL

----------------------------------------------------------------------------------------------------

CASE: B12V1

GOLD-STANDARD                   PNEUMON-IA (before)                  PNEUMON-IA (after)
--------------------------      --------------------------           --------------------------
(MYCOPLASMA)                    (MYCOPLASMA)                         (MYCOPLASMA)
NIL                             (VIRUS)                              (VIRUS)
NIL                             NIL                                  NIL
(LEGIONELLA CLAMIDIA FEBRE-Q)   (CLAMIDIA FEBRE-Q)                   NIL
(PNEUMOCOC)                     (PNEUMOCOC)                          (CLAMIDIA FEBRE-Q LEGIONELLA)
NIL                             NIL                                  (PNEUMOCOC)

                                FN: (LEGIONELLA)                     FN: NIL
                                FP: (VIRUS)                          FP: (VIRUS)
                                OM: NIL                              OM: NIL

----------------------------------------------------------------------------------------------------

CASE: B13V1

GOLD-STANDARD                   PNEUMON-IA (before)                  PNEUMON-IA (after)
--------------------------      --------------------------           --------------------------
(ANA)                           (ANA)                                (ANA)
NIL                             NIL                                  (ENTEROBACTERIES)
(ENTEROBACTERIES PNEUMOCOC)     NIL                                  NIL

                                FN: (PNEUMOCOC ENTEROBACTERIES)      FN: (PNEUMOCOC)
                                FP: NIL                              FP: NIL
                                OM: NIL                              OM: NIL

----------------------------------------------------------------------------------------------------
```

```
CASE: B14V1

GOLD-STANDARD                  PNEUMON-IA (before)               PNEUMON-IA (after)
--------------------------     ---------------------------       --------------------------
(PNEUMOCOC)                    (PNEUMOCOC)                       (PNEUMOCOC)
(LEGIONELLA)                   (LEGIONELLA)                      (LEGIONELLA)

                               FN: NIL                           FN: NIL
                               FP: NIL                           FP: NIL
                               OM: NIL                           OM: NIL

------------------------------------------------------------------------------------------------------

CASE: B15V1

GOLD-STANDARD                  PNEUMON-IA (before)               PNEUMON-IA (after)
--------------------------     ---------------------------       --------------------------
NIL                            (SA)                              NIL
(ANA SA)                       (PNEUMOCOC)                       (PNEUMOCOC SA)
(ENTEROBACTERIES)              (ENTEROBACTERIES)                 (ENTEROBACTERIES)
NIL                            (ANA)                             (ANA)

                               FN: NIL                           FN: NIL
                               FP: (PNEUMOCOC)                   FP: (PNEUMOCOC)
                               OM: (ANA ENTEROBACTERIES)         OM: (ANA ENTEROBACTERIES)

------------------------------------------------------------------------------------------------------

CASE: B16V1

GOLD-STANDARD                  PNEUMON-IA (before)               PNEUMON-IA (after)
--------------------------     ---------------------------       --------------------------
(PNEUMOCOC)                    (PNEUMOCOC)                       (PNEUMOCOC)
NIL                            (CLAMIDIA)                        (ENTEROBACTERIES)
(LEGIONELLA ENTEROBACTERIES)   (ENTEROBACTERIES)                 (HEMOPHILUS)
NIL                            NIL                               (CLAMIDIA)

                               FN: (LEGIONELLA)                  FN: (LEGIONELLA)
                               FP: (CLAMIDIA)                    FP: (CLAMIDIA HEMOPHILUS)
                               OM: NIL                           OM: NIL

------------------------------------------------------------------------------------------------------

CASE: B17V1

GOLD-STANDARD                  PNEUMON-IA (before)               PNEUMON-IA (after)
--------------------------     ---------------------------       --------------------------
NIL                            NIL                               NIL
NIL                            NIL                               NIL
(MYCOPLASMA PNEUMOCOC LEGIONELLA)  (CLAMIDIA MYCOPLASMA PNEUMOCOC)  (MYCOPLASMA PNEUMOCOC LEGIONELLA)
(SA)                           (VIRUS)                           NIL

                               FN: (SA LEGIONELLA)               FN: (SA)
                               FP: (VIRUS CLAMIDIA)              FP: NIL
                               OM: NIL                           OM: NIL

------------------------------------------------------------------------------------------------------

CASE: B18V1

GOLD-STANDARD                  PNEUMON-IA (before)               PNEUMON-IA (after)
-------------------------      -----------------------------     --------------------------
(PNEUMOCOC)                    (PNEUMOCOC)                       (PNEUMOCOC)
NIL                            (ENTEROBACTERIES)                 NIL
(ENTEROBACTERIES LEGIONELLA)   (LEGIONELLA)                      (LEGIONELLA ENTEROBACTERIES)
NIL                            NIL                               (HEMOPHILUS)

                               FN: NIL                           FN: NIL
                               FP: NIL                           FP: (HEMOPHILUS)
                               OM: NIL                           OM: NIL

------------------------------------------------------------------------------------------------------
```

```
CASE: B19V1

GOLD-STANDARD                     PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------        ---------------------------         --------------------------
NIL                               (PNEUMOCOC)                         (PNEUMOCOC)
(PNEUMOCOC ENTEROBACTERIES)       NIL                                 (HEMOPHILUS)
NIL                               (ENTEROBACTERIES LEGIONELLA)        NIL
(ANA LEGIONELLA)                  (HEMOPHILUS)                        (ENTEROBACTERIES LEGIONELLA)

                                  FN: (ANA)                           FN: (ANA)
                                  FP: (HEMOPHILUS)                    FP: (HEMOPHILUS)
                                  OM: NIL                             OM: NIL

------------------------------------------------------------------------------------------------------

CASE: B20V1

GOLD-STANDARD                     PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------        ---------------------------         --------------------------
(ENTEROBACTERIES)                 (PNEUMOCOC)                         (PNEUMOCOC)
(PNEUMOCOC)                       NIL                                 NIL
NIL                               (ENTEROBACTERIES ANA)               NIL
(ANA LEGIONELLA)                  NIL                                 (ENTEROBACTERIES ANA LEGIONELLA)

                                  FN: (LEGIONELLA)                    FN: NIL
                                  FP: NIL                             FP: NIL
                                  OM: (PNEUMOCOC ENTEROBACTERIES)     OM: (PNEUMOCOC ENTEROBACTERIES)

------------------------------------------------------------------------------------------------------

CASE: G01V1

GOLD-STANDARD                     PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------        ---------------------------         --------------------------
NIL                               (MYCOPLASMA)                        (MYCOPLASMA)
(MYCOPLASMA PNEUMOCOC)            (VIRUS)                             NIL
(LEGIONELLA)                      (CLAMIDIA)                          (CLAMIDIA VIRUS)

                                  FN: (LEGIONELLA PNEUMOCOC)          FN: (LEGIONELLA PNEUMOCOC)
                                  FP: (CLAMIDIA VIRUS)                FP: (CLAMIDIA VIRUS)
                                  OM: NIL                             OM: NIL

------------------------------------------------------------------------------------------------------

CASE: G02V1

GOLD-STANDARD                     PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------        ---------------------------         --------------------------
(PNEUMOCOC)                       NIL                                 (LEGIONELLA)
(LEGIONELLA)                      (LEGIONELLA CLAMIDIA)               NIL
(ENTEROBACTERIES)                 NIL                                 (PNEUMOCOC ANA)
NIL                               NIL                                 (ENTEROBACTERIES)
NIL                               (ANA MYCOPLASMA VIRUS)              NIL
NIL                               NIL                                 NIL
NIL                               (PNEUMOCOC ENTEROBACTERIES)         NIL

                                  FN: NIL                             FN: NIL
                                  FP: (CLAMIDIA ANA MYCOPLASMA VIRUS) FP: (ANA)
                                  OM: (LEGIONELLA PNEUMOCOC)          OM: (LEGIONELLA PNEUMOCOC)

------------------------------------------------------------------------------------------------------

CASE: G03V1

GOLD-STANDARD                     PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------        ---------------------------         --------------------------
(MYCOPLASMA)                      (PNEUMOCOC)                         (PNEUMOCOC)
(CLAMIDIA)                        (CLAMIDIA)                          NIL
(FEBRE-Q)                         NIL                                 (MYCOPLASMA CLAMIDIA)
NIL                               (MYCOPLASMA VIRUS)                  NIL
(PNEUMOCOC LEGIONELLA)            NIL                                 NIL

                                  FN: (LEGIONELLA FEBRE-Q)            FN: (LEGIONELLA FEBRE-Q)
                                  FP: (VIRUS)                         FP: NIL
                                  OM: (MYCOPLASMA PNEUMOCOC)          OM: (PNEUMOCOC MYCOPLASMA)
                                      (MYCOPLASMA CLAMIDIA)               (PNEUMOCOC CLAMIDIA)

------------------------------------------------------------------------------------------------------
```

```
CASE: G04V1

GOLD-STANDARD                  PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------     ---------------------------      --------------------------
NIL                            (PNEUMOCOC)                      (PNEUMOCOC)
(HEMOPHILUS PNEUMOCOC)         NIL                              (LEGIONELLA)
(LEGIONELLA)                   NIL                              NIL

                               FN: (LEGIONELLA HEMOPHILUS)      FN: (HEMOPHILUS)
                               FP: NIL                          FP: NIL
                               OM: NIL                          OM: NIL

--------------------------------------------------------------------------------------------------------

CASE: G05V1

GOLD-STANDARD                  PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------     ---------------------------      --------------------------
(PNEUMOCOC)                    (PNEUMOCOC)                      (PNEUMOCOC)
(HEMOPHILUS)                   (ANA)                            (ANA)
NIL                            (ENTEROBACTERIES)                NIL
(LEGIONELLA ENTEROBACTERIES)   NIL                              (ENTEROBACTERIES HEMOPHILUS)

                               FN: (LEGIONELLA HEMOPHILUS)      FN: (LEGIONELLA)
                               FP: (ANA)                        FP: (ANA)
                               OM: NIL                          OM: NIL

--------------------------------------------------------------------------------------------------------

CASE: G06V1

GOLD-STANDARD                  PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------     ---------------------------      --------------------------
(PNEUMOCOC)                    (PNEUMOCOC)                      (PNEUMOCOC)
(MYCOPLASMA)                   NIL                              (LEGIONELLA)
(LEGIONELLA)                   NIL                              NIL

                               FN: (LEGIONELLA MYCOPLASMA)      FN: (MYCOPLASMA)
                               FP: NIL                          FP: NIL
                               OM: NIL                          OM: NIL

--------------------------------------------------------------------------------------------------------

CASE: G07V1

GOLD-STANDARD                  PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------     ---------------------------      --------------------------
NIL                            (MYCOPLASMA)                     (MYCOPLASMA)
(MYCOPLASMA LEGIONELLA)        (VIRUS)                          NIL
NIL                            (ANA)                            (LEGIONELLA ANA)
(ANA PNEUMOCOC)                NIL                              (PNEUMOCOC)

                               FN: (PNEUMOCOC LEGIONELLA)       FN: NIL
                               FP: (VIRUS)                      FP: NIL
                               OM: NIL                          OM: NIL

--------------------------------------------------------------------------------------------------------

CASE: G08V1

GOLD-STANDARD                  PNEUMON-IA (before)              PNEUMON-IA (after)
--------------------------     ---------------------------      --------------------------
(PNEUMOCOC)                    (PNEUMOCOC)                      (PNEUMOCOC)
(HEMOPHILUS)                   (ENTEROBACTERIES)                (LEGIONELLA)
(ENTEROBACTERIES)              NIL                              NIL
(LEGIONELLA)                   NIL                              (ENTEROBACTERIES HEMOPHILUS)

                               FN: (LEGIONELLA HEMOPHILUS)      FN: NIL
                               FP: NIL                          FP: NIL
                               OM: NIL                          OM: (HEMOPHILUS LEGIONELLA)
                                                                    (ENTEROBACTERIES LEGIONELLA)

--------------------------------------------------------------------------------------------------------
```

```
CASE: G09V1

GOLD-STANDARD                   PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------      ---------------------------         --------------------------
(PNEUMOCOC)                     (PNEUMOCOC)                         (PNEUMOCOC)
NIL                             NIL                                 (MYCOPLASMA)
(LEGIONELLA HEMOPHILUS)         (CLAMIDIA MYCOPLASMA)               NIL
NIL                             (VIRUS)                             NIL
NIL                             NIL                                 (VIRUS CLAMIDIA LEGIONELLA)

                                FN: (HEMOPHILUS LEGIONELLA)         FN: (HEMOPHILUS)
                                FP: (VIRUS CLAMIDIA MYCOPLASMA)     FP: (VIRUS CLAMIDIA MYCOPLASMA)
                                OM: NIL                             OM: NIL
----------------------------------------------------------------------------------------------------
CASE: G10V1

GOLD-STANDARD                   PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------      ---------------------------         --------------------------
(HEMOPHILUS)                    (PNEUMOCOC)                         (PNEUMOCOC)
NIL                             NIL                                 NIL
(ANA PNEUMOCOC)                 (ENTEROBACTERIES LEGIONELLA)        (ENTEROBACTERIES LEGIONELLA)
(ENTEROBACTERIES                NIL                                 (HEMOPHILUS)

                                FN: (HEMOPHILUS ANA)                FN: (ANA)
                                FP: (LEGIONELLA)                    FP: (LEGIONELLA)
                                OM: NIL                             OM: (HEMOPHILUS PNEUMOCOC)
                                                                        (HEMOPHILUS ENTEROBACTERIES)
----------------------------------------------------------------------------------------------------
CASE: G11V1

GOLD-STANDARD                   PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------      ---------------------------         --------------------------
(ANA)                           (PNEUMOCOC)                         (PNEUMOCOC)
NIL                             (LEGIONELLA)                        (ANA)
NIL                             (ANA)                               NIL
NIL                             (ENTEROBACTERIES)                   NIL

                                FN: NIL                             FN: NIL
                                FP: (ENTEROBACTERIES                FP: (PNEUMOCOC)
                                     LEGIONELLA PNEUMOCOC)
                                OM: NIL                             OM: NIL
----------------------------------------------------------------------------------------------------
CASE: G12V1

GOLD-STANDARD                   PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------      ---------------------------         --------------------------
(PNEUMOCOC)                     (PNEUMOCOC)                         (PNEUMOCOC)
NIL                             NIL                                 (LEGIONELLA)
(ENTEROBACTERIES LEGIONELLA)    (LEGIONELLA CLAMIDIA)               (ENTEROBACTERIES)
NIL                             NIL                                 NIL
NIL                             (ENTEROBACTERIES VIRUS)             NIL
NIL                             NIL                                 NIL
NIL                             (CMV ANA)                           NIL

                                FN: NIL                             FN: NIL
                                FP: (ANA CMV CLAMIDIA VIRUS)        FP: NIL
                                OM: NIL                             OM: NIL
----------------------------------------------------------------------------------------------------
CASE: G13V1

GOLD-STANDARD                   PNEUMON-IA (before)                 PNEUMON-IA (after)
--------------------------      ---------------------------         --------------------------
(PNEUMOCOC)                     (ANA)                               (PNEUMOCOC)
NIL                             NIL                                 (ANA)
(ENTEROBACTERIES ANA)           NIL                                 NIL
(LEGIONELLA)                    NIL                                 (ENTEROBACTERIES LEGIONELLA)
NIL                             (PNEUMOCOC ENTEROBACTERIES          (HEMOPHILUS)
                                 LEGIONELLA HEMOPHILUS)

                                FN: NIL                             FN: NIL
                                FP: (HEMOPHILUS)                    FP: (HEMOPHILUS)
                                OM: (PNEUMOCOC ANA)                 OM: NIL
----------------------------------------------------------------------------------------------------
```

```
CASE: G14V1

GOLD-STANDARD                        PNEUMON-IA (before)                   PNEUMON-IA (after)
--------------------------           ---------------------------           --------------------------
(PNEUMOCOC)                          NIL                                   (PNEUMOCOC)
(ANA)                                NIL                                   NIL
NIL                                  (PNEUMOCOC ANA CLAMIDIA)              (LEGIONELLA ANA)
NIL                                  NIL                                   (VIRUS)
(HEMOPHILUS LEGIONELLA               (HEMOPHILUS VIRUS)                    NIL
 ENTEROBACTERIES)
NIL                                  (ENTEROBACTERIES)                     (ENTEROBACTERIES HEMOPHILUS)

                                     FN: (LEGIONELLA)                      FN: NIL
                                     FP: (CLAMIDIA VIRUS)                  FP: (VIRUS)
                                     OM: NIL                               OM: NIL

------------------------------------------------------------------------------------------------------------

CASE: G15V1

GOLD-STANDARD                        PNEUMON-IA (before)                   PNEUMON-IA (after)
--------------------------           ---------------------------           --------------------------
(PNEUMOCOC)                          (PNEUMOCOC)                           NIL
NIL                                  NIL                                   (PNEUMOCOC LEGIONELLA)

                                     FN: NIL                               FN: NIL
                                     FP: NIL                               FP: (LEGIONELLA)
                                     OM: NIL                               OM: NIL

------------------------------------------------------------------------------------------------------------

CASE: G16V1

GOLD-STANDARD                        PNEUMON-IA (before)                   PNEUMON-IA (after)
--------------------------           ---------------------------           --------------------------
(PNEUMOCOC)                          NIL                                   (PNEUMOCOC)
(SA)                                 (SA PNEUMOCOC)                        (SA)

                                     FN: NIL                               FN: NIL
                                     FP: NIL                               FP: NIL
                                     OM: NIL                               OM: NIL

------------------------------------------------------------------------------------------------------------

CASE: G17V1

GOLD-STANDARD                        PNEUMON-IA (before)                   PNEUMON-IA (after)
--------------------------           ---------------------------           --------------------------
(FEBRE-Q)                            (VIRUS)                               (MYCOPLASMA)
(PNEUMOCOC)                          (MYCOPLASMA)                          NIL
NIL                                  NIL                                   NIL
(LEGIONELLA MYCOPLASMA)              NIL                                   (VIRUS PNEUMOCOC ANA)
(CLAMIDIA)                           (TUBERCULOSI PNEUMOCOC ANA)           (ENTEROBACTERIES)
NIL                                  (ENTEROBACTERIES)                     NIL

                                     FN: (CLAMIDIA FEBRE-Q LEGIONELLA)     FN: (CLAMIDIA FEBRE-Q LEGIONELLA)
                                     FP: (ENTEROBACTERIES VIRUS            FP: (ENTEROBACTERIES VIRUS ANA)
                                          TUBERCULOSI ANA)
                                     OM: (MYCOPLASMA PNEUMOCOC)            OM: (MYCOPLASMA PNEUMOCOC)

------------------------------------------------------------------------------------------------------------

CASE: G18V1

GOLD-STANDARD                        PNEUMON-IA (before)                   PNEUMON-IA (after)
--------------------------           ---------------------------           --------------------------
(PNEUMOCOC)                          (PNEUMOCOC)                           (PNEUMOCOC)
(FEBRE-Q)                            NIL                                   (HEMOPHILUS)

                                     FN: (FEBRE-Q)                         FN: (FEBRE-Q)
                                     FP: NIL                               FP: (HEMOPHILUS)
                                     OM: NIL                               OM: NIL

------------------------------------------------------------------------------------------------------------
```

```
CASE: G19V1

GOLD-STANDARD                 PNEUMON-IA (before)                PNEUMON-IA (after)
---------------------------   ---------------------------        --------------------------
(PNEUMOCOC)                   (PNEUMOCOC)                        (PNEUMOCOC)
(LEGIONELLA)                  (CLAMIDIA)                         (LEGIONELLA)
NIL                           (MYCOPLASMA)                       NIL
NIL                           (VIRUS)                            (MYCOPLASMA CLAMIDIA)
NIL                           NIL                                (VIRUS)

                              FN: (LEGIONELLA)                   FN: NIL
                              FP: (VIRUS CLAMIDIA MYCOPLASMA)    FP: (VIRUS CLAMIDIA MYCOPLASMA)
                              OM: NIL                            OM: NIL
-----------------------------------------------------------------------------------------------

CASE: G20V1

GOLD-STANDARD                 PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------    ---------------------------        --------------------------
(PNEUMOCOC)                   (PNEUMOCOC)                        (PNEUMOCOC)
(CLAMIDIA)                    (ENTEROBACTERIES)                  NIL
NIL                           NIL                                (ENTEROBACTERIES HEMOPHILUS)

                              FN: (CLAMIDIA)                     FN: (CLAMIDIA)
                              FP: (ENTEROBACTERIES)              FP: (ENTEROBACTERIES HEMOPHILUS)
                              OM: NIL                            OM: NIL
-----------------------------------------------------------------------------------------------

CASE: G22V1

GOLD-STANDARD                 PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------    ---------------------------        --------------------------
(PNEUMOCOC)                   (PNEUMOCOC)                        NIL
(ANA)                         (ENTEROBACTERIES)                  (PNEUMOCOC ANA)
NIL                           (TUBERCULOSI)                      (SA)
(SA STR-A)                    NIL                                NIL

                              FN: (STR-A SA ANA)                 FN: (STR-A)
                              FP: (TUBERCULOSI ENTEROBACTERIES)  FP: NIL
                              OM: NIL                            OM: NIL
-----------------------------------------------------------------------------------------------

CASE: G23V1

GOLD-STANDARD                 PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------    ---------------------------        --------------------------
(PNEUMOCOC)                   NIL                                (PNEUMOCOC)
(HEMOPHILUS)                  (PNEUMOCOC ENTEROBACTERIES)        (ENTEROBACTERIES)
(TUBERCULOSI)                 NIL                                NIL
NIL                           (CLAMIDIA VIRUS)                   NIL
NIL                           (MYCOPLASMA)                       (HEMOPHILUS CLAMIDIA VIRUS)
NIL                           NIL                                NIL
NIL                           NIL                                (TUBERCULOSI MYCOPLASMA)

                              FN: (TUBERCULOSI HEMOPHILUS)       FN: NIL
                              FP: (MYCOPLASMA CLAMIDIA           FP: (MYCOPLASMA CLAMIDIA
                                  ENTEROBACTERIES VIRUS)             ENTEROBACTERIES VIRUS)
                              OM: NIL                            OM: NIL
-----------------------------------------------------------------------------------------------

CASE: G24V1

GOLD-STANDARD                 PNEUMON-IA (before)                PNEUMON-IA (after)
--------------------------    ---------------------------        --------------------------
NIL                           (CLAMIDIA)                         (MYCOPLASMA)
NIL                           NIL                                NIL
(MYCOPLASMA CLAMIDIA FEBRE-Q) NIL                                (VIRUS CLAMIDIA)
NIL                           NIL                                (FEBRE-Q)

                              FN: (FEBRE-Q MYCOPLASMA)           FN: NIL
                              FP: NIL                            FP: (VIRUS)
                              OM: NIL                            OM: NIL
-----------------------------------------------------------------------------------------------
```

```
CASE: G25V1

GOLD-STANDARD                        PNEUMON-IA (before)                    PNEUMON-IA (after)
---------------------------          ---------------------------            --------------------------
NIL                                  (PNEUMOCOC)                            (PNEUMOCOC)
(ENTEROBACTERIES PNEUMOCOC)          (CLAMIDIA)                             NIL
(LEGIONELLA)                         (ANA)                                  (ENTEROBACTERIES LEGIONELLA)
(HEMOPHILUS)                         NIL                                    NIL
NIL                                  (ENTEROBACTERIES HEMOPHILUS)           NIL

                                     FN: (LEGIONELLA)                       FN: (HEMOPHILUS)
                                     FP: (CLAMIDIA ANA)                     FP: NIL
                                     OM: NIL                                OM: NIL
--------------------------------------------------------------------------------------------------------------
```

Annotated Bibliography

An annotated bibliography on rule-based expert system validation follows, without aiming to be exhaustive. An entry in the bibliography is composed of a reference and a comment, with the following format:

Reference: author, year, title, journal/proceedings/book, pages
Comment: short description. *KEYWORD*.

The last word (o words) in the comment part is a keyword on the topic considered at the entry. The list of keywords is the following:

ARC:	Expert System Architectures
CAS:	Expert System Validation: Case Studies
CSM:	Conventional Software Methodologies
CSV:	Conventional Software Validation
ESM:	Expert Systems Methodologies
ESV:	Expert System Validation (general)
EVA:	Expert System Evaluation
KAC:	Knowledge Acquisition
MAI:	Expert System Maintenance
REF:	Knowledge Base Refinement
SPE:	Expert System Specification
TER:	Expert System Validation Terminology
TES:	Expert System Testing
VER:	Expert System Verification

Agarwal R., Tanniru M. (1991). A Petri-Net Based Approach for Verifying the Integrity of Production Systems. *Workshop on Knowledge-Based Systems: Verification, Validation and Testing, AAAI'91.*

Describes a method to test rule bases for redundancy, inconsistency and incompleteness at both local and global levels using Petri nets. *VER.*

Adrion W.R., Branstad M.A., Cherniavsky J.C. (1982). Validation, Verification and Testing of Computer Software. *Computing Surveys*, **14**(2), 159-192.

Excellent survey on validation of conventional software. Validation fundamentals are clearly explained, independently of the kind of software. It also contains a glossary defining a set of key terms on this field. *CSV.*

Anjewierden A. (1987). Knowledge Acquisition Tools. *AI Communications*, 0 (1) 29-38.

Description of tools to help knowledge engineers in knowledge acquisition. *KAC.*

Ayel M. (1988). Protocols for Consistency Checking in Expert System Knowledge Bases. *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, 220-225.

Describes the SACCO system, a non-exhaustive, heuristic-based inconsistency checker. Incoherence conjectures are used to guide verification. *VER.*

Ayel M., Rousset M.C. (1989). *La cohérence dans les bases de connaissances.* Collection Intelligence Artificielle, Cepandues Editions, Toulouse, France.

Studies the notion of coherence, a synonym of what is called consistency. It describes the coherence checking systems INDE, SUPER and COVADIS (exhaustive approaches), and the systems TIBRE and SACCO (heuristic approaches). *VER.*

Ayel M., Laurent J.P. (1991). *Validation, Verification and Test of Knowledge-Based Systems*, Ayel & Laurent eds., John Wiley & Sons.

Collection of papers on ES validation, produced from the workshop on KBS Validation associated with the ECAI'90 conference. *ESV.*

Bachant J., McDermott J. (1984). R1 Revisited: Four Years in the Trenches. *AI Magazine* **5**(3) 21-32.

Detailed description of R1 evolution during four years (1980-1984) of usage. Specially interesting are the performance data. *CAS.*

Barrett B. W. (1990). A Software Quality Specification Methodology for Knowledge-Based Systems. *Workshop on Knowledge-Based Systems: Verification, Validation and Testing, AAAI'90.*

Considers aspects of software quality for ESs, outlining a specification methodology. *ESV.*

Batarekh A. Preece A., Bennett A., Grogono P. (1991) Specifying an expert system. *Expert Systems with Applications*, **2**(3).

An ES specification is composed of a problem specification and a solution specification. Different aspects for these two parts are analyzed. *SPE*.

Beauvieux A., Dague P. (1990). A General Consistency (Checking and Restoring) Engine for Knowledge Bases. *Proceedings of the 9th European Conference on Artificial Intelligence*, 77-82.

Describes the GCE system, an incremental consistency checker for attribute-value rule bases. *VER*.

Bellman C. L. (1990). The Modelling Issues Inherent in Testing and Evaluating Knowledge-based Systems. *Expert Systems with Applications*, **1**(3), 199-216.

Highlights the role of models in ES development and testing. Rules cannot be seen in isolation, but they should be conceived and tested taking into account their supporting models. The idea of partial models (or minimodels) is introduced. *ESM*.

Belmonte M. (1990). *RENOIR: un sistema experto para la ayuda en el diagnóstico de colagenosis y artropatías inflamatorias*. PhD dissertation. Universitat Autonoma de Barcelona.

Describes RENOIR, an ES for rheumatology diagnosis. Interesting and substantive work in multi-expert validation. *CAS*.

Benbasat I., Dhaliwal J.S. (1989). A Framework for the Validation of Knowledge Acquisition. *Knowledge Acquisition*, **1**(2), 215-233.

Proposes a framework for validation in the ES life-cycle, with four validation stages: conceptual, elicitation, implementation and representation. Large list of criteria for validation assessment. *ESV*.

Bobrow D.G., Mittal S., Stefik M.J. (1986). Expert Systems: Perils and Promise. *Communications of the ACM*, **29**(9), 880-894.

Reviews a number of experiences in ESs, extracting a number of guidelines for application selection, knowledge acquisition and ES development. *ESM*.

Boehm B. W., Brown J.R., Kaspar H., Lipow M., MacLeod G.J., Merrit M.J. (1978). *Characteristics of Software Quality*. North-Holland.

Detailed enumeration of conventional software quality characteristics, criteria and measurements. *CSM*.

Boehm B. W. (1988). A Spiral Model of Software Development and Enhancement. *Computer*, May, 61-72.

Describes the spiral model of software development. Interesting summary of previous software models. Also in [Gupta 91]. *CSM*.

Breuker J. Wielinga B. (1988). Models of Expertise in Knowledge Acquisition. In *Topics In Expert Systems Design: methodologies and tools*, Guida & Tasso eds., North Holland.

Presents the KADS conceptual modelling language, that can be used as an intermediary representation between data on expertise and the design/implementation of an ES. *KAC.*

Buchanan B. G., Barstow D., Bechtel R., Bennett J., Clancey W., Kulikowski C., Mitchell T. and Waterman D. (1983). Constructing an expert system. In *Building Expert Systems*, Hayes-Roth F. D., Waterman A. and Lenat D. B. eds, 127-168. Addison-Wesley.

Contains the first comprehensive description of ES development. Clear and detailed. Locates validation at the last development step. *ESM.*

Buchanan B.G., Shortliffe E.H. (1984). *Rule-Based Expert Systems, The MYCIN Experiments of the Stanford Heuristic Programming Project.* Buchanan and Shortliffe eds. Addison-Wesley.

Exhaustive description of the MYCIN project, very valuable for the ES topic. Regarding validation, chapter 8 describes the first verifier, and chapters 30 and 31 provide a clear and vivid description of the MYCIN testing and evaluation procedures. Chapter 34 contains a survey assessing physicians opinions on ESs in medicine. Very interesting summary of learned lessons. *ESV, VER, TES, EVA, CAS.*

Buchanan B. (1987). Artificial Intelligence as an Experimental Science. *Synthese.*

Introduces some useful concepts to evaluate ES complexity. *EVA.*

Cain T. (1991). The DUCTOR: A Theory Revision System for Propositional Domains. *Proceedings of the 8th International Workshop on Machine Learning*, 485-489.

Explains the theory revision component of the DUCTOR system, an integrate learner that combines deductive, inductive and abductive methods. *REF.*

Chang C.L., Combs J.B., Stachowitz R.A. (1990). A Report on the Expert Systems Validation Associate (EVA). *Expert Systems with Applications*, **1**(3), 217-230.

The EVA system is described as an extensive set of verification utilities with a generic aim. It is based on the notion of metapredicate (verification predicate on predicates of rule based languages). *VER.*

Chandrasekaran B. (1983). On Evaluating AI Systems for Medical Diagnosis. *AI Magazine*, 4(2), 34-37.

Describes specific issues for evaluating medical ESs. Proposes a type of Turing test to cope with these issues. *EVA.*

Chandrasekaran B. (1987). Towards a Functional Architecture for Intelligence Based on Generic Information Processing Tasks. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence*, 1183-1192.

Describes the generic task approach for problem solving. A generic task is an abstraction regarding the task function, the knowledge representation and the control strategy. Generic tasks aims at being the basic building blocks for problem solving. *ARC.*

Childress R, Valtorta M. (1991). EVA and the Verification of Expert Systems Written in OPS5. *Workshop on Knowledge-Based Systems: Verification, Validation and Testing, AAAI'91.*

Describes the process of translating OPS5 applications into the EVA environment. *VER.*

Constantine M.M., Ulvila J.W. (1990). Testing Knowledge-Based Systems: The State of the Practice and Suggestions for Improvement. *Expert Systems with Applications*, **1**(3), 237-248.

Report on a survey performed on US Army members with experience in developing and testing KBS. Some recommendations are given. *CAS.*

Cragun B.J. and Steudel H.J. (1987). A Decision-Table-Based Processor for Checking Completeness and Consistency in Rule-Based Expert Systems. *International Journal of Man-Machine Studies*, **26**, 633-648.

Describes the ESC system, which follows the ONCOCIN RULE CHECKER approach using decision tables. Some efficiency improvements are obtained. *VER.*

Craw S., Sleeman D. (1990). Automating the Refinement of Knowledge-Based Systems. *Proceedings of the 9th European Conference on Artificial Intelligence*, 167-172.

Describes KRUST, a refinement system for propositional rule bases. Rule priorities are taken into account to generate and select refinements. *REF.*

Craw S., Sleeman D. (1991) The Flexibility of Speculative Refinement. *Proceedings of the 8th International Workshop on Machine Learning*, 28-32.

Explains KRUST on the basis that all possible refinements are generated. Empirical results of KRUST acting on manually altered rule bases are given. *REF.*

deKleer, J. (1986). An Assumption-based TMS. *Artificial Intelligence* **28**:127-162.

The first paper on ATMS (there are two more on the same issue). Here deKleer introduces ATMS basic concepts, particularly the concepts of label and environment, used later for verification purposes.

Evertsz R. (1991). The Automatic Analysis of Rule-based System Based on their Procedural Semantics. *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, 22-27.

Describes AbsPS, a verifier that checks redundancy, cycles and chuncking on rule bases underlying first order logic. AbsPS is based on an abstract interpretation of production systems, considering the procedural semantics of their language. *VER.*

Fujii M. S. (1977). Independent Verification of Highly Reliable Programs. *COMPSAC 77*, IEEE. 1, 38-44.

Introduces the concept of verification as an independent activity, that can be performed in different ways at the different stages of software development. Also in [Miller & Howden, 78]. *CSV.*

Gaines B.R. (1987). An overview of knowledge-acquisition and transfer. *International Journal of Man-Machine Studies* **26**, 453-472.

Study about the expertise, the role it plays in society and techniques for its acquisition. *KAC.*

Gaschnig J., Klahr P., Pople H., Shortliffe E., Terry A. (1983). Evaluation of Expert Systems: Issues and case Studies. In *Building Expert Systems*, Hayes-Roth, Waterman, Lenat eds., Addison Wesley.

The first comprehensive approach to the problem of ES testing and evaluation. Two case studies are included: R1 and ORNL. *TES, EVA, CAS.*

Geissman J. R., and Schultz R. D. (1988). Verification and Validation of Expert Systems. *AI Expert*, February 1988, 26-33.

Describes the rapid prototyping approach and provides six validation guidelines on it. Also in [Gupta 91]. *ESV.*

Ginsberg A., Weiss S., Politakis P.G. (1985). SEEK2 : A Generalized Approach to Automatic Knowledge Base Refinement. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence (IJCAI'85)*, 367-374.

Introduces the SEEK2 system as a set of extensions and improvements of SEEK. *REF.*

Ginsberg A. (1986). A Metalinguistic Approach to the Construction of Knowledge Base Refinement Systems. *Proceedings of the Fifth National Conference on Artificial Intelligence (AAAI'86)*, 436-441.

Describes RM, a metalanguage for refinement, on which SEEK2 was successfully reimplemented. *REF.*

Ginsberg A. (1988a). *Automatic Refinement of Expert System Knowledge Bases.* Pitman Research Notes in Artificial Intelligence.

Describes the SEEK2 system, based on SEEK with enhanced features. Among them, the RM metalanguage for knowledge base refinement. Previous work on SEEK2 and RM can be found in [Ginsberg et al, 85] and [Ginsberg 86]. *REF.*

Ginsberg A. (1988b). Knowledge-Base Reduction: A New Approach to Checking Knowledge Bases for Inconsistency & Redundancy. *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88)*, 585-589.

Describes the reduction of propositional rule bases in terms of labels and environments. The KB-REDUCER system checks reduced rule bases for inconsistency and redundancy. Reduction is the first step in the refinement schema proposed in [Ginsberg 88c]. Also in [Gupta 91]. *VER.*

Ginsberg A. (1988c). Theory revision via Prior Operationalization. *Proceedings of the Seventh National Conference on Artificial Intelligence (AAAI'88)*, 590-595.

Introduces refinement of theories as a three step process and describes the second step: the RTLS system that refines reduced propositional theories. *REF.*

Ginsberg (1990). Theory reduction, theory revision and retranslation. *Proceedings of the Eighth National Conference on Artificial Intelligence(AAAI'90)*. 777-782.

Describes the top-down retranslation, a method solving the third step in the theory revision schema proposed by Ginsberg. *REF*.

Gladden G.R. (1982). Stop the Life-Cycle, I Want to Get Off. *ACM Software Engineering Notes*, April 82, 35-39.

Enumerates several lacks and deficiencies of the waterfall model. *CSM*.

Goodenough J.B., Gerhart S.L. (1975). Towards a theory of test data selection. *IEEE Transactions on Software Engineering*, **SE-1**(2).

Landmark paper on formal treatment of test data selection, including the fundamental theorem of testing. Also in [Miller & Howden 78]. *CSV*.

Green C.J.R., Keyes M.M. (1987). Verification and Validation of Expert Systems. *Western Conference on Expert Systems*, 38-43.

Introduces the global question of ES validation (with mention to the "vicious circle"), and suggests a validation methodology composed of five tasks. *ESV*.

Grogono P., Batarekh A., Preece A., Shinghal R., Suen C. (1992). Expert System Evaluation Techniques: A Selected Bibliography. *Expert Systems*, to appear.

Extense collection of references on ES evaluation, including verification and validation. A detailed analysis of the different issues on the topic is given. *ESV*.

Gupta U.G., Biegel J. (1990). RITCaG: A Rule-Based Intelligent Test Case Generator. *Workshop on Knowledge-Based Systems: Verification, Validation and Testing, AAAI'90*.

Describes the RITCaG system, a test case generator for ART rule bases. *TES*.

Gupta U.G. (1991). *Validating and Verifying Knowledge-Based Systems*. IEEE Computer Society Press, Los Alamitos, California.

Collection of papers on ES verification, validation and testing, including related topics as development methodologies and case studies. Many of them are referenced individually in this bibliography. *ESV*.

Hamilton D., Kelley K., Culbert C. (1991). State-of-the-Practice in Knowledge-based System Verification and Validation. *Expert Systems with Applications*, **3**, 403-410.

Describes the results of a survey on KBS developers and users assessing the validation degree in actual KBS applications. The survey included both questionnaires and interviews. Some recommendations are given. *CAS*.

Hart A. (1986). *Knowledge Acquisition for Expert Systems*. Kogan Page Ltd.

Detailed introduction to KA, with an enumeration of KA techniques. *KAC*.

Herod J.M., Bahill A.T. (1991). Ameliorating the Pregnant Man problem: a verification tool for personal computer based expert systems. *International Journal of Man-Machine Studies* **35**, 789-805.

Introduces a tool to verify the correctness of question sequencing in ES. *VER*.

Hollnagel E. (1989). *The Reliability of Expert Systems*. Hollnagel ed. Ellis Horwood Limited. John Wiley & Sons.

Collection of papers on ES reliability, result of a seminar on the topic. *ESV*.

Hoppe T., Meseguer P. (1991). On the Terminology of VVT. *Proceedings of the European Workshop on Verification, Validation and Testing of KBS (EUROVAV'91)*, 3-14.

Comparative analysis of validation terminologies used by several authors, with a synthesis proposal. The concepts of formalizable and informal specifications are first introduced. *TER*.

Howden W.E. (1976). Reliability of of the path analysis testing strategy. *IEEE Transactions on Software Engineering*, **SE-2**, 3.

Shows the no existence of an adequate algorithm for the fundamental theorem of testing [Goodenough & Gerhart 75]. Also in [Miller & Howden 78]. *CSV*.

Howden W.E. (1978). A Survey of Dynamic Analysis Methods. In *Tutorial: Software Testing & Validation Techniques*, Miller and Howden eds. IEEE Computer Society.

Detailed survey of dynamic testing methods for conventional software. The underlying ideas of these methods can be applicable to ES. *CVS*.

Krause P., O'Neil M., Glowinski A. (1991). Can we Formally Specify a Medical Decision Support System?. *Proceedings of the European Workshop on Verification, Validation and Testing of KBS (EUROVAV'91)*, 247-258.

Describes an experience using the formal language Z to formally specify a medical ES. They conclude that are strong limitations for an ES to be completely specified. *SPE*.

Landauer C. (1990). Correctness Principles for Rule-Based Expert Systems. *Expert Systems with Applications*, **1**(3), 291-316.

Introduces five correctness principles in rule bases (consistency, completeness, irredundancy, connectivity and distribution), providing a set of methods for their analysis. *VER*.

Laurent J. P. (1992) Proposals for a valid terminology in KBS validation. To appear in ECAI'92 conference.

Introduces the concepts of formal and pseudo-formal specifications for ESs. Divides validation in two main parts, verification and evaluation. *TER*.

Liebowitz J. (1986). Useful Approaches for Evaluation Expert Systems. *Expert Systems* 3(2) 86-96.

Differenciates evaluation from validation, giving a set of ES evaluation criteria. A practical experience on ES evaluation is reported. *EVA.*

Lopez B., Meseguer P., Plaza E. (1990). Knowledge Based Systems Validation: A State of the Art. *AI Communications*, 3(2), 58-72.

Study of available KBS validation techniques, centered on verification, refinement and evaluation. A set of recommendations for future research are given. *ESV.*

Lopez B. (1991). CONKRET: A Control Knowledge Refinement Tool, in *Verification, Validation and Test of Knowledge-Based Systems*, Ayel and Laurent eds., John Wiley and Sons.

Describes CONKRET a refinement tool specialized on control knowledge. *REF.*

McCracken D. D. and Jackson M. A. (1982). Life-Cycle Concept Considered Harmful. *ACM Software Engineering Notes*, April 82, 29-32.

An attack to the waterfall model, proposing the evolutionary model. *CSM.*

McDermott J. (1981). R1's formative years. *AI Magazine* 2(2), 21-29.

Description of R1 experience when the system was still not fully operational. See the continuation of this paper in [Bachant & McDermott 84]. *CAS.*

McDermott J. (1988). A Taxonomy of Problem Solving Methods. In *Automating Knowledge Acquisition for Expert Systems*, ed. S. Marcus, 225-256, Kluvier.

Introduces solving methods as generic components in the problem solving activity. These methods can be used in different ES tasks. *ARC.*

Meseguer, P. (1990). A New Method to Checking Rule Bases for Inconsistency: A Petri Net Approach. *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, 437-442.

Describes the PENIC system, an inconsistency checker for propositional rule bases using Petri nets. Inconsistencies are detected solving linear equation systems. *VER.*

Miller E, Howden W. E. (1978) *Tutorial: Software Testing & Validation Techniques.* IEEE Computer Society, Catalog No. EHO 138-8.

Collects a number of early papers (some of them are key papers, widely referenced) in conventional software verification, validation and testing. *CSV.*

Miller L. A. (1989). A Comprehensive Approach to the Verification and Validation of Knowledge-Based Systems. *Workshop on Verification, Validation and Testing of KBS, IJCAI'89.*

Describes a formal set of verification and validation activities (reviews, documents) to be performed at the different stages of ES development. *ESV.*

Miller L. A. (1990). Dynamic Testing of Knowledge bases Using the Heuristic Testing Approach. *Expert Systems with Applications*, **1**(3), 249-269.

Introduces the heuristic testing approach, specially suitable for component testing of rule bases. Most serious faults are tested first. Test cases are generated. *TES*.

Mullerburg M. (1984). Software Validation: A Bibliography. In *Software Validation*, Hausen ed., Elsevier Science Publishers, 335-359.

Contains an annotated bibliography on conventional software validation. A set of keywords are associated with each entry. *CSV*.

Nazareth D.L. (1989). Issues in the verification of knowledge in rule-based systems. *International Journal of Man-Machine Studies*, **30**, 255-271.

Contains a taxonomy of verification issues and verification methods. Extense list of references. *VER*.

Nazareth D.L., Kennedy M.K. (1991). Verification of rule-based knowledge using directed graphs. *Knowledge Acquisition* **3**, 339-360.

Provides a formal characterization of verification issues (redundancy, conflict, circularity, dead-end rules and unreachable goals) using directed graphs. *VER*.

Newell A. (1982). The Knowledge Level. *Artificial Intelligence*, 18, 87-127.

Analyzes the problem of knowledge and its representation it computers. Introduces a new level, denominated the knowledge level, that fits into existing computer levels. Explores consequences of the existence of this new level, and analyzes the relation with the the symbol (program) level.

Nguyen T.A., Pekins W.A., Laffey T.J., Pecora D. (1985). Checking an Expert System Knowledge Base for Consistency and Completeness. *Proceedings of the Ninth International Joint Conference on Artificial Intelligence* (*IJCAI'85*), 375-378.

Describes the CHECK system, which detects redundancy, subsumption, conflict, gaps, cycles and other issues in a LES rule base. Verification algorithms are explained. *VER*.

Nguyen T.A., Pekins W.A., Laffey T.J., Pecora D. (1987). Knowledge Base Verification. *AI Magazine*, **8**(2), 69-75.

A more detailed description of CHECK, including possible extensions to rules with certainty degrees. Also in [Gupta 91]. *VER*.

Nonfjall h., Larsen H.L. (1992). Detection of Potential Inconsistencies in Knowledge Bases. *International Journal of Intelligent Systems*, **7**, 81-96.

Presents an inconsistency checking method that is more efficient than the COVADIS system. This method can be extended to deal with uncertain knowledge bases. *VER*.

O'Keefe R.M., Balci O., Smith E.P. (1987). Validating Expert System Performance. *IEEE Expert*, Winter 87, 81-89.

This paper addresses some basic questions for ES validation (what, when, how) and provides quantitative and qualitative criteria for ES testing. Some statistical measures are suggested. Also in [Gupta 91]. *ESV, TES.*

O'Keefe R.M., and Lee S. (1990). An Integrative Model of Expert System Verification and Validation. *Expert Systems with Applications*, **1**(3), 231-236.

Describes a ES development methodology based on the evolutionary and spiral models, including verification at each development step. Specially focused on small-to-medium ES. *ESM.*

O'Keefe R.M, O'Leary D.E. (1991) The Verification and Validation of Expert Systems. *Tutorial MP5 KBS Verification Validation and Testing, AAAI'91.*

Extense description of current issues in ES verification, validation and testing. Some statistics methods are given. Long list of references. *ESV.*

Ourston D. and Mooney R. J. (1990). Changing the Rules: A Comprehensive Approach to Theory Refinement. *Proceedings of the Eighth National Conference on Artificial Intelligence* (*AAAI'90*), 815-820.

Describes a refinement method for propositional theories. ID3 is used for inductive learning. *REF.*

Pipard E. (1988). Detection d'incoherences et d'incompletitudes dans les bases de regles: le systeme INDE. *Proceedings of the 8th International Conference of Expert Systems and Their Applications* (*AVIGNON'88*), 15-33.

Explains the INDE system, which checks inconsistency, incompleteness and unfireable rules in rule bases using Petri nets. *VER.*

Polat F., Guvenir H.A. (1991). UVT: A Unification Based Tool for Knowledge Base Verification. *Proceedings of the European Workshop on Verification, Validation and Testing of KBS* (*EUROVAV'91*), 147-163.

Describes UVT, a verifier like CHECK which add the inferred rules before checking. A inferred rule is a condensed form to represent a rule chain. *VER.*

Politakis P., Weiss S.M. (1984). Using Empirical Analysis to Refine Expert System Knowledge Bases. *Artificial Intelligence*, **22**(1), 23-48.

Summary description of SEEK, one of the early systems of knowledge base refinement. See also [Politakis 85]. *REF.*

Politakis P.G. (1985). *Empirical Analysis for Expert Systems.* Research Notes in Artificial Intelligence. Pitman. Boston.

Detailed description of SEEK, a semi-automatic refinement system for tabular rule bases. See also [Politakis & Weiss 84]. *REF.*

Prakash G.R., Subramanian E., Mahabala H.N. (1991). A Methodology for Systematic Verification of OPS5-Based AI Applications. *Proceedings of the 12th International Joint Conference on Artificial Intelligence* (*IJCAI'91*), 3-8.

Describes the SVEPOA system, a verifier for OPS5. Verification issues are detected using a linear system of inequalities and testing for feasible integer solution. *VER.*

Preece A.D., Shinghal R. (1991). COVER: A Practical Tool for Verifying Rule-Based Systems. *Workshop on Knowledge-Based Systems: Verification, Validation and Testing, AAAI'91.*

Explains the COVER system, a verifier checking propositional rule bases for deficiency, ambivalence, redundancy and circularity. *VER.*

Radwan A. E., Goul M., O'Leary T. J., and Moffitt K. E. (1989). A Verification Approach for Knowledge-Based Systems. *Transportation Research-A*, **23A**(4), 287-300.

Develops an independent verification framework based on conventional software engineering to be added to Buchanan's model. Also in [Gupta 91]. *ESV.*

Rousset M.C. (1988). On the Consistency of Knowledge Bases: the Covadis System. *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, 79-84.

Describes the COVADIS system, an inconsistency checker for forward chaining, attribute-value rule bases. *VER.*

Royce W. W. (1970). Managing the Development of Large Software Systems. *Wescon 70*, IEEE, 1-9.

Describes the waterfall model of software development, introducing the concept of life-cycle. *CSM.*

Rushby J. (1988a). Validation and Testing of Knowledge-Based Systems. How Bad Can It Get?. *Workshop on Knowledge-Based Systems: Verification, Validation and Testing, AAAI'88.*

Shows the necessity of detailed requirements for KBS, introducing the concepts of service and competence requirements, which are subdivided into desired and minimum. Also in [Gupta 91]. *SPE.*

Rushby J. (1988b). *Quality Measures and Assurance for AI Software.* SRI-CSL-88-7R, SRI Project 4616.

Excellent report on software quality assurance. Commonalties, differences and weaknesses of AI software versus conventional software are identified. Conventional software methods (verification, testing, metrics) are analyzed for their transfer into the AI realm. Extense list of references. *CSV, ESV.*

Sedgewick R. (1988). *Algorithms.* Second edition, Addison-Wesley.

A good book about general-purpose algorithms.

Shadbolt N. (1991). Building Valid Knowledge Bases: An ACKnowledge Perspective. *Proceedings of the European Workshop on Verification, Validation and Testing of KBS (EUROVAV'91)*, 195-210.

Describes the work performed in the ACK project regarding validation of knowledge acquisition and conceptual models. *KAC.*

Shapiro E.Y. (1984). Alternation and the Computational Complexity of Logic Programs. *Journal of Logic Programming* **1**, 19-33.

Defines a set of complexity measures in logic programs that can be useful for ES evaluation. *EVA.*

Sierra C. (1989). *MILORD: Arquitectura multinivell per a sistemes experts en classificacio*, PhD dissertation, Universitat Politecnica de Catalunya.

Describes the MILORD shell, on which the ES model used in this thesis is based. *ESM.*

Sierra C., Agustí-Cullell J., Plaza E. (1991). Verification by Construction in MILORD. Proceedings of the *European Workshop on Verification and Validation of Knowledge-Based Systems* (*EUROVAV'91*), 211-226.

Describes a methodology for KB construction based on the notion of module, with the operations of composition and refinement. Verification is included in these operations. *ARC.*

Slage J.R., Gardiner D.A., Han K. (1990). Knowledge Specification of an Expert System. *IEEE Expert*, August 90, 29-38.

Introduces a methodology to knowledge specification, based on Sowa's conceptual graphs. A practical experience is reported. *SPE.*

Soloway E., Bachant J., Jensen K. (1987). Assessing the Maintainability of XCON-in-RIME: Coping with the Problem of a Very Large Rule Base. *Proceedings of the Sixth National Conference on Artificial Intelligence* (*AAAI'87*), 824-829.

Addresses the problem of XCON maintainability, which has a very large rule base (6,200 rules) with a 50% of change per year. *MAI.*

Steels L. (1990). Components of Expertise. *AI Magazine.*11(2) 28-49.

Introduces a framework for expressing expertise in terms of three types of components: tasks, models and problem-solving methods. A summary of other knowledge architecture approaches is given. *ARC.*

Suwa M., Scott A.C., Shortliffe E.H. (1982). Completeness and Consistency in Rule-Based Expert Systems. *AI Magazine*, **3**(4), 16-21.

Describes the ONCOCIN RULE CHECKER system, which checks redundancy, subsumption, conflicts and missing rules on ONCOCIN rule bases. Also in [Buchanan & Shortliffe 84] and in [Gupta 91]. *VER.*

Tao Y., Zhijun H., Ruizhao Y. (1987). Performance Evaluation of the Inference Structure in Expert System. *Proceedings of the Tenth International Joint Conference on Artificial Intelligence* (*IJCAI'87*), 945-950.

Describes a quantitative measure to evaluate the structural efficiency of a set of rules, based on Petri net concepts. *EVA*.

Valtorta M. (1991). Some results on the computational complexity of refining confidence factors. *International Journal of Approximate Reasoning*, **5**(2).

Shows that different instancies of refinement problems in rule bases with confidence factors are NP-hard. *REF*.

Verdaguer A. (1989). *PNEUMON-IA: Desenvolupament i validacio d'un sistema expert d'ajuda al diagnostic medic*. PhD dissertation. Universitat Autonoma de Barcelona.

Describes the PNEUMON-IA expert system, that has been validated using in-depth ii and improver. Specially interesting are the results of the multi-expert validation performed. *ESV*.

Vignolet L., Ayel M. (1990). A Conceptual Model for Building Sets of Test Samples for Knowledge Bases. *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI'90)*, 667-672.

Uses a conceptual model for test case building, including specific knowledge for the testing task. *TES*.

Weitzel J. R., Kerschberg L. (1989) Developing Knowledge Based Systems: Reorganizing The System Development Life Cycle. *Communications of the ACM*, **32**(4), 482-488.

Introduces a methodology for ES development based in the concept of process (that can be executed concurrently), mapping the basic phases of conventional software development. *ESV*.