

Evaluation of Verification Tools for Knowledge-Based Systems

Alun D Preece*

*University of Aberdeen
Department of Computing Science
Aberdeen, Scotland, UK*

Stéphane Talbot

*ESIGEC, Department of Artificial Intelligence
Université de Savoie
Chambéry, France*

Laurence Vignollet

*Laboratoire d'Intelligence Artificielle (LIA)
ESIGEC, Department of Artificial Intelligence
Université de Savoie
Chambéry, France*

Abstract

Validation has emerged as a significant problem in the development of knowledge-based systems (KBS). Verification of KBS correctness and completeness has been cited as one of the most difficult aspects of validation. A number of software tools have been developed to perform such verification, but none of these are in widespread use. One of the reasons for this is that little quantitative evidence exists to demonstrate the effectiveness of the tools. This paper presents an experimental study of three KBS verification tools: a consistency checker, a completeness checker, and a testing tool (for correctness). The tools are evaluated on their ability to reveal plausible faults seeded into a complex, realistic KBS application. The cost of using the tools is also measured. It is shown that each tool is independently effective at detecting certain kinds of fault, and that the capabilities of the tools are complementary — a result not revealed by previous studies.

Short running title: **Evaluation of Verification Tools for KBS**

*The work described in this paper was conducted while the first author was visiting LIA, Université de Savoie, Chambéry, France.

1. INTRODUCTION

In software engineering, validation is the process by which developers determine whether or not a system meets its users' requirements; it is therefore a crucial part of the development of any software. Recent surveys have shown that the importance of validation is recognised by developers of knowledge-based systems (KBS) ([O'Leary, 1991] and [Hamilton, Kelley and Culbert, 1991]). Validation is performed by KBS developers — up to and sometimes exceeding allotted budgets — but it is considered to be a difficult, time-consuming, and problematic process. Chief among the difficulties identified by KBS developers are the verification of KBS completeness and correctness [O'Leary, 1991]. Verification is the part of validation concerned with checking formally-defined properties of the system, such as consistency, instances of correctness (such as defined by test cases, for example), and instances of completeness (coverage of the input domain, for example) [Laurent, 1992].¹

A number of software tools have been developed over the past decade to assist developers in verifying the completeness and correctness of KBS. Three common types of tool can be identified: *consistency checkers* which detect conflicting and redundant knowledge in a knowledge base (for example, COVADIS [Rousset, 88], In-Depth [Meseguer, 1992], KB-Reducer [Ginsberg, 1988] and SACCO [Ayel and Vignollet, 1993]); *completeness checkers* which detect missing or deficient knowledge (for example, COCTO [Lounis and Ayel, 1995] and COVER [Preece, 1993]); and *testing tools* for KBS correctness using test cases (for example, EVA [Chang, Combs and Stachowitz, 1991] and SYCOJET [Ayel and Vignollet, 1993]). Despite the availability of such tools, however, few are in widespread use by KBS developers at present. One reason for the slow adoption of such tools is that little concrete evidence exists for their utility. Developers lack information to help them decide under what circumstances particular tools are likely to be useful to them.

1.1. Related Studies

Few experimental studies have been performed to evaluate the efficacy of the different types of tool on realistic KBS applications. Furthermore, from the studies that have been performed, important questions remain unanswered. A summary of some significant results and limitations of previous studies follows:

The Minnesota study Kirani, Zualkernan and Tsai [Kirani et al, 1992] at the University of Minnesota, USA, report on the application of several testing techniques to a sample KBS in the domain of VLSI manufacturing. With the exception of a simple consistency checking tool, all of the methods used were manual testing techniques. The KBS itself was a 41-rule production system based

¹Laurent in [Laurent, 1992] provides a detailed set of definitions for concepts in KBS validation and verification. For our purposes, we define *correctness* as the ability of the KBS to produce output which agrees with some external standard, *completeness* as the ability to produce output for every required input case, and *consistency* as the ability never to produce output which is contradictory (consistency is therefore a special case of correctness). Our use of the term verification in this paper agrees with [Benbasat and Dhaliwal, 1989].

upon well-understood physical properties of semiconductors, into which a variety of plausible faults were seeded. The results of the study showed that the manual testing techniques, though labour-intensive, were highly effective, while the consistency checker performed poorly in detecting the seeded faults. Unfortunately, the success of the manual testing techniques could be attributed to the fact that this KBS application was exhaustively testable. Furthermore, given that the consistency checker employed was of only the most basic type (able to compare pairs of rules only for conflict and redundancy), it is unsurprising that it performed poorly. Therefore, this study does not provide clear evidence — positive or negative — for the utility of modern KBS verification tools.

The SRI study Rushby and Crow [Rushby and Crow, 1990] at SRI, USA, like the Minnesota study, compared manual testing techniques with a simple rule checker. The application used was a 100-rule production system in an aerospace domain, but the structure of the system was largely "flat" and highly simple. Faults were not seeded in this study — instead, actual faults were discovered in the real application! — so there was no control on the results. While interesting, it does not demonstrate verification tool effectiveness.

The Concordia study Preece and Shinghal [Preece and Shinghal, 1992] at Concordia University, Canada, examined the use of a particular consistency and completeness checking tool, COVER, on a variety of KBS in different domains. COVER was shown to detect genuine and potentially-serious faults in each system to which it was applied. However, this study did not compare the effectiveness of different kinds of tool.

The SAIC study Miller, Hayes and Mirsky [Miller, Hayes and Mirsky, 1993] at SAIC, USA, performed a controlled experiment on two KBS built in the nuclear power domain. Faults were seeded in each system, and groups of KBS developers and domain experts attempted to locate the faults using three different verification techniques: manual inspection, the VERITE tool (an enhanced version of COVER [Preece and Shinghal, 1992]), and MetaCheck, a simulated tool based on a conceptual enhancement of VERITE. The VERITE tool and the MetaCheck pseudo-tool were shown to provide significant assistance to both the groups of KBS developers *and* domain experts in locating faults. However, like the Concordia study, only consistency and completeness checking was considered: testing tools were not employed.

1.2. Objectives of This Study

In summary, none of these previous studies has performed a controlled experimental comparative evaluation of a representative set of state-of-the-art KBS verification tools. The objective of the work described herein was to perform such a study, with the following characteristics:

- using one of each of the three categories of tool: consistency checker, completeness checker, testing tool;
- applying the tools to finding faults in a realistically complex KBS application;
- controlled by seeding known plausible faults independently into several versions of the application;

- gathering data on the individual and relative abilities of the tools to locate the faults, and the costs of using them.

The paper is organised as follows: Section 2 describes the KBS application used in the study; Section 3 describes the method used, including the types of fault seeded, and the three tools employed; Section 4 presents the results concerning fault-detection; Section 5 presents the results concerning cost; Section 6 considers bias in the study; Section 7 concludes.

The study confirms some of the results of previous studies, showing that each of the three methods is effective at detecting certain types of fault. However, the results also demonstrate that the three verification tools are highly *complementary*: each detects faults that the others do not. This was not shown by the previous studies. It is hoped that the results reported here will promote the use of such verification tools, thereby making KBS validation less difficult, and KBS more reliable. The results also indicates areas in which current KBS verification tools are lacking, suggesting future research and development directions.

2. CONTEXT OF THE STUDY

2.1. The Application: GIBUS

2.1.1. General Description

GIBUS (for "Gestion Intelligente de Batteries Utilisées sur Satellite") is a KBS application developed by the Electronique Serge Dassault company for the European Space Agency (ESA). It was designed to supervise the evolution of batteries used on low-orbital satellites and to assist the operator in short-term management of these batteries. The accumulators used on a low-orbital satellite must supply electrical power to the equipment of the satellite during eclipse periods (35 to 40 minutes). The performance that the accumulators have to achieve during these eclipse periods depend only upon the needs of the satellite. The behaviour of an accumulator during the discharge stage can give some idea of the state of this accumulator. Knowing the state of the accumulators is quite important because, for example, an excessive discharge of the batteries will reduce their lifetime. In extreme cases it can also cause a polarity reversal in some accumulators, which will then produce hydrogen — potentially leading to the destruction of the accumulators and of the satellite itself.

The GIBUS KBS uses data given by the batteries at every orbit and detects abnormal behaviours of the batteries (excessive discharges, excessive recharges, short circuits, thermal runaway, etc). Then GIBUS gives advice to the operator so that these abnormal behaviours don't recur and that the lifetime of the batteries doesn't decrease.

2.1.2. The KOOL Version of GIBUS

The original version of GIBUS was developed with EMICAT 2.0 and Quintus Prolog 2.4. When the application was chosen for an evaluation of the validation

tools developed by partners of the Esprit project ViVa (Verification, Improvement and Validation of KBS), GIBUS was reimplemented in KOOL (an object- and rule-based system shell from the BULL company). The study we present here is based on this KOOL version of GIBUS. The GIBUS application is somewhat hybrid, using an object-oriented representation for facts (states of the batteries, etc) and a rule-based representation for the reasoning knowledge. A more detailed description of the implementation is provided in Appendix A.

GIBUS fact base The facts are defined with an object-oriented representation. The objects have attributes and one or several values can be attached to each attribute. Every attribute is described with descriptors (or facets) that specify the characteristics of the attribute: default value, type, etc. The objects are organised in a taxonomy of classes using inheritance.

GIBUS rule base The rules have a classical "if-then" form and are based on first order logic (i.e. variables are used inside the rules). Most of the rules (124 of them) are used in backward chaining, 75 in forward chaining and 8 in forward and backward chaining. The rule base contains 207 rules and is split into five subsets:

- 47 anomaly detection rules and 35 aberration detection rules are used to detect erroneous inputs and anomalies in the states of batteries;
- 42 numerical modelling rules are used to calculate non-measured values and the range of acceptable values for measured data. These rules correspond to a mathematical model of batteries established by the SAFT company, an accumulator manufacturer;
- 64 diagnostic rules detect the abnormal behaviours of the battery;
- 19 hint rules are used to give advice to the operator.

Appendix A shows that the KOOL implementation of GIBUS is fairly typical of current KBS, being of comparable size and complexity to other systems which have been described in the literature, and using knowledge representation and programming techniques common in modern KBS-building tools.

2.2. Access to Experts and Documentation

When GIBUS was chosen for the ViVa project the application was already developed. Therefore, none of the instigators of this study were involved in the original knowledge acquisition. However, for the ViVa project, ESA gave access to experts to acquire specific knowledge needed by the verification tools. One such expert was involved in the development of GIBUS and another worked on batteries at ESA. Two interviews (of approximate duration 4 hours each) were recorded and transcribed on paper. Unfortunately, because ESA was not directly associated with the study described here, neither these experts nor others were available.

However, all essential GIBUS documentation was available:

- the Detailed Design Document [Detailed Design Document, ESD] of the GIBUS project;

- the Final Report [Final Report, ESD] of the GIBUS project;
- the SAFT's Model Validation Report [SAFT's Model Validation Report, ESA];²
- a listing of GIBUS's EMICAT version;
- the transcriptions of two interviews of the experts recorded for the ViVa project [CISI 93a] and [CISI 93b].

Therefore, we were sufficiently aware of what GIBUS should do and how and, even if we could not play the role of domain expert, we had a good acquaintance with the application itself.

2.3. Four Versions of GIBUS

For this study, we chose not to use directly the KOOL version of GIBUS to test the different verification tools for the following reasons: first, GIBUS had already been validated, so almost all of the faults had already been removed; second, because our lack of access to experts prevented us from acquiring the actual expected results. Instead, it was decided to use the stable KOOL version of GIBUS as a *reference* version. We then seeded sets of faults into this version to obtain three KOOL *faulty* versions to verify against the reference version. Thus, the reference version could be employed as an "expert" to predict the "good" results and to compare them with the ones obtained using the faulty versions.

The faulty versions of GIBUS were created by first listing plausible classes of fault (see Section 3) and, based on these, creating three sets each containing about 25 specific GIBUS faults. Each fault set was created independently by different people with experience of developing and debugging KBS. Care was taken *not* to take into account the capabilities of the verification tools when creating faults.

3. METHOD

3.1. Classes of Fault

All of the three verification techniques used in the study are applied to the implementation of a KBS. This is because, in the nature of KBS development, the implementation is typically the sole formal description of the system (although other semi-formal descriptions — such as KADS-like conceptual models — may exist), and thus the sole description amenable to consistency checking, completeness checking, and testing. In view of this, our fault classes are based on the implementation. They are similar to those of the SAIC study [Miller, Hayes and Mirsky, 1993], and a subset of those in the Minnesota study [Kirani et al., 1992] (which defined fault classes for requirements and design documents also). We accept that faults may be introduced prior to implementation (for example, in a KADS-like conceptual model, or in a requirements document), but we argue that such faults manifest themselves in the implementation within the classes we define here.

²This report summarises the validation of the numerical model of the batteries. This validation was conducted by comparing the SAFT model results with experimental results.

The eight basic classes of fault we identify below are based upon the structure of an object and rule knowledge representation formalism. It is not possible to define a mapping between these types of faults and their manifestation in instances of incorrectness, incompleteness and inconsistency. Any of the faults may cause any manifestation. For example, a simple typographical error in a rule may lead to incorrectness, incompleteness or inconsistency in the KBS as a whole. We note that these types of faults were observed in the study described in [Preece and Shinghal, 1992].

Editing faults These are caused by trivial typographical or text-editing errors, such as transposed or missing characters, and accidental cuts or pastes (possibly leading to truncated or duplicated logic — for example a duplicate rule as described in [Preece and Shinghal, 1992]). While the *effect* of this class of fault may be similar to several of the following classes, the essential difference here is that there is no *rationale* for the fault: it just happened "by accident".

Object faults These are faults in either the object hierarchy (inheritance taxonomy) or in the referencing of specific objects or class names. For example, an object *O* may be mis-defined as belonging to a class *C'* instead of *C*, where *C* and *C'* are semantically similar (i.e. the fault is plausible).

Attribute reference faults An incorrect attribute is referenced in a rule clause. For example, an attribute *A* is referenced in place of *A'*, where *A* and *A'* are semantically similar (the fault is plausible).

Attribute value faults Similar to the previous case, but an incorrect symbolic value is given for an attribute in a rule clause.

Numerical faults Similar to the previous case, but an incorrect numeric value is given in an arithmetic expression in a rule.

Premise faults A rule is missing a premise clause, or has an additional unnecessary premise, or has an incorrect premise. An example of the latter would be the erroneous exchange of two premises from related rules.

Conclusion faults A rule is missing a conclusion clause, or has an additional unnecessary conclusion, or has an incorrect conclusion. As with premise faults, an example of the latter would be the erroneous exchange of two conclusion clauses from related rules.

Rule deletion faults A rule is missing.

3.2. Finding the Faults

To find the faults, three kinds of verification tool were employed: a completeness checker, a consistency checker, and a testing tool. Each tool was used on the three faulty versions, independently of each other (i.e. we did not take into account the results obtained with one tool before using another) and the faults were not dynamically corrected (i.e. we first used the tools on each version, and

then analysed the results of each tool to find the faults). The fault-finding was divided into two phases, as follows.

Search for fault symptoms (failures, in IEEE-standard terminology) To obtain these symptoms the tools were run on the faulty versions and the output compared with those obtained with the reference version. Here, the reference version was employed as a substitute for documented correct results (a "gold standard") or a human expert (an "oracle"). Possible bias resulting from this use of the reference version is considered in Section 6.

Identification of faults (i.e. determining which faults correspond to the symptoms and how those faults could be corrected) The faulty versions were compared with the GIBUS documentation (and in particular with the EMICAT version), but *never* with the reference version. Here, the GIBUS documentation embodies the correct knowledge from human experts.

Finally, the faults found (and the corrections proposed) were compared with the faults introduced in each version — this comparison indicates the "raw" power of the tools in detecting faults. Before examining the results in Section 4, we describe the procedure used to apply each verification tool.

3.2.1. *Use of the Consistency Checking Tool*

The tool used for consistency checking was SACCO (see Appendix B for more details, and [Ayel and Laurent, 1991; Ayel and Vignollet, 1993] for complete descriptions). SACCO performs static³ checks on a knowledge base for the following logical anomalies:

- Rule verification:
 - duplicate premises;
 - duplicate conclusions;
 - identical premises and conclusions;
 - tautologies;
 - anti-tautologies;
 - rules without premises or conclusions;
 - malformed premises or conclusions.
- Cycle detection.
- Consistency:
 - verifications of types and values for attributes;
 - etc.
- Redundancies between rules (subsumption).

These checks were performed on the four versions of GIBUS (the three faulty ones and the reference version). When SACCO indicated that there was an anomaly somewhere in a faulty version and, when this anomaly could not be found in the reference version, it was concluded that SACCO had identified a "possible" fault. On the other hand, when the anomaly was also found in the reference version, it was concluded that the anomaly did not correspond to any

³"Static" in the sense that the KBS is not executed with its inference engine.

fault. In effect, the verification of the reference version was used as an "expert judgement" on the anomalies, to discriminate between "real" faults and others.

3.2.2. *Use of the Completeness Checking Tool*

The tool employed checks basic completeness properties only; a detailed description of basic completeness properties together with more advanced ones is given by [Lounis and Talbot, 1993; Lounis and Ayel, 1995; Preece, 1993]. Our tool assumed that the attributes of the object model can be split into four subclasses:

- the initial attributes, which should be used only in the premises of the rules;
- the terminal attributes which should be used only in the conclusions of the rules;
- the intermediate attributes which should be used both in the premises and in the conclusions of the rules;
- the unused attributes which are not supposed to be used in the rules.

The tool assumes also that the rule base can be split into four subclasses:

- the initial rules which should use only initial attributes in premises;
- the terminal rules which should use only terminal attributes in conclusions;
- the intermediate rules;
- the rules which are both initial and terminal.

It should be noted that normally these partitions would (and can) be made in interaction with the experts. In this study they were determined using the reference version (again the reference version was used to replace the experts).

The completeness check determined — for each faulty version of GIBUS — the status of each attribute and each rule (terminal, initial, etc), which was then compared with their "theoretical" status (the one obtained with the reference version). When there were differences, we tried — using only the documentation (including the EMICAT version) and the current faulty version — to identify the possible faults which could explain the differences.

3.2.3. *Use of the Testing Tool*

To perform software testing, three major tasks must be done:

- create a set of test cases (sufficiently representative to provide significant results);
- determine the "expected results" for each test case;
- run the cases on the system, and compare the obtained results with those expected.

The first and third tasks are most amenable to assistance from automatic software tools. In our study, we used a new version of the SYCOJET testing tool which contains a test data generation module, a test execution module, and a run-time tracing module (this version of SYCOJET is described in Appendix C, but for a more detailed description of the SYCOJET test data generation module see [Ayel and Vignollet, 1993]). SYCOJET is used to generate automatically input data for test cases. The expected results for each test case cannot be generated automatically unless a reference model of the system exists; for example, such a model was built to test the ONCOCIN expert system [Shwe, Tu and Fagin,

1989], and in our case we were able to use the reference version of GIBUS, as explained below. Note that SYCOJET does not automate the process of comparing the obtained results with those expected, although this would be a useful enhancement to the tool.

Generation of test set We defined three sets of test cases, each of which was generated using the SYCOJET test data generator on the reference version of GIBUS. Our goal with these three sets of test cases was to evaluate how testing results are dependent upon the rule coverage:

- the first set — expected to fire 10% of the rules (14 test cases);
- the second set — expected to fire 37.5% of the rules (29 test cases);
- the third set — expected to fire 46% of the rules (38 test cases).

These rule coverage percentages were chosen arbitrarily but to be different and to allow reasonable time for test data generation. Domain experts were not available to provide the expected results; therefore, after the generation of the tests cases' input data, they were run on the reference version to obtain output which we considered to be the "expected results".

It should be noted that the use of SYCOJET to generate test data is intended to make the testing process easier and less biased. However, this does not preclude the use of additional (or replacement) cases provided by domain experts — if such are available. Certainly, this does not change the way in which the cases are used to perform testing.

Results of test case runs We assumed that domain experts would be able to provide, for each test case, the correct reasoning that should be used (i.e. the set of rules which should be fired) and the correct results (i.e. the final or intermediate deductions on which the reasoning is based). Therefore, we added capabilities to the testing tool to store, for each run of a test case:

- the set of facts used as inputs;
- the names of the rules fired;
- a set of deduced facts (all "interesting" deduced facts):
 - the outputs of GIBUS (hints and diagnostics);
 - internal characteristics of the batteries calculated by GIBUS at different orbits and stages of these orbits.

Identification of the faults All test cases were run on the three faulty versions of GIBUS and then compared with the results obtained from the reference version. When there were differences (i.e. the set of facts deduced was not identical or the rules fired were not the same) we have supposed that this failure was due to some error in the corresponding faulty version. For each test case we tried subsequently to make a distinction between *primary failures* (i.e. the minimal set of failures which could explain the incorrect behaviour) and *secondary failures* (i.e. failures which could be due to previous ones). This work was done manually using only the corresponding faulty version.

Once the sets of primary failures were defined, it was possible to identify faults and plausible corrections by the comparison of the faulty version with the documentation — particularly with the EMICAT GIBUS. For each primary failure, this entailed searching for a rule which could contain a fault and explain

the failure, verifying (using the EMICAT version and the documents) if those rules were correct or not, and proposing the fault and a refinement. Thus, there were three stages to make the identification of the faults:

- 1) identifying all failures which occurred during the runs of the test cases
- 2) attempting to define the primary failures
- 3) attempting to identify and to propose a correction for the faults.

4. RESULTS

4.1. Faults on Each Version

Tables 1–3 below summarise, for each faulty version of GIBUS, the number of faults seeded in each class, and the number of faults found by the three verification tools.

| |
|---|
| Table 1: Results from faulty version 1 |
|---|

| |
|---|
| Table 2: Results from faulty version 2 |
|---|

| |
|---|
| Table 3: Results from faulty version 3 |
|---|

Some general conclusions can be made from these tables. Particularly, we notice that more than 61% of the faults were detected and that a correction was proposed for those faults. Moreover, SACCO always discovered more than 35% of the faults, SYCOJET allowed at least 31% of the faults to be detected, and the completeness tool always discovered more than 27% of the faults.

If we look more closely at the results, to try to establish what kind of fault is more easily discovered by one method or another, first of all we note that all the faults made on the **object reference** were detected. Moreover, SACCO always detected more than 40% of these faults. Concerning **attribute value** faults, very good results were obtained since more than 80% of these faults were detected, more than 60% of which by SACCO. Regarding the results for **attribute reference** faults, 100% of these faults were detected in the versions 1 and 2; but, only 1 fault on the three introduced was detected in version 3. Globally, our results for this kind of fault are good. We also notice that the completeness tool is well adapted to the detection of **rule deletions** since in two versions all these faults were detected and 2 of the 3 faults were detected in the other version.

Regarding the results on **premise** faults, no "global" conclusion can be made. Examining the obtained results, taking into account the precise type of fault: a **premise deletion** fault was always detected in the last two versions (2 in version 2 and 3 in version 3); 2 such faults were introduced in version 1 and one was detected by a testing process. We have the same results when two premises were exchanged (2 in version 3). Poor results were obtained when a premise was duplicated or adjusted because none of these faults were detected by our tools.

Nothing can be concluded on **editing** faults because the results are not uniform; however, we can confirm that SACCO's functionality allowed it to discover 100% of the rule duplication faults (which are included in the editing faults). For **conclusion** faults, the number of introduced faults is not statistically significant. More experiments would need to be performed.

On the other hand, these tables show that some faults are difficult to detect with the tools and methods used here. This is the case for the **numerical** ones, but we should note that our tools are not designed to detect this class of fault at present (recall that, in creating the fault classes, care was taken *not* to take into account the abilities of the verification tools). The consistency checker and the completeness checker are not capable of detecting such faults and, for the testing process, SYCOJET does not generate test data using boundary values⁴ (and we have used only test data generated by SYCOJET). An improvement of this test data generation, taking into account boundary values, has been proposed in [Bendou, 1995]. However, it is not our purpose to discuss here the improvement of the methods and tools used.

In Table 4, we have grouped the results of the three versions, to give a rough average. This table reinforces the conclusions given above.

Table 4: Results with all three versions together

Of course, some faults are detected twice, and sometimes by all the methods. It is interesting to study which types of faults are affected by multiple detections. On the other hand, some tools (or methods) seem to be "specialised" in discovering certain types of faults. In the next section, we summarise the faults detected by only one method — which we call single detections — and the multiple detections.

4.2. Single and multiple detections

It is interesting to compare the multiple detections with the faults found by only one method or tool. All these results are summarised in Tables 5–7, for each version. In Table 5, 8 faults were discovered by at least 2 methods, that is 50% of the found faults and 31% of the initial faults. In Table 6, 8 faults were discovered by at least 2 methods. Indeed, one of the **attribute reference** faults was detected by all the methods so it is counted several times here. Thus 36% of the found faults and 31% of the initial faults were discovered by at least 2 methods. In Table 7, 8 faults were discovered by at least 2 methods. One of the **premise** faults was detected by all the methods and is counted several times. Thus 47% of the found faults and 32% of the initial faults were discovered by at least 2 methods.

Table 5: Single and multiple detections for version 1

Table 6: Single and multiple detections for version 2

Table 7: Single and multiple detections for version 3

Concerning the single detections, SACCO is the tool which discovers the fewest faults. However, it costs the least to employ (see section 5). Moreover, not using SACCO would have left 12% of faults not discovered, which is not insignificant. These tables show that the completeness tool is well-adapted for specific faults (rule deletion, for instance). Indeed, it was difficult, with the other two methods, to discover such faults. Improvements and addition of functionalities could improve of the results (for example, specific functionalities to discover faults on premise or conclusion deletions). Our testing process allowed 16% of the faults to be detected by itself, with a maximum rule coverage of 46%. Obtaining better test case coverage would have cost a great deal more in computing cases, but conceivably this result could be improved with faster hardware and optimised algorithms for SYCOJET. In any case, it is hard to imagine that a validation process would *not* include a testing process.

4.3. Rule coverage and testing

Considering the testing method, it is interesting to relate the rule coverage with the detected faults. Recall that the three sets of test cases were generated from the reference version, so that the same sets were used with the three versions. As they are "side results" of our study, we do not show all the results in tables as before. However, we examine two illuminating graphs: Figure 1 relates the faults discovered (the proportion of the number of faults in each version) to the number of test cases executed; Figure 2 relates the same proportional number of faults with the real rule coverage on each version.

Figure 1: Proportional number of faults discovered related to the number of test cases

Figure 2: Proportional number of faults discovered related to the real rule coverage

For version 2 and version 3 the results are intuitive: the more test cases, the better the real coverage, the better the results. However, this is not the case for version 1. There is no obvious reason for this, because the introduced faults in version 1 are not ostensibly different in nature to the ones in the two other versions. The results obtained with versions 2 and 3 (Figure 1) seem to indicate that the number of faults discovered is in proportion (linear increase) to the number of test cases.

⁴We call a boundary value a value which appears specifically in a rule; for instance, if $(X < 3)$ appears in a rule, 3 can be considered as a boundary value.

Conversely, it seems that the number of faults discovered grows faster than linearly (Figure 2). However, we have too few data points to offer strong conclusions here. So, these two graphs challenge some intuitive ideas, and call for further investigation.

5. COST-ORIENTED ANALYSIS

As we have shown in the previous section, the faults which were identified are not always the same: the faults found change with each of the three techniques we have compared. The number of faults found using each tool, and the proportion of those which were discovered just by one of them or discovered by more than one of them, gives us a first impression of the relative effectiveness of the different validation techniques. However, this is not sufficient if we want to have a fair comparison. Indeed, we have also to take into account the costs associated to the use of each technique. This is the topic of this section.

5.1. Testing

The cost of testing is very significant: we examine each component cost below.

5.1.1. *Generation of Test cases and expected results*

First it is necessary to choose the set of test cases which will be used for the test. The benefits of testing are extremely dependent on this choice. In particular it is very important to have a good coverage of each part of the knowledge base. Therefore, the sets of test cases must be built very carefully. The use of a test data generator like that of SYCOJET can help a great deal in the sense that the user can specify the knowledge base coverage that a set of test cases should achieve during its execution. When the test data have been generated, it is then necessary to define the results the KBS should give for each case. Without "expected results", there is no way to judge if the results produced by the application are acceptable or not — faults will not be identifiable.

Unfortunately, this essential step is expensive. If an expert is required to provide the test cases, then the cost is entirely in the expert's time. When an automatic test data generator, such as that provided by SYCOJET, is used to build the test samples, the cost of the test case creation includes the computer time needed to generate the data. To illustrate, in this study SYCOJET spent:

- 19 hours 30 minutes to generate the labels (an extension of De Kleer's labels [De Kleer, 1986] for first order rule bases) needed by SYCOJET prior to the test data generation itself;
- 60 seconds to generate data for the first set of test cases (14 test cases and a rule coverage of approximately 10% of the rules);
- 2 hours 20 minutes to generate data for the second set of test cases (29 test cases and a rule coverage of approximately 35% of the rules);

- 17 hours 30 minutes to generate data for the last set (38 test cases and a rule coverage of approximately 45% of the rules).

Note that the label generation has to be done just one time. Once the labels are built, the user can state how many sets of test cases are needed for each test set, without having to rebuild the labels.

The hardware used here was a diskless DPX-1000 (BULL Unix workstation) with a 68000 processor (and no co-processor). Thus, on a more modern machine with a careful implementation of SYCOJET, the times given above could be divided by 10 to 100. For example on a DPX-20 (BULL Unix workstation using a POWER-PC processor) LISP programs can run 20 times faster than on the DPX-1000. Under these conditions⁵ the time needed to generate the sets of test data are much more reasonable: approximately one hour to build the labels and another hour to generate the most expensive set of test data.

To determine the expected results associated with the generated test data, we ran the data on the reference version. Obviously this is not the normal way to proceed (except for non-regression testing). Consequently we cannot evaluate the cost of this particular stage.

5.1.2. *Runs of test cases*

The next step consists of running the test cases and storing the execution results (traces, etc). The costs are those associated with the use of the application. In fact, the cost depends not only upon the size and the number of the test cases but also upon the application (and the version). Table 8 summarises the costs for each version and each set of test cases:

| |
|---|
| Table 8: Costs for each version and each set of test cases |
|---|

5.1.3. *Analysis of the runs*

Finally the results produced have to be analysed. This analysis can be divided into four steps:

- verifying if the obtained results correspond to the expected ones;
- identifying, when they are not correct, the primary failures (to do this it is necessary to rebuild at least partially the reasoning of the application);
- identifying possible causes (faults) of these primary failures (these causes can be far away from the place of the failures in the deductive path); and finally
- proposing corrections.

With GIBUS, the analysis of the test case results took two days per version (note that all verifications were done manually, so the times given here are times spent by humans):

- one day to identify all the execution failures;

⁵ We were obliged to use this particular hardware platform because our version of KOOL was tied to it.

- one day to identify the primary failures and their causes.

5.2. Completeness

The cost for the verification of completeness is also quite important because, often, the anomalies that the tool identifies are far away from the faults which explain them.

5.2.1. Validation knowledge acquisition

The first difficulty in verifying the completeness of a KBS is to obtain from the experts the necessary "knowledge about completeness". Insofar as we have extracted this knowledge from the reference version of GIBUS (recall that we did not have access to the experts) it is difficult for us to estimate the cost of this stage.

5.2.2. Run of the completeness tool

The tool we have used for completeness checking is quite simple and the checks it makes are not very expensive. Most of information is computed, in fact, during the load of the knowledge base. So the tool completes the verification very quickly (less than one second for GIBUS). Of course, the cost would be greater for a more complex tool (such as those described in [Lounis and Ayel, 1995], [Lounis, 1993], and [Preece, 1993]) but we would expect the results to be better.

5.2.3. Analysis

As we have explained before, the analysis of the results of the completeness checker can be difficult; for this study, about half a day (four hours) was taken for each version of GIBUS.

5.3. Consistency

5.3.1. Validation knowledge acquisition

As for completeness, one of the major problems in checking the consistency of a knowledge based system is the acquisition of the "consistency knowledge" that such a tool needs. During the ViVa project there were two interviews with ESA experts ([CISI, 1993a] and [CISI, 1993b]), but the knowledge about consistency (for GIBUS) that could be extracted from these was not large [Talbot, 1993a]. This is because, as with all knowledge acquisition, it is difficult to "ask the right questions" in such interviews (especially when initially unfamiliar with the application). In this case, another problem was that the interviews took place too long after the implementation of GIBUS, and the experts had difficulty remembering the necessary details. Consequently, we have not used all the functionalities of SACCO, but only the ones which were less dependant on the "consistency knowledge".

5.3.2. *Running the tool*

The checks performed by SACCO were listed in Section 2. Table 9 gives the cost for these checks in computer time (as before, the computer used was a diskless DPX-1000 with an 68000 processor and no co-processor. Thus, on a modern machine with a good implementation of SACCO the times given above could be divided at least by 10 to 100):

| |
|---|
| Table 9: Duration of each consistency verification |
|---|

5.3.3. *Analysis of the results*

Unlike the previous validation techniques, the analysis of the results of a consistency checking tool like SACCO is very easy to do: the tool indicates clearly where the anomalies are (in which rule and in which part of the rule) and what their nature is. So the only remaining task is to verify — against the knowledge acquisition documents (in our case against the EMICAT version of GIBUS) or with the experts — if the anomalies found correspond to actual faults and, if they do, to propose corrections. With GIBUS we took about 2 hours to analyse and to find the faults of each version.

5.4. **Summary**

An evaluation of the cost for each validation technique has to take into account the following costs:

- For testing:
 - test case generation;
 - acquisition of the expected results of each test case;
 - running of the test cases;
 - analysis of the execution results.
- For consistency checking:
 - acquisition of the consistency validation knowledge;
 - use of the consistency checker;
 - analysis of the problems pointed out by the tool.
- For completeness checking:
 - acquisition of the validation knowledge about completeness;
 - use of the completeness checker;
 - analysis of the problems pointed out by the tool.

We summarise the evaluation of the costs (per version) obtained for each validation technique in Table 10.

| |
|--|
| Table 10: Cost evaluation for each validation technique |
|--|

A complete evaluation is not really possible because the acquisition costs have not been estimated. Even if it is very difficult to say anything without real

experiences of such a knowledge acquisition (i.e. without having experienced this acquisition for an application under development), we are convinced that it should not be underestimated at least for two reasons. Firstly, access to the experts is often difficult and entails significant overheads. Secondly, the quality of this acquisition will affect considerably the quality of the validation process. On the positive side, the acquisition has to be done just once, when this is not the case with the other stages. So our conclusions about evaluation costs are that:

- for testing, the most important costs are those related to the generation of the test data, the generation of the expected results and the analysis of test case execution;
- for completeness checking, the most important costs concern the acquisition of the knowledge about completeness and the fault analysis;
- for consistency checking, the most important costs relate to the acquisition of the consistency knowledge.

The consistency checking tool is the one for which the analysis of the results is the least significant, while testing is the technique for which this cost is the most significant. Of course, analysis costs can be decreased if we use tools for this task. For example, the comparison between the expected results and the real ones can be, at least partially, automated. However, consistency checking will always have the advantage over testing and completeness, because the anomalies found with testing or completeness checking are more likely to be further away from the location of the faults which cause them than is the case for consistency checking.

Regarding knowledge acquisition costs (test data and expected results, knowledge about what can be considered as consistent or not, and knowledge about completeness) a priori we will obtain the same order as for analysis costs. However we have to take into account that the completeness tool was very simple and that the methods used for consistency checking are those that are the least domain-dependant.

6. BIAS

We have proposed a method to evaluate verification tools. Of course this method influences the conclusions of the corresponding study. Thus, we need to examine the reliability of the evaluation process described above. In this section, we try to answer the following questions:

- Does the use of a reference version (instead of an expert) facilitate fault discovery?
- Does it penalise some stage of our process?
- Is the cost analysis biased?
- Is our fault generation realistic?

For the first question, we can consider that using a reference rule base, allowing comparisons between the fired rules in the reference version and those fired with the other versions, introduces a bias. For instance, it could favour the discovery of removed rules or of removed premises in the faulty versions. To avoid this, when such differences appeared in the traces, we didn't take them into account. Such a fault was pointed out only if another anomaly could indirectly lead to it.

In the same way, we can consider that having a reference version to compare numerical results could facilitate testing. Indeed, it could be difficult for an expert to determine the presence of a failure when it is a very small numerical error. With the comparison we made, every difference, as small as it can be, is detected.

Conversely, one could argue that our method can penalise the evaluation process. The first point concerns testing. We have generated the test data automatically. However, it would have been preferable to submit the generated test cases to an expert to guarantee their pertinence and their realism. This was not done. Intuitively, we could hope that more pertinent and more realistic test cases would increase fault discovery. Moreover, the analyses of the results obtained after the execution of the different tools or after the execution of the test cases on any faulty version are difficult to perform in the absence of a domain expert.

We can also consider that there exists bias on the cost analysis. Indeed, the cost analysis does not take into account time required to learn the validation tools. In fact, since we were already very familiar with the tools used, it was not possible to evaluate this cost. In the same way, the cost analysis should have taken into account, for completeness and consistency checking, the cost of validation knowledge acquisition (completeness knowledge and consistency knowledge). However it is not possible to analyse this cost considering the unavailability of human experts for this study. Similarly, for testing, experts were not involved in the expected results acquisition: instead, we obtained these from reference version runs. Thus it was not possible to evaluate this acquisition cost.

Finally, one could imagine that our fault generation could influence the evaluation. We can say that the previous study of the potential and classical faults guarantee the realism of the faults. Moreover, tool functionality was not allowed to influence the proposed faults (this can be seen when some faults are not found by any of the three validation processes). We believe that the fault generations were "honest" and that we have tried to minimise this bias.

7. CONCLUSIONS AND PERSPECTIVES

It was not surprising to us that each of the approaches was effective at detecting certain classes of fault. This confirms the previous studies which have used manual or automated versions of these kinds of verification. In particular, our results agree with the Concordia and SAIC studies of consistency and completeness checking ([Preece and Shinghal, 1992; Miller, Hayes and Mirsky, 1993]), and the SRI and Minnesota studies of testing ([Rushby and Crow, 1990; Kirani, Zualkernan and Tsai, 1992]).

However, one of the most interesting results of this study is that the three verification tools we have used are *complementary*. Each of them lead to the detection of more than 30% of the faults and, grouped together, to the detection of more than 70% of the faults. Moreover, the number of faults which were detected twice or three times using different tools is quite small. This is not an obvious result because there was no mapping between the eight fault classes and the functionalities of the tools. This result was not shown by the previous studies.

Therefore, to obtain the best results in KBS validation, we would recommend that knowledge engineers take advantage of using all three types of tool.

The relative efficacy of the consistency and completeness checking methods appear to contradict directly the conclusions of the Minnesota study [Kirani et al., 1992]. Of course, neither the tools, the application, nor the method are the same and this can explain the differences. For example the application used in [Kirani et al., 1992], MAPS, is very small (forward chaining, propositional logic, with only 41 rules) compared to the application we used, GIBUS (object-oriented, forward and backward chaining, first order logic, with 207 rules). The use of a small application can be justified by the fact that evaluation and analysis are easier. However, the results obtained using a more realistic application (in terms of complexity), can also be considered as more realistic. Particularly, the more complex the knowledge base, the more difficult it is to analyse the results of testing or of completeness checking. So using a small application could distort the comparison between the methods. Moreover, for large knowledge bases, the number of test cases needed for testing the application at a sufficient level, is also considerably larger. Thus, if in [Kirani et al., 1992] the application was exhaustively tested (it was possible regards to the size and the complexity of the knowledge base), it would explain the very good results of testing. Conversely, the complexity of consistency checking seems to be less dependent on the size of the knowledge base.

We do not believe that the difference in size between the two applications is sufficient to explain our conflicting conclusions, but it is certainly one of the factors which can have a great influence on the results of such a comparative evaluation of verification techniques. To perform a more complete comparison of the results, we would need to compare the methods and the functionalities of the tools used in the two studies. However, the descriptions of the tools and methods employed in [Kirani et al., 1992] do not allow us to make a good comparison, with the exception of the consistency checker which was less sophisticated than SACCO. This could well explain the differences in our findings.

The other conclusions we reach are less surprising. The study has confirmed that rule coverage generally gives a good idea of the quality of a set of test cases, but also that the results obtained using a set of test cases can depend greatly upon the knowledge base used. For example, while the number of faults found in GIBUS in general seems to grow with the rule coverage of the sets of test cases, for the first version we have found less faults with the third set of test cases (46% of rules fired) than with the second one (37,5% of rules fired) — remembering that the same set of test cases was used with each version.

In terms of cost, consistency checking is cheapest, completeness checking is a little more expensive, and testing is of much greater expense.

It is difficult to generalise our results to all knowledge based systems and, of course, further evaluations of other applications are necessary to confirm (or challenge) our conclusions. However, since the method we have used minimises the need for experts' interpretation of the faults, we can reasonably conclude that if we use an application of similar size and complexity to GIBUS, we would

expect to obtain similar results. Consequently, since our application has a size and a complexity which is representative of actual practice, we would expect that consistency and completeness checking, in addition to testing, would be an effective combination of methods to validate many of the knowledge based systems actually under development.

This study also points towards some research issues:

- First, it is clear that some effort should be made to facilitate the analysis of the results of testing and to interpret them in terms of faults. This is also true for completeness checking.
- Secondly, the results obtained with a very simple completeness checking tool, like the one we have used, prove that the researches about completeness checking are potentially an important area for the validation of knowledge based systems.
- Thirdly, new evaluations of applications under development are necessary to give a more precise idea of costs related to the acquisition of all this knowledge which has to be acquired for the validation process (i.e. knowledge about what is consistent or not, knowledge about the possible inputs of the application, knowledge about the possible outputs, knowledge about the validity of these outputs and so on).
- Finally, a complementary study could be performed to investigate the phenomena where the increasing of rule coverage does not always increase the number of faults discovered (we have seen that this last number could *decrease* significantly). Can general conclusions be made on this problem?

REFERENCES

- Ayel, M. & Laurent, J-P. (1991). SACCO-SYCOJET: two different ways of verifying knowledge-based systems. In M. Ayel & J-P. Laurent, Eds. *Validation, Verification and Test of Knowledge-Based Systems*, pp. 63–76, New York: Wiley.
- Ayel, M. & Vignollet, L. (1993). SYCOJET and SACCO, two tools for verifying expert systems. *International Journal of Expert Systems: Research and Applications*, **6**, pp. 357–382.
- Benbasat, I. & Dhaliwal, J. S. (1989). A framework for the validation of knowledge acquisition. *Knowledge Acquisition*, **1**, pp. 215–233.
- Bendou, A. (1995). CT-DATAGEN: a constraint-based test data generator. *Proceedings of the 3rd European Symposium on Validation and Verification of KBS (EUROVAV-95)*, Saint Badolph - France, June, pp. 19–29.
- Chang, C., Combs, J. & Stachowitz, R. (1992). A report on the Expert systems Validation Associate (EVA). *Expert Systems with Applications*, **1**, pp. 217–230.
- CISI (1993). *Transcription of the first ESTEC experts interview*. Technical Note 205, ViVa, Esprit Project 6125, January.

- CISI (1993). *Transcription of the second ESTEC experts interview*. Technical Note 207, ViVa, Esprit Project 6125, February.
- De Kleer, J. (1986). An assumption-based truth maintenance system. *Artificial Intelligence*, **28**, pp. 127–162.
- ESD (1989). *Detailed Design Document, ESA GIBUS — Système Expert de Gestion de Batteries pour Satellite en Orbite Basse*. ESA Report number Contrat ESTEC 7957/88/NL/RE, document ESD NE 392 668.
- ESD (1990). *Final Report, ESA GIBUS — Système Expert de Gestion de Batteries pour Satellite en Orbite Basse*. ESA Report number Contrat ESTEC 7957/88/NL/RE, document ESD NE 408 176 Ed. 2.
- Ginsberg, A. (1988). Knowledge-base reduction: A new approach to checking knowledge bases for inconsistency & redundancy. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI 88)*, vol. 2, pp. 585–589.
- Hamilton, D., Kelley, K. & Culbert, C. (1991). State-of-the-practice in knowledge-based system verification and validation. *Expert Systems with Applications*, **3**, pp. 403–410.
- Kirani, S., Zualkernan, I. A., & Tsai, W. T. (1992). *Comparative Evaluation of Expert System Testing Methods*. Technical report TR 92-30, Computer Science Department, University of Minnesota, Minneapolis.
- Laurent, J-P. (1992). Proposals for a valid terminology in KBS validation. In B. Neuman, Ed. *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, pp. 829–834. New York: Wiley.
- Lounis, R. (1993). *Specification of ViVa Completeness Checking Tool*. Technical Note 645, ViVa, Esprit Project 6125, December.
- Lounis, R. & Ayel, M. (1995). Completeness of KBS. *Proceedings of the 3rd European Symposium on Validation and Verification of KBS (EUROVAV-95)*, Saint Badolph - France, June, pp. 31–46.
- Lounis, R. & Talbot, S. (1993). *Functionalities for a Completeness Checking Tool*. Technical Note 642, ViVa, Esprit Project 6125, November.
- Meseguer, P. (1992). Incremental Verification of Rule-Based Expert Systems. In B. Neuman, Ed. *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, pp. 840–844. New York: Wiley.
- Miller, L., Hayes, J. & Mirsky, S. (1993). *Evaluation of Knowledge Base Certification Methods*. SAIC Report for U.S. Nuclear Regulatory Commission and Electrical Power Research Institute NUREG/CR-6316 SAIC-95/1028 Vol. 4.

- O'Leary, D. (1991). Design, development and validation of expert systems: A survey of developers. In M. Ayel & J-P. Laurent, Eds. *Validation, Verification and Test of Knowledge-Based Systems*, pp. 3–20. New York:Wiley.
- Preece, A. (1993). A new approach to detecting missing knowledge in expert system rule bases. *International Journal of Man-Machine Studies*, **38**, pp. 661–688.
- Preece, A. & Shinghal, R. (1992). Verifying knowledge bases by anomaly detection: An experience report." In B. Neuman, Ed. *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, pp. 835–839. New York: Wiley.
- Rousset, M-C. (1988). On the consistency of knowledge bases: the COVADIS system, *Computational Intelligence*, **4**, pp. 166–170.
- Rushby, J. & Crow, J. (1990). *Evaluation of an expert system for fault detection, isolation, and recovery in the manned maneuvering unit*. NASA Contractor Report CR-187466, SRI International, Menlo Park CA.
- Shwe, M., Tu, S. & Fagan, L. (1989). Validating the knowledge base of a therapy planning system. *Methods of Information in Medicine*, **28**, pp. 36–50.
- Talbot, S. (1993). *Description of Gibus Validation Knowledge for SACCO and SYCOJET*. Technical Note 616, ViVa, Esprit Project 6125, April.
- Talbot, S. (1993). *Specification of Viva Consistency Checking Tool*. Technical Note 644, ViVa, Esprit Project 6125, December.
- Talbot, S. & Vignollet, L. (1993). *Functionalities for a Test Case Generator and a Consistency Checking Tool*. Technical Note 641, ViVa, Esprit Project 6125, November.
- Vignollet, L. & Ayel, M. (1990). A conceptual model for construction of sets of test samples for knowledge bases. *Proceedings of the 9th European Conference on Artificial Intelligence (ECAI-90)*, pp. 667–672. New York: Wiley.
- Vignollet, L. (1993). *Specification of Viva Test Case Environment Generator*. Technical Note 646, ViVa, Esprit Project 6125, December.
- Vignollet, L. & Lelouche, R. (1993). Test case generation using KBS strategy. *Proceedings of 13th International Joint Conference on Artificial Intelligence (IJCAI'93)*, pp. 483–488.

APPENDIX A: GIBUS in KOOL

This appendix provides additional information on the KOOL implementation in GIBUS, to demonstrate that it is typical of current KBS implementations.

Table 11 compares the KOOL implementation of GIBUS with other KBS for which complexity data are available. These data are from the study in [Preece and Shinghal, 1992]. We see that GIBUS is of average size in terms of its number of rules, and its ratio of rules to data items is similar to other systems, which tend to exhibit a low ratio. We would therefore regard GIBUS as fairly typical of real-world KBS applications. The complexity of the search space of GIBUS is high compared to the other systems, and this should be borne in mind when considering the costs of performing validation.

Description of KOOL

KOOL is a shell produced by the firm Bull for building hybrid expert systems. It has more or less the same functionality as Intellicorp's Kappa product, and is fairly typical of current hybrid KBS building tools. KOOL provides extensive object-oriented facilities. It is possible to define classes and sub-classes which are organised in inheritance trees. In each class, it is possible to define attributes which can be inherited by the subclasses and by the instances of the class. The attributes are monovalued or multivalued and, as in frame-based languages, descriptors are attached to each attribute which define the behaviour of the attribute (its type, inheritance properties, etc). It is also possible to attach methods to classes, as in object-oriented programming languages.

Rules in KOOL can be used to assert or retract facts (a fact is an instance attribute value). For each rule, it is possible to specify if it should be used in forward chaining, backward chaining or both. Some example rules appear below. Here, object variables are typed by class and have the following general form: *xClass where x is an integer and Class a name of a class.

```
{ Rule + : FirstExampleRule
  If *1Human:Has_Child = *2Human
     *2Human:Has_Child = *3Human
  Then *3Human:Has_Grandparents = *1Human
}
```

This defines a rule named `FirstExampleRule` which will be used in forward chaining. `*1Human`, `*2Human` and `*3Human` are variables which can take their values in the instances of the class named `Human`. `Has_Child` and `Has_Grandparents` are attributes defined in the class `Human`. The effect of such a rule will be: for each instance of the class `Human`, `H1`, `H2` and `H3` such that `H2` is one of the values of the attribute `Has_Child` of `H1`, and `H3` is one of the values of the attribute `Has_Child` of `H2`, `H1` will be added in the set of values of the attribute `Has_Grandparents` of `H3`.

```
{ Rule - : SecondExampleRule
  If *1Human:Salary = *1Number
     *1Human:Dept = *2Number
```



```

      (< (divide 7 *2Number) (multiply 12 *1Number))
Then *1Human:PossibleNewLoan =
      (minus (multiply 80 *1Number) *2Number)
}

```

This defines a rule named `SecondExampleRule` which will be used in backward chaining to calculate the residuary loan in response to a query. In the rule, `(< (divide 7 *2Number) (multiply 12 *1Number))` and `(minus (multiply 80 *1Number) *2Number)` are LISP expressions which are used to define additional conditions and to calculate values.

APPENDIX B: The SACCO Tool Used in the Study

SACCO assumes that the knowledge base is split into three parts: a factual part describing the *domain* (which can be empty); a factual part describing the *problem* that the KBS has to solve (this part changes with each particular problem); and a deductive part describing the *expertise* required to solve problems. Using the KOOL implementation of SACCO, the factual parts are defined using KOOL objects and the deductive part with KOOL rules.

SACCO assumes also that there is additional knowledge which defines what is consistent or inconsistent:

- type and legal values of each attribute;
- monovalued attributes which cannot not have more than one value at the same time;
- multivalued attributes for which a maximum number of values can be defined;
- values of a multivalued attribute which cannot be held simultaneously (for example, a dog can be small and cute in the same time, but not small, aggressive and cute);
- couples of mutually exclusive attributes (for example, the attributes `parents` and `children` of one instance of the class `Human` cannot have the save value in the same time: one cannot be both the parent and the child of somebody);
- generic inconsistencies which are specified using sets of uninstantiated facts (effectively a first order extension of De Kleer's *no-goods* [De Kleer, 1986]).

SACCO performs two kinds of consistency verification: static and dynamic checks. For the dynamic checks, it tries to generate consistent fact bases which lead to the deduction of inconsistencies. For these checks SACCO takes into account the full deductive power of the knowledge base, which is not the case for the static checks. Because we have not used these dynamic verifications on GIBUS, we will describe here only the anomalies searched by SACCO when it performs static checks on a knowledge base.

Rule Verifications

SACCO examines each rule to find:

- a rule has duplicate premises;
- a rule has duplicate conclusions;

- a rule has identical premises and conclusions;
- a rule is without premises or conclusions;
- tautologies: some constraint in a rule is always true;
- anti-tautologies: some constraint in a rule is always false;
- malformed premises or conclusions: there is a reference to an attribute which does not exist.

Cycle Detection

A sequence of rules R_1, \dots, R_n is detected as a cycle if there are n variable substitutions s_1, \dots, s_n such that

- for i between 1 and $(n-1)$, s_i unifies one conclusion of R_i and one premise of $R_{(i+1)}$,
- s_n unifies one conclusion of R_n and one premise of R_1 .

Consistency Verifications

Verification of types and values for attributes The values should have types which are compatible with the ones defined for the attributes; the sets of values should not contain sets of values defined as inconsistent; monovalued attributes should not have more than one value.

Detection of conflicting rules There is a conflict between two rules R_1 and R_2 when their premises are compatible but not their conclusions.

Detection of contradictory sequences of rules A sequence of rules R_1, \dots, R_n is contradictory if there are n variable substitutions, s_1, \dots, s_n such that each substitution s_i unifies one conclusion of R_i with one premise of $R_{(i+1)}$ and the premises of R_1 are not compatible with the conclusions of R_n .

Redundancy and Subsumption between Rules

A rule R_1 is redundant if there is another rule R_2 and a variable substitution, s , such that:

- each premise of $s(R_2)$ is included in the premises of R_1 and
- each conclusion of R_1 is included in the conclusions of $s(R_2)$.

So each premise of R_2 matches a premise of R_1 and each conclusion of R_1 matches a conclusion of R_2 .

APPENDIX C: The SYCOJET Tool Used in the Study

The version of SYCOJET used in the study is written in KOOL and contains three modules: a test data generation module, a test execution module and a run-time tracing module. We briefly describe these three modules (the vocabulary used here is essentially the one used by J.De Kleer [De Kleer, 1986]).

Test Data Generation Module

This module is comprised of two components: a label calculator and a value generator. It has the architecture shown in Figure 3. The classification of the attributes (as input or output) and their domains are extra knowledge given by the expert. Label calculation is based on a backward chaining process from output to input attributes. During each environment calculation, the label calculator aggregates also the predicates it encounters in the rules. The resulted conjunction of predicates on the environment's variables constraints the values these variables can take.

The value generator chooses environments inside the labels which will lead to the expected coverage (this expected coverage is in this study the percentage of rules potentially firable when executing the test data generated). Several types of generation are supported:

- random generation (the values will be chosen randomly);
- limit generation (the values will be chosen at the limit of their domains);
- robustness generation (the values will be chosen at the outside of their domains).

In this study we only used random generation. The value generator checks *a posteriori* if the values proposed for the variables of an environment verify the conjunction of predicates associated with the environment.

The result of this test data generation module is a set of test data. It cannot provide the expected outputs.

Description of the Test Execution Module

As SYCOJET is written in KOOL and the knowledge based systems used are also written in KOOL, the test execution module supplies the test data values to the attributes of the objects, and then the reasoning is started by KOOL itself. This test execution module also calls *a priori* the run-time tracing module.

Description of the Run-Time Tracing Module

This tracing module runs in parallel with the application; it records which rules are fired and which facts are deduced. It gives as output the initial test data, the set of rules fired during the run and a set of deduced facts (users can specify the attributes of specific objects for which they want to know the values).

| Type | Number | Found | % | By SACCO | % | By Testing | % | Completen. | % |
|---------------------|--------|-------|------|----------|----|------------|----|------------|----|
| Editing | 3 | 1 | 33 | 1 | 33 | 1 | 33 | | 0 |
| Object | 4 | 4 | 100 | 3 | 75 | 3 | 75 | 1 | 25 |
| Attribute reference | 3 | 3 | 100 | 2 | 67 | 1 | 33 | 1 | 33 |
| Attribute value | 5 | 4 | 80 | 3 | 60 | 1 | 20 | 3 | 60 |
| Numerical | 4 | 1 | 25 | | 0 | 1 | 25 | | 0 |
| Premise | 4 | 1 | 25 | | 0 | 1 | 25 | | 0 |
| Conclusion | - | - | - | | | | | | |
| Rule deletion | 3 | 2 | 67 | | 0 | | 0 | 2 | 67 |
| Total: | 26 | 16 | 61.5 | 9 | 35 | 8 | 31 | 7 | 27 |

Table 1: Results from faulty version 1

| Type | Number | Found | % | By SACCO | % | By Testing | % | Completen. | % |
|-------------------------|--------|-------|------|----------|----|------------|----|------------|-----|
| Editing | 2 | 1 | 50 | 1 | 50 | 1 | 50 | | 0 |
| Object | 5 | 5 | 100 | 2 | 40 | 1 | 20 | 4 | 80 |
| Attribute reference | 6 | 6 | 100 | 4 | 67 | 3 | 50 | 4 | 67 |
| Attribute value | 5 | 5 | 100 | 3 | 60 | 3 | 60 | | 0 |
| Numerical | 2 | 0 | 0 | | 0 | | 0 | | 0 |
| Premise | 3 | 2 | 67 | 1 | 33 | 1 | 33 | | 0 |
| Conclusion ⁶ | 1 | 1 | 100 | | 0 | | 0 | 1 | 100 |
| Rule deletion | 2 | 2 | 100 | | 0 | | 0 | 2 | 100 |
| Total: | 26 | 22 | 84.6 | 11 | 42 | 9 | 35 | 11 | 42 |

Table 2: Results from faulty version 2

⁶A conclusion fault corresponds here to the deletion of a rule conclusion.

| Type | Number | Found | % | By SACCO | % | By Testing | % | Completen. | % |
|---------------------|--------|-------|-----|----------|-----|------------|----|------------|-----|
| Editing | 2 | 2 | 100 | 2 | 100 | 1 | 50 | 0 | 0 |
| Object | 4 | 4 | 100 | 3 | 75 | 1 | 25 | 2 | 50 |
| Attribute reference | 3 | 1 | 33 | | 0 | 1 | 33 | 1 | 33 |
| Attribute value | 4 | 4 | 100 | 3 | 75 | 1 | 25 | 3 | 75 |
| Numerical | 4 | 0 | 0 | | 0 | | 0 | | 0 |
| Premise | 6 | 5 | 83 | 2 | 33 | 4 | 67 | 1 | 17 |
| Conclusion | 1 | 0 | 0 | | 0 | | 0 | | 0 |
| Rule deletion | 1 | 1 | 100 | | 0 | | 0 | 1 | 100 |
| Total: | 25 | 17 | 68 | 10 | 40 | 8 | 32 | 8 | 32 |

Table 3: Results from faulty version 3

| Type | Number | Found | % | By SACCO | % | By Testing | % | Completen. | % |
|---------------------|--------|-------|-----|----------|----|------------|----|------------|----|
| Editing | 7 | 4 | 57 | 4 | 57 | 3 | 43 | | |
| Object | 13 | 13 | 100 | 8 | 62 | 5 | 38 | 7 | 54 |
| Attribute reference | 12 | 10 | 83 | 6 | 50 | 5 | 42 | 6 | 50 |
| Attribute value | 14 | 13 | 93 | 9 | 64 | 5 | 36 | 6 | 43 |
| Numerical | 10 | 1 | 10 | | 0 | 1 | 10 | | 0 |
| Premise | 13 | 8 | 62 | 3 | 23 | 6 | 46 | 1 | 8 |
| Conclusion | 2 | 1 | 50 | | 0 | | 0 | 1 | 50 |
| Rule deletion | 6 | 5 | 83 | | 0 | | 0 | 5 | 83 |
| Total: | 77 | 55 | 71 | 30 | 39 | 25 | 32 | 26 | 34 |

Table 4: Results with all three versions together

| Type | No. | % | SACCO only | % ⁷ | Testing only | % | Compl. only | % | SACCO & Testing | % | SACCO & Compl. | % | Testing & Compl. | % |
|----------------|-----|------|---------------|----------------|-----------------|----|----------------|----|-----------------------|----|----------------------|----|------------------------|----|
| Editing | 3 | 33 | | | | | | | 1 | 33 | | 0 | | 0 |
| Object | 4 | 100 | | | 1 | 25 | | | 2 | 50 | 1 | 25 | | |
| Attribute ref. | 3 | 100 | 2 | 67 | | | | | | | | | 1 | 33 |
| Attribute val. | 5 | 80 | | | | | 1 | 20 | 1 | 20 | 2 | 40 | | |
| Numerical | 4 | 25 | | | 1 | 25 | | | | 0 | | 0 | | 0 |
| Premise | 4 | 25 | | | 1 | 25 | | | | 0 | | 0 | | 0 |
| Conclusion | - | - | | | | | | | | | | | | |
| Rule deletion | 3 | 67 | | | | | 2 | 67 | | 0 | | 0 | | 0 |
| Total: | 26 | 61,5 | 2 | 8 | 3 | 12 | 3 | 12 | 4 | 15 | 3 | 12 | 1 | 4 |

Table 5: Single and multiple detections for version 1

| Type | No. | % | SACCO only | % | Testing only | % | Compl. only | % | SACCO & Testing | % | SACCO & Compl. | % | Testing & Compl. | % |
|----------------|-----|------|---------------|----|-----------------|----|----------------|-----|-----------------------|----|----------------------|----|------------------------|----|
| Editing | 2 | 50 | | | | | | | 1 | 50 | | | | |
| Object | 5 | 100 | | | | | 3 | 60 | 1 | 25 | 1 | 25 | | |
| Attribute ref. | 6 | 100 | | | | | 2 | 33 | 3 | 50 | 2 | 33 | 1 | 17 |
| Attribute val. | 5 | 100 | 2 | 40 | 2 | 40 | | | 1 | 20 | | | | |
| Numerical | 2 | 0 | | | | | | | | | | | | |
| Premise | 3 | 67 | 1 | 33 | 1 | 33 | | | | | | | | |
| Conclusion | 1 | 100 | | | | | 1 | 100 | | | | | | |
| Rule deletion | 2 | 100 | | | | | 2 | 100 | | | | | | |
| Total: | 26 | 61,5 | 3 | 12 | 3 | 12 | 8 | 31 | 6 | 23 | 3 | 12 | 1 | 4 |

Table 6: Single and multiple detections for version 2

⁷in relation to the number of initial errors

| Type | No. | % | SACCO only | % | Testing only | % | Compl. only | % | SACCO & Testing | % | SACCO & Compl. | % | Testing & Compl. | % |
|----------------|-----|------|---------------|-----|-----------------|----|----------------|-----|-----------------------|----|----------------------|----|------------------------|----|
| Editing | 2 | 100 | 1 | 100 | | | | | 1 | 50 | | | | |
| Object | 4 | 100 | 1 | 25 | 1 | 25 | | | | | 2 | 50 | | |
| Attribute ref. | 3 | 33 | | | | | | | | | | | 1 | 33 |
| Attribute val. | 4 | 100 | | | | | 1 | 25 | 1 | 25 | 2 | 50 | | |
| Numerical | 4 | 0 | | | | | | | | | | | | |
| Premise | 6 | 83 | 1 | 17 | 3 | 50 | | | 1 | 17 | 1 | 17 | 1 | 17 |
| Conclusion | 1 | 0 | | | | | | | | | | | | |
| Rule deletion | 1 | 100 | | | | | 1 | 100 | | | | | | |
| Total: | 25 | 61,5 | 3 | 12 | 4 | 16 | 2 | 8 | 3 | 12 | 5 | 20 | 2 | 8 |

Table 7: Single and multiple detections for version 3

| | 1st set of test cases | 2nd set of test cases | 3rd set of test cases |
|--------------------|-----------------------|-----------------------|-----------------------|
| Reference version | 1'20" | 19'09" | 24'57" |
| 1st faulty version | 1'23" | 54'22" | 52'49" |
| 2nd faulty version | 1'29" | 23'16" | 15'26" |
| 3rd faulty version | 1'22" | 30'27" | 32'03" |

Table 8: Costs for each version and each set of test cases

| | 1st faulty version | 2nd faulty version | 3rd faulty version |
|--------------------|--------------------|--------------------|--------------------|
| cycle detection | 47 seconds | 32 seconds | 47 seconds |
| rule verifications | 241 seconds | 246 seconds | 244 seconds |
| consistency | 123 seconds | 123 seconds | 124 seconds |
| redundancy | 503 seconds | 524 seconds | 570 seconds |

Table 9: Duration of each consistency verification

| | consistency | testing | completeness |
|--|-----------------------|--|---------------------|
| acquisition or test case generation | not evaluable | 2 days (computer) | not evaluable |
| acquisition of the expected results | not pertinent | not evaluable | not pertinent |
| test case execution or use of the tool | 16 minutes (computer) | run of the application: (±1 hour (computer)) | 1 second (computer) |
| results analysis | 2 hours (human) | 2 days (human) | 4 hours (human) |

Table 10: Cost evaluation for each validation technique

| Name | Function | Rules | Declarations | Breadth/Depth |
|----------|------------|-------|--------------|-------------------|
| MMU-FDIR | Diagnosis | 105 | 65 | 6/1 |
| TAPES | Selection | 150 | 80 | 3/1.5 |
| NEURON | Diagnosis | 190 | 155 | 4/2 |
| DISPLAN | Planning | 350 | 55 | 2/2 |
| DMS1 | Diagnosis | 550 | 510 | 5/2 |
| GIBUS | Monitoring | 207 | 181 | 6/14 ⁸ |

Table 11: Comparison of the complexity of various KBS

⁸This is the maximum, not mean value.

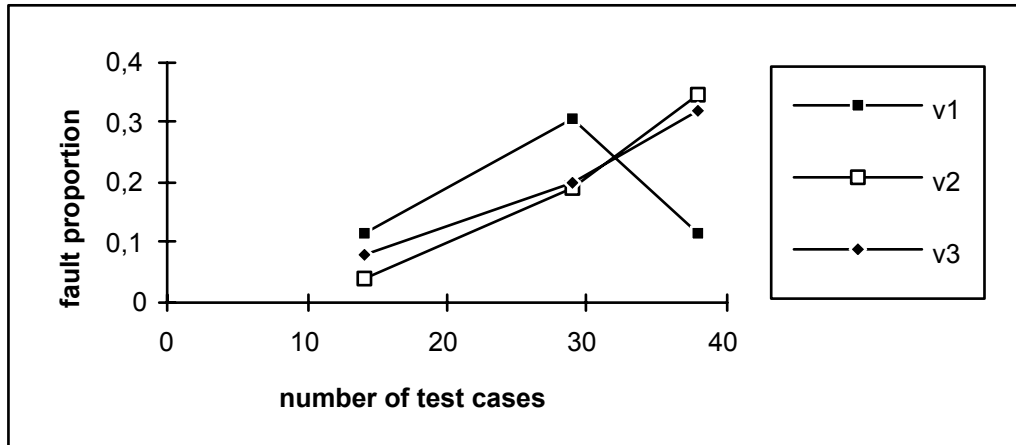


Figure 1: Proportional number of faults discovered related to the number of test cases

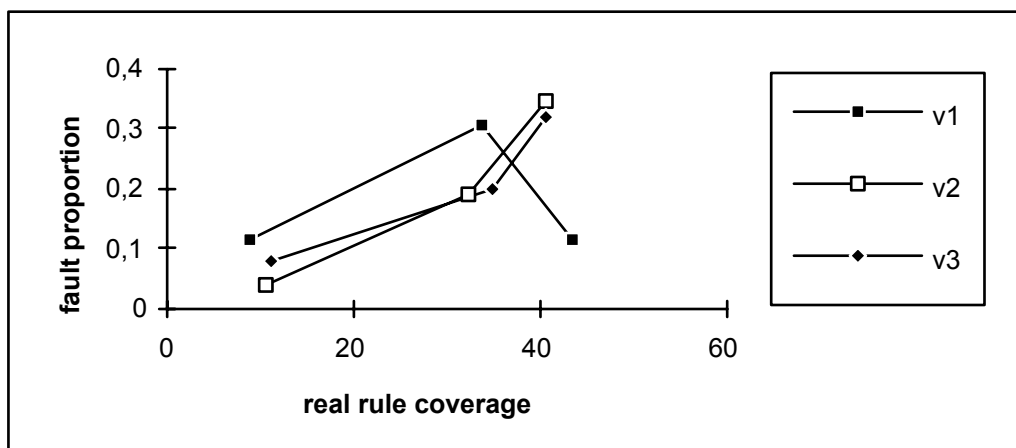


Figure 2: Proportional number of faults discovered related to the real rule coverage⁹

⁹The abscissas are not the same for each version because the execution of the same set of test cases does not fire the same rules for the three versions. So the number of rules fired and the rule coverage change with the version used.

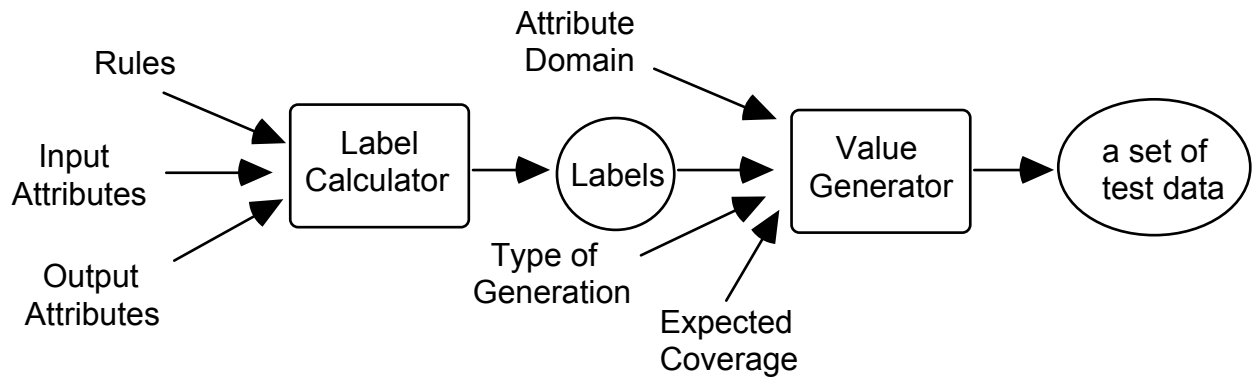


Figure 3: Architecture of the SYCOJET tool used in the study