6

Testing Methods for Decision Support Systems

Jean-Baptiste Lamy[1], Anis Ellini[1], Jérôme Nobécourt[1],
Alain Venot[1] and Jean-Daniel Zucker[2]
[1]*Laboratoire d'Informatique Médicale et Bioinformatique (LIM&BIO), UFR SMBH, Université Paris 13*
[2]*LIM&BIO, UFR SMBH, Université Paris 13; Institut de Recherche pour le Développement*
France

1. Introduction

Decision support systems (DSS) have proved to be efficient for helping humans to make a decision in various domains such as health (Dorr et al., 2007). However, before being used in practice, these systems need to be extensively evaluated to ensure their validity and their efficiency. DSS evaluation usually includes two steps: first, testing the DSS under controlled conditions, and second, evaluating the DSS in real use, during a randomised trial. In this chapter, we will focus on the first step.

The test of decision support systems uses various methods aimed at detecting errors in a DSS without having to use the DSS under real use conditions; several of these methods were initially developed in the field of expert systems, or software testing (Meyer, 2008). DSS testing methods are usually classified in two categories (Preece, 1994):

- *static methods* do not require to use the DSS. They usually consist in the review of the DSS' knowledge base (Duftschmid & Miksch, 2001), either manually by human experts, or automatically, using programs that search for syntactic, logical or semantic errors in the knowledge base. Static methods are sometimes called *verification*, as they consist in checking whether the DSS meets the requirements specified by the users (are you building the system right?) (Preece, 1998).
- *dynamic methods* do require the use of the DSS. They consist in using the DSS to solve a set of test cases. Various methods have been proposed for (a) choosing test cases that are meaningful for testing purpose, and then (b) for determining whether the DSS outputs are considered as erroneous or not, generally by asking human experts to solve the test cases by hand. Dynamic methods are sometimes called *validation*, as they aim at verifying whether the DSS satisfies the actual users' requirement (are you building the right system?) (Preece, 1998).

  Recently, we have proposed a dynamic method for testing almost exhaustively a DSS (Lamy et al., 2008); it involves a very large set of test cases, including potentially all possible cases. Consequently, the DSS outputs are very numerous and cannot be reviewed directly by a human expert. Thus, the method relies on learning or visualization algorithms (Andrews, 2002) to help reviewing the DSS outputs.

Source: Decision Support Systems, Book edited by: Chiang S. Jao,
ISBN 978-953-7619-64-0, pp. 406, January 2010, INTECH, Croatia, downloaded from SCIYO.COM

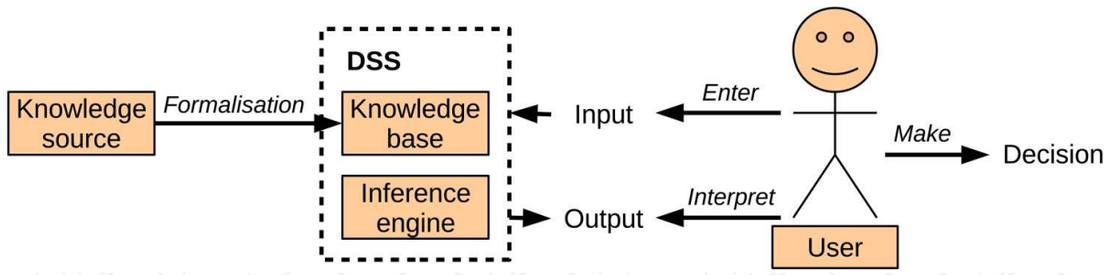


Fig. 1. General schema of a DSS.

In this chapter, we will first propose a classification of the errors that can be found in DSSs. Then, we will describe the various DSS testing methods that have been proposed, and finally we'll conclude by giving advice for choosing DSS testing methods.

2. Classification of errors in DSS

DSSs are usually built from a non-structured knowledge source, for instance a clinical practice guideline (a textual guide that provides recommendations to the physicians about the diagnosis or the therapy for a given disease), a set of cases (for a system using case-based reasoning) or a group of domain experts; this knowledge source is then structured into a knowledge base, for instance a set of rules or a case database, and finally, an inference engine applies the knowledge base to the system's input and determines the output (Fig. 1). Consequently, we can distinguish four main types of error:

- **Errors in the knowledge source**, *e.g.* the French clinical practice guideline for arterial hypertension (HAS, 2005) says "For diabetic patient, angiotensin converting enzyme inhibitors or angiotensin II antagonists are recommended, from the stage of microalbuminuria. For diabetic type 2 patient with other risk factors, angiotensin converting enzyme inhibitors are recommended". The recommendation is ambiguous because it is not clear whether it is "other risk factors than diabetes type 2" or "other risk factors than microalbuminuria", and this can lead to interpretation errors.
- **Errors in the knowledge base**, *i.e.* the structured knowledge base does not exactly translate the knowledge source, *e.g.* the following rule "if patient is diabetic and patient has microalbuminuria, then recommend angiotensin converting enzyme inhibitors or angiotensin II antagonists" does not correspond exactly to the first sentence of the previously cited guideline for hypertension. In fact, the guideline says "from the stage of microalbuminuria", and thus also includes the stages above (such as proteinuria), whereas the rule does not.
- **Errors in the inference engine**, which include both errors in the strategy for executing the knowledge base, *e.g.* in a rule based-system, an engine that does not apply the rules in the desired order, and software bugs in the inference engine.
- **Errors in the use of the DSS**, *i.e.* errors when the user enters the system's input, and errors when the user reads and interprets the system's output. These errors are not located in the DSS itself. However, as DSS are expected to help a human user to make a decision, it sometimes make sense to evaluate the user-DSS couple. Moreover, a badly-designed DSS can mislead the user, for instance, by providing uncommon or incoherent default values for some input. E Coiera *et al.* have studied these errors in the medical context (Coiera et al., 2006); in particular, errors during data entry seem to be quite frequent, and represent an important cause of medication errors.

Errors in the knowledge source are the most problematic, but as they can only be detected and fixed by referring to another source of knowledge, typically human experts, there is few works on them. Errors in the inference engine are less problematic, as the inference engine is normally domain-independent, it can be tested as any other software. As a consequence, most works focus on the errors in the knowledge base.
The errors in the knowledge base are divided in several categories:

- **Syntax errors** occur when the knowledge base does not respect the expected grammar, *e.g.* unbalanced parentheses in a rule database.
- **Logical anomalies**; we speak of "anomaly" and not "error", because a logical anomaly does not always lead to an error in the output of a DSS (Preece & Shinghal, 1994), for instance duplicating a rule in a rule-based system is an anomaly, but it has no influence in the system behavior. However, logical anomalies are often clues of other errors in the knowledge base, such as knowledge errors (see below), for instance a duplicated rule can actually be the same that another rule because a part of the rule has been forgotten. Four types of logical anomalies are considered (Santos et al., 1999; Preece & Shinghal, 1994):
  - **Inconsistency** (also called ambivalence) occurs when the knowledge base can lead to incompatible conclusions for a given input. For instance, a rule-based DSS having the following rules: "if the patient's diastolic blood pressure is inferior to 90 mmHg, the patient does not suffer from hypertension" and "if the patient is diabetic and his diastolic blood pressure is superior to 80 mmHg, the patient suffer from hypertension", because, for a diabetic patient with diastolic blood pressure between 80 and 90 mmHg, the rules conclude that the patient both suffers from hypertension and does not.
  - **Deficiency** occurs when there is missing knowledge in the knowledge base, *i.e.* there are some situations for which the knowledge base leads to no conclusion.
  - **Redundancy** occurs when there is useless elements in the knowledge base, *i.e.* removing these elements from the knowledge base does not affect the DSS's behavior at all. In particular, redundancy includes (but is not limited to) duplicated elements and unsatisfiable conditions, *e.g.* a rule that can never be triggered.
  - **Circularity** occurs when the knowledge base includes some statements that depend only on themselves. For example, the following rules define a circular dependency: "if patient is treated by insulin, then patient's glycemia should be monitored" and "if patient's glycemia is monitored, then patient should be treated by insulin".

The importance of the various types of anomaly depends of the application domain (Preece & Shinghal, 1994).

- **Semantic errors** occur when the knowledge base includes elements that are correct from the logic point of view, but conflicting with domain-specific knowledge. For example, it is a semantic error to conclude that a male patient is pregnant, or to consider a human body temperature of 60°C.
- **Knowledge errors** occur when the knowledge base does not correspond to the knowledge source, although it is syntactically, logically and semantically correct. For example, a clinical guideline says "For diabetic type 2 patient, then it is recommended to start the treatment by a diet", and the associated rule-based knowledge base states that "If patient is diabetic type 2, then start the treatment by prescribing metformin". The example given in the "Errors located in the knowledge base" paragraph is also a knowledge error.

Errors in a DSS can have a more or less important impact, both in term of frequency and gravity. However, the importance of errors is domain dependent. For example, when computerizing a clinical guideline, the guideline is assumed to be the "gold standard", and therefore errors in the knowledge source are not considered, since the source is considered as being the truth. On the contrary, when using a patient database in case-based reasoning, the patient database may be biased and not representative of the new patients for which the DSS is used.

3. Static methods

Static methods test a DSS without requiring to use the DSS. They usually consist in the inspection of the DSS' knowledge base (Duftschmid & Miksch, 2001), either manually by human experts, or automatically. By definition, static methods cannot detect errors in the inference engine or in the use of the DSS.

3.1 Manual approaches

Manual static methods consist in the inspection of the knowledge base by one or more domain experts. Expert inspections can detect errors in the knowledge source or in the knowledge base, however, since humans are not error-proof, they do not guarantee to detect *all* errors of these categories.

Usual recommendations for expert inspections of the knowledge base are the following (Wentworth et al., 1995):

- If the knowledge base has been designed with the support of some domain experts, the inspection should not be done by the same experts, for detecting errors in the knowledge source.
- If possible, the inspection should be done by more than one expert. To obtain an error rate of about 5%, it is required to have at least 4 experts that all agree on the knowledge base content.
- The knowledge base content should not be presented to the experts by someone they know well, for instance a well-known expert of their field (because it could bias their opinion on the knowledge base).

When setting up an expert inspection, several choices must be done. First, a way of presenting the knowledge base should be chosen. The formal representation of the knowledge base is usually not understandable by domain experts. Thus, the knowledge base should be translated into a more human-readable form, such as a set of "if-then" rules expressed in natural language or a decision tree. A more original presentation consists in a set of machine-generated examples for verifying intention-based definitions in the knowledge base (Mittal & Moore, 1996). When the knowledge base is complex, it is often possible to split it into several parts, for instance several decision trees corresponding to various situations. However, one should also verify that the knowledge base and its human-readable translation are really equivalent.

Second, the "gold standard" the knowledge base is compared to, can be either the expert's own knowledge, or the knowledge source used to build the DSS. In the first case, both errors in the knowledge source and in the knowledge base are detected, usually with a stress on the first ones, whereas in the second case, only errors in the knowledge base are detected. For instance, when computerizing a clinical guideline, the experts can be asked to check the knowledge base against their own knowledge, or against the paper guideline.

Finally, if several experts are inspecting the knowledge base, one must decide how to deal with expert disagreements. Disagreements are usually treated by searching a consensus between all the experts, however other methods such as voting have also been proposed.
In conclusion, expert inspections are very interesting for detecting errors in the knowledge source. However, the main drawback of these methods are the difficulties to express the knowledge base in a human-readable way, and to find the experts, since experts are often more motivated by the testing of the complete DSS (especially if the DSS is potentially useful for the expert), than the tedious reading of the DSS's knowledge base.

3.2 Automatic approaches

Automatic static methods rely on programs that search the knowledge base for syntactic, logical or semantic errors. They are sometimes called *verification*. Many of these methods have been proposed for verifying rule-based knowledge bases in expert systems, in the 1980 decade. More recently, some of these methods have been adapted for the verification of other forms of knowledge, such as ontologies (Gómez-Pérez, 1999) or structured clinical guidelines (Duftschmid & Miksch, 2001).

3.2.1 Check for syntax errors

Syntax errors can be found using traditional grammars, such as BNF (Backus-Naur Forms). Pre-formatting tools can also be used when writing the knowledge base, to help preventing syntax errors.

3.2.2 Check for logical anomalies

For rule-based systems, three algorithms have been proposed for detecting logical anomalies (Preece & Shinghal, 1994).

- **Integrity check** considers each rule individually, and checks its validity. It can detect only a few anomalies, such as unsatisfiable conditions.
- **Rule pair check** considers each pair of rules separately. It can detect all anomalies that involve only two rules, such as two inconsistent rules. However, some inconsistency may involve more than two rules, and are not detected.
- **Extension check** considers all the possible paths in the rule of the knowledge base. It can detect all logical anomalies.

Rule pair check requires more computation time than integrity check, and extension check requires even more time. However, it has been shown that even extension check can be achieved in an acceptable computation time on real-world knowledge bases for medical diagnosis, fault diagnosis, and product selection (Preece & Shinghal, 1994), and for power system control centers (Santos et al., 1999). Specific methods have also been proposed for verifying temporal constraints (Duftschmid et al., 2002).

3.2.3 Check for semantic errors

Checking semantic errors requires that the testing program includes some domain-specific knowledge. This additional knowledge typically consists in parameters' possible values (*e.g.* the human body temperature should be within 36°C and 43°C), and combinations of incompatible parameters values (*e.g.* the following combination sex=male and pregnant=true is incompatible) (Duftschmid & Miksch, 2001).
The detection of semantic errors is usually performed at the same time than the detection of logical anomalies (Preece & Shinghal, 1994), for instance integrity check can verify the

parameters' values, and rule pair check or extension check can detect sets of rules that lead to conclusions that are semantically inconsistent. The additional knowledge considered for semantic error can also be taken into account in the detection of logical anomalies; for example it is not a deficiency anomaly to consider only three possible cases, a female pregnant patient, a female non-pregnant patient and a male non-pregnant patient, because the fourth case, a male pregnant patient, is not semantically correct.

In conclusion, automatic static verification methods are very interesting for detecting syntactic, logical or semantic errors and anomalies. Their main advantage is their automatic nature: it is easy to perform the test again when the DSS has been modified, and they ensure to detect all anomalies of a given type in the knowledge base (whereas an expert that manually reviews a knowledge base might not see an error). However, they also have several drawbacks. First, they cannot detect errors in the knowledge sources, the inference engine, and knowledge errors in the knowledge base. In many situations, such as the implementation of clinical practice guidelines, the main difficulty is to structure the knowledge source, and therefore knowledge errors are the more problematic ones. In these situations, automatic static methods are not helpful. Second, these methods work only on declarative knowledge, but not on procedural knowledge, and it is not always easy to transform a procedural knowledge into a declarative one. Finally, the detection of semantic errors requires the addition of domain-specific knowledge, which makes these methods less automatic (a domain expert may be necessary) and raise the question of the verification of this additional knowledge. For all these reasons, automatic static methods are no longer the more active field in DSS verification.

4. Dynamic methods

Dynamic methods test the DSS by running it over some test cases, and they often require the intervention of domain experts for checking the results obtained in the test cases.

4.1 Test bases

Traditional dynamic methods, sometimes called empirical testing, involve the use of a test base that includes a limited number of test cases (compared to the usually very high number of possible cases). To set up such a study, the first step is to build the test base; several methods have been proposed for choosing the cases in the test base.

First, when they are available, real cases can be used, *e.g.* from a patient database for a medical DSS, or a server log for a network monitoring DSS. However, it may be difficult to obtain all the input values required by the DSS, and it is sometimes required to complement the cases.

Second, the test cases can be arbitrarily chosen. A group of final users or domain experts can be asked to write a set of test cases, or, during evaluation, each evaluator can be asked to enter test cases of his choice. DSS designers can also choose and propose test cases that correspond to the difficulties they encountered during the design of the DSS, such as test cases for ambiguous situations or for situations that previously lead to an error in the DSS's outputs (to ensure that these errors have not been reintroduced).

Third, test cases can be generated at random. A basic method consists in creating a case by randomly choosing a value for each DSS input variable. More sophisticated methods can involve semantic constraints (*e.g.* to avoid generating test cases involving a male pregnant patient).

Fourth, various methods have been proposed for the automatic generation of "optimal" test cases, using heuristics. A first approach is to partition the input domain in several subdomains, each of them associated to a sub-domain of the output domain, and then to generate one or more test cases for each sub-domain. A. Preece reviewed the methods for partitioning (Preece, 1994): in *equivalent class partitioning*, the input domain is partitioned in subdomains that lead to the same output; in *risk-based partitioning*, input and output domains are partitioned in ten partitions according to the level of risk they can cause in real life, in particular, various metrics can be used to determine which test cases are the more complex to deal with; in *structure-based partitioning*, one partition is created for each *path* in the DSS (the definition of *path* being DSS-dependent, and potentially subject to discussion). Vignollet *et al.* (Vignollet & Lelouche, 1993) proposed another "optimal" approach for rulebased systems, which take into account the inference engine strategy, and generate a test base that triggers every rules of the knowledge base at least once during the test. Sensitivity analysis (Sojda, 2007) is a third approach, which considers cases that test the behaviour of the DSS for extreme input values, and ensure that the output evolves as expected when an input value increases or decreases. For instance, in a DSS for diabetes type 2 therapy, glycosilated haemoglobin is a marker of the gravity of the disease, and therefore, when glycosilated haemoglobin increases, the recommended treatment should not be weaker.

Finally, it is possible to build a test base by mixing several methods, *e.g.* by including both real and random cases. For validating a DSS, a good test base should typically include (a) realistic test cases (either real cases or cases written by final users or domain experts), (b) test cases for ambiguous or problematic situations, written by DSS designers, (c) randomly generated test cases, and (d) possibly "optimal" test cases.

Depending on how the right DSS output for each test case is determined and who runs the DSS, there are several possible protocols for the evaluation:

1. For real test cases corresponding to past data, the expected DSS outputs can be observed in the real life. In this case, the DSS designers can run the DSS and compare the DSS outputs to the observed ones. For instance, a DSS for predicting the evolution of bird populations have been tested on real past data (Sojda, 2007).
2. A group of experts is asked to determine the right output for each test case, according to their own expertise. In case of disagreement between experts, a consensus should be obtained. Then the DSS designers run the DSS and compare the DSS outputs to the expert's ones.
3. When a gold standard is available, such as a clinical guideline in the medical domain, a group of experts is asked to interpret the gold standard and determine the right output for each test case according to the gold standard (even if the experts disagree with it). In case of disagreement between experts, a consensus should be obtained. Then the DSS designers run the DSS and compare the outputs to the expert's ones.
4. Each expert runs the DSS and compares the DSS outputs to his personal opinion.

In the three first protocols, the right outputs for each test case are determined first, and then the DSS is run by the DSS designers (or a technician). In the last protocol, the DSS is run by the experts and there is no absolute "right" output for each test case, since each expert is free to compare the DSS output to his own opinion, possibly different from the ones of the other experts. To avoid bias, the experts involved in the testing should not have been involved in the DSS design.

Several measures have been proposed for quantifying the effectiveness of a DSS over a test base (Guijarro-Berdiñas & Alonso-Betanzos, 2002): contingency tables (including false

positive and false negative rates), percentage agreement and the Kappa statistic for pair tests (*i.e.* comparing the DSS to a gold standard or a single expert), and Williams' index, cluster analysis and Multi-Dimensional Scaling (MDS) for group tests (*i.e.* comparing the DSS with a group of several experts).

Depending on the choice done for generating the test cases, and the evaluation protocol, validation over a test base can be used in various situations, and it can potentially discover all types of errors listed in section 2. The main drawback of this method is that the number of test cases is necessarily limited, and therefore it cannot ensure the absence of errors in the DSS for other cases.

4.2 "Exhaustive" method

Recently, we have proposed a new dynamic testing method that runs the DSS over a very high number of test cases, allowing an almost exhaustive testing (Lamy et al., 2008). As test cases are far too numerous to let human experts review the DSS outputs for each test case, the method relies on the use of learning algorithms or visualisation techniques to help verifying the DSS's outputs. The method includes three steps:

1. Generate an exhaustive (or almost exhaustive) set of the DSS input vectors, and run the DSS on each input vector to obtain the associated output. It is possible to generate an exhaustive set of input vectors by considering a set of variables expressing the various elements of input for the DSS, and generating all possible combinations of the variables' values. If the input vector includes continuous variables, they should be limited to a few values. Semantic constraints can be added to exclude impossible or infrequent cases.
2. Extract knowledge from the set of (input vector, output result) pairs by applying learning or generalization algorithms, or generate a graphical representation of the (input vector, output result) pairs.
3. Let an expert review the knowledge or the graphical representation produced at step 2, and compare them to the original knowledge source used to design the DSS, or to his own opinion.

We applied this method for testing the ASTI critiquing module, a medical DSS implementing therapeutical recommendations from clinical guidelines, and aimed at raising alerts whenever a physician's prescription does not follow the recommendations. The ASTI critiquing module includes knowledge bases for six diseases: type two diabetes, hypertension, tobacco addiction, dyslipaemia, atrial fibrillation and thrombo-embolic risk. In a first study (Lamy et al., 2008), we used Quinlan C4.5 algorithm (Quinlan, 1993) to generate a decision tree from an almost exhaustive set including hundreds of thousands (input vector, output result) pairs for each disease. To ensure 0% of error in the decision tree, pruning was disabled. However, for hypertension, the extracted decision tree was too huge for being human reviewed, and thus this testing method has not been applied to this disease.

To evaluate this approach, errors were introduced in the DSS. All the errors introduced were clearly visible on the decision tree.

In a second time, we built tables from more limited set of about thousands (input vector, output result) pairs. We divided the input vectors in two parts: the clinical profile (including comorbidities, and various patient characteristics such as age or sex; each clinical guideline lead to about ten profiles), and the treatments (including the current treatment, and the

Current treatment: no treatment	diet	gemfibrozil	anionic resin	ezetimib	nicotinic acid	pravastatin	simvastatin	atorvastatin	rosuvastatin	statin + ezetimib	statin + resin	statin + nicotinic acid	Prescribed treatment
○	○	○	○	○	○	○	○	○	○	○	○	○	no treatment
●	○	○	○	○	○	○	○	○	○	○	○	○	diet
○	○	●	●	●	●	◑	◑	◑	◑	○	○	○	gemfibrozil
○	○	●	●	●	●	◑	◑	◑	◑	○	○	○	anionic resin
○	○	●	●	●	●	◑	◑	◑	◑	○	○	○	ezetimib
○	○	●	●	●	●	◑	◑	◑	◑	○	○	○	nicotinic acid
○	●	●	●	●	●	●	●	●	●	●	●	●	pravastatin
○	●	●	●	●	●	●	●	●	●	●	●	●	simvastatin
○	●	●	●	●	●	●	●	●	●	●	●	●	atorvastatin
○	○	●	●	●	●	●	●	●	●	○	○	○	rosuvastatin
○	○	○	○	○	○	○	○	○	○	○	○	○	statin + ezetimib
○	○	○	○	○	○	○	○	○	○	○	○	○	statin + resin
○	○	○	○	○	○	○	○	○	○	○	○	○	statin + nicotinic acid

Table 1. Example of the use of table for the graphical visualisation of the inputs and outputs of the ASTI critiquing module for patient with hypercholesterolaemia (a type of dyslipaemia) with no other risk factors. The current treatment of the patient is shown horizontally, and the treatment prescribed by the physician vertically. The DSS output is indicated by the following symbols: ● means that the DSS considers the prescribed treatment as conform to the recommendations, ○ that the DSS considers the prescribed treatment as non conform, and ◑ that the DSS considers the prescribed treatment as conform only if the current treatment is poorly tolerated by the patient (but not if it is inefficient). This table summarizes 338 test cases.

treatment prescribed by the physician). Then, for each clinical profile, we built a table displaying the current treatment horizontally, the prescribed treatment vertically, and at the intersections the corresponding DSS outputs, *i.e.* the conformity of prescribed treatment to the recommendations. Table 1 show an example of such a table.

Although not as exhaustive as the decision trees of the previous approach, these tables provided an interesting overview of the DSS behaviour. In particular, it is easy to visually detect some patterns on the graphical presentation, for instance, it is easy to see in Table 1 that the DSS behaviour is the same if the current treatment is a statin (pravastatin, simvastatin, atorvastatin or rosuvastatin). All the six diseases supported by the ASTI critiquing module were tested using this approach, and it allowed us to find several errors that were present but not discovered in the decision trees.

Compared to the standard dynamic methods that rely on small test bases, the "exhaustive" method tests the system over a much larger number of test cases. However, it is more complex to set up, and may not be suitable for all DSS.

The first approach, based on learning algorithms, should be quite easy to adapt to other DSSs. However, when the generated decision tree is too huge for being human-reviewed, the method cannot be applied. In addition, if the DSS knowledge base includes rules that are more complex that basic "if-then" rules with and / or operators, such as "if x out of y statements are true, then...", it might be necessary to use more sophisticated learning algorithms than C4.5. Several knowledge representations can also be used as alternative to decision trees, such as production rules or flowcharts; Wentworth *et al.* review them in the chapter 6 of their book (Wentworth et al., 1995).
The second, graphical, approach can only be used if it is possible to represent the inputs and outputs of the DSS in one or a few tables; this point is highly domain-dependent. Other visualisation technics could be used as alternatives to tables, such as 2D or 3D bar charts, or star-plot glyphs (Lanzenberger, 2003). Our experiments shew that both approaches are complementary, as they allowed to find different errors.

5. Conclusion: How to choose a testing method?

In the preceding sections, we have presented four main categories of testing methods: manual static methods, automatic static methods, test bases, and "exhaustive" dynamic methods. We have seen that all these methods have their own advantages and drawbacks: there is no perfect or ideal DSS testing method. In addition, it has been shown that the various methods do not detect the same errors (Preece, 1998). Therefore we recommend to apply several methods.
Table 2 shows the types of errors that can be found by the various testing methods. One should combine several testing methods so that the combination of methods covers all types of errors. In addition, one should typically:

- combine both static and dynamic methods,
- combine both automatic methods and methods relying on domain experts, and
- in a test base, mix test cases chosen randomly or by the system developers and test cases as close as possible to real cases (either real test cases or test cases written by final users).

In table 2, automatic static methods cannot detect knowledge errors in the knowledge base. As a consequence, these methods are not very useful when knowledge errors are frequent, which typically occurs when the knowledge source used to create the DSS is complex and difficult to interpret.
Another important element to take into account when choosing DSS testing methods is whether there is a "gold standard" knowledge source in the field covered by the DSS, or not. For instance, when designing a DSS to implement a clinical practice guideline, the decisions recommended by the guideline are assumed to be the best possible decisions, and therefore it is a "gold standard" knowledge source. In this case, errors in the knowledge source are not to be considered, and consequently one should favor protocol #3 for test base evaluation.
Finally, another question is related to the order in which the various testing methods should be applied. It is usually admitted that verification and static methods should be performed before validation and dynamic methods. Another advice is to perform automatic methods before methods relying on experts, because, if the DSS was heavily modified consequently to the first test, it is usually easier to perform again automatic testing rather than the work with the experts.

Error types	Static methods		Dynamic methods	
	Manual	Automatic	Test base	Exhaustive
Errors in the knowledge source	++[1]/-	-	+[2]/-	++[1]/-
Errors in the knowledge base				
Syntax errors	++	++	+	++
Logical anomalies				
Inconsistencies	++	++	+	++
Deficiencies	++	++	+	++
Redundancies	++	++	-	-
Circularities	++	++	+	++
Semantic errors	++	++	+	++
Knowledge errors	++	-	+	++
Errors in the inference engine				
Strategy errors	-	-	+	++
Software bugs	-	-	+	++
Errors in the use of the DSS				
Input entry errors	-	-	+[3]/-	-
Output interpretation errors	-	-	+[3]/-	-

Table 2. The various types of error in a DSS, and the test methods that can be used to detect them. "-" indicates that the method cannot detect error of this type. "+" indicates that the method can detect the errors of this type and covers only a part of the knowledge source, knowledge base or inference engine functionalities. "++" indicates that the method can detect the errors of this type and covers the whole knowledge source, knowledge base or inference engine functionalities.

In conclusion, many methods have been proposed for testing DSS, each of them having its own advantages and weaknesses. Correctly used, these methods can detect a lot of errors in a DSS. After testing the DSS thoroughly, the next step in the DSS evaluation is to set up a randomized trial in real use conditions, in order to ensure that final users really perform significantly better with the DSS than without, but also that the use of the DSS does not introduce other sources of errors (Coiera et al., 2006), such as automation bias, *i.e.* the user follows the DSS recommendations without question at all, or on the contrary errors of dismissal, *i.e.* the user totally ignore the DSS recommendations (or deactivate the system, if the user is allowed to).

6. References

Andrews, K. (2002). *Information visualisation: tutorial notes*, Graz University of Technology.

Coiera, E., Westbrook, J. & Wyatt, J. (2006). The safety and quality of decision support systems., *Yearbook of medical informatics* pp. 20–25.

[1] only if the gold standard is the expert knowledge, and not the knowledge source.

[2] protocols #1, #2 and #4 only (see section 4.1).

[3] protocol #4 only.

Dorr, D., Bonner, L., Cohen, A., Shoai, R., Perrin, R., Chaney, E. & Young, A. (2007). Informatics systems to promote improved care for chronic illness: a literature review, *J Am Med Inform Assoc* 14(2): 156–163.

Duftschmid, G. & Miksch, S. (2001). Knowledge-based verification of clinical guidelines by detection of anomalies, *Artif Intell Med* 22: 23–41.

Duftschmid, G., Miksch, S. & Gall,W. (2002). Verification of temporal scheduling constraints in clinical practice guidelines, *Artif Intell Med* 25(2): 93–121.

Gómez-Pérez, A. (1999). Evaluation of taxonomic knowledge in ontologies and knowledge bases, *Proceedings of the North American Workshop on Knowledge Acquisition, Modeling, and Management (KAW)*, Vol. 2, Banff, Alberta, Canada.

Guijarro-Berdiñas, B. & Alonso-Betanzos, A. (2002). Empirical evaluation of a hybrid intelligent monitoring system using different measures of effectiveness, *Artif Intell Med* 24(1): 71–96.

HAS (2005). Prise en charge des patients adultes atteints d'hypertension art´erielle essentielle, Available at http://www.has-sante.fr/portail/display.jsp?id=c 269118.

Lamy, J.-B., Ellini, A., Ebrahiminia, V., Zucker, J.-D., Falcoff, H. & Venot, A. (2008). Use of the C4.5 machine learning algorithm to test a clinical guideline-based decision support system, *Stud Health Technol Inform* 136: 223–228.

Lanzenberger, M. (2003). The interactive stardinates - design considerations, *Proceeding of Human-Computer Interaction (INTERACT'03)*, IOS Press, Zurich, Switzerland, pp. 688–693.

Meyer, B. (2008). Seven principles of software testing, *IEEE Computer* 41(10): 99–101.

Mittal, V. & Moore, J. (1996). Detecting knowledge base inconsistencies using automated generation of text and examples, *Proceeding of the 16th conference on Artificial Intelligence (AAAI-96)*, Portland, Oregon, pp. 483–488.

Preece, A. (1994). Validation of knowledge-based systems: The state-of-the-art in north america, *Journal of communication and cognition - Artificial intelligence* 11: 381– 413.

Preece, A. (1998). Building the right system right - Evaluating V&V methods in knowledge engineering, *Proceedings of the eleventh workshop on Knowledge Acquisition, Modeling and Management (KAW'98)*, Voyager Inn, Banff, Alberta, Canada.

Preece, A. D. & Shinghal, R. (1994). Foundation and application of knowledge base verification, *Int J Intell Syst* 22(8): 23–41.

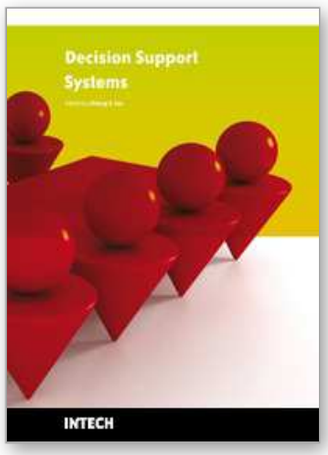
Quinlan, J. R. (1993). *C4.5: Programs for machine learning*, Morgan Kaufmann.

Santos, J., Faria, L., Ramos, C., Vale, Z. & Marques, A. (1999). *Multiple approaches to intelligent systems*, Vol. 1611/2004 of *Lecture Notes in Computer Science*, Springer Berlin / Heidelberg, chapter Verification of knowledge based-systems for power system control centres, pp. 316–325.

Sojda, R. (2007). Empirical evaluation of decision support systems: Needs, definitions, potential methods, and an example pertaining to waterfowl management, *Environmental Modelling & Software* 22: 269–277.

Vignollet, L. & Lelouche, R. (1993). Test case generation using KBS strategy, *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI'93)*, Chamb´ery, France, pp. 483–488.

Wentworth, J., Knaus, R. & Aougab, H. (1995). *Verification, validation, and evaluation of expert systems*, Vol. 1, A FHWA Handbook.



Decision Support Systems

Edited by Chiang S. Jao

ISBN 978-953-7619-64-0
Hard cover, 406 pages
Publisher InTech
Published online 01, January, 2010
Published in print edition January, 2010

Decision support systems (DSS) have evolved over the past four decades from theoretical concepts into real world computerized applications. DSS architecture contains three key components: knowledge base, computerized model, and user interface. DSS simulate cognitive decision-making functions of humans based on artificial intelligence methodologies (including expert systems, data mining, machine learning, connectionism, logistical reasoning, etc.) in order to perform decision support functions. The applications of DSS cover many domains, ranging from aviation monitoring, transportation safety, clinical diagnosis, weather forecast, business management to internet search strategy. By combining knowledge bases with inference rules, DSS are able to provide suggestions to end users to improve decisions and outcomes. This book is written as a textbook so that it can be used in formal courses examining decision support systems. It may be used by both undergraduate and graduate students from diverse computer-related fields. It will also be of value to established professionals as a text for self-study or for reference.

INTECH
open science | open minds

InTech Europe
University Campus STeP Ri
Slavka Krautzeka 83/A
51000 Rijeka, Croatia
Phone: +385 (51) 770 447
Fax: +385 (51) 686 166
www.intechopen.com

InTech China
Unit 405, Office Block, Hotel Equatorial Shanghai
No.65, Yan An Road (West), Shanghai, 200040, China
中国上海市延安西路65号上海国际贵都大饭店办公楼405单元
Phone: +86-21-62489820
Fax: +86-21-62489821

IntechOpen