Centrum voor Wiskunde en Informatica
Centre for Mathematics and Computer Science

R.C. van Soest

Consistency and completeness of a knowledge base

Computer Science/Department of Software Technology Note CS-N8608 December

The Centre for Mathematics and Computer Science is a research institute of the Stichting Mathematisch Centrum, which was founded on February 11, 1946, as a nonprofit institution aiming at the promotion of mathematics, computer science, and their applications. It is sponsored by the Dutch Government through the Netherlands Organization for the Advancement of Pure Research (Z.W.O.).

Consistency and Completeness of a Knowledge Base

René C. van Soest
Department of Mathematics, University of Amsterdam,
Roetersstraat 15, Amsterdam, The Netherlands.

This paper introduces a method for detecting rules which can cause inconsistency or incompleteness in a knowledge base containing production rules. For the greater part this method is based on the structure of a knowledge base as used together with the DELFI-2 system, an expert system shell which has been developed at the Delft University of Technology. However, this method can be used for similar rule-based systems as well. The main idea of this method is to make use of the declarative knowledge before considering the production rules.

1980 Mathematics Subject Classification : 69K11
1982 CR Categories : I.2.1
Key Words & Phrases : expert systems, consistency and completeness of a knowledge base

1. Introduction

In the field of expert system design, problems concerning building a knowledge base are important topics of investigation. One of those problems is the way in which the knowledge items in the knowledge base are interrelated. As will become clear in the next sections, there exist situations in which some knowledge items are interrelated in a way which is not permitted in the field of expert systems. A knowledge base is called "inconsistent" if it contains illegal relationships. On the other hand, another often met problem when constructing a knowledge base is caused by leaving out some of the, for proper functioning, required knowledge. This means that the knowledge base does not contain all information it should contain on the basis of its current contents. Such a knowledge base is called "incomplete". This paper provides definitions of these topics and describes a method for detecting inconsistency and incompleteness of knowledge bases. The knowledge base on which this method is based, is a rule-based one, that forms an expert system if it is combined with the expert system shell DELFI-2 [8]. A DELFI-2 knowledge base can be separated into two parts (as will be considered in section 1.1), and resembles an EMYCIN [4] knowledge base.

In the next sections some fundamentals of expert systems and in particular of knowledge bases are introduced. In addition, the problems which occur with the construction of production rules are described and a system which assists in solving some of these problems is considered. Some basic definitions about propositional and first-order logic, which will be used in other sections, are reviewed in section two. Section three deals with the consistency of a knowledge base and in section four completeness is discussed. Section five considers an implementation and section six finally arrives at a conclusion.

1.1. Fundamentals of expert systems

From the beginning of the construction of DELFI-2, the aim was to arrive at an expert system which was separated into two parts (see fig. 1) :

-- A *knowledge base*, with domain specific knowledge,
-- A domain independent *consultation program.*

A major advantage of such a separation is the opportunity to consider the two parts independently. In addition, it is possible to construct a new expert system, just by combining a new knowledge base

Note CS-N8608
Centre for Mathematics and Computer Science
P.O. Box 4079, 1009 AB Amsterdam, The Netherlands

with the expert system shell. Because of this possibility to consider the two parts independently, this paper only considers the knowledge base. A description of the consultation program can be found in [1] and [8].

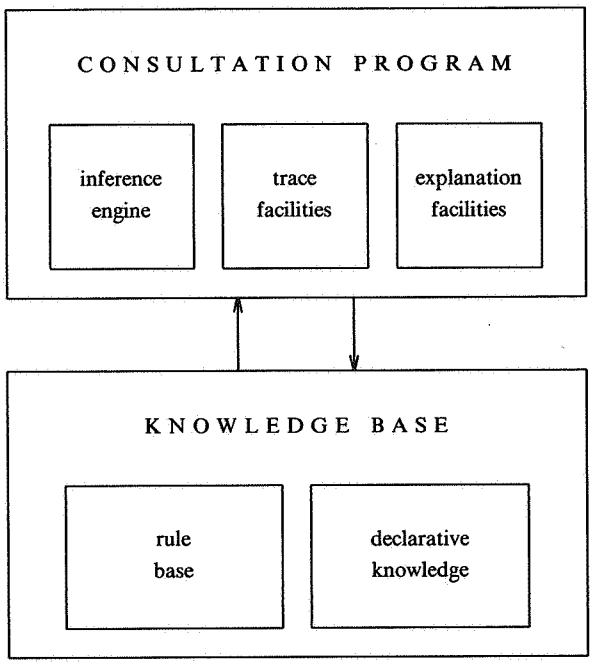

FIGURE 1. General structure of an expert system.

One remark to be made about the "inference engine", is that it operates in a *top-down*, or *goal-driven*, manner, which means that a specific part of the knowledge contains goal-descriptions that initiate the inference process.

As shown in figure 1, a knowledge base is also separated into two parts. The first part, which will be considered is the "declarative knowledge" part. In DELFI-2 and EMYCIN, declarative knowledge is stored in a "context tree". A more general name, which will be used throughout this paper is "object tree". An object tree consists of objects and attributes with the following definition :

```
<object>    ::= object
                  <identifier1>
                  <identifier2>
                 {<attribute>}+
                 {<object>   }*
                end
```

```
<attribute> ::= attribute
                    <identifier1>
                    <identifier2>
                    <class>
                    <domain>
                   {<value> }+
                end

<class>  ::= goal | notgoal
<domain> ::= text | numerical | boolean
```

The identifier of an object or an attribute contains the name of that object or attribute respectively. That name is divided into two parts. The first part is a one word name which is used internally in the system. The second part of the name is a more descriptive name, which is used in the communication between the expert system and its user(s). An attribute of class "goal" is an attribute whose derivation is a goal in the consultation of the expert system, for which the knowledge base is used. With an object, certain attributes are associated. Objects and attributes are organized in a tree, called an object tree, and there is one object called the root of that tree. An object can have some subobjects. The choice of dividing a knowledge base into objects and subobjects is made because of the possibility to split up a large knowledge base into smaller surveyable parts. A (sub)object can have a father, but also sons and/or brothers in the object tree. In figure 2, object A, which has no father, is the root of the tree. Object B is the son of A. Both C and D are sons of B, thus they are called brothers. As the figure shows, each object has its own attributes, so it is possible to have two different attributes with the same identifier. The combination of object and attribute, however, is unique.

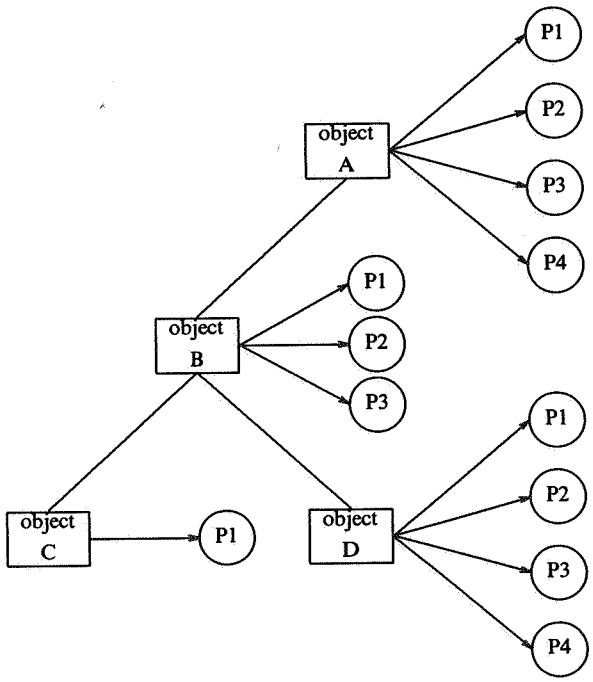

FIGURE 2. Example of an object tree.

The objects and attributes being defined, they can be refered to in production rules having the following syntactic structure :

if <antecedent> **then** <consequent> **fi**

with

```
<antecedent> ::= <clause> { and <clause> }*
<consequent> ::= <conclusion> { and <conclusion> }*
<clause>     ::= <condition> { or <condition> }*
<condition>  ::= <predicate><object><attribute>{<value>}*
<conclusion> ::= <predicate><object><attribute><value> ; <CF>
<predicate>  ::= same | notsame | contains | subsetof | known | notknown |
                 definite | notdefinite | lessthan | lessequal | greaterthan |
                 greaterequal | equal | notequal | . . .
```

Some predicates are restricted to a certain domain. For example the predicates "lessthan" and "lessequal" are permitted for attributes with a numerical domain only. Another example is the predicate which is used in a conclusion of a production rule. Only the predicate "same" is permitted to occur at that place. Each conclusion contains a certainty factor (CF). This certainty factor is an indication on a scale of [-1,1] how certain this conclusion is in respect to the antecedent of the rule. For the purpose of this paper, the certainty factor can be ignored in most situations. In situations in which the certainty factor is relevant, it will be introduced.

1.2. Faulty constructions of a knowledge base

Most problems in building a knowledge base arise in the construction of the production rules. The major part of these problems are due to syntactical errors. The use of a special knowledge base editor supports in avoiding these problems [1]. Such an editor assists in the construction of production rules. For example, an editor checks if objects and attributes which occur in a rule are defined. Similarly, the values of an attribute are checked with respect to the domain associated with the attribute. So, if a knowledge base editor is used, it will be assured that the production rules are syntactically correct and consistent with the object tree. Production rules that are constructed with an editor are all *legal* production rules.

DEFINITION *A production rule in a knowledge base is called legal, if and only if*
(i) *All predicates used in the conditions and the conclusions are defined, and used properly,*
(ii) *All objects are defined in the object tree,*
(iii) *All attributes are defined, and combined with the right object, and*
(iv) *All values are within the domain of their attribute.*
A rule is called illegal if and only if it is not legal.

Thus the problems which can be solved by using a special editor, are syntactical problems. The other problems that arise can be called empirical problems and semantical problems. An empirical problem arises when a production rule is semantically and syntactically all right, but encodes real-world knowledge inadequately. Most of these rules have to be detected by the experts in the domain, because they are the ones who can decide whether a certain conclusion can be drawn upon the fulfillment of the conditions or not. A well-known system that assists an expert in this task is TEIRESIAS[1] [2]. In section 1.3, TEIRESIAS will be considered in some detail. The last kind of

1. The program is named for the blind seer in *Oedipus the King,* since the program, like the prophet, has a form of "higher-order" knowledge.

problems are the semantical problems. These problems can be detected by checking the whole set of production rules by comparing the rules mutually. The semantical problems can be separated into the consistency of a knowledge base and the completeness of a knowledge base, and are the main topic of this paper.

1.3. TEIRESIAS

TEIRESIAS co-operates with the expert system MYCIN and assists an expert in modifying and developing a production rule, when during a consultation, an error is detected. In a kind of "ask and answer-game" the expert and the system produce or modify a production rule. One remarkable fact of TEIRESIAS is that it uses "rule models". Rule models are "abstract descriptions of subsets of rules, built from empirical generalizations about those rules and used to characterize a typical member of the subset" [4]. Rather than being hand-tooled, the models are assembled by TEIRESIAS on the basis of the current contents of the knowledge base. Because of this rule models, TEIRESIAS "knows" that rules about a certain topic have some specific conditions. So, if a rule is added lacking one or more specific conditions, TEIRESIAS will suggest that something is forgotten and asks about it. This does not mean that TEIRESIAS adds missing things to the knowledge base all by itself; the expert always has the last word, and TEIRESIAS waits for approval before proceeding. Additionally the remark can be made that TEIRESIAS does not find empirical or semantical problems by itself. If the user of the expert system locates a fault, TEIRESIAS only assist the user finding the rule which caused that fault and assists in fixing that fault. TEIRESIAS gives the expert the opportunity to add rules to a knowledge base without the assistance of a knowledge engineer. Thus TEIRESIAS demonstrates that it is possible to forge a direct link between an expert and a knowledge-based system, and allow the expert to test, evaluate and correct the performance of the system.

2. Some background in logic

This section introduces some definitions from propositional and first-order logic [5], because some of the results of first-order logic are needed in the next sections. Because first-order logic is preceded by propositional logic some propositional definitions are needed too. As the name suggests, the propositional logic deals with propositions. A proposition is a declarative sentence that is either *true* or *false*, but not both. Symbols that are used to denote propositions are called *atoms*. Compound propositions are built with *logical connectives*. In the propositional logic five logical connectives are used :

$\neg$ (not), $\wedge$ (and), $\vee$ (or), $\rightarrow$ (if...then), and $\leftrightarrow$ (if and only if).

With this knowledge next definition can be given.

Definition *A well-formed formula, or formula for short, is defined recursively :*

(i) *An atom is a formula.*
(ii) *If G is a formula, then ($\neg$G) is a formula.*
(iii) *If G and H are formulas, then (G$\wedge$H), (G$\vee$H), (G$\rightarrow$H) and (G$\leftrightarrow$H) are formulas.*
(iv) *All formulas are generated by applying the above rules.*

Each atom is assigned a truth value that is either **T** (true) or **F** (false). A formula can be composed of several atoms. The truth values of formulas are related to the truth values of the atoms in the formula. Next table represents that relationship.

G	H	$\neg$G	G$\wedge$H	G$\vee$H	G$\rightarrow$H	G$\leftrightarrow$H
T	T	F	T	T	T	T
T	F	F	F	T	F	F
F	T	T	F	T	T	F
F	F	T	F	F	T	T

TABLE 1. Truth values of basic formulas

If the relationship between the truth value of a formula and the truth values of the atoms it is composed of, is defined, the *interpretation* of a propositional formula can be defined.

DEFINITION *Given a propositional formula G, let $A_1,...,A_n$ be the atoms occurring in G. Then an interpretation of G is an assignment of truth values to $A_1,...,A_n$ in which every A_i is assigned to either* **T** *or* **F**, *but not both.*

In the field of a knowledge base of an expert system, it is necessary to introduce some definitions from the first-order logic. Before presenting a definition of a formula in the first-order logic, four more basic definitions are discussed.

DEFINITION *Terms are defined recursively as follows :*
(i) *A constant is a term.*
(ii) *A variable is a term.*
(iii) *If* f *is an* n*-place function symbol, and* $t_1,...,t_n$ *are terms, then* $f(t_1,...,t_n)$ *is a term.*
(iv) *All terms are generated by applying the above rules.*

A predicate is a mapping that maps a list of constants to {**T**,**F**}.

DEFINITION *If* P *is an* n*-place predicate symbol, and* $t_1,...,t_n$ *are terms, then* $P(t_1,...,t_n)$ *is an atom.*

Once atoms are defined, the same five logical connectives as in the propositional logic can be used to build up formulas. Furthermore, since variables are introduced, two special symbols $\forall$ and $\exists$ are used to characterize variables. The symbols $\forall$ and $\exists$ are called, respectively, the *universal* and *existential quantifiers*. If x is a variable, then $(\forall x)$ is read as *"for all x", "for each x"* or *"for every x"*, while $(\exists x)$ is read as *"there exists an x", "for some x"* or *"for at least one x"*. A variable in the first-order logic can be *bound* or *free*.

DEFINITION *An occurence of a variable in a formula is bound if and only if the occurence is within the scope of a quantifier employing the variable, or is the occurrence in that quantifier.*
An occurrence of a variable in a formula is free if and only if this occurrence of the variable is not bound.

DEFINITION *A variable is free in a formula if at least one occurrence of it is free in the formula.*
A variable is bound in a formula if at least one occurrence of it is bound.

Now, enough preparation is done to define a formula in the first-order logic.

DEFINITION *Well-formed formulas, or formulas for short, in the first-order logic are defined recursively as follows :*

(i) *An atom is a formula.*
(ii) *If G and H are formulas, then ¬(G), (G∧H), (G∨H), (G→H) and (G↔H) are formulas.*
(iii) *If G is a formula and* x *is a free variable in G, then (∀x)G and (∃x)G are formulas.*
(iv) *Formulas are generated only by a finite number of applications of (i), (ii) and (iii).*

In the propositional logic, an *interpretation* is an assignment of truth values to atoms. In the first-order logic, since there are variables involved, more has to be done. To define an interpretation in the first-order logic, the domain and an assignment to constants, function symbols and predicate symbols occuring in the formula have to be specified. The following is the formal definition of an interpretation of a formula in the first-order logic.

DEFINITION *An interpretation of a formula G in the first-order logic consists of a nonempty domain D and an assignment of values to each constant, function symbol and predicate symbol occurring in G as follows :*

(i) *To each constant, an element in D is assigned.*
(ii) *To each* n*-place function symbol, a mapping from D^n to D is assigned.*
(iii) *To each* n*-place predicate symbol, a mapping from D^n to* {**T**,**F**} *is assigned.*

For every interpretation of a formula over a domain D, the formula can be evaluated to **T** or **F** according to the following rules :

1. If the truth values of formulas G and H are evaluated, then the truth values of the formulas ¬(G), (G∧H), (G∨H), (G→H) and (G↔H) are evaluated by using the table in the beginning of this section.
2. (∀*x*)G is evaluated to **T** if the truth value of G is evaluated to **T** for every *x* in D; otherwise it is evaluated to **F**.
3. (∃*x*)G is evaluated to **T** if the truth value of G is evaluated to **T** for at least one *x* in D; otherwise it is evaluated to **F**.

The next two definitions can be used both in propositional and in first-order logic.

DEFINITION *A formula G is said to be satisfiable if and only if there exists an interpretation I such that G is evaluated to* **T** *in I (I satisfies G). A formula G is said to be unsatisfiable if and only if there exists no interpretation that satisfies G.*

DEFINITION *Given formulas $F_1,...,F_n$ and a formula G, G is said to be a logical consequence of $F_1,...,F_n$ (or G logically follows from $F_1,...,F_n$) if and only if for every interpretation I in which $F_1 \wedge ... \wedge F_n$ is true, G is also true.*

The definitions of interpretation and satisfiability are the two most important ones with respect to the domain of building a knowledge base of an expert system. As will become clear in the next section just a specific part of the first-order logic will be used.

3. Consistency

When designing an expert system, one of the most important aims is, to arrive at a system which does not contain any undesirable rule. Especially, when more and more production rules are added to the knowledge base the chance of introducing faults increases. In addition, many knowledge bases are constructed by several people. So, for example, it is possible to have two rules which succeed in the same situation with the same conclusion(s). Also possible is a situation in which two different rules conflict, meaning that two different rules have the same conditions, but conflicting conclusions. If two rules are exactly the same, there are two reasons making one of those rules unwanted. First, the possibility exists that one of those rules will be forgotten in a situation that both have to be modified. Second, when those rules are used in the consultation program of an expert system, the weight they give to their conclusion(s) is more than wished. Additionally, it is not of any use that rules occur more than once in a knowledge base, because it only increases space and time requirements of computation. In this section a method is presented to detect situations in which such faults occur.

3.1. Basic definitions

To consider a production rule of a knowledge base in a more logical formulation, the following definitions are given :

Definition *An attribute qualification is a variable of the form object.attribute.*

Definition *A rule-atom is an atom P(x,c), with "P" a 2-place predicate symbol, "x" an attribute qualification and "c" a constant.*

Definition *A production rule R (also written as $F \rightarrow G$) is defined as*
if $F_1 \wedge ... \wedge F_n$ **then** $G_1 \wedge ... \wedge G_m$ **fi**,
where $F_1 \wedge ... \wedge F_n$ is the antecedent, *and $G_1 \wedge ... \wedge G_m$ the* consequent *of that rule, and with each F_i (i=1,...,n) a finite disjunction of rule-atoms, called a PR-clause, and each G_j (j=1,...,m) a rule-atom.*

In relation with these definitions the (first-order logic) definition of interpretation can be reformulated for the "special" case of knowledge bases.

Definition *An KB-interpretation of a PR-clause G, consisting of rule-atoms $A_1,...,A_n$ is an assignment of truth values to $A_1,...,A_n$, in which every A_i is assigned to either* **T** *or* **F**, *but not both.*

This definition of KB-interpretation is a mixture of the definitions of interpretation of propositional and first-order logic.

3.2. A rule base consisting of just one production rule

To find out whether there are any situations in which a specific production rule is undesirable the following definition of *dependency* is required.

Definition *A consequent G of a production rule $F \rightarrow G$ is called dependent on F if and only if there exists an attribute qualification "a" for which there is a rule-atom A(a,c) in G and also a rule-atom P(a,d) in F. The consequent G is called independent of F if and only if G is not dependent on F.*

If G is independent of F, then $F \rightarrow G$ does not show any relationship between antecedent and consequent. On the other hand, if the consequent G of a rule is dependent on F, that rule is called a *self-referencing* rule, because that rule concludes about an attribute qualification which has been specified in the antecedent of the rule as well.

EXAMPLE IF a value for X is not known (after trying to establish one)
THEN conclude that the value of X is Z
(Default reasoning) [4]

Many inference engines are not capable of dealing with these rules and therefore self-referencing rules should be detected in a knowledge base. In dealing with a self-referencing rule, which could be applied by the inference engine, the expert of the knowledge has to decide whether or not this rule is useful. Rules like the one in the last example can be very useful in a knowledge base because of their capability to confer default values to attribute qualifications which can not be established another way, but the inference engine has to provide the capability to handle such rules, else they are not permitted. A problem arises if in a self-referencing production rule there exist a conclusion G_j and a condition F_i such that $G_j \wedge F_i$ is unsatisfiable.

EXAMPLE **if** `notsame(plant.colour,red)` **then** `same(plant.colour,red)` **fi**

Rules of this form have to be removed or modified into a rule like the first example of default reasoning, if default reasoning is allowed.

Another problem occurs if in a production rule $F \rightarrow G$, with $F=F_1 \wedge ... \wedge F_n$, two PR-clauses F_i and F_j exist, with $F_i \wedge F_j$ is unsatisfiable. The consequence of such a situation is that there are no KB-interpretations of $F_1,...,F_n$ such that F is true. In a knowledge base such rules are worthless, because the consequent G of that rule will never come true and so these rules will never contribute to the answer of a question in the consultation program.

If the problems that can arise, dealing with one single production rule, are considered, a definition can be given for rules which may occur in a knowledge base without causing undesirable situations. These rules are called *correct* rules.

DEFINITION *If $F \rightarrow G$ is a production rule, with $F=F_1 \wedge ... \wedge F_n$, and $G=G_1 \wedge ... \wedge G_m$, then that rule is called correct if and only if*
(i) *$F \rightarrow G$ is a legal production rule.*
(ii) *There are KB-interpretations of $F_1,...,F_n$ such that F is true (F is satisfiable), with each F_i a PR-clause.*
(iii) *For every conclusion G_j, there is no i such that $F_i = \neg G_j$ ($F \wedge G$ is satisfiable).*
(iv) *There are KB-interpretations of $G_1,...,G_m$ such that G is true (G is satisfiable).*
Otherwise, $F \rightarrow G$ is called incorrect.

In terms of a knowledge base, to check if a rule is correct, first, it is necessary to look for KB-interpretations such that the antecedent of the rule is true. Each rule-atom of a rule consists of a predicate symbol, an attribute qualification and a constant. If two rule-atoms contain different attribute qualifications, they can be true and false independently. Only if rule-atoms contain the same attribute qualification, their truth values can be dependent on each other.

EXAMPLE
```
if
same(plant.colour,blue)    ∧
same(plant.colour,yellow)  ∧
lessthan(plant.leaves,6)   ∧
greaterthan(plant.leaves,7)
then
same(plant.name,...)
fi
```

The first two conditions of this rule both relate to the colour of a plant. Now it depends on the definition of the attribute "colour" whether these two conditions can be true simultaneously. If a plant can have multiple colours, for example blue and yellow, the two conditions do not cause any trouble. The last two conditions contain another attribute qualification than the first two, so they do not influence the first two conditions. Looking at condition three and four, a contradiction is discovered, because they can not be true at the same time. Attribute "leaves" of object "plant" can not be less than 6 and greater than 7 at the same time. On behalf of this fact the rule in this example is incorrect, because there are no KB-interpretations of the PR-clauses such that the antecedent is **T** (true).

To find situations in a rule in which contradicting conditions exist the following algorithm is used :

```
function Rule_correct (new : rule) : boolean;

  begin
    correct:=true;
    repeat read(conditionA);
           Findconditions(same attribute qualification);
             {Search for conditions with the same object and attribute}
           if found
           then if contradiction(conditionA,condition)
                  then correct:=false
                  fi
           fi
    until all conditions have been checked or not correct
    Rule_correct:=correct
  end;
```

The algorithms for the third and fourth part of the definition are analogous.

3.3. A rule base consisting of more than one production rule

The chance of having a rule in a knowledge base which is not correct is rather small, because most rules are written by one author, who has a good view at that specific rule. More problems arise if we look at a set of production rules, which is very often constructed by several people. In this section the definition of *consistency* of a set of production rules in a knowledge base of an expert system is discussed.

According to [7], inconsistencies in the knowledge base appear as :

Conflict	Two rules succeed in the same situation, but with one or more conflicting conclusions.
Redundancy	Two rules succeed in the same situation and have some conclusions in common.
Subsumption	Two rules have some conclusions in common, but one contains additional restrictions on the situation in which it will succeed. Whenever the more restrictive rule succeeds, the less restrictive rule also succeeds, resulting in redundancy.

According to [3] one more situation causes inconsistency :

Circularity	Two or more rules form a cycle. The system will enter an infinite loop at run time, unless the system has a special way of handling circular rules.

With those specific situations the definition of *unwished related* can be formulated.

DEFINITION *Production rules in a knowledge base are called unwished related if and only if*
(i) *they are conflicting, or*
(ii) *they are redundant or partly redundant (subsumption), or*
(iii) *they have a relationship, which can cause an infinite loop.*

In this definition, subsumption is classified together with redundancy because rules which satisfy this part of the definition, have some conclusions in common. Finally, the definition of circularity is weakened to "cause an infinite loop" because in a system which prevents infinite looping, it may not be necessary to detect rules which are circular.

With the definition of a "correct" rule and the definition of "unwished related", the main definition of section three is constructed.

DEFINITION *A set of production rules of a knowledge base is called consistent if and only if*
(i) *every rule of that set is a* correct *rule, and*
if the set contains more than one rule,
(ii) *no two or more rules of that set are unwished related.*

A set of production rules containing only one production rule is consistent, if that rule is correct. Next, consider a consistent set of production rules consisting of more than one rule. Adding one new rule to that set, can render this (new) set to be inconsistent. Thus, it is necessary to check what harm a single rule can do to the consistency of a set of production rules of a knowledge base. In other words, how are the rules interrelated and in which situation is this relationship unwished.

DEFINITION *Two production rules $F \rightarrow K$ and $G \rightarrow L$ are called related if and only if there exist an attribute qualification "a", for which there are rule-atoms $F_i(a,c)$ and $G_j(a,d)$ in F and G respectively, or $K_i(a,c)$ and $L_j(a,d)$ in K and L respectively. Two production rules are called unrelated if and only if they are not related.*

The reason, to search for related rules just by looking at the attribute qualifications, is for example that while searching for rules with conditions in common also rules with contradicting conditions can be found. For example :

```
rule 1 : if  same(plant.colour,red)  then  ...  fi
rule 2 : if  notsame(plant.colour,red)  then  ...  fi
```

These two rules are found as related to each other, and as will become clear later on, it depends on their conclusion whether their relationship is unwished or not.

3.3.1. Conflict, redundancy and subsumption. According to the definition of "related to", when a new rule (say rule A) is added to a knowledge base, different sets of rules related to this new rule can be made. This way, for each condition of rule A a set of rules can be constructed which are related to A by means of the attribute qualification of that specific condition. The sets, that are created this way, "belong" to a certain condition of rule A and they are therefore called *condition sets*. A characteristic of these sets is that the elements are rules which have an attribute qualification in at least one of their conditions in common.

If for each condition of rule A a condition set is created, it is possible to conclude that rules which are not a member of the union of these condition sets, are not of interest for the purpose of finding

rules that can cause inconsistencies. Rules that are no member of that union are dealing with totally different attribute qualifications in their conditions than the attribute qualifications which are used in conditions of rule A. So, they need not be compared with rule A.

Using the definition of "unwished related", rules can be selected that are a real threat to the consistency of the knowledge base. First take the union of all condition sets of rule A and call this set $\mathfrak{U}_A$. The elements of set $\mathfrak{U}_A$ are rules, which are related to rule A in such a way that attribute qualifications exist in rule A which also occur in conditions of the elements of set $\mathfrak{U}_A$. If set $\mathfrak{U}_A$ is an empty set it is certain that there are no two conflicting or (partly) redundant rules, because a rule which has some rule-atoms in its antecedent in common with rule A has to be a member of set $\mathfrak{U}_A$. If set $\mathfrak{U}_A$ is a nonempty set, it can be divided into three disjunct sets, set I, set II and set III.

Set I contains rules for which the set of conditions is an improper superset of the set of conditions of rule A or vice versa. So, if rule X is an element of set I and if X_{cond} is the set of conditions of rule X, then $X_{cond} \supseteq A_{cond}$ *or* $X_{cond} \subseteq A_{cond}$. (A condition is represented by a rule-atom of the form P(a,c), thus a combination of predicate, attribute qualification and value.)

$$I = \{ X \in \mathfrak{U}_A \mid X_{cond} \supseteq A_{cond} \} \cup \{ X \in \mathfrak{U}_A \mid A_{cond} \supseteq X_{cond} \}$$

Set II is generated by taking all rules from the difference of set $\mathfrak{U}_A$ and set I, which are related on the basis of their conclusions. It means that elements of set II have at least one conclusion which deals with the same attribute qualification as a conclusion of rule A.

$$II = \{ X \in \mathfrak{U}_A \setminus I \mid \text{an attribute qualification exist, which is used in a rule-atom of a conclusion of X and also in a rule-atom of a conclusion of A} \}$$

Set III finally contains all rules which are an element of set $\mathfrak{U}_A$, but are neither an element of set I nor of set II.

$$III = \mathfrak{U}_A \setminus (I \cup II)$$

The elements of set III are not relevant any more, because they do not have any attribute qualification in their conclusions in common with attribute qualifications in conclusions of rule A. Additionally, elements of set III have at least one condition which differs from all conditions of rule A, and rule A has at least one condition which differs from all conditions of rule X, if rule X is a member of set III. Rules which differ in at least one condition and in all conclusions can not be conflicting nor causing (partial) redundancy. The problem of circularity will be treated separately.

The elements of set II have at least one conclusion concluding on an attribute qualification of a conclusion of rule A. Additionally, at least a part of their conditions deal with some of the attribute qualifications of the conditions of rule A. To decide if a member of set II and rule A can be reformulated in one single rule or whether or not one of those rules has to be removed because of a contradiction between them, most rules of set II have to be compared with rule A by the knowledge engineer (Maybe also the expert in the domain has to assist in solving certain problems.). On forehand it is difficult to say which situation can occur and how to handle in that situation. One situation can be considered explicit. It is when two or more rules have exactly the same conclusion (Rules which have more than one conclusion, can be temporarily divided into more rules with each just one conclusion.). As shown in the next example, rules which have a conclusion in common have to be modified into one new rule, with that conclusion as consequent. The antecedent of the new rule is constructed by taking the disjunction of the antecedents of the original rules.

EXAMPLE X: **if** A(a,x) ∧ (B(b,y) ∨ C(c,z)) **then** K(f,u) **fi**
A: **if** A(a,x) ∧ (B(b,y) ∨ D(d,w)) **then** K(f,u) **fi**

Rule X is a member of set II, because rule-atom C(c,z) of rule X is no condition of rule A and rule-

atom D(d,w) of rule A does not occur in rule X. However they have a conclusion in common. The problem of these rules is that whenever rule-atoms A(a,x) and B(b,y) are evaluated to true, rule-atom K(f,u) is evaluated to true twice, on the ground of the same conditions, but different rules (A and X). This means that rule A and rule X cause redundancy. If rule A is right that K(f,u) may be concluded if A(a,x) and D(d,w) are true, then rule A has to be removed and rule X has to be modified into :

X: **if** A(a,x) $\wedge$ (B(b,y) $\vee$ C(c,z) $\vee$ D(d,w)) **then** K(f,u) **fi**

In this specific example, the two conclusions of the rules were equal, but, for example, it is also possible that two conclusions only differ with respect to their certainty factors. Then another solution has to be found. This way, set II can have several rules which have some relationship which has to be examined more accurate.

The set which remains is set I. If rule X is a member of set I, two situations can appear :
(i) $X_{cond}=A_{cond}$
(ii) $X_{cond} \supset A_{cond}$ or $X_{cond} \subset A_{cond}$.

Let us first discuss $X_{cond}=A_{cond}$. Even if both rules A and X have exactly the same conditions, the relationship between these conditions can be different.

EXAMPLE X: **if** A(a,x) $\vee$ B(b,y) **then** R(d,v) **fi**
A: **if** A(a,x) $\wedge$ B(b,y) **then** Q(c,z) **fi**

Both rules contain identical condition sets, but have no further relationship. Because of this, the elements of set I with $X_{cond}=A_{cond}$ have to be examined by the knowledge engineer. One situation which can be considered on forehand is when the antecedent of rule X is exactly the same as the antecedent of rule A. This situation needs special attention, because now it is possible that rule A and rule X are conflicting or redundant. If two rules exist with the same antecedent three different situations can occur. First, it is possible that they have one or more conflicting conclusions. Second, they can have one or more conclusions in common and finally their conclusions can be independent of each other. The problems created by such rules can be solved by creating one new rule Z, which has the same antecedent as the rules A and X. The consequent of that rule Z has to consist of the conjunction of all conclusions of rule A and all conclusions of rule X, on the condition that :
-- if two conclusions conflict with each other, one of them has to be removed (the expert of the knowledge usually has to decide which one)
-- if two conclusions are exactly the same this conclusion may only occur once in the consequent of rule Z.
If the creation of rule Z is finished, rule A and rule X have to be removed.

REMARK The creation of a new rule Z is not always possible. For example if production rules of a knowledge base are represented by Horn-clauses, each rule can have just one conclusion. This situation makes it necessary that rules exist which only differ in respect to their conclusion. The creation of a new rule Z has to be followed then by a division of rule Z into rules with the same antecedent, but with one conclusion in their consequent, in such a way that all conclusions of rule Z will be used in exactly one rule.

The situation in which $X_{cond} \supset A_{cond}$ will be considered next. ($X_{cond} \subset A_{cond}$ is treated analogously.) This means that rule X has more conditions than rule A, but all conditions of rule A are also conditions of rule X. In this situation the same problem occurs as with $X_{cond}=A_{cond}$. Although all conditions of rule A are also conditions of rule X it is not sure that both rules have an antecedent that is partly in common. This time again the rules of set I are selected for which the antecedent of rule A is exactly the same as a part of the antecedent of rule X. The other rules have to be examined

"by hand".

EXAMPLE X: **if** P(a,x) ∧ Q(b,y) ∧ R(c,z) **then** ... **fi**
A: **if** P(a,x) ∧ Q(b,y) **then** ... **fi**

A distinguishing mark of the rules which are considered in this section is that the antecedent of rule X consist of the antecedent of rule A conjuncted with some other PR-clause(s). From this collection, rules have to be detected that are inconsistent because of subsumption. The best way to find those rules is to find out whether some of the conclusions of rule X are related to conclusions of rule A. If related conclusions are found, a difference has been made between conclusions which are exactly the same and conclusions which are only related because they conclude on the same attribute qualification. If two conclusions are exactly the same, rule X and rule A have to be modified into rules without a subsumption relation (As will be considered in next example.). As defined earlier in this paper, a conclusion of a production rule contains a rule-atom and a certainty factor (cf). If two related conclusions conclude about different values, they do not cause inconsistency in the knowledge base, because they conclude a different fact. In an example the situation of two rules with a rule-atom in their consequent in common, is considered.

X: **if** F(a,x) ∧ G(b,y) **then** $K(c,z)_{cf=0.8}$ ∧ $L(d,u)_{cf=0.7}$ **fi**
A: **if** G(b,y) **then** $K(c,z)_{cf=0.3}$ ∧ $M(e,v)_{cf=0.9}$ **fi**

Rule A is added to a knowledge base in which rule X already exists. The antecedent of rule A is partly identical to the antecedent of rule X, and they have one conclusion in common which deals with the same rule-atom (K(c,z)). New in this example is the occurrence of the certainty factor which indicates on a scale of [-1,1] how certain a conclusion is. If the certainty factors of K(c,z) had been equal, then rule A and rule X should have been partly redundant, because rule X contains additional restrictions on the situation in which it will succeed compared with rule A, and thus whenever rule X succeeds also rule A succeeds with the same conclusion. Now, with different certainty factors, those two rules have to be modified into three rules. Rule X will remain the same. Rule A has to be divided in two parts :

Aa: **if** G(b,y) **then** $K(c,z)_{cf=0.3}$ **fi**
Ab: **if** G(b,y) **then** $M(e,v)_{cf=0.9}$ **fi**

The rules X and Ab may exist together, but rule Aa is still redundant. The expert of the knowledge could decide to remove rule Aa, but a more usual solution is to modify rule Aa :

Aa': **if** ¬F(a,x) ∧ G(b,y) **then** $K(c,z)_{cf=0.3}$ **fi**

This rule Aa' expresses that K(c,z) can be concluded with cf=0.3 if only condition G(b,y) is true and rule X expresses that K(c,z) can be concluded with cf=0.8 if also condition F(a,x) is satisfied. In the situation that both certainty factors are equal, the expert has to decide whether the whole rule Aa' (or Aa) has to be removed or just the conclusion $K(c,z)_{cf}$ of rule X.

So far, rules which are unwished related, by means of the first two parts of the definition, are detected. In the next part of this section the third part of the definition is considered.

3.3.2. Circularity. The last situation in which relationships between rules occur, is when two or more rules form a cycle. For example :

```
(1): if same(skin.colour,grey)       then same(animal.nose,very_long) fi
(2): if same(animal.nose,very_long) then same(animal.name,elephant)  fi
(3): if same(animal.name,elephant)  then same(skin.colour,grey)      fi
```

In DELFI-2 this kind of rules does not cause very much harm, because every rule can be used only once, thus it is impossible to have an infinite loop. But nevertheless, when checking a knowledge base for consistency, it is important to know which rules form a cycle. A method of finding such cycles is the following. Start with a knowledge base, which does not contain cycles. Next, add one new rule A to that knowledge base and look whether this rule causes a cycle.

To find a cycle two methods can be used. The first method is according the following recursive procedure :

```
procedure Find_cycles (new,pr : rule ; found : boolean);

  begin
    Concl:=[]; {Set of rules}
    repeat read(conclusion of pr);
           repeat Findcondition(conclusion of pr);
                    {Search for a rule with a condition,
                     in which the conclusion of pr is used}
                  if found
                  then Concl+:=[rule]
                  fi
           until all rules have been checked
           for each element of Concl
           do if element<>new
              then Find_cycles(new,element,found)
              else found:=true
              fi
           od
    until all conclusions of pr have been checked
  end;
```

If this procedure is started for the first time, the arguments "new" and "pr" are both the new rule which is added to the knowledge base (rule A). Then for each conclusion of rule A, a set of rules is created, which have that conclusion as condition. The members of those sets are treated the same way as rule A, until rule A is found as a member of one of those sets, meaning that a cycle has been found. To find the conclusions and conditions which were causing that cycle, the rules that were used in the cycle shall have to be traced. If a cycle has been found and the system with which the knowledge base will be used does not prevent infinite looping, at least one rule in the cycle has to be removed or modified in such a way that no longer a cycle exists. The second method for finding circularities is almost the same as the first. This time the procedure starts to read conditions and looks for conclusions in which that condition is used.

3.4. Modification of a production rule

If a rule is deleted from the knowledge base, the consistency of that knowledge base is not harmed, because a deletion can not cause an unwished relationship between other rules. The modification of a rule can cause inconsistency, so if a rule is modified it has to be treated as a new rule which is added to the knowledge base.

3.5. Conclusion

The algorithms described in section three are meant to help the knowledge engineer in detecting situations in which a knowledge base and in particular a set of production rules can be inconsistent. The algorithms detect most inconsistencies, but in a lot of situations the knowledge engineer or even the expert in the domain has to solve these problems. In addition to the detection of inconsistencies, the described algorithms select rules which have some relationship, and gives the knowledge engineer the opportunity to rewrite some rules.

4. Completeness

Where consistency is the property that all production rules are mutually compatible, completeness is the property that with respect to the current contents of the knowledge base no lack of information exists, meaning that according to the current knowledge base, its contents seems to be consisting of all desired knowledge. To check a knowledge base for completeness, the point of departure is the definition of the attribute qualifications. In the first place, every attribute qualification should be used in at least one production rule. In the special case of an attribute qualification with domain text, each legal value has to be used in at least one production rule. Obviously, an attribute qualification that is not refered to in any production rule is redundant. Therefore, in this section each defined attribute qualification is considered to be required in the knowledge base and thus is used in at least one production rule. The same argument counts for the legal values of an attribute qualification with domain text.

To check if really all defined attribute qualifications and their values occur, all production rules have to be checked. If an attribute qualification or legal value is used in a rule, a mark can be placed in the object tree, to indicate that a certain attribute qualification or legal value is used. If all production rules have been checked, it is possible to identify all unmarked attribute qualifications or values. In addition, the knowledge engineer shall have to decide either to remove these values or to add rules in which the unmarked attribute qualifications occur. Values that do not occur in production rules are causing so-called *missing rules*, one of three situations in which a knowledge base can be incomplete. According to [3] two other types of rules cause incompleteness :

(i) Unreachable rules
(ii) Deadend rules

The definitions of unreachable and deadend rules depend on the way the production system operates. In a data-driven production system these definitions should be interchanged, meaning that an unreachable rule in a data-driven production system is a deadend rule in a goal-driven production system and vice versa. In this paper the knowledge base is considered to be one of a goal-driven production system, because it is based on the earlier mentioned DELFI-2 system.

4.1. Unreachable rules

In a goal-driven production system, a conclusion of a rule should either contain an attribute qualification which is a goal of the consultation or match a condition of another rule. Otherwise that rule is called *unreachable*. Unreachable rules are worthless in a knowledge base, because they never take part in the search process. Finding unreachable rules is important, because those rules can be a guide for detecting more missing rules. This detection of missing rules can be considered in two ways. First it is possible that an unreachable rule contains attribute qualifications or values that only appear in that rule. Those attribute qualifications or values play no role in the knowledge base, so in fact they do not exist in the knowledge base, and are causing *missing rules*, in the way described in the beginning of this section. On the other hand, it can be necessary to add one or more rules to the knowledge base to make those unreachable rule(s) reachable.

4.2. Deadend rules

A *deadend* rule is a rule in which the conditions are neither askable nor appearing as conclusion of another rule. For a deadend rule it is never possible to decide whether it succeeds or not, because at least one condition can not be evaluated. The problems which occur when dealing with a deadend rule are rather similar to the problems of an unreachable rule. This way it is possible that deadend rules contain attribute qualifications or values that only occur in that specific rule, and it can also be necessary to add one or more rules to make deadend rules "alive" again.

4.3. Detecting unreachable and deadend rules

To solve a situation in which unreachable or deadend rules occur, one can just remove those rules. However this procedure can cause other rules to become unreachable or deadend. To detect rules that are unreachable or deadend in a knowledge base of a goal-driven expert system, next algorithm can be followed :

```
procedure Find_unreachable_and_deadend;

  begin
    deadend:=[];
    unreachable:=[];
    repeat read(rule);
          repeat read(condition);
                if condition is not askable
                then Findconclusion(condition)
                      {Search for a rule which concludes
                       about the condition}
                     if not found
                     then deadend+:=[rule]
                     fi
                fi
          until all conditions have been checked
          repeat read(conclusion);
                if conclusion is notgoal
                then Findcondition(conclusion)
                      {Search for a rule with a condition,
                       in which the conclusion is used}
                     if not found
                     then unreachable+:=[rule]
                     fi
                fi
          until all conclusions have been checked
    until all rules have been checked
  end;
```

In the next section another algorithm of finding rules which cause incompleteness of a knowledge base will be discussed.

4.4. Modification of a production rule

The deletion of a rule can harm the knowledge base in respect to its completeness. It is possible that unreachable or deadend rules are created, or that the last reference of an attribute qualification or value is removed from the rule base. To check if the deletion of a rule really causes incompleteness of the knowledge base, the attribute qualifications and values which occur in that rule have to be checked. If the deletion of a rule causes incompleteness this problem can be solved by also deleting those attribute qualifications or values from the object tree, or by adding a new rule to the rule base in which those attribute qualifications or values are used. A new rule, which is added to a complete knowledge base only causes incompleteness, if that rule is unreachable or deadend. For each new rule only those two situations have to be checked. If a rule in a complete knowledge base is modified, some conclusions and/or conditions are deleted and others are added. So a modified rule has to be treated as a deleted rule if conclusions and/or the conditions are deleted, and it should be treated as a new rule for the new part of it.

5. Implementation of the checking algorithm

The difference between the methods of checking for consistency and for completeness is the way the knowledge base is treated. Checking for consistency can be done by using induction. This makes it possible to check for consistency while adding a new rule to the knowledge base. Additionally this makes it possible to keep a not yet finished knowledge base consistent. If it is necessary to check a finished knowledge base for consistency, only the rules which are already checked have to be compared with the rule which is checked next. This means that for a knowledge base with n rules, maximal ½n(n-1) checks have to be done, instead of n(n-1) checks which would be necessary if each rule is compared with all others.

Checking for completeness can only be done if a knowledge base is finished, because completeness deals with the knowledge base as a whole.

5.1. The checking algorithm in practice

One way to find rules which are causing inconsistency or incompleteness in practice is the following. While loading the rules it is possible to connect each rule to the attribute qualifications and values that are used in that rule in conditions and conclusions. The attribute qualifications are loaded in an object tree. If each attribute qualification, and for an attribute qualification with domain text each value, has a connection to the rule in which it is used in a condition and to the rule in which it is used in a conclusion, then a lot of the search process can be done with the help of these explicit stored relationships. For example it is possible to find rules which are related to each other in the sense of section three, just by looking at the object tree. Rules which are related to each other are all connected to a certain attribute qualification which is used in these rules. To find rules that are related to a certain rule, it is sufficient to use the structured object tree and it is not necessary to search for related rules by checking other rules in the knowledge base. In relation with completeness not only missing rules can be found this way, but also deadend and unreachable rules. To find a deadend rule, for example, it is sufficient to select attribute qualifications which are not askable. If one of those attribute qualifications (or values of those attribute qualifications) is connected to rules in which it is used in a condition, then that attribute qualification is causing deadend rules if there is no connection with a rule in which it is used as a conclusion.

Based on these words, the following proposition can be stated :

Proposition *Production rules which are candidate for causing inconsistency or incompleteness of a knowledge base can be traced by just using the object tree.*

As was considered in the beginning of this section, the checking of a knowledge base for consistency is possible at the moment that a new rule is added to the knowledge base. This makes it attractive to built a consistency checker as part of a knowledge base editor, instead of building a separate checking system. This knowledge base editor then produces production rules which are syntactical correct, and in addition the knowledge base containing the rules is consistent.

The checking for completeness can be done separately from the editing of a knowledge base. In other words, if building the knowledge base is finished, according to the knowledge engineer or any other person who is building an expert system, it is the right moment to start a separate completeness checker. On the other hand, it can be usefully to check the knowledge base for completeness at some earlier time, just to detect objects, attributes or values that are not used in a production rule yet. A first experimental implementation, based on the algorithms described in this paper, produced satisfiable results, meaning that missing rules and unreachable and deadend rules were located, in several knowledge bases.

6. Conclusion

The aim of this paper was to define and detect inconsistency and incompleteness of a set of production rules of a knowledge base and to discuss some algorithms to achieve this aim within the practice of knowledge engineering. The last proposition states a method which is practical in tracing inconsistency and incompleteness. If this method is integrated with a knowledge base editor, the constructing of knowledge bases can be done with much more reliability in respect to syntactical and semantical problems. The first experimental implementations seemed successful, so future efforts can lead to an allround knowledge base editor which assists a knowledge engineer in building syntactical and semantical faultless knowledge bases.

References

1. R. C. van Soest (May 1986). *Een KNOWLEDGE BASE EDITOR voor DELFI-2 (in Dutch),* University of Amsterdam.
2. R. Davis, D. B. Lenat (1982). *Knowledge-Based Systems in Artificial Intelligence (Part Two : "TEIRESIAS: Applications of Meta-Level Knowledge"),* Mc Graw-Hill, Inc.
3. T. A. Nguyen, W. A. Perkins, T. J. Laffey, D. Pecora. *Checking an expert systems knowledge base for consistency and completeness,* Lockheed Research and Development O/92-10 B/254E.
4. B. G. Buchanan, E. H. Shortliffe, eds. (1984). *Rule Based Expert Systems,* Adisson-Wesley Publishing Company.
5. C. Chang, R. C. Lee (1973). *Symbolic Logic and Mechanical Theorem Proving,* Academic Press New York.
6. D. R. Hofstadter (1979). *Gödel, Escher, Bach: an eternal golden braid,* Basic Books.
7. M. Suwa, A. C. Scott, E. H. Shortliffe (1982). *An approach to verifying completeness and consistency in a rule-based expert system,* Stanford University Report No. STAN-CS-82-922.
8. Ir. H. de Swaan Arons, E. P. Janssen, Drs. P. J. F. Lucas, Dr. Ir. H. Stienen (February 1985). *DELFI-2 Handleiding (in Dutch),* TH-Delft.
9. P. J. F. Lucas (April 1986). *Knowledge representation and inference in rule-based systems,* Centre for Mathematics and Computer Science, Report CS-R8613.