Building Validation Tools For Knowledge-Based Systems

R.A. Stachowitz, C.L. Chang, T.S. Stock and J.B. Combs
Lockheed Missiles & Space Company, Inc.
Lockheed Artificial Intelligence Center, O/90-06, B/30E
2100 East St. Elmo Road
Austin, Texas 78744

Abstract

The Expert Systems Validation Associate (EVA), a validation system under development at the Lockheed Artificial Intelligence Center for more than a year, provides a wide range of validation tools to check the correctness, consistency and completeness of a knowledge-based system.

Using a declarative meta-language (higher-order language), we want to create a generic version of EVA to validate applications written in arbitrary expert system shells.

In this paper we outline the architecture and functionality of EVA. The functionality includes Structure Check, Logic Check, Extended Structure Check (using semantic information), Extended Logic Check, Semantic Check, Omission Check, Rule Refinement, Control Check, Test Case Generation, Error Localization, and Behavior Verification.

INTRODUCTION

The growing reliance on knowledge-based systems (KBS's) in industry, business and government requires the development of appropriate methods and tools to validate such systems to ensure their correctness, consistency and completeness. Furthermore, as these systems become operational, an increasing number of knowledge engineers will be involved in their development and maintenance. Hence, insuring the integrity of a particular KBS over its entire life cycle makes the need for automated validation even more crucial.

Early attempts at validating KBS's did not progress beyond basically "syntactic" validation [Nguyen et al. 1985, Nguyen 1987, Reubenstein 1985, Suwa et al. 1982]. Significantly more advanced validation tools are being built at the Lockheed AI Center using semantic information and meta-knowledge for KBS validation [Stachowitz et al. 1987a, 1987b].

The system under development is called the Expert Systems Validation Associate (EVA). Our goal is to create a generic version of EVA which can validate applications written in arbitrary expert system shells, such as ART*, KEE**, OPS5, and others, by mapping an application written in any specific shell into internal data structures in a general and declarative meta-language (higher-order language).

The purpose of EVA is to improve the validation process by finding mistakes and omissions in the knowledge base, by proposing knowledge base extensions and modifications, and by showing the impact of changes to the knowledge base. In other words, EVA addresses not only the question "Is a KBS application correct?", but also the question "Is the knowledge used by the expert for the application correct?"

In this paper, we outline the architecture and functionality of EVA. The functionality includes structure check, logic check, extended structure check (using semantic information), extended logic check, semantic check, omission check, rule refinement, control check, test case generation, error localization, and behavior verification.

EVA ARCHITECTURE

To permit the addition of future functionality EVA is designed to be a *meta*knowledge-based system shell. All the validation modules (checkers) will be built on a unifying, extensible platform. The unifying architecture will be based upon: (a) A single user interface for all checkers; (b) A single meta-knowledge base for all checkers; and (c) A common meta-language for specifying constraints. The advantages of the architecture are uniformity and extensibility.

We intend to implement all the checkers in Quintus Prolog [Quintus 1985]. That is, information about an application represented in a knowledge base, algorithms for the checkers, and domain knowledge will be represented by clauses in Quintus Prolog. Then, invoking a checker is essentially comparable to entering a goal (query) in Quintus Prolog. Our selection of Prolog is based on the following considerations:

(1) The validation of KBS's requires extensive automatic theorem proving facilities such as a unification subroutine. Prolog has a built-in automatic resolution-based theorem prover. By using Prolog we can expedite the development of EVA.

(2) Maintaining the meta-knowledge base requires the functionality of a database management system. Prolog provides a built-in database management system, which makes it unnecessary to develop such functionality separately.

(3) The meta-language required for representing validation criteria as meta-statements can be defined and implemented in Prolog. This makes it unnecessary to develop a separate abstract machine to interpret this meta-language.

EVA FUNCTIONALITY

The functionality of EVA, represented by means of a data flow diagram, is depicted in Figure 1.

An application is written in an object shell, while validation statements for semantic and control constraints, and behavior descriptions are written in a very expressive metashell (meta-language). The metashell will provide many higher-order constructs to support high-level predicates such as *symmetric, nonsymmetric, transitive, nontransitive, reflexive, irreflexive, mandatory, synonymous, compatible, incompatible*, etc.

An analyzer uses conversion algorithms to translate the application and the validation statements into the format to be used by the EVA validation modules; each of which performs a static or dynamic analysis of the application. The analyzer will build a connection graph from facts and rules in the application. An arc in the connection graph denotes a match between a literal in the LHS (Left-Hand Side) of a rule and a literal in the RHS (Right-Hand Side) of a rule. Note that a fact is considered as a rule consisting of a RHS only.

For static analysis, EVA will provide the structure checker, logic checker, extended structure checker, extended logic checker, semantics checker, omission checker, rule refiner and control checker. For dynamic analysis, EVA will provide the test case gen-

*Automated Reasoning Tool, Inference Corporation

**Knowledge Engineering Environment, IntelliCorp

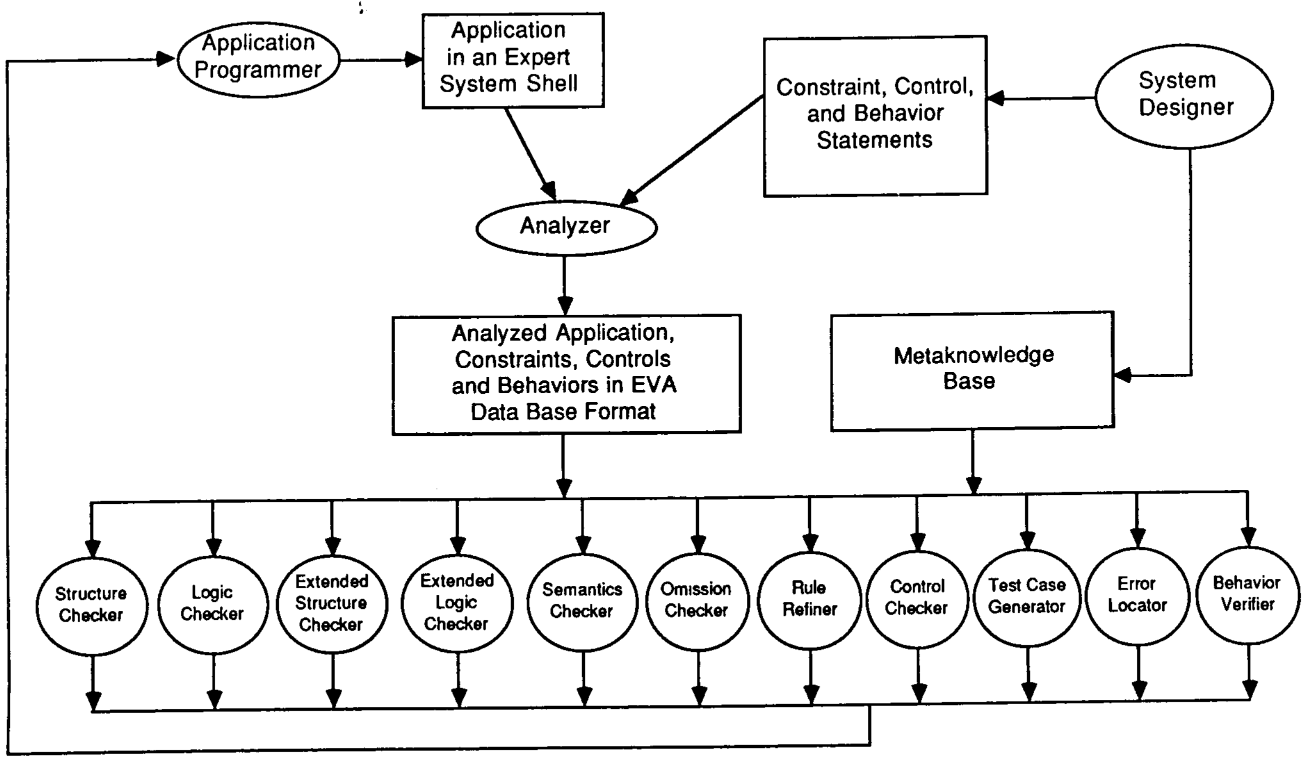

Figure 1. Validation Functionality

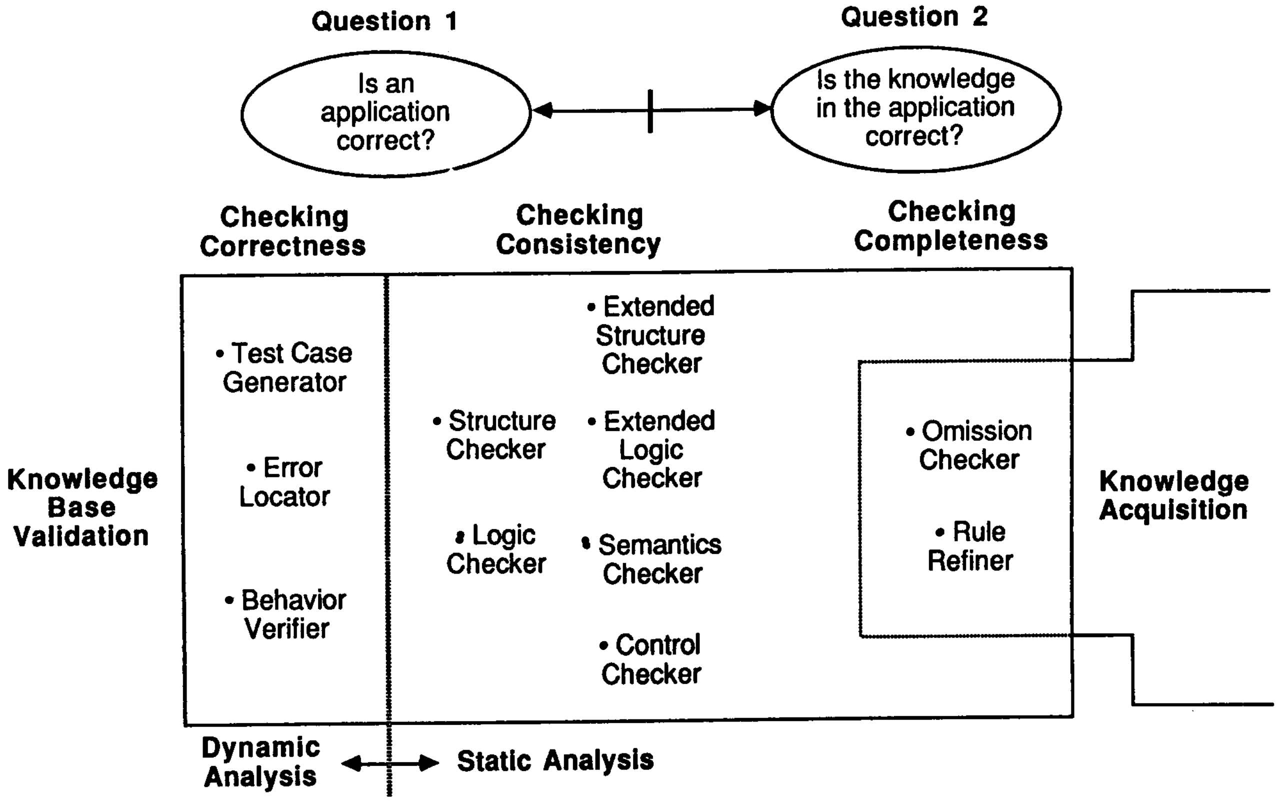

Figure 2. Validation Objectives and Functionality

erator, error locator and behavior verifier. The validation objectives and functionality of EVA are shown in Figure 2.

Using ART and LISP, we have implemented prototypes of the structure checker, logic checker and semantic checker for ART-based expert systems. We have also implemented considerable portions of the structure checker and extended structure checker in Quintus Prolog. Other modules, and more elaborate functions for the structure checker, logic checker and semantics checker are being designed and will be implemented in the future.

We now give detailed descriptions of the EVA modules as follows:

STRUCTURE CHECKER

As discussed before, a knowledge base can be represented by a connection graph. That is, the connection graph basically shows *the structure* of the knowledge base. The purpose of the structure checker is to detect any anomalies in the connection graph. For example, it will identify rules and facts which will never be used, rules which are superfluous, and rules which possibly lead to an infinite loop.

Reachability

One type of "common" error is *the use of an undefined* literal in the LHS of a rule. This may occur in top-down and bottom-up specifications. In the top-down approach, the specifier starts with the highest modules, planning to define the lower modules afterwards. However, he may forget them. Therefore, they are left undefined. In the bottom-up approach, the lower modules are defined first. However, when the specifier tries to use them, he may type the module names incorrectly.

A LHS literal may be undefined because either its predicate is not defined or it cannot be unified with any RHS literal. In terms of the *connection graph, a literal is undefined* if it is not pointed to by any arc.

Another type of related anomaly is the case where RHS literals are not linked to any LHS literals. That is, *they are defined but not* used. They are called "unreachable" literals. The presence of the unreachable literals may indicate some anomalies, namely, omissions. This is analogous to the experience we have with repairing a car. We may disassemble a carburetor. However, when we put the parts back again, we may find some parts are left unused.

In terms of the connection graph, a RHS literal is unreachable if there is no arc going from it to another literal.

Redundancy

The types of redundancies the structure checker will check are duplications and subsumptions. The duplication and subsumption checks can be done by the same algorithm because duplication is a special case of subsumption.

Given two rules R1 and R2, if the LHS of R1 subsumes the LHS of R2, then whenever R2 can be fired R1 also can be fired. Therefore, any actions in the RHS of R2 which also occur in the RHS of R1 can be eliminated. If after the elimination no actions are left in the RHS of R2, then R2 can be eliminated.

The following are examples of duplication and subsumption:

Duplication:

$\text{male}(X) \wedge \text{parent}(X) \rightarrow \text{father}(X).$

$\text{parent}(Y) \wedge \text{male}(Y) \rightarrow \text{father}(Y).$

Subsumption:

$\text{tenured}(X) \wedge \neg\text{staff}(X) \rightarrow \text{university\_member}(X).$

$\text{tenured}(X) \rightarrow \text{university\_member}(X).$

Cycles

In a knowledge base, cycles will occur if recursive rules are used to define predicates. Some cycles are harmless, while others cause problems. EVA will identify "potentially bad" cycles.

First, we consider the case where rules are used to define predicates [Chang 1976, 1978, 1981]. A predicate is called a basic predicate if it is used only for representing facts. A predicate is called a derived predicate if it is defined in terms of basic predicates, or in terms of basic and derived predicates by a rule or a set of rules. For example, the predicate "ancestor" may be defined in terms of the basic predicate "parent" as follows:

$\text{parent}(X,Y) \rightarrow \text{ancestor}(X,Y).$

$\text{parent}(X,Y) \wedge \text{ancestor}(Y,Z) \rightarrow \text{ancestor}(X,Z).$

The first rule is a terminating rule and the second rule is a recursive rule. If the terminating rule is not given by the developer, it will cause the second rule to fire repeatedly, ie., result in an *infinite* loop. Therefore, if EVA finds that cycles of rules are used to define a derived predicate, it will try to check whether terminating rules are given.

Second, we consider the case where rules are used to establish equivalent predicates. For example, consider the following rules:

$\text{human}(X) \rightarrow \text{person}(X).$

$\text{person}(X) \rightarrow \text{human}(X).$

These two rules result in a cycle. However, the developer may intend to show that predicates "human" and "person" are equivalent or synonymous. In this case, he should choose one of the predicates as the standard predicate and delete one of the rules.

Finally, we consider the case where a cycle (loop) is used to perform a repeated task in a distributed or non-distributed environment. In this case, cycles are allowed.

EVA will also check if a knowledge base contains overlapped cycles (loops) [Chang 1978]. A knowledge base with overlapped cycles is less modularly structured than one without any overlapped cycles. Since we can always represent a knowledge base without using overlapped cycles, we should enforce this software methodology. This is analogous to structured programming where (since only single-entry and single-exit statements are allowed) loops are well structured in structured programs.

LOGIC CHECKER

The logic checker checks if at some particular time *inconsistent or* conflicting actions can be triggered by facts in a knowledge base.

For example, consider the rules:

$\text{big}(X) \rightarrow \text{expensive}(X).$

$\text{big}(Y) \rightarrow \neg\text{expensive}(Y).$

If the fact

big(truck)

is in the knowledge base, then the rules will assert the inconsistent facts:

expensive(truck).

$\neg$expensive(truck).

On the other hand, consider the rules:

$\text{new}(X) \wedge \text{big}(X) \rightarrow \text{expensive}(X).$

$\text{broken}(Y) \wedge \text{big}(Y) \rightarrow \neg\text{expensive}(Y).$

Depending upon what set of facts exists at a particular time, these two rules may or may not assert inconsistent facts. For example, if we only have the facts

new(truck1).

big(truck1).

big(truck2).

broken(truck2).

then the rules will assert the facts

expensive(truck1).

$\neg$expensive(truck2).

which are not inconsistent. However, if we only have the facts

new(truck1).

big(truck1).

broken(truck1).

then we will get inconsistent facts:

expensive(truck1).

$\neg$expensive(truck1).

The question is whether such a set of facts is ever possible. If the developer thinks it is impossible that an object can be simultaneously new, big and broken, then the inconsistency may not arise. Otherwise, he should be warned. Note that to make sure that a certain set of facts is impossible, the knowledge base will be verified by the semantic checker to be discussed later. That is, the knowledge base will contain "negative" semantic constraints to be used by the semantic checker. For the above example, the negative constraint is represented as

incompatible(new(X), big(X), broken(X)).

Given a set S of rules, the logic checker finds if there exists a set T of facts that may lead the rules in S to generate inconsistent facts. If such a set T exists, the validation system will prompt the developer if T is possible. If he says that T is impossible, a corresponding negative constraint will be added into the knowledge base.

EXTENDED STRUCTURE CHECKER

The extended structure checker checks for reachability, redundancy and cycles caused by generalization hierarchy or synonymy.

Most expert system shells support generalization hierarchy based upon the subclass relationship "isa". For example, if

submarine **isa** ship,

then "submarine" may not be undefined if "ship" is defined, and "ship" may not be unreachable if "submarine" is used in a LHS-literal of a rule.

Similarly, generalization hierarchy can affect the redundancy and cycle properties of a knowledge base. For example, given the rules

$submarine(X) \rightarrow launch(X).$

$ship(X) \rightarrow launch(X).$

clearly the second rule subsumes the first one.

Now, let us consider synonymy. An expert system may be developed by more than one person. Different persons may use different schemas (names or structures) for the same type of objects. When their knowledge is merged, some standard should be established. EVA will provide the meta-language to map the schemas into a standard schema by using rules. For example, if "sub" and "submarine" are synonymous, and if we choose "submarine" to be the standard name, then the mapping rule is given as

$sub(X) \rightarrow submarine(X).$

Mapping rules map a slot or a function of many slots into a standard slot. For example, the slots "year" and "date-of-birth" may be mapped into the slot "age". This mechanism is similar to "view" definitions in a relational data base.

Once the standard schemas are established, the functionality of the extended structure checker for the knowledge base is essentially the same as the functionality of the structure checker for the knowledge base appended by the mapping rules.

EXTENDED LOGIC CHECKER

The purpose of the extended logic checker is to check for inconsistencies and conflicts caused by generalization hierarchy, incompatibility, or synonymy. If the application contains rules that can derive contradictory conclusions from the same set of facts with the properties of generalization hierarchy, incompatibility, or synonymy, then the application is inconsistent.

Inconsistency Under Generalization Hierarchy

Given the metafact that *submarine* is a subclass of *ship*, rules 8 and 9 are inconsistent.

Rule 8: $E(X) \wedge F(X) \rightarrow \neg ship(X).$

Rule 9: $E(Y) \wedge F(Y) \rightarrow submarine(Y).$

Inconsistency Under Incompatibility

Given the metafact that *boy* and *girl* are incompatible, rules 10 and 11 are inconsistent.

Rule 10: $A(X) \wedge B(X) \rightarrow boy(X).$

Rule 11: $A(Y) \wedge B(Y) \wedge C(Y) \rightarrow girl(Y).$

Inconsistency Under Synonymy

Given the metafact that *submarine* and *sub* are synonymous, rules 12 and 13 are inconsistent.

Rule 12: $E(X) \wedge F(X) \rightarrow submarine(X).$

Rule 13: $E(Y) \wedge F(Y) \rightarrow \neg sub(Y).$

In each of the above examples, the conditions in each of the rule pairs are exactly the same or one condition is a proper subset of another. However, the inconsistency can occur in a more general case. The examples below are in **conflict** if **all** of the facts A(e), B(e), C(e), and D(e) are present in the knowledge base **at the same time**.

Conflict Under Generalization Hierarchy

Given the metafact that *submarine* is a subclass of *ship*, rules 14 and 15 are in conflict.

Rule 14: $A(X) \wedge B(X) \rightarrow \neg ship(X).$

Rule 15: $C(Y) \wedge D(Y) \rightarrow submarine(Y).$

Conflict Under Incompatibility

Given the metafact that *boy* and *girl* are incompatible, rules 16 and 17 are in conflict.

Rule 16: $A(X) \wedge B(X) \rightarrow boy(X).$

Rule 17: $C(Y) \wedge D(Y) \rightarrow girl(Y).$

Conflict Under Synonymy

Given the metafact that *submarine* and *sub* are synonymous, rules 18 and 19 are in conflict.

Rule 18: $A(X) \wedge B(X) \rightarrow \neg sub(X).$

Rule 19: $C(Y) \wedge D(Y) \rightarrow submarine(Y).$

If the conditions A(X), B(X), C(X), and D(X) can be satisfied by future facts in the application (recognized from assertions in the RHS of rules), the logic checker warns of *potential conflict*.

The extended logic checker will use an algorithm similar to the one used by the logic checker. The extended logic checker will first choose a goal consisting of a complementary pair of RHS literals or a set of incompatible RHS literals. It will then try to find a set of facts from the goal by performing unifications and substitutions on literals, using rules in the knowledge base as rewriting rules.

EVA provides the meta-predicate "incompatible" for the developer to specify a set of incompatible literals. It has the following structure

incompatible(L1,...,Ln),

where L1,...,Ln are literals for n equal to 1 or more. The meaning of this statement is that the conjunction of L1,...,Ln can not be true at any time. Incompatible statements are interpreted as "negative constraints". In a knowledge base or meta-knowledge base, there are general and domain-specific negative constraints. The following are some examples of incompatibility statements:

(1) incompatible(A, $\neg$A)

means that an atomic formula A and its negation are incompatible. Note that A may contain a first-order or higher-order predicate.

(2) incompatible(assert(X), retract(X))

means that asserting and retracting the same fact at the same time are incompatible, where X is a literal.

(3) incompatible(add(O,S,V1), delete(O,S,V2),
single_value(O,S))

means that adding a value V1 to a slot S of an object O and deleting a value V2 from the same slot of the same object are incompatible if the slot is a single-valued slot.

(4) incompatible(modify(O,S,V1), delete(O,S,V2),
single_value(O,S))

means that modifying the value of a slot S of an object O to V1 and deleting a value V2 from the same slot of the same object are incompatible if the slot is a single-valued slot.

(5) incompatible(room(X), in(john,X), in(mary,X))

means that for a particular application we have the negative constraint "John and Mary can not be in the same room".

(6) incompatible(on(X,X))

means that an object X can not be on itself.

SEMANTICS CHECKER

The semantics checker has two major functions: cnecking facts in a knowledge base of an application written in the object shell for violations of the semantic constraints, and checking the semantic constraints themselves for internal consistency and agreement with other metaconstraints represented in the metashell.

We can have "positive constraints" and "negative constraints". The facts in the knowledge base must satisfy the former, but must not satisfy the latter. The constraints will be stated by using the meta-predicates of the meta-language. A negative constraint can be represented by using the meta-predicate "incompatible" as discussed before. Some of the meta-predicates (meta-relations) for specifying positive constraints such as range constraints, minimum/maximum cardinality constraints, legal value constraints, value compatibility constraints, subrelations, inverses, data types, etc. are shown below:

(1) lower_upper(slot, class, lower, upper)

This metarelation defines the legal range of numerical values: the values for the <slot> of the <class> must be between <lower> and <upper>. EVA discovers and flags values that exceed these bounds.

EVA not only checks facts against semantic constraints, but also checks that the semantic constraints themselves are consistent. Since a "child" is a "person", the age range of "child" must fall within the age range of "person". Thus the following semantic constraints would be inconsistent.

is_a(child, person)
lower_upper(age, person, 1, 110)
lower_upper(age, child, 0, 12)

(2) legal_value(slot, class, values)

This metarelation defines the legal values of a slot: the values for the <slot> of the <class> must be listed in <values>.

For example, the following semantic constraint

legal_value(gender, student, [male,female,hermaphrodite])

states that the gender of a student must be male, female, or hermaphrodite. EVA flags any "student" record where "gender" has a nonlegal value.

(3) relation(rel(domain-1,...,domain-n))

This metarelation defines both the number of legal arguments of a relation <rel> and the legal data type of each argument; <domain-i> is either a class of objects or a set of values for i=1,...,n.

This kind of semantic constraint is used to permit EVA to enforce strong data-type checking for relations.

For example, given the two facts

person(charlie)

and

dog(snoopy)

and the metafact

relation(murderer_of(person,person))

EVA flags as erroneous the fact

murderer_of(charlie,snoopy).

Another example of constraints

relation(enroll(freshman,{math101,english101,...}))
relation(enroll(sophomore,{math201,art201,...}))

states that a freshman can only enroll in Math101, English101,... that a sophomore can only enroll in Math201, Art201, ..., and so on.

(4) min_max_rel(relation, domain, min, max)

This metarelation specifies the minimum and maximum number of tuples (records) of a relation: the number of records of <relation> with object in <domain> must be between <min> and <max>.

For example, the following semantic constraint

min_max_rel(enroll, student, 0, 5000)

means that up to 5000 student enrollments are allowed in the data base at any one time. The enrollments are represented by records or tuples of the relation "enroll".

(5) min_max_role(relation, domain, min, max)

This metarelation defines that each object in <domain> must have at least <min> and at most <max> objects for the relation <relation>.

Thus the semantic constraint

min_max_role(enroll, sophomore, 3, 4)

states that each sophomore must enroll in at least 3 and at most 4 courses. EVA flags any sophomore who enrolls in fewer than three or more than four courses.

(6) subrelation(relation1, relation2)

This metarelation defines that <relation1> is a subrelation of <relation2>. EVA checks that the number of arguments for <relation1> is greater than or equal to the one for <relation2>, and that the data types of the arguments of <relation1> are subclasses of the corresponding arguments of <relation2>.

For example, EVA determines that the semantic constraints

relation(killer_of (animate_obj, animate_obj))
relation(murderer_of (person,thing))
subrelation(murderer_of, killer_of)

are inconsistent since the second argument "thing" of "muroerer_of" is not a subclass of the second argument "animate_obj" of "killer_of".

EVA also checks that the inverse of <relation1> is a subrelation of the inverse of <relation2>, and creates the missing inverse of a subrelation, if one does not exist.

Our meta-language will also permit the developer to define properties of predicates or relations, such as *transitive, non-transitive, symmetric, non-symmetric, reflexive, irreflexive, antonymous* (male-female, ie., non-male implies female and vice versa), *contrary* (young-old, ie., non-young does not imply old), etc.

OMISSION CHECKER

Knowledge can be organized around the concept of set, eg., a class of objects, a class of relations, a class of rules, and a set of values for a multi-value slot. Given a set written by the developer, the basic question to ask is "Is the set complete?". In other words, does the set contain all the necessary elements or lack some elements? The goal of the omission checker is to answer this question by investigating and identifying useful techniques and representations for defining completeness of a knowledge base. Some of these techniques are given as follows:

(a) Class Taxonomy

If the developer creates only the classes for BOY, MAN and WOMAN,

PERSON(sex:{male,female},age:integer)
>>> BOY(sex=male,age≤12)
>>> MAN(sex=male,age>12)
>>> WOMAN(sex=female,age>12)

(where the classes are written in upper case, the slots in lower case, and the subclass relationship is denoted by >>>,) then the omission checker will prompt him if there should be a class for persons who are female and not older than 12.

(b) Relation Taxonomy

Given a knowledge base as shown below,

```
PERSON >>> MAN
       >>> WOMAN

PARENT-OF(person,person)
 v
 v
 v
FATHER-OF(man,person)
```

the omission checker will find that the taxonomy for PARENT-OF is incomplete because the PERSON class for the first argument of PARENT-OF seems to split into the MAN and WOMAN classes. Therefore, it will prompt the developer whether there should be another subrelation of PARENT-OF that holds between WOMAN and PERSON, namely, MOTHER-OF.

(c) Omission of Rules Or Facts

Arguments of predicates may be associated with classes. By analyzing and comparing the arguments, the omission checker may detect that certain facts or rules for some classes are missing.

Consider the following knowledge base:

```
PERSON>>>ADULT>>>WOMAN
              >>>MAN
     >>>CHILD >>>GIRL
              >>>BOY
```

GO(passenger,from,to).

TAKE(passenger,flight,fare).

DEPARTMENT(department_name,floor).

ADULT(x) ∧ GO(x,austin,atlanta) → TAKE(x,f#7,150)

CHILD(x) ∧ GO(x,austin,atlanta) → TAKE(x,f#7,75)

DEPARTMENT(man,2).

DEPARTMENT(woman,1).

If we look at the rules defining TAKE, we know the domain for the first argument of TAKE is the union of ADULT and CHILD, namely, PERSON. However, the domain for the first argument of DEPARTMENT is the union of MAN and WOMAN, namely, ADULT. The idea of checking missing rules or facts is to find the minimal class that is the domain for some argument of a predicate. If two minimal classes are related by the subclass relationship, then the rule or fact set associated with the smaller minimal class is likely to be incomplete. For the above example, the fact set for DEPARTMENT is incomplete. That is, the omission checker will prompt the developer on which floor the "child" department is.

(d) Incomplete Slot Values

There may be a set of typical objects for a slot of an object. For example, a room is a complex object that contains many other objects as parts. The room can be represented by a schema which has a slot named "containing". A value of this slot is a set of other objects, typically such as table, chair, board, PC, etc. These typical parts of an object can be stored in the meta-knowledge base. When a specific object is created and it does not contain some of the typical parts, the omission checker will tell the developer.

RULE REFINER

A rule may be too general or too restrictive. Specific test cases will be chosen from the knowledge base to prompt the expert if the rule should apply to the test cases. Any "no" answer will indicate that the rule is too general, and more specific rules will be proposed. If he answers "yes" to all the test cases, it may indicate that the rule is too restrictive, and other test cases in sibling classes of the generalization hierarchy will be chosen.

Consider a knowledge base given as follows:

```
PERSON>>>MAN---MAN(Sam,22,USA)
               ---MAN(Ted,42,USA)
               ---MAN(Ray,52,France)
      >>>WOMAN---WOMAN(Sara,37,USA)
```

where the schema for PERSON is PERSON(*name,age,place_of_birth*).

If we have the following rule saying that every man can be the president of USA

MAN(x) → CAN-BE-PRESIDENT-OF-USA(x),

then the rule refiner will test the rule by presenting some instances of MAN and asking the developer if Sam, Ted and Ray can be the president of USA. The answer will be "no" for Sam and Ray, and "yes" for Ted. Since there are "no" answers, the rule is too general, so that the developer will change the rule to

MAN(x) ∧ ≥age(x,35) ∧ place_of_birth(x,USA)
→ CAN-BE-PRESIDENT-OF-USA(x).

Now, the rule refiner will test the modified rule by presenting an instance of WOMAN and ask if Sara can be the president of USA. The answer will be "yes". This means that the modified rule is too restrictive. Therefore, the rule will be changed to

PERSON(x) ∧ ≥age(x,35) ∧ place_of_birth(x,USA)
→ CAN-BE-PRESIDENT-OF-USA(x).

The goal of the rule refiner is to help the developer refine his rules. Since this is an interactive process, a good and comprehensive user interface is required.

CONTROL CHECKER

As larger knowledge bases for complex applications are implemented, some software engineering methodology [Jacob and Froscher 1986] should be developed. One method is to partition a large knowledge base into smaller subsets of facts and rules. These subsets can be labeled by meaningful names such as activity names. Conversely, we can start with the activity names and then implement each activity by a set of facts and rules.

The implicit execution model of a knowledge-based system is given as follows: A rule has a RHS and LHS. The rule will be fired if the LHS is satisfied by the knowledge base. When the rule is fired, the RHS tells the system to add, change or delete facts and objects.

In an application, there may be "ordering" constraints (called control constraints) among the activities. The control checker permits the developer to specify the control constraints, and then verify if rules in a knowledge base will be executed in a sequence that does not violate the control constraints. For example, in an office system, there are the activities for clearing and publishing papers. An ordering constraint is that a paper must be "cleared" before it is "published". Assume the activities are specified as follows:

CLEARING_ACTIVITY:

paper(X) ∧ approved(X) → cleared(X).

PUBLISHING_ACTIVITY:

paper(X) ∧ accepted(X) → publish(X).

EVA will recognize that the control constraint is violated because there are no data dependencies between these two activities. That is, there are no RHS-literals in the first activity used in the LHS of the rule in the second activity.

TEST CASE GENERATOR

As discussed before, a knowledge base consisting of facts and rules can be represented by a connection graph. In the connection graph, there are two kinds of leaf nodes, namely, *input* nodes and *output* nodes. An input node is a node representing a fact that is connected to LHS-literals of some rules. An output node is a node representing a RHS-literal that is not connected to any LHS-literals.

There are two ways to test the knowledge-based system. One approach is to generate different sets of input nodes (input test cases) to exercise the system and observe data produced at the output nodes. The goal is to traverse each arc in the connection graph at least once. The input test cases must satisfy semantic constraints that are imposed on the system.

The other approach is to specify requirements on data at the output nodes. Each requirement will be represented as a query. EVA will check that all input facts will satisfy all the queries.

Consider an example where the speed of an engine is controlled by the position of a valve of a fuel system. A value of the position and a value of the speed are input and output data, respectively. We may generate different values of the position and observe the values of the speed. On the other hand, we can specify that the speed should fall within a certain range and then check if such a requirement can be fulfilled.

ERROR LOCATOR

The error locator is to locate "incorrect" rules which derive "incorrect" facts from input facts. For example, consider an adder that is specified by the following rule:

input(1,N,V1) ∧ input(2,N,V2) ∧ adder(N)
→ *output*(N, V1+V2).

The adder has two input ports 1 and 2. It takes values V1 and V2 at the input ports, and produces the sum of the values at its output port. If we have the following input facts for adder a

input(1,a,10)
input(2,a,50)

the system should produce

output(a,60).

Now, if the adder is specified by the incorrect rule

input(1,N,V1) ∧ input(1,N,V2) ∧ adder(N)
→ output(N, V1+V2),

the system will produce the incorrect output

output(a,20).

To help the developer, the error locator will present him the deduction tree of the incorrect fact so that he can debug it. For the above example, the deduction tree uses only one fact, namely, input(1,a,10), with the rule. This should give the developer the necessary hint to correct the incorrect rule.

BEHAVIOR VERIFIER

A system may be decomposed into many subsystems. A subsystem may be represented by a collection of facts and rules in the object shell. However, the subsystem must have external input/output interfaces to communicate with the outside world. For example, in the space shuttle flight software system, the navigation controller is a subsystem that sits in a control loop, collects and analyzes data, and then sends control signals to the vehicle manipulator.

The behavior of the subsystem is a description of relationships among the external input/output interfaces and internal states of the subsystem. The subsystems are connected together to form the total system. The purpose of the behavior verifier is to prove that the intended behavior of the system can be derived from the behaviors of the subsystems and the description of their connections.

CONCLUSION

This paper has described the architecture and functionality of EVA. It is evident that EVA provides a powerful means for representing knowledge about an application domain and for verifying that the knowledge is correct, consistent and complete. EVA increases the reliability of knowledge-based systems, speeds up their development, and assists in their continuing modification. The necessity for such validation tools will continue to grow as future knowledge-based systems play a more critical role in business, industry, government, and the sciences.

ACKNOWLEDGMENTS

The authors would like to thank Linda B. Burris of Lockheed Artificial Intelligence Center for her careful reading of the paper and many useful comments.

REFERENCES

Chang, C.L. [1976] DEDUCE --- A deductive query language for relational data bases, in *Pattern Recognition and Artificial Intelligence* (C.H. Chen, Ed.), Academic Press, Inc., New York, 1976, pp.108-134.

Chang, C.L. [1978] DEDUCE 2: Further investigations of deduction in relational data bases, in *Logic and Data Bases*(H. Gallaire and J. Minker, Eds.), Plenum Publishing Corp., New York, 1978, pp.201-236.

Chang, C.L. [1981] On evaluation of queries containing derived relations in a relational data base, in *Advances in Data Base Theory --- Volume 1* (H. Gallaire, J. Minker and J.M. Nicolas, Eds.) Plenum Publishing Corp., 1981, pp.235-260.

Jacob, R.J.K., and Froscher, J.N. [1986] *Developing a software engineering methodology for knowledge-based systems*, NRL Report 9019, Computer Science and Systems Branch, Information Technology Division, Naval Research Laboratory, Washington, D.C. 20375-5000.

Nguyen, T.A. [1987] Verifying consistency of production systems, *Proc. of the 3rd IEEE Conference on AI Applications*, February 1987, pp.4-8.

Nguyen, T.A., Perkins, W.A., Laffey, T.J., and Pecora, D. [1985] Checking an expert systems knowledge base for consistency and completeness, *Proc. of the 9th International Joint Conference on Artificial Intelligence*, 1985, pp.375-378.

Quintus [1985] *Quintus Prolog Reference Manual*, Quintus Computer Systems, Inc., 2345 Yale Street, Palo Alto, CA 94306.

Reubenstein, H.B. [1985] *OPMAN: An OPS5 rule base editing and maintenance package*, MIT Master's thesis, MIT, AI Laboratory, 545 Technology Square, Cambrideg, Ma 02139.

Stachowitz, R.A., and Combs, J.B. [1987a] Validation of Expert Systems, *Proc. of the 20th Hawaii International Conference on Systems Sciences*, 1987, pp.686-695.

Stachowitz, R.A., Combs, J.B., and Chang, C.L. [1987b] Validation of Knowledge-Based Systems, *Proc. of the 2nd AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, Arlington, Virginia, March 9-11, 1987.

Suwa, M., Scott, A.C., and Shortliffe, E.H. [1982] An approach to verifying completeness and consistency in a rule-based expert system, *The AI Magazine*, 1982, pp.16-21.

THE AUTHORS

Dr. Stachowitz received his Ph.D. in Linguistics from the University of Texas at Austin in 1969. His background includes design and development of a large-scale knowledge-based mechanical translation system, computer hardware and software performance evaluation, and research in applicative programming languages, semantic data models, and analytic modeling and performance evaluation of data base machine architectures. He also has performed research in logic and functional knowledge base manipulation and query languages. He is currently a research scientist at Lockheed's Artificial Intelligence Center where he is the co-principal investigator of the Knowledge-Based Systems Validation project, the topic of this paper.

Dr. Chang received his Ph.D. in Electrical Engineering from the University of California, Berkeley, CA in 1967. His background includes design and developemnt of large-scaled knowledge-based systems, and research in program generation, very high level languages, compilers, rapid prototyping, relational data bases, natural language query systems, mechanical theorem proving, and pattern recognition. He wrote two books "Symbolic Logic and Mechanical Theorem Proving" and "Introduction to Artificial Intelligence Techniques", and published more than 50 papers. He is currently a co-principal investigator of the Knowledge-Based Validation project.

Todd Stock received his B.A. in Computer Science from the University of Texas at Austin in 1986. He worked for the UT CS department doing research and implementation of artificial intelligence and machine learning systems under Dr. Bruce Porter. He is currently at Lockheed's Artificial Intelligence Center where he is implementing the Knowledge-Based Systems Validation project in Prolog.

Ms. Combs received her M.S. in Computer Science/Mathematics from the University of Texas at Dallas in 1984, where she was awarded a research assistantship in the area of artificial intelligence. She was a member of the technical staff of the Corporate Engineering Center at Texas Instruments in Dallas, where her work included the design and development of process control knowledge-based systems and expert system shells. She is presently working at Lockheed's Artificial Intelligence Center where she is the associate principal investigator on the Knowledge-Based Systems Validation project.