Technical Note 494 • December 1990

Proving Properties of Rule-Based Systems

Prepared by:

Richard J. Waldinger, Principal Scientist
Mark E. Stickel, Principal Scientist

Artificial Intelligence Center
Computing and Engineering Sciences Division

An abbreviated version of this paper is to appear in the proceedings of the Seventh IEEE Conference on Artificial Intelligence Applications, Miami Beach, Florida, February 1991.

This research is supported by the United States Air Force, Rome Air Development Center under Contract F30602-87-D-0094 and the National Science Foundation under Grant CCR-8904809. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies, either expressed or implied, of the Rome Air Development Center, the United States Air Force, the National Science Foundation, or the United States government.

Abstract

Rule-based systems are being applied to tasks of increasing responsibility. Deductive methods are being applied to their validation, to detect flaws in these systems and enable us to use them with more confidence.

Each system of rules is encoded as a set of axioms that define the *system theory*. The operation of the rule language and information about the subject domain are also described in the system theory. Validation tasks, such as establishing termination, unreachability, or consistency, or verifying properties of the system, are all phrased as conjectures. If we succeed in establishing the validity of the conjecture in the system theory, we have carried out the corresponding validation task.

If the proof is restricted to be *sufficiently constructive*, we may extract from it information other than a simple yes/no answer. For example, we may obtain a description of a situation in which an error or anomaly may occur.

A method for the gradual formulation of specifications based on the attempted proof of a series of conjectures has been found to be suitable for rule-based systems. Such a specification can serve as the basis for a reengineering of the system using conventional software technology.

Validation conjectures are proved or disproved by a new theorem-proving system, SNARK, which implements (nonclausal) resolution and paramodulation, an optional constructive restriction, and some facilities for proof by induction. The system has already been applied to prove properties of a number of simple rule-based systems.

1 Introduction

Languages based on rules are an appealing implementation vehicle for expert systems. The system can be developed incrementally without much preliminary planning. In introducing a new rule, one supposedly need have little understanding of how the rest of the system behaves. The rules may embody the advice of many different experts, who are ignorant

of each other's opinions and may even disagree with each other. Proponents of rule-based methodologies have found that they can develop running systems far more quickly than with conventional programming languages.

As a consequence of this success, expert systems based on rules have been proposed for tasks of increasing responsibility, including aircraft and spacecraft fault diagnosis as well as financial and medical advice. For this reason, the question arises of how we can establish that these systems will be worthy of our confidence?

This is where a conflict emerges. Accumulated experience suggests that to be trustworthy, a system must be constructed in a systematic way that begins with an attempt to formulate its specification prior to the implementation effort. This doctrine is antithetical to the rule-based system methodology, in which the intended behavior of the system changes at each stage of its implementation. The system is developed in the absence of specifications; in fact, the methodology may be regarded as a framework for rapid prototyping, in which we gradually formulate an executable specification through experimentation. On the other hand, a complex nondeterministic system of rules is rarely acceptable as a specification; it is difficult for anyone, including its developers, to predict what it will do.

It is not the purpose of this paper to criticize or improve the rule-based system methodology. Rather, we shall attempt to apply deductive techniques to support the methodology as it is practiced. We shall provide techniques to determine what a rule system does, to identify its faults, and to establish confidence in it. One may be able to formulate a single specification that characterizes the intended behavior of the system. That specification may then be used as the basis for a reimplementation of the system using conventional software-engineering techniques. We may hope that the reimplemented system will be more efficient, concise, and reliable than the original.

In other cases, in which the system is too complex to allow a full specification to be verified or even formulated, we can use deductive methods to assist in the testing of the system. We can generate sets of test cases that exercise all the rules of the system or that cause certain inconsistencies or anomalies to occur. We can detect that certain rules will

never be executed. The same deductive framework can serve a variety of these purposes.

Because rules look like logical sentences, it is tempting to treat them that way, and analyze them for properties such as consistency. In fact, rules cannot usually be understood in a purely declarative way. They are imperative constructs with the intended side effects of adding elements to and deleting them from a single data structure, the "working memory." In this paper, we treat a rule language as an imperative language. Because all side effects in the language alter a single structure, the language is more amenable to logical analysis than most imperative languages, such as those with general assignment statements.

Special problems arise because of the nondeterministic nature of rule-based languages. A situation can occur in which more than one rule is applicable, and the system implementation must choose between them. Different implementations may make different choices and a system may behave correctly in one implementation and not in another. It may be difficult to anticipate what different implementations may do.

Conventional program verification often assumes that a full specification for a correct system is available. In this work, we recognize that the system may be incorrect and the specification may be only partial. We detect faults and formulate the specification gradually, as a result of attempts to prove a series of conjectures about the system. We may also attempt to prove the conjecture's negation, which will hold if the system fails to possess the desired property.

In most work on program verification, a proof gives us at best a yes/no answer as to whether a system meets its specification. Implicit within a proof, however, is other potentially valuable information, which is usually discarded. For example, if we prove the existence of a fault in a system, we may be able to extract from the proof a description of the conditions under which that fault occurs. Program synthesis techniques for extracting programs from constructive proofs (e.g., Manna and Waldinger [9]) may also be applied to extract other sorts of information.

1.1 Validation Tasks

In keeping with these goals, we consider a variety of validation tasks, most of which have both a positive and a negative aspect.

- (+) Verification: Proving that a system will always satisfy a given condition.

  (−) Fault detection: Exhibiting an input that causes a system to fail to satisfy a given condition.

- (+) Termination: Proving that a system will always terminate.

  (−) Loop detection: Exhibiting an input that will cause a system to fail to terminate.

- (+) Firing: Exhibiting an input that causes a given rule to fire.

  (−) Unreachability: Proving that no input will cause a given rule to fire.

- (+) Consistency: Proving that no input can produce an inconsistent working memory.

  (−) Inconsistency: Exhibiting an input that will produce an inconsistent working memory.

Some of these problems are significantly more difficult than others. For example, to prove that a system always terminates will generally require considering all execution histories beginning from any possible input, at least in principle. Exhibiting an input that causes Rule A to fire may require considering a small number of inputs and only part of the system. Thus, we can expect to be successful at this smaller task more readily than at establishing termination.

1.2 The System Theory

Our approach is deductive. For a given system of rules, we develop a *system theory*, which is defined by a set of axioms that express the actual behavior of the system. The system theory also incorporates any background knowledge we wish to take into account in validating the system. For each validation task, there is an associated conjecture. If we can manage

to establish the validity of the conjecture in the system theory, we have performed the associated validation task. Typically, to perform the negative aspect of a task, we prove the negation of the conjecture associated with its positive aspect. If we want to exhibit an input or other object as part of our task, we must restrict the proof syntactically to be *sufficiently constructive*; this means that the proof will tell us how to build such an object. The description of the object can then be extracted from the proof automatically.

1.3 Related Work

There have been several efforts to apply conventional testing techniques to rule-based systems (e.g., Becker et al. [1], Kiper [8]). The body of work closest to ours is that of Chang, Combs, and Stachowitz [11, 4, 5] at the Lockheed Artificial Intelligence Center. The Lockheed work, like ours, deals with some specifications of the expected properties of the rule-based system and uses deductive methods to establish them. It uses Prolog as an inference system, which limits it to properties expressed as Horn clauses. The SNARK theorem prover we are developing accepts properties in a full first-order logic, with mathematical induction. Unlike most researchers, we deal with rules that may delete as well as add elements of the working memory. Our work is also original in that it presents a unified theoretical framework for expressing and establishing properties of rule-based systems.

1.4 Outline of this Paper

We first provide a description of a somewhat idealized rule language in Section 2. For a given system of rules, we show how to construct a corresponding system theory in Section 3 and in Section 4 how to translate validation tasks into conjectures in the theory. In Section 5, we exhibit portions of a validation proof and present a scenario leading to the formulation of a specification for a textbook rule-based system. Finally, in Section 6 we describe the SNARK theorem prover we have been developing to prove validation conjectures and to extract information from validation proofs.

2 Rule-Based Systems

In this section we present a prototype rule-based system framework that will serve as the focus of our effort.

2.1 The Rule Language

Our rule language is a smoothed-up version of OPS5 [7, 3]. A rule describes an operation to be performed on working memory. The *working memory* is a (finite) set of atoms $p(t_1, \ldots, t_k)$, where p is a predicate symbol and $t_1, \ldots, t_k$ ($k \geq 0$) are terms. The atoms in working memory are *ground*, that is, they contain no variables, only constant, function, and predicate symbols. An atom $p()$ with no arguments will be written p. For example,

$$\{farmer(john), banker(mother(john))\}$$

is a working memory.

A *rule* is an expression of the form

$$L_1, \ldots, L_m \longrightarrow R_1, \ldots, R_n \quad .$$

Here each element L_i of the left side is a *literal*, that is, either an atom $p(t_1, \ldots, t_k)$ or the negation *not* $p(t_1, \ldots, t_k)$ of an atom. Each element R_i of the right side is an (unnegated) atom. We do not require rule elements to be ground, that is, they may contain variables. For example

$$red(x),\ not\ big(x) \longrightarrow blue(x)$$

is a rule. We impose certain restrictions on rules; these will be discussed after we have described rule application.

2.2 Rule Application

To apply a rule to working memory, we first select a ground instance of the rule, that is, we replace each of its variables with a corresponding ground term. We require that the

instances of the positive (unnegated) atoms on the left side of the rule be present in working memory and that the instances of the negated atoms be absent.

For example, the rule

$$\textit{Rule } B: \quad red(x),\ \textit{not } big(x) \longrightarrow blue(x)$$

is applicable to the working memory

$$\{red(b), big(a)\}.$$

The appropriate rule instance is obtained by replacing x with the ground term b. The instance $red(b)$ of the positive atom $red(x)$ is present in this working memory; the instance $big(b)$ of the negated atom $big(x)$ is absent.

To apply an applicable rule to the working memory, we delete the selected instances of the positive atoms of the left side and add the instances of the atoms of the right side. For example, to apply Rule B to the working memory $\{red(b), big(a)\}$, we delete $red(b)$ and add $blue(b)$, to obtain the new working memory

$$\{blue(b), big(a)\}.$$

Note that instances of the positive atoms on the left side are deleted as the rule is applied. This means that if any of these atoms is to be retained, it must appear on the right side as well, so that it can be added back into working memory.

For example, the rule

$$red(x), big(x) \longrightarrow blue(x)$$

will delete instances of both $red(x)$ and $big(x)$ from working memory. If it is intended that the rule retain the instance of $big(x)$, the rule must read

$$red(x), big(x) \longrightarrow blue(x), big(x).$$

(This rule describes a situation in which big red blocks are to be painted blue, but remain big.) The rationale for this convention is that the positive atoms of the left side are replaced

in working memory by the atoms of the right side; thus the rule behaves as a rewriting of working memory.

One restriction we impose on the language is that every variable that occurs anywhere in a rule must occur in some positive atom on the left side. This implies that once we have instantiated these positive atoms, we have instantiated the entire rule. For example, the rule

$$red(x),\ not\ big(y) \longrightarrow red(z)$$

violates this restriction for two reasons: the variable y from the negated atom $big(y)$ and the variable z from the right side are not present in the positive atom $red(x)$ on the left side. If we attempt to apply this rule, it is unclear which instance of $big(y)$ is to be absent from working memory and which instance of $red(z)$ is to be added.

We do not allow explicit negation signs in the working memory. This is not an essential limitation. If it is desired to express that an atom $p(t_1, \ldots, t_k)$ is false, we can introduce a new predicate symbol $negp$, and include $negp(t_1, \ldots, t_k)$ in the working memory, with the understanding, to be expressed in the theory, that $negp(t_1, \ldots, t_k)$ is the complement of $p(t_1, \ldots, t_k)$.

A *rule system* is an (unordered) set of rules. To apply a rule system to a working memory, we repeatedly apply any of the rules to the working memory until no rule is applicable. The final working memory is the result of applying the system. Application of a rule system is nondeterministic; by selecting different rules, or different instances of the same rule, we may obtain different results.

2.3 Explicit Halting

Our systems halt only when no rule is applicable. Some rule-based languages offer an explicit *halt* statement: if the special symbol $HALT$ appears on the right side of a rule that is applied, the system will halt at once, even if some rule is applicable. This is a convenience that does not increase the logical power of the language. Any system with the halt feature can be transformed into one that behaves the same way without the feature.

The transformed system includes the negated atom *not halt* (that is, *not halt*()) on the left side of each rule. If any rule contains the special symbol $HALT$ on its right side, it is transformed into a rule with the atom *halt* on its right side instead. If such a rule should fire, it adds the atom *halt* to working memory. Then no rule will be applicable, and the transformed system will halt, without invoking any special halt feature.

For example, the system

$$red(x) \longrightarrow blue(x)$$
$$yellow(x) \longrightarrow HALT$$

with the halt feature is transformed into the system

$$red(x),\ not\ halt \longrightarrow blue(x)$$
$$yellow(x),\ not\ halt \longrightarrow halt$$

without the halt feature. The two systems behave the same way.

2.4 Conflict Resolution Strategy

When several rules are applicable to the same working memory, the system invokes a conflict resolution strategy to choose a single rule to be applied. Conflict resolution strategies tend to be complex. The spirit of the rule-based methodology suggests that the correctness of the system should not depend on the details of the strategy. The choices of the strategy may influence the efficiency of the system or the understandability of its execution sequence, but the correctness of the final outcome should be independent of these choices. Consequently, we have developed an approach that will perform our validation tasks regardless of the conflict resolution strategy. If some aspects of the strategy turn out to be crucial to the correctness of the system, they may be expressed in an augmented system theory.

An exception is made in the case of the specificity aspect of the conflict resolution strategy, which does affect the correctness of the system. According to the specificity principle, a more specific rule is to be preferred to a more general one. This principle allows us to state a rule in its greatest generality, and then to add exceptions by introducing new

rules, without the need to qualify the original rule. For example, in the system

$$\begin{aligned}
&\textit{Rule } 1: \quad bird(x) \longrightarrow bird(x), fly(x)\\
&\textit{Rule } 2: \quad bird(x), penguin(x) \longrightarrow bird(x), penguin(x), negfly(x)\\
&\textit{Rule } 3: \quad bird(x), penguin(x), inairplane(x) \longrightarrow \\
&\qquad\qquad bird(x), penguin(x), inairplane(x), fly(x)
\end{aligned}$$

each rule is more specific than the previous one, which it qualifies. It might be erroneous to apply the earlier rule if the later rule were applicable. For example, the second rule should not be applied if the penguin is in an airplane, because then the third rule should supersede it. Thus the second rule has the implicit condition *not* $inairplane(x)$.

Because specificity has a bearing on correctness concerns, we do want it reflected in the system theory. We achieve this by transforming rules so that the implicit conditions imposed by the strategy are made explicit. For example, Rule 2 would be transformed to

$$\begin{aligned}
&\textit{Rule } 2': \quad bird(x), penguin(x),\ not\ inairplane(x) \longrightarrow \\
&\qquad\qquad bird(x), penguin(x), negfly(x)
\end{aligned}$$

Rule $2'$ cannot be applied if Rule 3 is applicable. Rule 1 above would be transformed into

$$\textit{Rule } 1': \quad bird(x),\ not\ penguin(x),\ not\ inairplane(x) \longrightarrow bird(x), fly(x)$$

Rule $1'$ cannot be applied if either Rule $2'$ or Rule 3 is applicable.

This transformation has no effect on the execution of the system, but we shall apply it so that the specificity aspect of the conflict resolution strategy will be reflected in our validation.

3 The System Theory

In this section, we describe how a given rule system is described in a corresponding system theory, so that questions about the system may be phrased as conjectures within the theory.

Consider a rule

$$P_1, \ldots, P_i,\ not\ Q_1, \ldots,\ not\ Q_j \longrightarrow R_1, \ldots, R_l$$

with variables $x_1, \ldots, x_k$.

We define a relation $applic(r, \langle t_1, \ldots, t_k \rangle, w)$, which holds if rule r is applicable to working memory w with variables $x_1, \ldots, x_k$ instantiated to ground terms $t_1, \ldots, t_k$, respectively, by the axiom

$$applic(r, \langle t_1, \ldots, t_k \rangle, w) \equiv \left[\begin{array}{c} P_1[t_1, \ldots, t_k] \in w \\ \wedge \\ \vdots \\ \wedge \\ P_i[t_1, \ldots, t_k] \in w \\ \wedge \\ Q_1[t_1, \ldots, t_k] \notin w \\ \wedge \\ \vdots \\ \wedge \\ Q_j[t_1, \ldots, t_k] \notin w \end{array} \right]$$

for all ground terms $t_1, \ldots, t_k$ and any working memory w. Here $\langle t_1, \ldots, t_k \rangle$ is a tuple of ground terms and r is a constant that names the rule. We write $P[t_1, \ldots, t_k]$ for the result of replacing each variable $x_1, \ldots, x_k$ in P with the corresponding term $t_1, \ldots, t_k$. In other words, the appropriate instances of the positive atoms must be present in working memory and the corresponding instances of the negated atoms must be absent. A separate such axiom is provided for each rule of the system.

For example, for the rule

$$Rule\ 1 : parent(x, y),\ not\ male(x) \longrightarrow parent(x, y), mother(x, y)$$

we provide the axiom

$$applic(rule1, \langle s, t \rangle, w) \equiv [parent(s, t) \in w \wedge male(s) \notin w].$$

Note that each predicate symbol in a rule is represented by a function symbol in the system theory. Thus, *parent* and *male* are function symbols.

We also define a function $apply(r, \langle t_1, \ldots, t_k \rangle, w)$, whose value is the result of applying the rule r to working memory w with variables $x_1, \ldots, x_k$ instantiated to ground terms $t_1, \ldots, t_k$, by the axiom

$$\begin{array}{lll} if & applic(r, \langle t_1, \ldots, t_k \rangle, w) & \\ then & apply(r, \langle t_1, \ldots, t_k \rangle, w) = w & - \quad P_1[t_1, \ldots, t_k] \\ & & \quad \vdots \\ & & - \quad P_i[t_1, \ldots, t_k] \\ & & + \quad R_1[t_1, \ldots, t_k] \\ & & \quad \vdots \\ & & + \quad R_l[t_1, \ldots, t_k]. \end{array}$$

Here $w - u$ and $w + u$ are the results of deleting the element u from the set w and adding u to w, respectively.

For example, for Rule 1, we provide the axiom

$$\begin{array}{lll} if & applic(rule1, \langle s, t \rangle, w) & \\ then & apply(rule1, \langle s, t \rangle, w) = w & - \quad parent(s, t) \\ & & + \quad parent(s, t) \\ & & + \quad mother(s, t). \end{array}$$

When we leave a variable free, we mean it to have implicit universal quantification. Also, we use our vocabulary to indicate the sorts of the objects involved. For example, the above axiom is to apply to any working memory w and all ground terms s and t.

Note that when a positive atom occurs on both the left and the right sides of the rule, the corresponding term is first deleted from and then added to the set w in the *apply* axiom. Because the term is certain to occur in the set, this will always leave the set the same. For example, in the *apply* axiom for Rule 1, the term $parent(s, t)$ is first deleted, then added. To simplify the axioms (and the corresponding proofs), these operations will be omitted. The *apply* axiom for Rule 1 will actually read

$$if\ applic(rule1, \langle s, t \rangle, w)\ then\ apply(rule1, \langle s, t \rangle, w) = w + mother(t).$$

The following axiom tells us that if a rule is applicable to a working memory, that rule must be one of the rules of the system:

$$\begin{aligned} &if \quad applic(r,t,w) \\ &then \quad r = rule1 \vee \cdots \vee r = rulen, \end{aligned}$$

where t is a tuple of ground terms.

More complex versions of this axiom may be substituted if one desires to express more subtle aspects of the conflict resolution strategy.

We shall say that the entire system is applicable to a working memory w, denoted by $appl(w)$, if some rule, with some instantiation, is applicable. This is expressed by the axiom

$$appl(w) \equiv (\exists r, t) applic(r,t,w).$$

A working memory w to which no rule is applicable, that is, $\neg appl(w)$, is called a *final* working memory.

This translation of rules into axioms depends on our formulation of the rule language; the translation mechanism for other languages will differ slightly.

3.1 Histories

A history is a description of a finite initial segment of a possible computation of the system. In the theory, a *history* is a finite tuple of pairs

$$\langle \langle r_1, t_1 \rangle, \ldots, \langle r_n, t_n \rangle \rangle.$$

Each r_i is the rule applied at the i^{th} stage of the computation. Each t_i is a tuple of ground terms indicating how the variables of the rule r_i are instantiated at the i^{th} stage.

A history $h = \langle \langle r_1, t_1 \rangle, \ldots, \langle r_n, t_n \rangle \rangle$ is *applicable* to a working memory w, denoted by $hist(h,w)$, if there is a finite sequence of working memories

$$w_1, w_2, \ldots, w_{n+1},$$

where $w = w_1$, such that

$$w_{i+1} = apply(r_i, t_i, w_i);$$

that is, each memory is obtained from the previous one by applying the corresponding rule from the history. This is expressed by the axioms

$$hist(\langle\rangle, w)$$
$$hist(\langle r, t\rangle \bullet h, w) \equiv applic(r, t, w) \wedge hist(h, apply(r, t, w))$$

for all working memories w, rules r, tuples of ground terms t, and histories h. Here $\langle r, t\rangle \bullet h$ is the history obtained by inserting the pair $\langle r, t\rangle$ at the beginning of the history h.

The result $sys(h, w)$ of applying an applicable history h to a given working memory w is expressed by the axioms

$$sys(\langle\rangle, w) = w$$
$$if\ hist(\langle r, t\rangle \bullet h, w)\ then\ sys(\langle r, t\rangle \bullet h, w) = sys(h, apply(r, t, w)).$$

The second axiom states that the result of applying a nonempty history is the same as that of applying its first rule and instantiation, and then applying the remainder of the history. The result is itself a working memory.

Note that a history describes an initial segment of a computation of a rule system, not necessarily a full computation; more rules may be applicable to the resulting working memory. We say that h is a terminating history starting from w, denoted by $ter(h, w)$, if h is applicable to w and results in a final working memory. This is expressed by the axioms

$$ter(\langle\rangle, w) \equiv \neg appl(w)$$
$$ter(\langle r, t\rangle \bullet h, w) \equiv applic(r, t, w) \wedge ter(h, apply(r, t, w)).$$

It can be established that

$$ter(h, w) \equiv hist(h, w) \wedge \neg appl(sys(h, w)).$$

3.2 Finite Sets and Unique Names

Because we use finite sets of expressions to represent working memory, it is necessary to incorporate the theory of finite sets into our system theory. In particular, we include axioms

that describe the set addition function $w + v$ and the set membership relation $u \in w$:

$$u \notin \{\}$$

$$u \in w + u$$

$$if\ u \in w\ then\ u \in w + v$$

$$if\ u \neq v\ then\ if\ u \in w + v\ then\ u \in w.$$

For the set deletion function $w - v$ we have

$$v \notin w - v$$

$$if\ u \in w - v\ then\ u \in w.$$

We include a general well-founded induction principle, described in the next subsection, which applies to finite sets and finite histories as well.

Properties of tuples, for reasoning about histories and about tuples of ground terms, are also included but will not be presented here. Theories of finite sets and tuples are described more fully in Manna and Waldinger [10].

Although it may seem pedantic, we must include axioms that tell us that distinct function symbols correspond to distinct predicate symbols in working memory. For example, $red(s)$ and $blue(t)$ cannot stand for the same atom. This is expressed by the axiom

$$red(s) \neq blue(t).$$

We must provide such an axiom for each pair of function symbols corresponding to predicate symbols in working memory.

We must also provide an axiom stating that if two terms are distinct, they cannot be made identical by applying any of the predicate symbols from the working memory; e.g., for the predicate symbol red, we have

$$if\ red(t_1) = red(t_2)\ then\ t_1 = t_2.$$

A similar axiom is provided for each function symbol corresponding to a predicate symbol from working memory.

3.3 Well-Founded Induction

Many proofs require use of an induction principle. We use well-founded induction (also called Noetherian induction). This principle has the following form.

To prove a sentence

$$(\forall w)P[w]$$

prove the *inductive step*

$$(\forall w)\left[\begin{array}{l} if\ (\forall w')\,[if\ w' \prec w\ then\ P[w']] \\ then\ P[w] \end{array}\right].$$

Here $\prec$ is a well-founded relation, that is, one that admits no infinite decreasing sequences

$$w_1 \succ w_2 \succ w_3 \succ \cdots$$

We provide definitions of many known well-founded relations. For example, the proper subset relation is known to be well-founded over the finite sets because there are no infinite sequences of finite sets

$$w_1 \supset w_2 \supset w_3 \supset \cdots$$

We also include as lemmas other properties of the proper subset relation, e.g.,

$$if\ u \in w\ then\ w - u \subset w.$$

Well-founded relations over the tuples are also provided.

4 Validation Conjectures

Many validation tasks may be phrased as conjectures within the system theory; if we can establish the validity of the conjecture in the theory, we have carried out the validation task.

For example, suppose we wish to determine whether, for the given rule system, big, red objects will be painted yellow. We may phrase this task as:

$$(\forall w, h, t) \left[\begin{array}{ll} if & red(t) \in w \wedge \\ & big(t) \in w \wedge \\ & ter(h, w) \\ then & yellow(t) \in sys(h, w) \end{array} \right].$$

If we can prove this sentence in the system theory, we have shown that the system satisfies the condition.

Note that the sentence does not establish termination of the system, but only that if a history does terminate, the condition will be satisfied. To show that the system does always terminate, it suffices to establish the following *termination condition*:

$$if\ applic(r, t, w)\ then\ apply(r, t, w) \prec w$$

for some well-founded relation $\prec$. In other words, we show that each applicable rule reduces the size of working memory with respect to some well-founded relation. Because well-founded relations do not admit infinite decreasing sequences, this means we must ultimately reach a working memory to which no rule is applicable, i.e., a final working memory.

In our examples, we require the user to provide a well-founded relation for the termination proof. For example, the user might define

$$w_1 \prec w_2 \equiv \left[\begin{array}{c} (\forall t)\,[if\ red(t) \in w_1\ then\ red(t) \in w_2] \\ \wedge \\ (\exists t')\,[red(t') \in w_2 \wedge red(t') \notin w_1] \end{array} \right].$$

Therefore, with respect to $\prec$, if one working memory is less than another, it has fewer red objects. This is well-founded because working memories are finite. (We could define a similar relationship in terms of the number of red objects in working memory, but this would require reasoning about nonnegative integers, as well as sets, in the proof.) A more ambitious effort, which we have not yet attempted, would require the system to discover the well-founded relation as part of the proof process.

If we fail to prove that all red, big objects will be painted yellow, we may attempt to establish the opposite, namely, that some red, big objects will not be painted yellow. This can be established by proving the negation of the original conjecture, i.e., that

$$(\exists w, h, t)\left[\begin{array}{l} red(t) \in w \wedge \\ big(t) \in w \wedge \\ ter(h, w) \wedge \\ yellow(t) \notin sys(h, w) \end{array}\right].$$

Proving this conjecture will establish the falseness of the original condition. If we restrict a proof to be sufficiently constructive, we may extract from the proof a description of the objects whose existence it establishes. From this proof, we may extract a description of a case in which the condition fails to hold. In particular, we obtain a description of an initial working memory, a terminating history, and an object such that the object is initially big and red, but executing the history will produce a final working memory in which the object is not painted yellow.

Suppose we cannot prove termination and we suspect that our system sometimes fails to terminate. To prove this, we must provide a description of a possible infinite execution history, in the form of three functions, r, t, and w, which compute the i^{th} rule, instantiation, and working memory, respectively. These functions must satisfy the property

$$\begin{aligned} &applic(r(i), t(i), w(i)) \wedge \\ &apply(r(i), t(i), w(i)) = w(i+1) \end{aligned}$$

for all integers $i \geq 1$. We have not yet experimented with proving nontermination.

If we want to establish that a particular rule, say Rule 1, can fire, we may prove the conjecture

$$(\exists w, h, t)[hist(h, w) \wedge applic(rule1, t, sys(h, w))].$$

If we restrict the proof to be sufficiently constructive, we may obtain descriptions of the initial working memory w, history h, and instantiation t, such that executing h in initial working memory w will produce a working memory in which Rule 1 is applicable, with

instantiation t. Proving the negation of the above conjecture will establish that Rule 1 is unreachable, i.e., cannot be executed under any circumstances.

The notion of consistency depends on an application domain. Our rule language does not include explicit negation, but it may include predicate symbols that are understood to be mutually inconsistent. We may expect, for example, that an object cannot be both red and blue, or that the predicate symbol *negp* is to be the complement of the predicate symbol p.

If we want to show that our system can never produce an object that is simultaneously red and blue, we may attempt to prove

$$(\forall w, h, t)\left[\begin{array}{ll} if & hist(h, w) \\ then & \neg\left[red(t) \in sys(h, w) \wedge blue(t) \in sys(h, w)\right] \end{array}\right].$$

The above conjecture implies that an object cannot be both red and blue at any stage of the computation. If we are only concerned with the final state of the computation, we may prove

$$(\forall w, h, t)\left[\begin{array}{ll} if & ter(h, w) \\ then & \neg\left[red(t) \in sys(h, w) \wedge blue(t) \in sys(h, w)\right] \end{array}\right].$$

If we can prove the negation of either of the above conjectures, we have established that the system can produce an inconsistency, in either an intermediate state or a final state, respectively. Restricting the proofs to be sufficiently constructive will enable us to extract a description of how the inconsistency can occur.

5 Example: The Billing Category System

To illustrate the formation of a system theory, we adapt an example from a standard expert-systems text (the customer-billing example from Brownston et al.[3]). The system is to assign each of a fixed, finite pool of customers to a billing category, either normal or priority, depending on the history of the customer. The rules of the system are as follows:

$$Rule\ 1: \quad good(x),\ not\ set(x) \longrightarrow good(x), set(x), priority(x).$$

That is, if the category of a good customer has not been set, assign the customer to the priority category.

$$Rule\ 2: \quad bad(x),\ not\ set(x) \longrightarrow bad(x), set(x), normal(x).$$

That is, if the category of a bad customer has not been set, assign the customer to the normal category.

$$Rule\ 3: bad(x),\ not\ set(x), long(x) \longrightarrow bad(x), set(x), long(x), priority(x).$$

That is, if the category of a bad but long-term customer has not been set, assign the customer to the priority category. Note that, by the specificity principle, Rule 3 is intended to supersede Rule 2 when both are applicable; that is, Rule 2 is not meant to apply to long-term customers.

5.1 The Rule Axioms

These rules are represented by the following axioms in the system theory.

$$if\ applic(r, t, w)\ then\ r = rule1 \vee r = rule2 \vee r = rule3.$$

In other words, the only rules that can be applicable to a given customer t are Rule 1, Rule 2, or Rule 3.

Axioms for Rule 1:

$$applic(rule1, t, w) \equiv good(t) \in w \wedge set(t) \notin w$$

Note that because the rules have only one variable, we simplify the axioms by using t, rather than $\langle t \rangle$, throughout.

$$if\ applic(rule1, t, w)\ then\ apply(rule1, t, w) = w + set(t) + priority(t)$$

Because the atom $good(x)$ occurs on both sides of Rule 1, the corresponding term $good(t)$ is neither deleted nor added by the axiom.

Axioms for Rule 2:

$$applic(rule2, t, w) \equiv bad(t) \in w \ \wedge long(t) \notin w \ \wedge set(t) \notin w$$

Note that we include in the applicability axiom for Rule 2 the condition, implied by the specificity principle, that the rule should not be applied to long-term customers.

$$if\ applic(rule2, t, w)\ then\ apply(rule2, t, w) = w + set(t) + normal(t)$$

Axioms for Rule 3:

$$applic(rule3, t, w) \equiv bad(t) \in w \wedge long(t) \in w \wedge set(t) \notin w$$

$$if\ applic(rule3, t, w)\ then\ apply(rule3, t, w) = w + set(t) + priority(t)$$

The other axioms of the system theory are the same from one system to the next.

5.2 Conjectures: A Scenario

Suppose we wish to determine whether a good customer will always be placed in the priority category. Then we may conjecture

$$(\forall w, h, t)\left[\ if\ ter(h, w)\ then \quad \left[\begin{array}{ll} if & good(t) \in w \\ then & priority(t) \in sys(h, w) \end{array}\right]\right].$$

In fact, we cannot prove the above conjecture. We can, however, prove its negation

$$(\exists w, h, t)\left[\begin{array}{l} ter(h, w) \wedge \\ good(t) \in w \wedge \\ priority(t) \notin sys(h, w) \end{array}\right].$$

If we restrict the proof to be constructive, we may extract a description of the initial working memory,

$$w : \{good(t), set(t)\}$$

and the history

$$h : \langle\rangle,$$

the empty history. (The variable t is not instantiated during the proof, so it can be replaced by any ground term that denotes a customer.) In other words, no rule is applicable to the initial working memory $\{good(t), set(t)\}$, because customer t has been marked as if his billing category has already been set. Therefore, the final working memory $sys(h, w)$ is the same as the initial working memory w, and $priority(t) \notin w$.

We attempt to refine our conjecture accordingly. We speculate that if a good customer has no set billing category, he will ultimately be placed in the priority category. We attempt to prove

$$(\forall w, h, t)\left[\begin{array}{ll} if & ter(h, w) \\ then & if\ good(t) \in w \wedge set(t) \notin w \\ & then\ priority(t) \in sys(h, w) \end{array}\right].$$

Again we fail to prove this, but succeed in proving its negation. If the proof is restricted to be constructive, we may extract the (inconsistent) initial working memory

$$w : \{bad(t), good(t)\}$$

and the one-element history

$$h : \langle\langle rule2, t\rangle\rangle.$$

This example has again defied our expectations. Because customer t is bad as well as good (and not a long-term customer), Rule 2 can be applied, producing the final working memory

$$sys(h, w) : \{bad(t), good(t), set(t), normal(t)\}.$$

In other words, good customer t has not been put into the priority category.

This scenario illustrates the pitfalls we may face in formulating a specification for a rule-based system (or any system). It also illustrates how a proof system may help us break through some preconceptions. With further experimentation, we may attempt to formulate and prove a full specification for a rule system, which characterizes its intended behavior.

For the billing category system, we propose the following conjecture:

$$if\ (\forall t)\left[\begin{array}{l}[good(t) \in w \underline{\vee} bad(t) \in w] \wedge \\ set(t) \notin w \wedge \\ priority(t) \notin w \wedge \\ normal(t) \notin w\end{array}\right]$$

$$then\ (\forall h)\left[\begin{array}{ll} if & ter(h,w) \\ then & [priority(t) \in sys(h,w) \equiv good(t) \in w \vee long(t) \in w] \wedge \\ & [normal(t) \in sys(h,w) \equiv bad(t) \in w \wedge long(t) \notin w] \wedge \\ & set(t) \in sys(h,w) \wedge \\ & [good(t) \in sys(h,w) \equiv good(t) \in w] \wedge \\ & [bad(t) \in sys(h,w) \equiv bad(t) \in w] \wedge \\ & [long(t) \in sys(h,w) \equiv long(t) \in w]\end{array}\right]$$

for all customers t and working memories w. In other words, we suppose that initially each customer is either good or bad, but not both ($\underline{\vee}$ is exclusive *or*). Initially no customer has had his billing category set, and no customer is in either normal or priority category. Then for each terminating history, we require that the customers in the final priority category be those that are initially good customers or long-term customers. Customers in the final normal category must be those that are initially bad customers and not long-term customers. Every customer must have his final billing category set. Furthermore, we may expect that customers in the final good-customer, bad-customer and long-term customer categories are the same as those that were in those categories initially.

The above conjecture relates the initial and final working memories. For the proof to succeed, we must generalize the theorem to relate the intermediate and final working memories. The generalized conjecture implies the above conjecture as a special case. We have not yet proved the above conjecture.

Termination of the system must be proved separately, by establishing the usual termi-

nation condition

$$\left[\begin{array}{ll} if & applic(r,t,w) \\ then & apply(r,t,w) \prec w \end{array}\right]$$

for some well-founded relation $\prec$. In this case, the well-founded relation $\prec$ may be defined by

$$w_1 \prec w_2 \equiv \left[\begin{array}{c} (\forall t)\,[if\ set(t) \in w_2\ then\ set(t) \in w_1] \\ \wedge \\ (\exists t')\,[set(t') \in w_1 \wedge set(t') \notin w_2] \end{array}\right].$$

In other words, with respect to $\prec$, one working memory is less than another if it has more customers with set billing category. This is well-founded because there are only a finite number of customers.

6 The SNARK System

To prove validation conjectures, a theorem prover requires an unusual combination of features. In particular, it must

- Prove sentences in full first-order logic.
- Deal expeditiously with equality and ordering relations.
- Prove theorems by mathematical induction.
- Handle finite sets and tuples.
- Restrict proofs to be sufficiently constructive to allow information extraction when necessary.
- Prove simple theorems without human assistance.

While some existing theorem provers excel in certain of these areas, they are typically deficient in others. The Argonne system [12], for example, is proficient at full first-order logic with equality, but has no facilities for proof by induction. The Boyer-Moore theorem

prover [2] specializes in proof by induction, but does not allow full first-order quantification. The Nuprl system [6] is certainly expressive enough—it allows full quantification and proof by induction—but is not geared to finding proofs automatically. Furthermore, it relies entirely on a constructive logic that may prove cumbersome when no information needs to be extracted from the proof.

For these reasons, we have been developing a new theorem prover, SNARK, for application in software engineering and artificial intelligence. SNARK is especially appropriate for the validation of rule-based systems.

SNARK operates fully automatically and uses an agenda to order inference operations. Similarly to the Argonne system, SNARK attempts to compute the deductive closure of a set of formulas. The user selects the inference operations and starting formulas to be used. Agenda elements are formulas in the set of support to be operated upon by all selected inference rules, and are ordered by symbol count.

The most important inference operations available in the current system are binary resolution and paramodulation. These rules allow SNARK to deal with predicate logic with equality, which underlies the system theory for rule-based systems. We have used extended versions of these inference rules that are applicable to nonclausal formulas as well as clauses. Hyperresolution can be simulated by control restrictions on the use of binary resolvents. Both clausal and nonclausal subsumption are available to eliminate redundant formulas.

Formulas can be simplified by user-given or derived equalities or equivalences. Innermost or outermost simplification strategies can be specified. Derived equalities can be oriented automatically by Knuth-Bendix or recursive decomposition simplification orderings. Truth-functional simplification is accomplished by rewriting rules, which makes it easy to add new connectives and their simplification rules.

SNARK can use either nonclausal formulas or the more restrictive clauses. If clauses are to be used, SNARK can automatically translate more general formulas to clauses. Even if clauses are primarily used, translation of formulas to clauses is not required to be done only at the beginning of the proof. Rewrites can specify that an atomic subformula of a

formula be rewritten to a formula; the result of rewriting may be a nonclausal formula that is later simplified to clause form. For example, the rewrite

$$u \in w + v \equiv u = v \vee u \in w$$

would result in the clause $a \notin s + x \vee C$ being rewritten to $\neg(a = x \vee a \in s) \vee C$, which could be replaced by two clauses.

Efficient formula- and term-indexing methods—a choice of path indexing or discrimination-tree indexing—are used to efficiently retrieve the relevant formulas or terms for inference, subsumption, and simplification operations. Efficient indexing is essential for solving difficult problems that require derivation of a large number of results.

SNARK uses sorted logic to efficiently represent the information that certain classes of objects being manipulated are disjoint. In the examples of this paper, several sorts are used: rules, working memories, working memory elements, customers, history lists, and the pairs that are history list members.

SNARK supports the use of special unification (and subsumption and equality) algorithms. Associative-commutative subsumption is widely used for truth-functional simplification, and commutative matching is used to efficiently implement symmetry of the equality relation.

Although at an early stage of development, SNARK has already been useful in proving properties of rule-based systems, including those indicated in the scenario (Section 5.2).

7 Summary and Plans

Our work suggests that deductive methods are appropriate to support testing and other validation tasks, besides verification, for rule-based systems. Preliminary results in applying the new deduction system SNARK to sample rule-based systems have been promising. By restricting the system to be sufficiently constructive, we have been able to extract information other than simple yes/no answers from proofs. We have found that a method for formulating specifications by proposing a series of conjectures is appropriate to rule-

based systems.

We intend to develop the system theory to apply to more realistic rule languages and to extend SNARK to more complex rule systems and more sophisticated properties and conjectures. The extension will be carried out by introducing inference rules targeted to the application (e.g., rules for reasoning about sets and ordering relations), strategic deduction (e.g., special treatment for inductive proofs), interactive controls, and parallel search for proofs.

References

[1] L. Becker, P. Green, R.J. Duckworth, J. Bhatnagar, and A. Pease. Evidence flow graphs for VV&T. In *Preliminary Proceedings IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems*, 1989. Detroit, MI.

[2] R.S. Boyer and J S. Moore. *A Computational Logic.* Academic Press, New York, 1979.

[3] L. Brownston, R. Farrell, E. Kant, and N. Martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming.* Addison-Wesley, Reading, MA, 1985.

[4] C. Chang and R. Stachowitz. Testing expert systems. In *Proceedings of the Space Operations Automation and Robotics (SOAR-88) Workshop*, 1988. Dayton, OH.

[5] C. Chang, R. Stachowitz, and J.B. Combs. Testing integrated knowledge-based systems. In *IEEE International Workshop on Tools for AI*, 1989. Fairfax, Virginia.

[6] R. Constable et al. *Implementing Mathematics with the Nuprl Proof Development System.* Prentice Hall, Englewood Cliffs, NJ, 1986.

[7] C.L. Forgy. OPS5 user's manual. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, July 1981.

[8] J. Kiper. Structural testing of rule-based expert systems. In *Preliminary Proceedings IJCAI-89 Workshop on Verification, Validation and Testing of Knowledge-Based Systems*, 1989. Detroit, MI.

[9] Z. Manna and R. Waldinger. A deductive approach to program synthesis. *ACM Transactions on Programming Languages and Systems*, 2:90–121, 1980.

[10] Z. Manna and R. Waldinger. *The Logical Basis for Computer Programming*, volume 1: Deductive Reasoning. Addison-Wesley, 1985.

[11] R. Stachowitz, J. Combs, and C. Chang. Validation of knowledge-based systems. In *Proceedings of the Second AIAA/NASA/USAF Symposium on Automation, Robotics and Advanced Computing for the National Space Program*, 1987. Arlington, VA.

[12] L. Wos, R. Overbeek, E. Lusk, and J. Boyle. *Automated Reasoning.* Prentice Hall, Englewood Cliffs, NJ, 1984.