NASA Contractor Report 181827

FORMAL VERIFICATION OF AI SOFTWARE

(NASA-CR-181827) FORMAL VERIFICATION OF AI SOFTWARE Final Report (SRI International Corp.) 156 p
CSCL 09B
N89-24811
Unclas
G3/61 0217233

John Rushby, R. Alan Whitehurst
SRI International
Computer Science Laboratory
333 Ravenswood Avenue
Menlo Park, CA 94025

NASA Contract 18226 (Task 5)
February 1989

National Aeronautics and
Space Administration

Langley Research Center
Hampton, Virginia 23665-5225

Abstract

The application of formal verification techniques to AI software, particularly expert systems, is investigated. Constraint satisfaction and model inversion are identified as two formal specification paradigms for different classes of expert systems. A formal definition of consistency is developed, and the notion of approximate semantics is introduced. Examples are given of how these ideas can be applied in both declarative and imperative forms.

Contents

Chapter 1

Introduction

Almost all computer software harbors faults; those faults can lead to failure, and failure can have serious consequences. Stringent measures are (or should be) taken to ensure that software faults in critical systems will not lead to failures that could endanger life or national security, cause harm to the environment, or have other undesired consequences disproportionate to the benefits provided by normal operation. The goal in software development for critical systems should be to develop *dependable* software—software for which justifiable reliance can be placed on the quality of service it delivers.

There are two main approaches for achieving dependability: fault *exclusion*, which aims to prevent the occurrence of faults, and fault *tolerance*, which seeks to detect and recover from the manifestations of faults before they can lead to failure.[1] Fault exclusion techniques include systematic design methodologies intended to prevent faults from being introduced in the first place, combined with thorough testing and examination procedures intended to identify and eliminate those faults that slip through the first stage. Fault tolerance is based on redundancy and run-time checking; because all software faults are design faults, the redundant components cannot be simple replicates of the original, but must be "diverse designs" [6].

Both fault exclusion and fault tolerance techniques can benefit from the observation that dependability is not the same as reliability: it may not be necessary to exclude or tolerate all faults, but only those with unacceptable consequences [35]. That is to say, dependability can be regarded as reliability with respect to unacceptable failures.

[1]What we have called fault exclusion is generally termed fault *avoidance*; we prefer the former term since it has a more active and positive connotation.

Dependability requirements for life-critical systems (such as fly-by-wire digital control for passenger aircraft) are onerous. For example, the FAA requires failures that could endanger a civil aircraft to be "extremely improbable" and "not expected to occur within the total life span of the whole fleet of the model"—the FAA has suggested that this should be interpreted as a probability of less than 10^{-9} in a 10 hour flight, or 10^{-10} per hour of flight. Substantiating reliabilities such as these through testing is infeasible. Furthermore, experimental and theoretical results cast doubt on the failure-independence assumption that underlies the belief that software fault tolerance techniques can deliver the required dependability [12, 21, 32, 37].

1.1 Formal Verification

Formal verification is a technique that can, *in principle*, guarantee the absence of faults. Formal verification demonstrates consistency between two different descriptions of a program. Often, one description is the program code itself and the other is its specification, although the method can be applied equally well to two specifications at different levels of detail. Formal verification treats specifications and programs as formal, mathematical texts, and the demonstration of consistency between them takes the form of a mathematical proof. Formal verification steps can be "stacked" on top of each other, so that a hierarchy of increasingly detailed specifications carries the demonstration of consistency all the way from a highly abstract "requirements" statement down to executable code. This guarantees the "correctness" of the executable code provided:

1. The notion of consistency that is employed is appropriate for the intended interpretation.

2. The requirements statement accurately captures the real (dependability) requirements on the system.

3. The meaning ascribed to the program code during its verification accurately reflects its behavior during execution.

4. The proofs are performed without error.

Consistency is usually equated with logical implication: it is proved that the properties of a lower level imply those of the upper one. In other words, it is proven that the lower levels do at least that which is specified in the upper levels. This excludes faults of omission, but not necessarily those of commission.

A requirements statement is just a set of marks on paper; there can be no conclusive demonstration that it is an accurate formulation of the dependability objectives for the system. The best that can be done is to employ notational formalisms that are expressive, precise and perspicuous, and to subject the requirements statement to considerable scrutiny. This may take the form of testing, either of the requirements specification itself (if the specification language is executable), or of a simulation or "rapid prototype," or of analytic scrutiny. The latter can be "formal testing" that involves proving conjectures ("if the requirements are right, then I ought to be able to prove that the following must be true"), checks for consistency and completeness, and informal review.

The problem of ensuring that the meaning ascribed to a program during its verification (i.e., its formal semantics) accurately reflects its behavior during execution (i.e., its operational semantics) is a difficult one. One approach is to carry the formal verification down through the specification and implementation of the programming language itself, its support software, and the hardware on which it runs until the bottom-level assumptions are descriptions of elementary logic gates. There has been substantial progress recently on each of these levels [8, 14, 15, 30, 39, 40], and some pioneering demonstrations that they can be "stacked." For the near-term future, however, it is reasonable to assume that the correctness of programming language implementations will be subject only to informal arguments.

It is in order to avoid the fourth class of error, errors made during logical reasoning, that truly formal, and mechanically assisted, verification is generally recommended. The theorems that need to be proven during verification are numerous, massively detailed, and of little intrinsic interest. Experience has shown that semiformal proofs, as employed in mathematics journals, are unreliable when applied to theorems such as these. Consequently, the use of mechanical theorem proving (or proof checking) is attractive.

The inherent limitations to formal verification identified above are likely to be exacerbated in practice by compromises (typically incomplete application, or verification of only limited properties) imposed by technical or resource limitations. Nonetheless, formal verification should be considered an important component in the demonstration of dependability for critical systems, and an even more important component in the process of constructing such systems. When integrated with the design and development process (rather than being an after-the-fact analysis), formal verification can contribute significantly to the development of dependable systems by encouraging systematic and conscious thought, and a preference for simple and perspicuous design and implementation.

1.2 AI Software and Expert Systems

This report is concerned with the application of formal verification techniques to AI software in general and expert systems in particular. By *AI software* we mean software that uses techniques from the field of Artificial Intelligence; Genesereth and Nilsson [24] give an excellent modern introduction to such techniques. Expert systems can be characterized as AI software that use highly problem-specific information (as opposed to general laws) in order to achieve performance comparable to human specialists in limited problem domains. Expert systems employ "surface" rather than "deep" formulations of knowledge; they codify empirical associations, special cases, and exceptions. These are often expressed in the form of "if–then" production rules, giving rise to the "rule-based expert systems" that are becoming widespread. A good survey of expert systems is given by Buchanan and Smith [11]; Harmon and King [28] provide a more elementary overview.

On the face of it, expert systems seem unlikely candidates for formal verification. In the first place, their requirements statements are notoriously vague: "build a system to do what Bill does" is not a gross parody of the specification for an expert system. Even when requirements are specific, they may not be amenable to formal analysis. For example, "build a system for loan approval that delivers no more than 4% bad loans by value" is not a requirement that can be verified analytically.

Second, the developmental process used for expert systems is largely experimental in nature; the knowledge base is refined "in response to errors exhibited in solving test cases" [11, page 42]. The hierarchy of specification and design that is the foundation of systematic engineering for conventional software, and of its formal verification, is absent for most expert systems. As Buchanan and Smith observe [11, page 47]:

> Often when one begins designing an expert system, neither the problem nor the knowledge required to solve it is precisely specified. Initial descriptions of the problem are oversimplified, so the complexity becomes known only as early versions of the system solve simple versions of the problem. Expert systems are said to approach competence incrementally.

Parnas puts it less sympathetically [43]:

> The rules that one obtains by studying people turn out to be inconsistent, incomplete, and inaccurate. Heuristic programs are developed by a trial-and-error process in which a new rule is added whenever one finds a case that is not handled by the old rules. This approach usually yields a program whose behavior is poorly understood and hard to predict.

Third, conventional verification is concerned with algorithmic software. The premise of conventional development methodology and of verification is that the goal of the exercise is to produce an efficient implementation of a systematic method (an algorithm) for satisfying the requirements. Expert systems do not satisfy this premise; solutions are found by search rather than algorithmically. The knowledge base is not a conventional program, but a collection of heuristic information used to guide the search.

Finally, the languages in which expert systems and their knowledge bases are encoded have not been developed with formal analysis in mind.

Chapter 2

Requirements and Specifications

The absence of precise requirements and specification documents for much AI software reflects the genuine difficulty in stating à priori the expectations and requirements for a system whose capabilities will evolve through a development process that is partly experimental in nature. However, if formal verification is to be applied to AI software, precise requirements and specifications are a necessity. We have proposed a possible solution to this dilemma in a previous report [48].

The basis of our proposal is first to separate the "inherently AI" aspects of AI software from the more conventional aspects—aspects that should be amenable to conventional software engineering and quality assurance. To this end, we distinguish two sets of requirements and specifications for AI software: the *competency* requirements and the *service* requirements.

Competency requirements pertain to those dimensions of the overall requirements that concern "knowledge" or appeal to comparison with human skills. We accept that such requirements may of necessity be vague and incomplete. Service requirements cover all other requirements and should be amenable to statements no less rigorous and formal than those for conventional software. Service requirements should include descriptions of the input and output formats expected, the processing rate, the explanation facilities required, and so on. Service requirements and their decomposition through levels of specification should be traceable, verifiable, and testable just like those of conventional software.

The present investigation is concerned with verification of competency requirements. Because we have already conceded that these may be vague and incomplete, it may seem contradictory to talk about their formal verification. However, it is not necessary to verify full functional correctness in order to accomplish something useful. For many critical applications, it is not necessary that the system should not fail, only that it should not fail "badly." Accordingly, we further subdivide competency requirements into "desired" and "minimum" requirements. The desired competency requirement will often be defined relative to human expertise and describes how *well* the system is expected to perform. The minimum competency requirement, on the other hand, defines how *badly* it is allowed to perform.

Whereas desired competency requirements may be hard to state explicitly, there is some hope that minimum competency requirements may sometimes be capable of precise definition. For example, one of the expert systems developed at SRI, by the Information Sciences and Technology Division, is the Automated Air Load Planning System—AALPS [5]. It is used to produce schedules and layouts for loading army divisions and their equipment (tanks, helicopters, etc.) onto air transports. One of the things to be considered is the unloading of the aircraft, especially if this is to be performed in flight, using parachute drops. As the heavy equipment is first moved around inside the aircraft prior to being dropped, and then actually dropped, the aerodynamic stability of the aircraft must remain undisturbed. Specifying the desired competency of AALPS is obviously difficult—we want a near-optimum loading plan, but an *optimum* loading plan is hard to define, and it is even harder to determine whether a given plan is optimal. But the minimum competency requirement can be given quite a sharp statement: the aerodynamic trim of the aircraft must remain within certain limits at all times during all stages of unloading in flight.

We believe that minimum competency requirements may be able to capture certain desired "safety" properties for AI software and that such requirements can often be cast as formal requirements specifications. In the following sections we will look more closely at the formal definition of minimum competency requirements for different classes of AI software.

2.1 Safety and Liveness Properties

All extensional properties of programs are conjunctions of *safety* and *liveness* properties [4]. A liveness property asserts that something good must eventually happen; termination is the archetypal liveness property. A safety property asserts that some bad thing does not occur during execution. Partial correctness is the archetypal safety property: the "bad thing" that must not occur is termination with the wrong answer. The term "safety property" is a technical one; such properties need have nothing to do with safety as it is commonly interpreted—though we believe that many safety properties in the conventional sense of the term can be captured formally by properties satisfying its technical sense.

We can formalize these notions as follows. Let I and O be the input and output domains of the program $\mathcal{P}$. The input/output behavior of $\mathcal{P}$ can then be represented by a relation on $I \times O$: $\mathcal{P}(i, o)$ will be true if execution of $\mathcal{P}$ can terminate with output $o \in O$ when given input $i \in I$. [1] $\mathcal{P}$ may not be intended to work for all possible inputs, so we let $\mathcal{V}$ be a predicate on I that identifies "valid" inputs. If we now let $\mathcal{C}$ be a predicate on O that identifies outputs satisfying the desired safety property, then the specification that $\mathcal{P}$ satisfies $\mathcal{C}$ for inputs satisfying $\mathcal{V}$ is simply:[2]

$$\forall i \in I, o \in O, \mathcal{V}(i) \wedge \mathcal{P}(i, o) \supset \mathcal{C}(o)$$

2.2 Classification of Expert System Applications

Applications of expert systems can be divided into those concerned with problems of ***interpretation*** (analysis) and those concerned with problems of ***construction*** (synthesis). Buchanan and Smith [11, pages 28 and 29] tabulate some of the major examples of each kind. Among the most common of the analytic systems are those concerned with equipment monitoring, fault diagnosis, and the screening and interpretation of data. Expert systems for synthesis include those concerned with planning, scheduling, loading (AALPS is in this category), configuration, and design. We believe that another way of looking at this dichotomy between analytic and synthetic expert systems sheds more light on their possible requirements specifications. Our alternative dichotomy is that between ***constraint satisfaction*** and ***model inversion***.

[1]We use relations rather than functions (i.e., we do not write $\mathcal{P}(i) = o$) to allow the possibility of nondeterministic behavior.

[2]We use $\neg$, $\wedge$, $\vee$, and $\supset$ to denote negation, conjunction ("and"), disjunction ("or"), and implication, respectively.

2.2.1 Constraint Satisfaction Problems

A constraint is a predicate over the output variables of a program. The constraint satisfaction problem is to find an assignment of values to those variables that satisfies the constraint. The important point about constraint satisfaction problems is that we can check whether purported solutions do indeed satisfy the constraint. ***Optimization problems*** are a variation on constraint satisfaction. We want a solution that not only satisfies the constraint, but also maximizes (or minimizes) some function. It may not be feasible to check whether a purported solution to an optimization problem is indeed optimal, but it is at least possible to check whether it satisfies the constraint. Often, truly optimal solutions are not necessary; in these cases bounds on acceptable values may be characterized as part of the constraint to be satisfied (e.g., "find a loading plan that uses no more than three aircraft"). We will call these ***bounded*** constraint satisfaction problems. We contend that minimum competency requirements for synthetic expert systems can often be formulated as bounded constraint satisfaction problems—and specifications for such problems can be formalized as safety properties. For example, let $\mathcal{P}$ be the AALPS program, $\mathcal{V}$ a definition of "valid AALPS input" and $\mathcal{C}$ a formalization of the constraint "the loading plan places the center of gravity of each aircraft within acceptable limits." Then the simple safety property

$$\forall i \in I, o \in O, \mathcal{V}(i) \wedge \mathcal{P}(i, o) \supset \mathcal{C}(o)$$

expresses the requirement that if AALPS is given valid input, and if it terminates, then any loading plan produced must place the center of gravity of each aircraft within acceptable limits.

2.2.2 Model Inversion Problems

A model is a "a simplified representation or description of a system or complex entity, especially one designed to facilitate calculations or predictions" (Collins English Dictionary).

Computer models can range from systems of equations capable of yielding highly accurate and precise numerical predictions (e.g., for orbital mechanics) to descriptions that provide only qualitative information. To qualify as a model, a system representation or description should have some explanatory or causal value: it should in some sense describe how the system being modeled actually works. A model has much in common with a physical theory, and need not be correct or complete in order to be useful—so long as its predictions are adequate for their purpose.

Models have parameters—quantities whose values differ from one instance of the model to another. In the case of a model for an automobile electrical system, for example, the parameters might include the voltage at the battery, and whether or not the starter motor is broken. Usually, certain parameters can be identified as inputs (i.e., their values are determined by factors external to the system being modeled), and others as outputs. Inputs can be regarded as "causes" and outputs as "effects"; the model is then a mechanism for reasoning from causes to effects.

Computer models are often used in this predictive manner: given values for the input parameters, the model is used to predict values for the output parameters. Such predictions are usually determined by direct calculation and are performed by conventional software.

The inverse problem—that of finding causes that explain observed effects—is what we call the problem of "model inversion." Whereas problems of prediction can usually be solved by direct calculation, problems of model inversion must often be solved by search—different values for the input parameters are tried in turn until a configuration is found that leads to a prediction matching the observed behavior. Of course, such explicit searching will often be unacceptably slow, so heuristic techniques may be employed to perform the search in an "intelligent" manner. Such heuristic techniques can be used in two ways: either they can be used *in conjunction* with the explicit model, or they can be used *instead* of it. In the first case, heuristics can be used to suggest values for the input parameters, the output values can be computed, and their divergence from those actually observed can be used by the heuristics to adjust the input parameters. (Of course, more sophisticated arrangements may sometimes be possible, in which the heuristics are more intimately connected to the structure of the model.) In the second case, heuristics are used to suggest values for the input values and these are accepted without further checks.

We contend that many, if not most, analytic expert systems can be characterized as heuristic model inversion procedures of this second kind—except that the model that they invert is never constructed explicitly. We further contend that the way to produce precise specifications for such expert systems is to construct the missing model.

We can then say that an analytic expert system is consistent with its model if the former is an inverse of the latter—that is, if outputs of the expert system, when supplied as inputs to the model, generate predicted effects that match those observed.

This can be expressed formally as follows. Let C and E be the domains of "causes" and "effects," respectively. Let $\mathcal{M}$, a predicate on $C \times E$, be an explicit model (from causes to effects), and let $\mathcal{P}$, a predicate on $E \times C$, be an expert system (that reasons from effects to causes). Then the expert system $\mathcal{P}$ is consistent with the model $\mathcal{M}$ if

$$\forall c \in C, e \in E : \mathcal{P}(e, c) \supset \mathcal{M}(c, e)$$

It is easy to construct variations on this definition. For example, if $\sqsupseteq$ is a relation on effects, with the interpretation that $e' \sqsupseteq e$ means the predicted effects e' do not contradict observed effects e, then we can define a weaker notion of consistency as follows:

$$\forall c \in C, e \in E : \mathcal{P}(e, c) \supset \exists e' : \mathcal{M}(c, e') \wedge e' \sqsupseteq e$$

We can also say that $\mathcal{P}$ is ***complete*** with respect to $\mathcal{M}$ if

$$\forall c \in C, e \in E : \mathcal{M}(c, e) \supset \mathcal{P}(e, c)$$

Consider an expert system to diagnose faults in the electrical subsystem of an automobile, for example. This will typically be a rule-based system containing rules such as

```
if  the starter motor turns
    and there is no spark at plugs
then
    there is a problem in the HT Coil
```

An explicit model, on the other hand, would link the components of the subsystem in cause-effect relationships in some suitable degree of detail. A highly detailed model might account for actual voltage, current, and resistance values, whereas a qualitative model might simply deal with the signs (i.e., positive or negative) of various quantities and their derivatives. For some diagnostic problems, a simple model of fault propagation is all that is needed (i.e., a component fault is assumed to affect all sensors "downstream" from that component). A simple model for the automobile electrical system might note that the battery is "upstream" from both the starter motor and the HT coil, and the HT coil is upstream from the spark plugs. (This sort of model is sometimes called a "causal net"). The expert system rule given above is consistent with this model. The rule

```
if the starter motor does not turn
then
   there is a problem in the battery
```

is weakly consistent with the model if we define $e' \sqsupseteq e$ to mean that the observations predicted abnormal in e' are a superset of those observed abnormal in e.

2.2.2.1 Abduction

There are many different representations for the many different types of models—differential equations, directed graphs, and predicate calculus, for example. In the case of predicate calculus representations, model-inversion can be described by a form of logical inference known as "abduction." Given the premise $(\forall x : P(x) \supset Q(x))$, the familiar process of *deduction* allows us to reason from the observation $P(a)$ to the conclusion $Q(a)$. *Abduction*, on the other hand, allows us to reason from the observation $Q(a)$ to the possible "explanation" $P(a)$. With a complex model, there can be many abductive explanations for a given observation and various criteria have been devised for selecting a preferred explanation [52].

Abduction has been proposed as a technique for model-based diagnosis [17, 41] and methods for mechanizing abductive inference have been developed [16, 45, 52]. We believe that abductive techniques can also serve to specify minimum competency requirements for certain model inversion problems.

2.3 Formal Requirements Specifications

We have introduced two classes of problems and shown how formal specifications can be given for each. We contended that constraint satisfaction could provide formal specifications for expert systems that perform synthesis while model inversion serves that purpose for analytic systems. Of course, this dichotomy is a little simplistic: some aspects of analytic expert systems may be best specified in terms of constraint satisfaction, while synthetic systems may have some model inversion attributes. And some requirements may lend themselves to neither constraint satisfaction nor model inversion formulations. (For example, planning probably requires the introduction of a third problem class.) Nonetheless, we believe that our two classes of problem formulation are adequate to specify the requirements of many expert system applications.

The question remains, do our formulations make "good" requirements statements? One of the principal characteristics desired of a requirements formulation is that it should be comprehensible: those responsible for the procurement and quality assurance of a system must be able to scrutinize its requirement statement and convince themselves that they truly understand its implications and limitations. We believe that the statements that any results produced by an expert system are to be consistent with "this set of constraints" or "that model" do satisfy this desideratum. A model that provides some plausible and systematic causal connection between inputs and outputs is far more comprehensible and defensible—and surely represents more "knowledge"—than a set of associations encoded as rules.

If constraints and models can serve so well as the requirements specifications for expert systems, why do we build rule-based expert systems at all? Why not build constraint satisfaction and model inversion procedures? We are convinced that the development of such procedures is the correct direction to pursue if reliable expert-like systems are to be produced. There are, in fact, substantial bodies of work on constraint satisfaction (the book by Leler [34] is a good introduction) and qualitative modeling (see the book edited by Bobrow [10]). Diagnosis of physical systems is a field in which direct model-based procedures have been studied extensively (see Reiter's work [46], for example, and the survey by Davis and Hamscher [18]).

In the short term, however, it is likely that expert systems will be able to outperform systems based on direct constraint satisfaction or model inversion. For one thing, many problems of interest lack good models (strategic warfare and medical diagnosis, for example) and a good ad hoc expert system may be able to do better than a simplistic model or constraint-based system. However, we believe that even simplistic models and constraints serve to establish useful minimum competency requirements for such expert systems.

Chapter 3

Languages

The languages in which expert systems are commonly written, whether "rule-based," "frame-based," or "object-oriented" have not been developed with formal analysis in mind. In particular, they lack formal semantics, and generally lack the regularity and conceptual simplicity that would make the construction of formal semantics feasible.

For concreteness, we will use forward-chaining production rule languages as our paradigm for languages used in the implementation of expert systems. These languages (for example, OPS5 [23]) are appropriate when all input data are known in advance, or are easy to collect, and when there are relatively few hypotheses to be explored. Such are likely to be the case in many near-term space and aviation applications of expert systems, where input data are usually provided by sensors.

3.1 Declarative Semantics

There is some similarity between rule-based notations and logic programming languages such as Prolog [13] and OBJ [26]. A pure logic programming language (e.g., OBJ) has a declarative semantics: a program can be interpreted as a theory in the logic, and computation is equivalent to deduction in that logic. The behavior of programs in execution can be predicted by proving theorems within the logic. Impure logic programming languages (e.g., Prolog) compromise the declarative semantics in order to improve performance and to provide for the deliberate "control" of execution behavior. Thus, most Prolog interpreters omit the "occurs check" (which renders them unsound), use depth-first search (which renders them incomplete), add "negation as failure" (whose logical interpretation is a controversial topic [22]), and provide extra-logical "control" constructs such as "cut," "assert," and "retract." (Although these features destroy hopes for a declarative semantics, it is possible to give them a denotational semantics [19].)

Production-rule languages add further control and other features (e.g., nonmonotonicity) that take them even further away from pure logic programming languages—and compromise still further any declarative semantics. Buchanan and Smith justify this as follows [11, page 34]:

> Many designers of expert systems are uncomfortable with mathematical logic as a representation language because it lacks expressive power. Numerous extensions must be made to express some of the concepts that are frequently used in applications: uncertainty, strategy knowledge, and temporal relations. Some logicians are uncomfortable with reasoning that is not theorem proving and with knowledge bases that are not axiomatic systems that allow proofs of consistency and completeness. The search for new logical formalisms that are more powerful than predicate calculus reflects the *tension* between simple, well-understood formalisms and expressive power.

3.2 Operational Semantics

If the need to consider control strategies and working memory makes the search for declarative semantics for rule-based languages rather unpromising, perhaps it will be better to consider more operational semantics—that is, semantics that explicitly consider the existence of "state" and control flow.

Hoare introduced a notation and deductive system for reasoning about the partial correctness of imperative programs in his seminal paper [29].

In this notation, if C is a program, and P and Q are conditions on the program variables used in C, then the notation $\{P\}C\{Q\}$ expresses the safety property that if execution of C begins in a state (i.e., an assignment of values to program variables) satisfying P, *and if execution terminates*, then it must do so in a state satisfying Q.

An alternative notation that makes the system state explicit is sometimes preferable. In this notation, conditions are modeled as predicates on states (assignments of values to program variables), and programs as relations on states. If s and t are states, then $P(s)$ is true if state s satisfies condition P, and $C(s,t)$ is true if program C, when started in state s, can terminate in state t. The connection between the two notations (see, for example, [27]) is given by

$$\{P\}C\{Q\} \equiv \forall s,t : P(s) \wedge C(s,t) \supset Q(t)$$

"Proof rules" for the elementary constructs of a programming language, together with rules for their combination, allow properties of imperative programs to be proved. An example of a proof rule is that for the "while" loop:

$$\frac{\{P \wedge B\}C\{P\}}{\{P\}\ \textbf{while}\ B\ \textbf{do}\ C\{P \wedge \neg B\}}$$

The hypotheses of the rule are written above the line, the conclusions are written below it. A formula P such that $\{P \wedge B\}C\{P\}$ is called an *invariant* of C for B.

The difficulty in applying these ideas to production-rule languages is that the control structures of production-rule systems are less explicit and regular than "while" loops, and the control (i.e., conflict resolution) strategies used are quasi-nondeterministic. It is possible to give Hoare-style semantics for pure nondeterministic constructs (e.g., for Dijkstra's guarded command language [20][1]), but the strategies used in production-rule languages are so operational that they defy tractable formalization.

It is important to understand just how complex some of these control features are; a brief description of the operation of OPS5 should make it clear.

OPS5 programs consist of rules called *productions*, each of which comprises a condition part (called the LHS) and an action part (called the RHS). Programs operate on a global database called *working memory* by repeatedly performing a sequence of actions called the *recognize-act cycle*:

Match: evaluate the LHSs of the productions to determine which are satisfied by the current contents of working memory (the set of eligible productions so formed is called the *conflict set*).

Conflict Resolution: select one production from the conflict set; if the conflict set is empty, return control to the user.

Act: Perform the actions specified in the RHS of the selected production (this is called *firing* the production).

[1]Dijkstra uses "weakest preconditions" rather than Hoare logic, but the differences between these formalisms are not significant to our discussion.

The elements of the conflict set are called ***instantiations***; they consist of a production and a binding of working memory elements to the variables in its LHS. The conflict resolution strategy is the following[2]:

1. Discard from the conflict set those instantiations that have already fired. (The definition of "already fired" is complex; roughly speaking, it means that there is some *n* such that this instantiation has been present in ***every one*** of the previous *n* conflict sets, and that it was selected for firing in one of those sets.)

2. Compare the recencies (the cycle on which they last changed) of the working memory elements matching the ***first*** condition elements in each instantiation. Retain those with the most recent elements.

3. Order the instantiations on the basis of the recencies of the ***remaining*** working memory elements using the following algorithm to compare pairs of instantiations. First compare the most recent elements from the two instantiations. If one is more recent than the other, the instantiation with that element dominates. If the two are equally recent, compare the second most recent elements from each instantiation, and so on, until either a difference is found or one instantiation runs out of elements. If one instantiation exhausts its elements before the other, the unexhausted instantiation dominates.

4. If no single instantiation dominates all others under the previous rule, use the one with the most conditions in its LHS; if that does not select a unique rule, choose one arbitrarily.

Forgy [23] defends these rules on the grounds that "they make it easy to add productions to an existing set and have the productions fire at the right time, and because they make it easy to simulate common control constructs such as loops and subroutine calls." Anyone concerned with formal reasoning (or just reasoning) about rule based systems might wish that control structures were explicit rather than "easy to simulate." (The same argument applied to imperative programming languages would justify the elimination of all control structures except the "goto," since it can also simulate those control structures.)

Additional conflict resolution strategies that are commonly employed include giving preference to rules according to their position in the rule-

[2]This is the strategy called MEA; OPS5 also has a strategy called LEX—see [23].

base (early rules have preference), or associating explicit priorities with rules (rules with high priority have preference).

3.3 Approximate Semantics

The inescapable conclusion from the previous two sections is that the construction of either declarative or Hoare-style semantics for current rule-based languages is a hopeless task. In the long term, we expect that concern for predictability and reliability will lead to the development of programming languages with tractable semantics for expert system applications (just as it has, to at least some extent, with conventional languages). In the near term, however, we either have to accept something less than perfect, or give up.

Because we have already abandoned the attempt to prove correctness for expert systems, being willing to settle instead for proofs of, possibly modest, safety properties, it may be that merely "approximate semantics" will suffice for our purposes.

Recall that the (exact) functional behavior of a program $\mathcal{P}$ is identified with the relation $\mathcal{P}(i, o)$. An approximate semantics will associate a somewhat different relation $\mathcal{P}_{\text{approx}}(i, o)$. To be useful, the approximate semantics must be conservative—that is, they must not "underestimate" the exact behavior (at least when applied to valid inputs). Thus we require:

$$\mathcal{V}(i) \wedge \mathcal{P}_{\text{approx}}(i, o) \supset \mathcal{P}(i, o)$$

It will then follow that safety properties established with respect to the approximate semantics will hold for the exact semantics as well.

3.3.1 Approximate Declarative Semantics

We have seen that rule based languages differ from pure logic programming languages most violently in providing implicit and explicit control over the order in which rules are selected. (The other serious difference is that rule-based languages allow values to be asserted and retracted; we will ignore this issue for now.) If there were no conflict resolution strategy—i.e., if any and all matching rules could fire—then we would be closer to a purely logical interpretation of the rules. Now the effect of conflict resolution strategies is to limit which rules can fire—so that fewer things will happen in the presence of conflict resolution than in its absence. This means that a logical interpretation of a rule-base (i.e., no conflict resolution) should indeed provide a conservative approximate semantics. Unfortunately, it will often be so conservative that nothing useful can be deduced. An example might help make this clear.

Consider the following rules from a system to handle hospital admissions, and suppose that conflict resolution is performed by giving priority to rules according to the order in which they appear.

```
if  head-injury(X)
    and time-since-injury(X) < 8
then
    send-to(X) := emergency-room


if  can-walk(X)
then
    send-to(X) := waiting-room
```

Now suppose we would like to establish the safety property "anyone who has had a head injury within the last 4 hours will be sent to the emergency room."

Interpreting the rules as a logical theory, we have

$$\text{head-injury}(X) \wedge \text{time-since-injury}(X) < 8 \supset \text{send-to}(X) = \text{emergency-room}$$

$$\text{can-walk}(X) \supset \text{send-to}(X) = \text{waiting-room}$$

It is clear that we can establish our safety property from the first of these formulas. However, because there is no ordering implied between the formulas when interpreted in logic, "firing" the first does not preclude the second one from firing also. Thus, if it happened that patient X was able to walk, we might *also* conclude that he should be sent to the waiting room. In approximating our rule base by a logical theory, we have lost too much information. For the conflict resolution strategy considered in this example, it is easy to see that the strategy can be encoded in the approximating theory: we simply conjoin to the antecedent of each formula the negations of the LHSs of earlier rules:

head-injury(X) $\wedge$ time-since-injury(X) $< 8 \supset$ send-to(X) = emergency-room

$\neg$(head-injury(X) $\wedge$ time-since-injury(X) < 8) $\wedge$ can-walk(X) $\supset$ send-to(X) = waiting-room

The dynamic nature of most other conflict resolution strategies precludes such simple static representations, however.

Although simply regarding a rule base as a logical theory may provide an excessively conservative semantics, it may still be possible to derive some benefit from the exercise. Suppose we were to modify the theory by adding clauses to the antecedents of its formulas until we were able to prove the desired safety properties. This would make explicit the requirements on the conflict resolution strategy to be employed—which could be valuable information. Informal arguments could then be used to justify the differences between the rules and the derived formulas (or the rules could be modified to make them less dependent on the conflict resolution strategy).

Consistency

There is a flaw in this reasoning, however. If the derived theory is inconsistent, then it can be used to prove any safety property whatsoever. Before we proceed to analyze this issue, it will be helpful to review some of the notation and terminology of logic.

A (propositional) theory $\mathcal{T}$ is simply a set of (propositional) formulas; a formula $\mathcal{F}$ is a *theorem* of $\mathcal{T}$ if $\mathcal{T} \vdash \mathcal{F}$—that is, if any model of $\mathcal{T}$ is also a model of $\mathcal{F}$. By the deduction theorem, this is equivalent to $\vdash \mathcal{T} \supset \mathcal{F}$—that is, $\mathcal{T} \supset \mathcal{F}$ must be *valid* (true in all interpretations). If $\mathcal{T}$ is inconsistent, then it has no model (by definition), so all interpretations will valuate $\mathcal{T}$ to false, and hence $\mathcal{T} \supset \mathcal{F}$ to true—whatever formula $\mathcal{F}$ may be.

It seems, therefore, that we must check any theories derived from rule bases for consistency before proceeding to draw any conclusions. In fact, the situation is rather more complex than this. For the theory derived from a rule base to be consistent simply means that it has a model (i.e., some interpretation of its constant, function, and predicate symbols that valuate all its formulas to true). However, it may be that the facts initially asserted into the working memory of the expert system are inconsistent with any models of its rule base. Again, we will be able to prove arbitrary formulas. The explanation is, of course, that it is not the rule base alone that induces the theory corresponding to an expert system—it is the rule base *plus* the initial facts.

This explains something that others have noted: the notions of inconsistency employed in logic and in rule based expert systems do *not* coincide. In logic, the following set of formulas is consistent

$$\begin{aligned} p &\supset q \\ p \wedge r &\supset \neg q \end{aligned}$$

because both formulas evaluate to `true` in any interpretation that assigns `false` to p. However, if these formulas are interpreted as rules, and p and r are input variables assigned the value true, then both q and $\neg q$ will be asserted—an obvious contradiction. The problem here is that although the rules are consistent on their own, they form an inconsistent theory when combined with certain initial values. That is, the following "instantiated theory" is inconsistent:

$$\begin{aligned} p &\supset q \\ p \wedge r &\supset \neg q \\ p & \\ r & \end{aligned}$$

This observation allows us to define *consistency* for the theory corresponding to a rule-based system as follows: let I_0 be any interpretation for the inputs to the system (i.e., assignment of initial values),[3] then there must exist a model for the rule base (interpreted as a theory) that is an extension to I_0.

Several authors have proposed methods for testing rule-based systems for consistency [9, 25, 42, 53] but none of them present a rigorous definition for rule-base consistency, though Ginsberg [25] does observe that it is different from the notion of consistency in a logical theory (the example above is due to him). We therefore believe that ours is the first precise definition of consistency for rule-based systems. It should be possible to use our definition to provide a proof of correctness for Ginsberg's KB-Reduction al-

[3]If there is a validity constraint on input values, then we can restrict the interpretations to those that satisfy it.

gorithm [25].[4] It should also be possible to accommodate the simpler forms of nonmonotonic reasoning (such as the closed-world assumption).

The notion of consistency derived above is a logical one: it simply assures us that there is some way of satisfying the rule base given any (valid) assignment of values to the inputs, without assuming a particular conflict resolution strategy. The "answer" produced by the system will be (part of) the interpretation that constitutes a model for the rule base. Now there may be several models in general, and so several different sets of outputs corresponding to the same inputs. This naturally prompts the introduction of a second notion for consistency: if the same set of inputs can produce several sets of outputs, then it may be desirable in many cases that each of these sets of outputs should be "similar" or "consistent" with each other.[5] That is, we should require:

$$\mathcal{V}(i) \wedge \mathcal{P}(i, o_1) \wedge \mathcal{P}(i, o_2) \supset o_1 \simeq o_2$$

where $\simeq$ denotes "similarity" (presumably an equivalence relation).

It is not clear how (or whether) one can verify this property of "output consistency" in general; it is similar to "sensitivity" or "continuity"—the requirement that inputs that are close together should generate outputs that are also close together. Empirical tests seem the only plausible way to validate such properties.[6] A property that is sometimes capable of verification is the stronger one that the outputs corresponding to any set of inputs should be unique. In the case of a rule-based system, this will be so if the order in which rules are fired does not affect the final values produced. This latter property is generally called the Church-Rosser property. Very effective tests for the Church-Rosser property are known for certain restricted logics (e.g., the Knuth-Bendix procedure [33] for equational theories), together with weaker results for more general cases (e.g., [47]). It is possible that these techniques also could be adapted to restricted rule bases.

[4]The methods of Suwa et al. [53] and Nguyen et al. [42] are so weak and ad hoc that they do not require further investigation. Bezem's method [9] is simply an algorithm for testing whether a particular initial interpretation I_0 can be extended to a model for the rule base.

[5]This notion of consistency is likely to be particularly appropriate for analytic expert systems: how much trust should one place in a system that is capable of producing totally different answers from the same inputs?

[6]If the equivalence classes of the similarity relation can be identified with some predicate, then the problem becomes one of constraint satisfaction—which we believe is verifiable.

Another property likely to be of some interest is completeness: a rule-based system is incomplete if it can fail to assign a value to some output. Analogous to our logical definition of inconsistency, we could say that the theory corresponding to a rule base is incomplete if it has models in which the interpretation of some output is unconstrained. It seems plausible that the KB-Reducer algorithm of Ginsberg [25] can be extended to this case.

We began with the idea that it might be possible to interpret a rule base as a logical theory and thereby deduce some approximate, but conservative, properties. We have seen that it is necessary to ensure a strong form of consistency for the theory so produced. Given such a consistent theory, it may then be possible to prove that certain constraints will be satisfied in any execution of the system. Performing such a proof might seem to require general theorem proving capability. In fact, all the information required is gathered as part of Ginsberg's KB-Reducer consistency-checking procedure. KB-Reduction is essentially a form of symbolic execution [31]; for each "hypothesis", "labels" are built up that indicate the combinations of input values that will cause that hypothesis to be asserted. The label for a hypothesis asserted on the right-hand side of a rule is computed by symbolic evaluation of its left-hand side. For example, if the "partial labels" for hypotheses D and A are respectively $p \vee q$ and p, then the rule $D \wedge \neg A \rightarrow B$ will add $(p \vee q) \wedge \neg p$ (which simplifies to $q \wedge \neg p$) to the label for B. Thus, to verify a constraint such as "anyone who has had a head injury within the last 4 hours will be sent to the emergency room," we simply check that the label for "being sent to the emergency room" is implied by the condition "head injury within the last 4 hours."

It should be perfectly straightforward to develop an implementation of Ginsberg's KB-Reducer procedure that supports this limited form of verification. It should be noted that there is great similarity between these topics and algorithms for rule-based systems and those for decision tables [36, 38].

3.3.2 Approximate Imperative Semantics

While the investigation of approximate declarative semantics has shed some light on the question of consistency, and suggests a way of verifying constraints for elementary rule bases, we believe that approximate imperative semantics provides a more suitable base for verifying more complex properties. As with the declarative approach, the more complex forms of conflict resolution will not be modeled, but the presence of an explicit state allows operational constructs to be modeled more accurately. In particular, the assignment of values to variables (and subsequent reassignment of new values) is easily accommodated.

The semantics of an individual rule

if $\mathcal{C}$ **then** $\mathcal{A}$

can be modeled by the standard axiom

$$\frac{\{\mathcal{P} \wedge \mathcal{C}\}\mathcal{A}\{\mathcal{P}\}}{\{\mathcal{P}\}\ \textbf{if}\ \mathcal{C}\ \textbf{then}\ \mathcal{A}\{\mathcal{P}\}}$$

Here the predicate $\mathcal{P}$ is an *invariant* maintained by the rule. If a number of rules all maintain the same invariant, then their nondeterministic combination will also maintain that invariant, as will the iteration of that combination.

The basic idea for rule base verification using approximate imperative semantics is to establish an invariant for the entire rule base and to verify, using the rule above, that it is indeed an invariant for each rule in the rule base. If the invariant is also guaranteed true for any valid input, then one may conclude that it will be true when the rules terminate.

It might seem that any formula that is true of the inputs to the system, and that is maintained true by all the rules in the system, can only encode a very weak property. In a strict sense, this is obviously true, but a great deal of information can be encoded in the invariant using control variables. These are variables that record and control the progress of computation in the rule base. When a rule that initiates a new stage of processing fires, it sets a control variable to a particular value to indicate that fact. Some rules will condition their LHSs on the values of control variables (e.g., **if** control-var = x **and** other conditions **then** actions), so that they are eligible for execution only during certain stages of processing. The system invariant can exploit this attribute by having different components for different control values—that is, it may have the form

$$\begin{array}{lcl} \text{control-var} = \text{x} & \supset & \text{invariant-component-x} \\ \wedge\ \text{control-var} = \text{y} & \supset & \text{invariant-component-y} \\ & \vdots & \\ \wedge\ \text{control-var} = \text{z} & \supset & \text{invariant-component-z} \end{array}$$

Informal arguments will probably have to be used to argue that, on termination, the control variable will have a value associated with an "interesting" component of the invariant.

In the case of constraint satisfaction problems, the constraint will provide the invariant, or at least one component of it, to be verified. For model inversion problems, the invariant to be verified will be that the components of the "answer" being constructed are consistent with the model employed. In particular, the rule "if symptoms **then** cause" should be supported by the argument that in the explicit model, "cause" leads to "symptoms." The next chapter presents a preliminary experimental investigation into the utility of these ideas.

Chapter 4

Examples

4.1 Introduction

In this chapter we report on some of our experiences and observations in attempting to apply techniques proposed in this report to the verification of rule-based expert systems. The experiments described in this chapter should not be considered as indications of our current capabilities; rather, they serve to illustrate what we believe to be some of the promising approaches and interesting challenges concerned with expert system verification.

These experiments were conducted using various notations and formalisms, including the P-BEST expert system shell and the EHDM verification environment [55]. P-BEST is a forward-chaining production system that resembles OPS5 [23] and other similar tools. For a more complete description of the P-BEST environment, see Appendix A. EHDM is a formal specification and verification environment constructed at SRI. For an introduction to EHDM, see Appendix B.

4.2 Example–Constraint Satisfaction

In this section, we consider the application of formal reasoning to the development and verification of constraint-satisfaction type expert systems. The example presented here considers the problem of constructing an expert system that plays the game "Tic-Tac-Toe."

4.2.1 Problem Description

The game of *tic-tac-toe* may be thought of as a constraint satisfaction problem. Given an arbitrary legal board configuration, the computer is required to place a marker in a position such that a certain goal is achieved. We considered the "classical" two-player version of tic-tac-toe as played on a 3x3 board (illustrated in Figure 4.1). The goal of the game is to place three

1,1	1,2	1,3
2,1	2,2	2,3
3,1	3,2	3,3

Figure 4.1: Tic-tac-toe Board Configuration

markers on the board so that they are in alignment either vertically, horizontally, or diagonally, while preventing the opponent from placing three markers in alignment. Play begins with an empty board, and proceeds as alternating players place markers on the board until either: (a) some player achieves the placement of three markers in a row, or (b) no remaining empty positions exist.

4.2.2 Rule-Based Implementation

We implemented an initial rule base to play tic-tac-toe. To further constrain the problem, we adopted the convention that the opponent would always get the first move. In an effort to model the process of knowledge engineering, in which insight into the problem space is gathered in a piece-wise fashion, we adopted a naive playing strategy that can be informally stated as follows:

- If the computer can win (i.e., place a marker so that there are three markers in a row, either vertically, horizontally, or diagonally), then have the computer place the marker in the winning position and terminate the game.

- If the computer must block to prevent the opponent from winning, then have the computer place a marker in the blocking position.

- Otherwise, have the computer place a marker in one of the following positions, arranged according to desirability: center, corner, middle.

Using P-BEST, a forward-chaining expert system shell, we defined the following fact templates. In P-BEST, a fact template must be declared for each possible assertion. In the case of the tic-tac-toe example, there were three possible types of assertions that could be made into the knowledge base:

- position assertions
- turn assertions
- opponent move assertions

Position assertions were used to represent the state of the playing board at any given point during the game. There were exactly nine such assertions at all times—one assertion for each of the nine possible positions on the playing board. At the beginning of play, each of these assertions contained the designation that the corresponding position was free of any marker. As play progressed, the position assertions were altered to reflect when a marker was placed in the corresponding location. Such markers were designated as "opp" short for opponent, and "com" for computer.

Turn assertions were used to record whose turn it was at any given point in the game and to keep track of how many turns had transpired.

Opponent-move assertions were used to communicate the intention of the human opponent to the knowledge base. Incorporated into the knowledge base were several rules that were responsible for querying for the opponent's move, checking its legality, and asserting a proper opponent-move assertion into the knowledge base. Because these rules do not effect the behavior of the program with respect to its strategic performance, they are not considered in the subsequent analysis.

Figure 4.2, gives an example of what the declaration of these fact templates, called "ptypes" or *predicate-types* in the terminology of P-BEST, looks like. For a more complete description of the P-BEST system, see Appendix A.

For an example of the syntax of the P-BEST rules, refer to Figure 4.3, which gives an example of the rule "make_random_move" from the tic-tac-toe rule base. This rule is given an explicit ranking, in this case -10, which affects the selection of which rule to execute when there are multiple candidates. The higher the rank of the rule, the greater priority given that rule during conflict resolution. The effect of this rule is that when it is the computer's turn to move, if no rules with higher ranks are eligible to fire and if there is

```
(define-ptype POSITION
   "A unique position in the 3x3 board."
   row                 ; integer – range 1 to 3.
   col                 ; integer – range 1 to 3.
   marker              ; free, com or opp.
   )
```

Figure 4.2: P-type Fact Template Declaration in P-BEST

a free position left on the board, the computer will select one of those free positions and replace the "free" designation with its marker, "com".

The completed initial rule base consisted of 11 rules:

- 3 rules to detect a condition where the computer can win
- 3 rules to detect a position where the computer must block
- 2 rules to recognize lost and tie games
- 2 rules to get and check human input
- 1 rule to make a random move as a default case

plus a handful of LISP functions that assisted in managing the display of the game board. The complete code for the rule base is given in Appendix C.

4.2.3 Requirements

Although it is sometimes quite difficult to identify expectation for a system whose capabilities will evolve through a developmental process, in order to validate the system, either through testing or through formal analysis, it is imperative that the requirements against which the system is to be evaluated be made explicit.

Tic-tac-toe can be viewed as an optimization problem; given a board position, the computer is expected to pick the *best* move possible. Optimally, we would like to have a system which wins; however, in this particular game it is a well-known fact that a win cannot be guaranteed. A win occurs only when the opponent makes a mistake; if no mistake is made by either player, the game terminates in a draw. Therefore, a ***desired competency requirement***, which is also a liveness property for this system, might be:

```
(defrule MAKE_RANDOM_MOVE
   "Select a pseudo-random position to place marker.
  Prioritise positions as: 1. center, 2. corners
  and 3. middles."
 at rank -10 states
 if there exists a turn called t1
      such that player is com and
             count is ?x and
    there exists a position called p1
      such that marker is free
 then forget p1 and
     forget t1 and
     remember a position
       which inherits from p1
         except that marker is com and
     remember a turn
       such that player is opp and
              count equals (1+ ?x))
```

Figure 4.3: Rule make_random_move From the Tic-Tac-Toe Rule Set.

- The system wins, when possible.

This implicitly includes the notion of playing to the standards of a human player; it implies that a strategy is pursued that encourages the opponent to make an error, and recognizes such situations in order to capitalize on mistakes and force a win. Such a notion would be very difficult to capture formally or to check with formal analysis. Even in the game of tic-tac-toe, which has limited number of possible system states, an optimal move in an arbitrary configuration would be difficult to recognize; in some sense it would involve recognizing the strategy of the opponent which may not be deducible from the board configuration alone.

A *minimum* competency requirement, which is also a safety requirement for this system, is somewhat more tractable; above all:

- The system shouldn't lose.

We have already noted that it is impossible for an opponent to force a win; therefore, the minimum competency we required of our system was that it never make a mistake that would result in the opponent's winning.

To formalize this minimum competency requirement, we introduced the notion of a safe state. We defined a safe state, which is expressed in terms of board configurations, to be those configurations in which the computer has won the game, or in which the opponent cannot win the game with the placement of the opponent's next marker. In the language of EHDM, this is expressed as:

```
safe_def: Axiom
    safe(kbx, $s_n$)
        $\Leftrightarrow$ won_game(kbx, $s_n$)
          $\vee$ $\neg$lose_next_move(kbx, $s_n$)
```

Under our initial formulation of rules, there was no reason to believe the expert system should be able to win a game of tic-tac-toe because it does not attempt to pursue a winning strategy (the computer simply selects a pseudo-random move when it cannot immediately win and need not block). However, it was reasonable, to hope that the computer would not lose under this strategy. Specifically, three rules were included in the rule base, **blk_row**, **blk_col**, and **blk_dia**, that prevent the opponent from placing a marker in any row, column or diagonal which would result in a lost game.

4.2.4 Formal Specification

We constructed a formal specification of the tic-tac-toe rule base and proposed to verify that specification against the minimum competency requirement that the system must not lose the game. The specifications for the tic-tac-toe rule base are contained in Appendix D. In order to create a formal specification for this application, it was necessary first to define an approximate semantics for the language of our expert system shell, P-BEST.

We expressed the nonmonotonicity of the rules by introducing an explicit notion of system state. As explained in Section 3.3, one way to accomplish this in the EHDM language is to add to value-returning functions an extra parameter that corresponds to a specific system state. For example, one of the modules in the specification describes a formal object called a **kb**, which is declared to be a set of facts (see module **kbs** in Appendix D). A function, **kb_inst**, of two parameters is provided to access the value of objects of this type; one parameter is the knowledge base for which to return the value; the other is the current system state. Therefore, the declaration for this function in the language of EHDM is:

kb_inst: functions[kb, state → set[fact]]

With this formulation, we can express that facts that are associated with a given knowledge base in a given state may possibly not be associated with that knowledge base (i.e., may have been retracted) in some later state, without the danger of such retractions leading to logical contradictions.

Each fact template, or ptype, declaration in the rule base becomes a type declaration in the formal specification. For instance, the type **position** is declared to be a subtype of type **fact**. This allows the declaration and use of logical variables that range over all possible positions. Along with the type declaration, a function is defined that tests for membership of a certain class of instances of a fact in a knowledge base at a particular state. With these declarations, a relatively straightforward translation of the rules into a logical notation can occur. This can be seen by comparing any of the rules in Appendix C with their corresponding formal specifications in Appendix D.

The proof strategy for establishing that the system conforms to the specification of our safety property involves simple induction over all valid sequences of moves. In such a scheme, the initial state of the system, in which no markers have been placed on the board, is shown to conform to the property in question. In this case, the proof of the safety of the initial state is trivial; since there are no markers on the board, there is no way for the opponent to win on the next move. Then, given an arbitrary "reachable" state that preserves the safety property, if one can establish that all reachable next states also preserve the property, by appealing to induction one has established that the property is maintained over all valid sequences of moves.

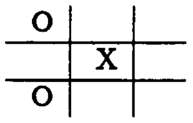

Figure 4.4: Example Configuration

In attempting to construct the proof, it soon became apparent that it was impossible to prove the safety property without considering the implicit control flow in the rule base and the sensitivity of the operational semantics to the ordering of assertions. In the tic-tac-toe rule base, for example, the rules dealing with blocking the opponent have precedence over the rule for placing the computer's marker at random. In a situation such as that shown in Figure 4.4, where the opponent's markers are represented as "O" and the computer's markers are represented by "X", it cannot be proved that the computer will not lose on the next move (one of the criteria for the safety property in question) if the computer places its marker in any position other than row 2, column 1. This is a direct reflection of the fact that it is unsafe to execute any rule in the above situation except the rule that explicitly deals with blocking an opponent's win on any particular column; however, if the rules are treated as being nondeterministically selected for execution, this crucial piece of knowledge is missing and it is impossible to reason formally about the safety of the system.

The informal operational semantics of the tic-tac-toe rule base include a ranking of rules. This ranking will prevent the make_random_move rule from firing when one of the blocking rules is also a candidate. Since exactly one rule fires per state transition, it is operationally impossible for the wrong rule to fire. We needed to find a formalism that could capture this implicit proceduralism of the rule base.

Making use of the concept of guarded commands as put forward by Dijkstra [20], we adopted a formulation that approximates the proceduralism of this application. To do this, we defined a set of operations, one for each of the rules, such that the precondition of each operation consisted of the conjunction of the negation of the preconditions of all operations of higher precedence with the precondition of the corresponding rule. This is illustrated in the module **guarded_ops** in Appendix D. This is only an approximation of the actual operational proceduralism because it results in a strict ordering of the rules; the actual rule base is only ordered with respect to groups of rules at differing precedence levels—rules within each group are nondeterministically selected. However, this strict ordering is sufficient to establish the safety property in question.

The operational semantics of the rule base is also sensitive to the order in which facts are asserted into the knowledge base. For example, the make_random_move rule (Figure 4.3) selects a free space to place a marker. Contrary to what the name of the rule suggests, the choice of free space is not random, but totally deterministic. As the documentation of the rule suggests, the center is always chosen if it is free, followed by the corners (beginning with the upper-right-hand corner and proceeding clockwise), and finally by the middles. This precedence occurs because the conflict resolution strategy of P-BEST gives priority to rule instances which reference the most recently-asserted facts. In order to capture this operational dependence upon the ordering of the initial assertions, additional axioms were introduced to make this ordering available during formal reasoning (see module ttt-rls1 in Appendix D).

The proof attempt uncovered four configurations of the board which could result in the computer being forced into a no-win situation; these configurations occur when the opponent can win in two possible ways, making it impossible for the computer to block the opponent in a single move. One such configuration is illustrated in Figure 4.5. We were able to prove that the

O		X
	X	
O		O

Figure 4.5: No-Win Sample Configuration

strategy of the tic-tac-toe rule base prevented the opponent from reaching two of these states. However, the formal analysis showed that the remaining two configurations could be reached and could cause a loss. Because of the existence of these situations, it was impossible to prove the safety invariant for the blocking rules as they were originally formulated. In other words, our original rule base was incorrect; it failed to consider situations which were both reachable and contradictory to our safety requirement.

With this insight, we were able to expand the definitions of our safety requirement to be:

$$
\begin{aligned}
&\text{safe\_def: } \textbf{Axiom}\\
&\quad \text{safe}(\text{kbx}, s_n)\\
&\quad\quad \Leftrightarrow \text{won\_game}(\text{kbx}, s_n)\\
&\quad\quad\quad \vee \neg(\text{lose\_next\_move}(\text{kbx}, s_n)\\
&\quad\quad\quad\quad\quad \vee \text{unsafe\_config\_1}(\text{kbx}, s_n)\\
&\quad\quad\quad\quad\quad\quad \vee \text{unsafe\_config\_2}(\text{kbx}, s_n)\\
&\quad\quad\quad\quad\quad\quad\quad \vee \text{unsafe\_config\_3}(\text{kbx}, s_n) \vee \text{unsafe\_config\_4}(\text{kbx}, s_n))
\end{aligned}
$$

To correct the rule base, two additional blocking rules were added at a higher priority with respect to the other blocking rules; these rules specifically countered the two remaining unsafe situations. With the addition of these two rules, the inclusion of their logical equivalents in the formal specification, and the expanded safety definition, it was possible for the proofs to be carried forward. Samples of proofs in the methodology of EHDM are included at the back of Appendix D. We did not attempt to complete all proofs; it was not necessary to formally complete all the proofs, nor to prove the induction hypothesis, in order to derive the benefits of the formal analysis.

4.2.5 Observations

From this experiment, we are able to draw the following conclusions.

The translation of expert system rules into logic requires a formulation that captures the state-dependent nature of the inference engine. An approximate semantics for rule-based systems can be formulated which provides an adequate framework for reasoning formally about some of the behavior of the system.

While it may be possible to establish very weak invariants independent of the conflict resolution strategies involved in the expert system, and independent of the inherent proceduralism involved, it would be impossible to establish many important safety properties, such as the one defined for this example system, without capturing the implicit control flow.

Reasoning formally about this application uncovered critical situations that were originally overlooked. While the rule base was not incomplete or inconsistent according to syntactic analysis, the original rule base was incorrect, because it was conceptually incomplete for a class of situations that could have led to violation of the safety property during operation. Such conceptual incompleteness would not have been discovered through any form of syntactic analysis of the rule base.

4.3 Example–Model Inversion

In this section we consider the application of formal reasoning to the development and verification of model-inversion type expert systems. The example presented here considers the problem of constructing an expert system to diagnose problems in a car's electrical system.

4.3.1 Problem Description

As in all diagnosis systems, the goal of our example program is to be able to identify the most-likely cause of a problem given a set of observed effects. To constrain the problem, we consider only a simplified subset of an actual electrical system. Our subset consists of the following components:

- battery cells
- battery
- ignition switch
- points
- coil
- distributor
- plugs
- solenoid
- starter

We limit the set of potential causes to be the set of failed components. In other words, we would consider the failure of the ignition switch to be a potential cause for the observation that power is failing to be delivered to the points and starter solenoid, but we would not consider the state of the wires connecting these components as potential sources of the problem. Further, we consider only single points of failure in this exercise—for any given set of observations, the expert system is required to find the single most-probable cause which accounts for the observations.

4.3.2 The Formal Model

We created a formal model of our simplified automotive electrical system in the language of EHDM; this model is contained in modules **components** and **engine_model** in Appendix H. The focal point of this model is the axiom **electrical_system_model** shown in Figure 4.6. The accuracy of this formalization is based on a number of assumptions (e.g., the fact that the car in question is not running); however, the model is adequate for the illustrative purposes of this discussion. The model asserts that there is a causal relationship between various components of the system. To have power at the ignition switch, for example, the model states that it is necessary that the battery be in a good state of repair and that the battery be delivering electrical_system_model: **Axiom**

$$
\begin{aligned}
&(\mathrm{power\_at}(\mathrm{battery}) \Leftrightarrow \mathrm{good}(\mathrm{cells}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{ignition}) \Leftrightarrow \mathrm{power\_at}(\mathrm{battery}) \wedge \mathrm{good}(\mathrm{battery}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{points}) \Leftrightarrow \mathrm{power\_at}(\mathrm{ignition}) \wedge \mathrm{good}(\mathrm{ignition}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{coil}) \Leftrightarrow \mathrm{power\_at}(\mathrm{points}) \wedge \mathrm{good}(\mathrm{points}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{distributor}) \Leftrightarrow \mathrm{power\_at}(\mathrm{coil}) \wedge \mathrm{good}(\mathrm{coil}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{plugs}) \Leftrightarrow \mathrm{power\_at}(\mathrm{distributor}) \wedge \mathrm{good}(\mathrm{distributor}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{solenoid}) \Leftrightarrow \mathrm{power\_at}(\mathrm{ignition}) \wedge \mathrm{good}(\mathrm{ignition}))\\
&\quad \wedge (\mathrm{power\_at}(\mathrm{starter}) \Leftrightarrow \mathrm{power\_at}(\mathrm{solenoid}) \wedge \mathrm{good}(\mathrm{solenoid}))
\end{aligned}
$$

Figure 4.6: Model of Simplified Automotive Electrical System

power. This is representative of the fact that the battery is "up-stream" from the ignition in our simple model, just as the ignition is up-stream from both the points and the starter solenoid.

4.3.3 Rule-Based Implementation

For this experiment, we developed a small backward-chaining rule interpreter, which is a variation on Winston's expert problem solver [56]. Backward-chaining systems start with an unconfirmed hypothesis and attempt to confirm it. This involves identifying rules that substantiate the hypothesis, and then attempting to verify the conditions upon which the applicable rules are dependent. Apart from some syntactic sugar in the form of extensions to simplify the expression of rules and hypothesis, there are two major differences between Winston's original system and our enhanced version:

1. Our system allows the appearance of "not" clauses in the LHS of rules

2. Our system allows the specification of questions to be used to obtain observations from the user, rather than attempting to synthesize such questions from the symbolic representation of the facts

We created two separate implementations of this rule base. One implementation was distilled from a set of troubleshooting and diagnosis procedures contained in Chilton's Import Car Repair Manual [54]. We would expect the quality of this rule base to approximate the results of most standard knowledge engineering efforts. A subset of the rule base is given in Appendix F. The other implementation was derived from the abstract model of our simplified system through a relatively mechanical process. This rule base is given in Appendix G.

The procedure for generating the rule base from the axiomatic specification of the model, as seen in Figure 4.6, was straightforward. For each equivalence relation specified by the model, a rule is generated to capture the equivalence. For example, the relation

$$(\mathrm{power\_at(starter)} \Leftrightarrow \mathrm{power\_at(solenoid)} \wedge \mathrm{good(solenoid)})$$

caused the following rule to be included in the rule base:

```
(define-rule d7
  (if (power at solenoid)
      (good solenoid))
  (then (power at starter)))
```

Then, rules were provided which define the possible observations and provide questions that may be asked of the user to acquire information about the observations. Finally, a set of rules were added which identified the global dependencies in the system; these rules serve to associate the failure hypothesis with the observations about system behavior.

4.3.4 Requirements and Validation

As suggested in Section 2.2.2, requirements for expert systems which have formal models can be expressed as

$$\forall c \in C, e \in E : \mathcal{P}(e, c) \supset \mathcal{M}(c, e)$$

where C is the set of all causes, and E is the set of effects. This leads to a procedure for validating the operation of the expert system against the expectations of the model for any given set of causes and effects.

Our simplified model identifies eight possible causes for the system to fail; these eight causes correspond to the failure of one of the eight system components. The effects in our model are limited to an observation of where power is available and where it is not. There is an assumption that the failure of a particular component causes the flow of power through that component to cease, making the power unavailable to all components down-stream from the failing component, and that the presence or absence of such power is observable. What we were interested in showing was, that given a certain set of observations presented to the expert system, an answer would be produced which, when presented to the model, would cause the correct set of effects to be predicted.

To allow the model to be used to make such predictions, we defined two additional EHDM modules, **test** and **predict** (see Appendix H), which we used to predict the values of the observable effects. To establish which effects should be observable according to the model, we set the constant `comp` in module **predict** to the value of the failed component. We then attempted to prove eight lemmas, each of which assert that power will be visible flowing through a particular system component. The proofs of these lemmas are derivable from the characteristics of the model, the value of `comp`, and the assumption that there is a single point of failure. Those lemmas for which the proof succeeds correspond to the set of predicted effects.

Consider the example of a failed ignition switch. To select the set of components through which the model predicts power will flow, we set

comp = ignition

and invoke the EHDM *provemodule* command. The results of this command is shown in Figure 4.7. In this case, the model predicted that there would

```
Proof summary for module predict

power_at_star   UNPROVED
power_at_sole   UNPROVED
power_at_plug   UNPROVED
power_at_dist   UNPROVED
power_at_coil   UNPROVED
power_at_poin   UNPROVED
power_at_igni   PROVED
power_at_batt   PROVED

Totals: 8 proofs, 8 attempted, 2 succeeded.
```

Figure 4.7: Result of EHDM ProveModule With comp=ignition

be observable power through the battery and the ignition. To validate the expert system, we simply answer the questions which correspond to a faulty ignition switch. A transcript of the results of the expert system run is shown in Figure 4.8. As can be seen from comparing the transcript of the

```
Does your car fail to start? (Yes or No) yes

Does the starter fail to turn when the key is
engaged? (Yes or No) yes

Are all of the voltage levels of the individual
battery cells normal? (Yes or No) yes

Rule <D1A>
 deduces: (GOOD CELLS)
Rule <D1>
 deduces: (POWER AT BATTERY)

Are there any cracks or damage to the battery
case? (Yes or No) no

Are the battery posts and cable clamps
corroded? (Yes or No) no

Rule <D2A>
 deduces: (GOOD BATTERY)
Rule <D2>
 deduces: (POWER AT IGNITION)

Does a voltmeter connected to the starter post
of the solenoid fail to move when the key is
turned to the start position? (Yes or No) yes

Rule <Y5>
 deduces: (BAD IGNITION)
Diagnosis:  BAD IGNITION.
```

Figure 4.8: Example Transcript From Expert System

expert system session with the results of the formal proof, the model and the expert system share an inverse relationship. The model was able to accurately predict the set of observations consistent with a failed ignition switch, and the expert system was able to correctly deduce that the ignition switch had failed (while making intermediate deductions consistent with the predictions of the model).

It is interesting to note that the expert system may not actually observe all of the predicted good components on its way to diagnosing a failure. For instance, in the case of a failed solenoid, the model's predictions are shown in Figure 4.9. The model predicts that there will be power throughout

```
Proof summary for module predict

power_at_star   UNPROVED
power_at_sole   PROVED
power_at_plug   PROVED
power_at_dist   PROVED
power_at_coil   PROVED
power_at_poin   PROVED
power_at_igni   PROVED
power_at_batt   PROVED

Totals: 8 proofs, 8 attempted, 7 succeeded.
```

Figure 4.9: Result of EHDM ProveModule With comp=solenoid

all system components with the single exception of the starter. The expert system, on the other hand, on its way to finding the correct answer, observed only three of the predicted seven good components, as can been seen from the transcript included in Figure 4.10. This is illustrative of the notion of weaker consistency as explained in Section 2.2.2. In this case, the $\sqsupseteq$ in

$$\forall c \in C, e \in E : \mathcal{P}(e, c) \supset \exists e' : \mathcal{M}(c, e') \wedge e' \sqsupseteq e$$

corresponds to the relation that the observed effects do not contradict the predicted effects and that the predicted effects are a superset of the observed effects.

4.3.5 Observations

Based on this and other similar experiments, we make the following observations and conclusions.

Development of the initial rules based upon heuristics was a tedious and arduous process which left us feeling very unsure about the quality and completeness of the rule base for even so small an example as the one under consideration here. Quite to the contrary, we found generation of diagnostic rules based upon an explicit representation of the model of system behavior to be a relatively straightforward process. Further, because we were working from an explicit model, we felt a much higher degree of confidence concerning the quality of the ensuing rule base. It should be possible to develop mechanical translators which are capable of doing much of the transformation from the logical notation used in expressing models to a rule-based representation, thereby reducing development costs and further increasing confidence that errors were not introduced during the transformation process.

We believe that the notion of model inversion has great potential for application to verification of diagnostic expert systems. While the ideas set forward here are still in a formative stage, the success experienced to date has left us feeling optimistic about the feasibility of developing verification and validation techniques that are both effective and well-grounded in theory.

```
Does your car fail to start? (Yes or No) yes

Does the starter fail to turn when the key is
engaged? (Yes or No) yes

Are all of the voltage levels of the individual
battery cells normal? (Yes or No) yes

Rule <D1A>
 deduces: (GOOD CELLS)
Rule <D1>
 deduces: (POWER AT BATTERY)

Are there any cracks or damage to the battery
case? (Yes or No) no

Are the battery posts and cable clamps corroded?
(Yes or No) no

Rule <D2A>
 deduces: (GOOD BATTERY)
Rule <D2>
 deduces: (POWER AT IGNITION)

Does a voltmeter connected to the starter post of
the solenoid fail to move when the key is turned
to the start position? (Yes or No) no

Does the needle of a voltmeter flicker when the key
is jiggled? (Yes or No) no

Rule <D3A>
 deduces: (GOOD IGNITION)
Rule <D3>
 deduces: (POWER AT POINTS)
Rule <D3>
 deduces: (POWER AT SOLENOID)

Does the starter buzz or turn the engine slowly when
a jumper is connected between the battery and starter
posts of the solenoid? (Yes or No) no

Does the starter show no response when a jumper is
connected between the battery and starter posts of
the solenoid? (Yes or No) yes

Rule <Y9>
 deduces: (BAD SOLENOID)
Diagnosis:  BAD SOLENOID.
```

Figure 4.10: Example Transcript From Expert System

Chapter 5

Conclusions and Recommendations

Formal verification is not a technique to be used in isolation: it should be a component of a comprehensive development methodology based on the systematic application of formal methods. Due to the cost and difficulty of applying such methodologies, they are usually reserved for very critical applications, where assured reliability and safety are paramount, and where simple and robust designs, conducive to formal analysis, may take precedence over functionality. For formal verification to be tractable, and its results credible, formal analysis must be part of the design process from the beginning and must influence the design as it evolves.

These are not attributes of expert system development and knowledge engineering as commonly practiced today. Consequently, formal verification has only limited applicability, and possibly even less utility, in contemporary expert systems development. This is emphatically not to say that formal verification has no part to play in the development of high-quality expert systems, but to point out that much more needs to be done besides development of the formal verification techniques themselves. It will be necessary to revise much of the practice of expert systems engineering, and some of its goals, in order to accommodate requirements for high reliability and predictable behavior assured by formal analysis.

We do not expect—or even advocate—that the field of expert systems should confess error and change its working practices and philosophical foundations. Rather, we suggest the creation of a subdiscipline that will seek to develop and apply, from expert systems engineering, the methods and insights that are applicable and beneficial to systems with exacting dependability requirements. Formal methods and verification should be a significant technical component of that subdiscipline—which should bear a similar relationship to its parent field as the engineering of conventional computer systems for life-critical applications does to the general field of computer and software engineering.

We believe this report makes three contributions to the development of a foundation for the subdiscipline of dependable expert systems:

- The identification of constraint satisfaction and model inversion as formal specification paradigms for synthetic and analytic expert systems, respectively.
- A formal definition of consistency for rule-based expert systems.
- Identification of the notion of approximate semantics, and examples of how these can be achieved and applied in both declarative and imperative forms.

Recommendations for Further Research

We suggest that research and development should continue on the following three levels of increasing rigor and formality:

1. The development of formally based analysis and anomaly detection tools—such as consistency checkers—that can provide useful, but not definitive, help in the construction and quality assurance of conventional expert systems.
2. The development of a methodology and tools for the creation of dependable expert systems, based on formal specifications, run-time checking, and fault-tolerance techniques. Constraint satisfaction and model inversion can provide not only formal specifications for expert systems, but executable tests for the satisfaction of those specifications. By testing the output of an expert system against its specification at run time, and invoking fault-recovery and tolerance mechanisms on failure (e.g., the execution of alternate subsystems or algorithms as in the recovery block method for software fault tolerance), it should be possible to develop highly dependable expert systems for certain, perhaps limited, applications without requiring a complete

change in the development and implementation strategy for the expert systems themselves.

3. The development of a formally-based methodology and tools for expert systems. This will require development of a systematic design methodology for expert systems in which knowledge engineering is combined with the systematic, top-down elaboration of requirements and specifications. It will also require development, for expert systems, of programming languages that are amenable to formal analysis. One slightly radical, but very promising, approach will be to abandon conventionally hand-crafted, rule-based expert systems for certain critical applications, and instead to develop those applications through explicit constraint satisfaction and model inversion techniques. The latter may involve either the exploration of an explicit model at run time, or the "compilation" of that model into an efficient set of production rules. Pearce [44] has demonstrated that this technique can be both more efficient and more reliable than a hand-crafted rule set.

We suggest that exploration of the second and third approaches identified above should be conducted and evaluated through modest prototyping experiments using potential applications of expert systems of interest to NASA and Space Station planners. Fault diagnosis (analysis) and scheduling (synthesis) seem to be the most promising applications, from both the scientific and practical points of view.

Appendix A

P-Best

The Production-Based Expert System Toolset (P-BEST) is a forward-chaining, LISP-based expert system development environment (sometimes referred to as a *shell*), which consists of a rule-development language, rule compiler, run time routines, and a debugger. It provides an integrated set of facilities for the creation and debugging of complex knowledge bases. Although developed primarily as a research vehicle in the exploration of expert systems, its features and facilities have been heavily influenced by the suggestions, critiques and demands of individuals involved in the applications of expert systems to real-world problems. P-BEST is intended to be portable, yet reasonably efficient in executing complex rules against large data sets.

A.1 How P-BEST Works

The basic P-BEST strategy revolves around maintenance of an expert system state. The system state is represented by the union of the states of the *fact base*, the *rule base*, and any external data structures unique to a given application. This union is referred to as the *knowledge base* (KB). Rules (*IF*... *THEN* ... pairs) are developed by the user and compiled by P-BEST into the rule base. A very simple rule might say something like:

```
if  is-a-man socrates,
then
   is-mortal socrates.
```

Once the rules are developed, the user may introduce facts (knowledge) into the KB through assertions. For example, a user might wish to assert that

```
is-a-man socrates
```

The P-BEST inference engine is a distributed mechanism for matching facts to rule antecedent patterns, performing binding analysis, and invoking rule consequents. It is distributed in the sense that for each rule in the rule base, the compiler will produce a number of primitive LISP functions that implement the ***semantics*** of the rule. As facts are asserted, each rule in the rule base that references the type of knowledge being asserted examines the fact for a potential match. When a pattern in the antecedent of a rule matches the fact being asserted, a binding is created and associated with both the fact and the rule. If such a binding is created, and if, as a result of this binding, all the patterns of a rule have been matched, then binding analysis is performed to determine if all of the variable values referenced by the rule/fact bindings are consistent.

Given the example rule stated above, if the fact were asserted

```
is-a-man socrates
```

then the rule would determine that its antecedent had been satisfied. Because the sample does not specify any additional constraints, the rule would pass consistency analysis and conclude that

```
is-mortal socrates
```

Of course, if the rule stated something like

```
      if  is-a-man socrates and
          is-alive socrates
      then
          is-mortal socrates
```

then asserting that `is-a-man socrates` would not be sufficient to complete the conditions of the rule.

Most expert systems will consist of a large number of rules and require the assertion of a substantial number of facts to reach meaningful conclusions. Under P-BEST, when a fact is asserted and matched against all candidate rules, the resulting set of rules flagged as complete (all patterns in the antecedent are bound to some fact) are examined to determine if the there is a valid binding combination. This second level of analysis which deals with value constraints is called ***binding analysis***. In general, variable values are examined and any additional ***test*** constraints are evaluated to determine if there is a consistent interpretation of the binding set that would allow a rule to reach its conclusion (or *fire*). The rules with rule/fact bindings that meet binding analysis are gathered into a conflict set, and the "best" rule in this set is selected and fired. The process of selecting a rule to fire from a number of possibilities is called *conflict resolution*. The consequent of the selected rule will fire and generally alter the state of the knowledge base, which will cause the creation of other rule/fact bindings. The process then repeats until no further candidates for firing remain. It is up to the individual application to assign an interpretation to this state: in some applications, exhausting all possible inferences might be considered the natural culmination of processing, while in other applications, reaching such a state might be considered an error.

A.1.1 Facts

Facts in the P-BEST system may be thought of as predicates asserted into the fact base. At the lowest level, they are LISP structures whose format is described by a user-provided definition statement. A fact's type is called its *ptype* (or *pattern-type*), and thus the function for defining fact types is called "define-ptype." A sample ptype definition might be:

```
(define-ptype is-mortal person).
```

This ptype, `is-mortal`, allows the assertion of facts into the system whose `person` field could contain something like `socrates`, or `reagan` or `5`. It is necessary to note the difference between syntactic constraint, which the system enforces, and semantic constraint, which is the responsibility of the knowledge engineer. In the above example, `is-mortal` is a syntactic structure that takes a single argument: this is the extent of the restrictions enforced by the system. Semantically, the intent of this predicate is that the value occupying the `person` field is a `mortal`. Asserting, for instance, that `(is-mortal air)` or `(is-mortal 5)` is syntactically correct, but is of dubious semantic worth.

Although facts are generally asserted into the knowledge base by rules, a user can directly alter the state of the knowledge base. The act of asserting a fact is handled by the function `pbest-assert`. The function `pbest-assert` allows the assertion of a single fact into the Fact Base. A fact assertion might look like this:

```
(pbest-assert '(is-mortal socrates))
```

$$\begin{tikzcd}
FB: & {[f1]} \arrow[r, Rightarrow] \arrow[d] & {[f2]} \arrow[r, Rightarrow] \arrow[d] & {[f3]} \arrow[r, Rightarrow] \arrow[d] & {[fn]} \arrow[d] \\
 & (R_xP_{x1}) \arrow[d] & (R_yP_{y1}) \arrow[d] & (R_zP_{z1}) \arrow[d] & nil \\
 & (R_xP_{x2}) \arrow[d] & nil & (R_xP_{x3}) \arrow[d] & \\
 & nil & & (R_yP_{y1}) \arrow[d] & \\
 & & & nil &
\end{tikzcd}$$

$$\begin{tikzcd}
 & C & C & nil \\
RB: & {[R_x]} \arrow[u] \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {[R_y]} \arrow[u] \arrow[r, Rightarrow] \arrow[d, Rightarrow] & {[R_z]} \arrow[u] \arrow[d, Rightarrow] \\
 & \begin{array}{c} P_{x1}B : f1 \\ P_{x2}B : f1 \\ P_{x3}B : f3 \end{array} & \begin{array}{c} P_{y1}B : f2, f3 \\ \neg P_{y2}B : nil \end{array} & \begin{array}{c} P_{z1}B : f3 \\ P_{z2}B : nil \end{array}
\end{tikzcd}$$

Figure A.1: Knowledge Base

This would result in the assertion of a fact (or predicate) of type `is-mortal`, whose attribute is `socrates`. The fact would be added to the global knowledge base, and appropriate rules would examine the fact in an attempt to bind to it.

A.1.2 Rule/Fact Bindings

What does it mean for rules and facts to be bound? When a fact is asserted into the fact base, all rules pertaining to that fact (i.e., which reference facts of a particular p-type) create links to it. These links are referred to as *bindings*. Consider the symbolic examples of Figure A.1, which represents the state of the system after three rules (R_x, R_y, R_z) have been added to the rule base, and three or more facts (f1, f2, f3,...fn) have been asserted to the fact base.

In this example, (where R=rule, P=pattern, and B=binding) rules R_x and R_y are candidates for binding analysis to determine if some combination of the current bindings constitutes a consistent binding context (i.e., all repeated variables have the same value, and all variable tests and restrictions are satisfied). You will notice that R_x is not a candidate for binding analysis because R_xP_{x2} is not bound to any fact. Therefore, the COMPLETE flag (represented by the vertical "C") is not set. R_y, on the other hand, will be considered in binding analysis because $\neg R_yP_{y2}$, which is a not-pattern, has no bindings; all other patterns in R_y have bindings.

What does it mean for a binding context to be consistent? Briefly, P-BEST checks each possible combination of the rule patterns and all matching facts until it finds a set of facts that are consistent. So, if a rule has two patterns, the first pattern having four facts that match it, and the second pattern having five facts that match it, then P-BEST may have to consider all possible pairs[1]. In this case, there are potentially 20 different comparisons to establish a consistent set of facts.

A.2 P-BEST Components

P-BEST consists of a production-rule language, a compiler, and a window debugger. These facilities are combined into an integrated environment for the creation and maintenance of knowledge-based systems.

A.2.1 The Language

The heart of any expert system shell is its facility for capturing knowledge. The P-BEST language is very expressive, allowing the knowledge engineer to capture complex relationships in a concise form, yet the syntax is constrained English, thereby greatly increasing the readability (and therefore, the maintainability) of P-BEST rules over other representations.

A.2.2 The Compiler

The implementation of P-BEST optimizes execution speed and avoids the use of an extensive run-time support package in favor of direct translation of the production rules into complete, independently compiled functions. Various run-time control logic and auxiliary functions are provided to support the execution of the rules, but the real heart of a P-BEST-based expert system is the generated functions that represent the rules themselves.

It is perhaps misleading to say that P-BEST contains a compiler; it does not. P-BEST contains a translator that converts the rules from its

[1]P-BEST's optimized search strategy allows it to terminate its search upon finding the first consistent binding context

own representation language to CommonLisp functions. P-BEST then depends upon the operation of the target system's CommonLisp compiler to complete the process of generating object code. This is all transparent to the knowledge engineer, however.

Each production rule is translated into three main functions that implement the semantic intent of the rule: a function for examining newly asserted facts and creating proper rule/fact bindings; a function for analyzing binding relationships and determining consistency; and a function that implements the results, or consequent, of the rule.

A.2.3 The Debugger

The expert system development environment presented by P-BEST is greatly enhanced by the addition of a windowing debugger. Although this debugger is not intended to be used as part of a completed expert system, it is very useful during the creation and debugging of the rule base.

The debugger provides multiple simultaneous views into the state of the knowledge base. In one window, the set of currently asserted facts is displayed along with binding information. In another window, the rules are displayed. The user is provided with numerous operations, such as commands to step through the execution of the rule base, commands to alter system state through assertion or negation of facts, and commands to activate tracing features.

Appendix B

Ehdm

The EHDM verification system was constructed at SRI International under the direction of Friedrich von Henke[1, 2, 3, 55]. The following sections provide a brief overview of some of the features of the EHDM methodology.

B.1 The EHDM Specification Language and Logic

This section provides a brief overview of the specification language of EHDM and the underlying logic.

The specification language is based on first-order typed predicate calculus, but also includes elements of richer logics, such as higher-order logic [49], lambda-calculus [7] and Hoare logic [29], for greater expressiveness. For example, higher-order terms are particularly useful for expressing induction schemas and requirements.

The specification language is strongly-typed; all entities must be declared with their type before use. The type system includes subtypes and function types. Specifications are written as definitions and formulas (axioms, theorems, and lemmas). In addition to the standard expressions, the language provides the Boolean connectives, polymorphic conditionals, and quantified expressions, including quantification over functions .

A sublanguage is included for modeling operational behavior and imperative programs, based on the notions of *state object* and *operation*. State objects correspond to "program variables" in programming languages. Operations express state transformations; they have an effect on state objects by possibly changing their values.

The sublanguage also includes constructs for composing operation expressions; these correspond to the common control structures of programming languages. The combination of all these features forms a sublanguage that is essentially equivalent to a simple subset of the Ada programming language. The semantics of operations are defined by Hoare formulas, which express properties of the states before and after the state transformation denoted by the operation.

Specifications are organized around the concept of parameterized modules. Modules are closed scopes with explicit importation and exportation of names; modules can be nested. Names are made unique by qualifying them with the name of their module of origin. Modules may be parameterized by types, constants, and functions. Semantic assumptions or constraints on module parameters can be expressed; these entail an obligation that must be justified for each module instantiation. This form of module parameterization is very general and powerful; it supports generic specifications and allows many complex constructs to be built from simple language primitives. Modules are the basic building blocks of specifications. A module may represent the theory describing a specification concept, an abstract data type, an abstract state machine, or an (abstract) program.

An important aspect of the EHDM language (and the EHDM approach in general) is the support of hierarchical development of specifications and proofs. The language supports hierarchical structuring of specifications, both with respect to composition of modules ("horizontal hierarchy") and levels of abstractions and refinements ("vertical hierarchy"). Vertical links between modules at adjoining abstraction levels are established by mappings, which generalize the notion of implementation.

The language has been designed so that it naturally supports a high degree of reusability of specifications and proofs. Reusability is also enhanced by the library facility described later.

B.2 The Theorem Prover of EHDM

The theorem-prover component of EHDM combines powerful heuristics for mechanically generating proofs in first-order predicate logic with efficient decision procedures for the following standard theories:

- Ground formulas in propositional calculus
- Equality over uninterpreted function symbols [50]

- Presburger arithmetic, that is, linear arithmetic with the usual ordering relations [51]

Equational reasoning similar to the use of rewrite rules is specially supported by the mechanized proof procedure. The prover also implements the main reduction rules of lambda calculus and a fragment of higher-order logic; however, the exact extent of this support is currently unclear.

The system supports both automated proof generation and interactive proof construction that depends on user guidance. Proofs (more precisely, proof steps) are declared in the proof part of a specification module; they are expressed as a conclusion to be proven and a list of formulas (axioms and lemmas) from which the conclusion can be deduced. The automated prover completes a proof by attempting to construct suitable instances of the formulas involved; the user can help in this process by providing some substitutions for free variables, either directly in the proof declaration or during proof construction. Completed proofs can be captured in augmented proof declarations and included in the specification text for later "replay." A proof-chain analysis tool checks for completeness of larger proof trees and helps keep track of dependencies.

A special procedure for reasoning about state transformations has built-in knowledge of the meaning of Hoare formulas and the constructs for expressing state objects and state transformations. This procedure provides the main support for code-level verification. It permits users to reason directly with Hoare formulas, without the traditional intermediate step of translating annotated programs into verification conditions (VCs); the procedure also supports reasoning about program fragments, as opposed to complete programs units (like subprograms). In these respects, the paradigm that is implemented by the procedure is more general than the traditional "verification condition generator" (VCG) paradigm. However, the equivalent of a VCG is available in the EHDM environment as a proof development tool.

B.3 The EHDM Environment

The EHDM environment is implemented as an integrated, interactive system that supports all activities involved in creating, analyzing, modifying, managing, and documenting specification modules and proofs. The standard user interface of EHDM uses the bit-map display and combines a display-oriented text editor (customized and enhanced EMACS) with multiple windows, menus, and mouse input. (A less enhanced editor-based interface is available for remote operation.) All operations can be invoked directly from the editor, including the basic operations of parsing, prettyprinting, and typechecking specification text, invoking the theorem prover, and requesting status information. In addition to the basic operations just mentioned, the system provides a number of further support tools, including: the Context and Library tools, the configuration control support, and the MLS Checker and the EHDM–to–Ada translator (described in [3]).

B.3.1 The Context and Library Manager

The system maintains an internal data base for keeping track of the state of individual modules and proofs (referred to as the *working context*), and of the interdependences among modules and libraries; the user can manipulate this working context or switch between contexts. Modules are the basic entities around which the EHDM system is organized, and the context feature virtually insulates the user from the underlying file system; the EHDM system creates and manages 'internal' files, which the user can ignore completely because all file manipulation happens as a side effect of the user interaction with the EHDM system.

The library mechanism permits users to group together standard modules in libraries of reusable concepts, theorems, and proofs. The environment offers tools for creating and maintaining module libraries and supports sharing of libraries among users and projects.

Contexts and libraries have been designed so that the novice user can completely ignore these facilities. A user always works within a context, but when a user starts EHDM for the first time, the system automatically establishes a working context; later, when the user leaves EHDM the system saves the context and restores it when work is resumed. Users need to know about contexts only when they want to work in more than one context; similarly, they can ignore libraries until they want to make use of that facility.

B.3.2 The Configuration Control Tool

A configuration and version control mechanism ensures that consistent versions of modules are used, and status checks report on the status of modules and proofs, and on dependencies among modules and proofs. At any given time, a module specification may be in one of several states. When a module is typechecked and also when a proof is performed, a "version check" is made to see that the typecheck information recorded for the transitive closure of all referenced modules is still valid. For example, if module A uses module B, then changes to module B will invalidate the typecheck information recorded for module A. This will be discovered when the typecheck information for module A is used during the typechecking of a module that uses A, or during the construction of a proof from A or a module that uses A.

Appendix C

Tic-Tac-Toe Rule base

```
;; Each position in the 3x3 board is represented in working
;;  memory by a ''position'' assertion of the following form:

(define-ptype position row col marker)

;; where row,col are integers in the range 1-3 and marker is an
;; element of the set free, opp, com.
;;
;;
;; A fact is asserted to keep track of whose turn it is
;;

(define-ptype turn player count)

;; where player is an element of the set opp, com, and count is
;; of type nat.
;;
;; The initial state consists of 9 position assertions and 1 turn
;; assertion.

;;
;; A fact is asserted as the result of a player move, prior to
;; altering the board position, so that it may be checked.
```

```
(define-ptype opp_move row col)

;; where row and column are integers in the range of 1-3 which
;; must correspond to a free position on the board.
;;
;;
;; The following macros are defined for convenience:
;;
(defmacro cornerp (x y) '(and (oddp ,x)(oddp ,y)))
(defmacro middlep (x y) '(or (and (oddp ,x)(evenp ,y))
                             (and (evenp ,x)(oddp ,y))))
(defmacro centerp (x y) '(and (equal ,x 2)(equal ,y 2)))
;;
;;




(defrule blk_cc states
  if there exists a turn called t1
        such that player is com and
                  count is 3 and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is opp and
     there exists a position
        such that row is ?r2 and
                  col is ?c2 and
                  marker is opp
        with (and (not (equal ?r1 ?r2))
                  (not (equal ?c1 ?c2))
                  (cornerp ?r1 ?c1)
                  (cornerp ?r2 ?c2))
     there exists a position
        such that row is 2 and
                  col is 2 and
                  marker is com and
```

```
      there exists a position called p1
         such that row is ?r3 and
                   col is ?c3 and
                   marker is free
         with (middlep ?r3 ?c3)
   then forget t1 and
        forget p1 and
        remember a position
           which inherits from p1
              except that marker is com and
        remember a turn
           such that player is opp and
                     count is 4)
```

```
(defrule blk_cm states
   if there exists a turn called t1
         such that player is com and
                   count is 3 and
      there exists a position
         such that row is ?r1 and
                   col is ?c1 and
                   marker is opp and
      there exists a position
         such that row is ?r2 and
                   col is ?c2 and
                   marker is opp
         with (and (not (equal ?r1 ?r2))
                   (not (equal ?c1 ?c2))
                   (cornerp ?r1 ?c1)
                   (middlep ?r2 ?c2))
      there exists a position
         such that row is 2 and
                   col is 2 and
                   marker is com and
      there exists a position called p1
```

```
        such that row is ?r3 and
                  col is ?c3 and
                  marker is free
        with (and (and (oddp ?r3)  ;; take corner which intersects
                       (oddp ?c3)) ;; both of opp's markers
                  (or (and (equal ?r3 ?r1)
                           (equal ?c3 ?c2))
                      (and (equal ?r3 ?r2)
                           (equal ?c3 ?c1))))
  then forget t1 and
       forget p1 and
       remember a position
          which inherits from p1
             except that marker is com and
       remember a turn
          such that player is opp and
                    count is 4)




(defrule win_col at rank 1 states
  if there exists a turn called t1
        such that player is com and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is com
     there exists a position
        such that row is ?r2 and
                  col is ?c2 and
                  marker is com
        with (and (not (equal ?r1 ?r2))
                  (equal ?c1 ?c2)) and
     there exists a position called pos
        such that col is ?c3 and
                  marker is free
        with (equal ?c2 ?c3)
     then
        forget t1 and
        forget pos and
        remember a position
           which inherits from pos
              except that marker is com)




(defrule win_row  at rank 1 states
  if there exists a turn called t1
        such that player is com and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is com
     there exists a position
        such that row is ?r2 and
                  col is ?c2 and
                  marker is com
        with (and (equal ?r1 ?r2)
                  (not (equal ?c1 ?c2))) and
     there exists a position called pos
        such that row is ?r3 and
                  marker is free
        with (equal ?r2 ?r3)
     then
        forget pos and
        forget t1 and
        remember a position
           which inherits from pos
              except that marker is com)
```

```
(defrule win_dia at rank 1 states
  if there exists a turn called t1
        such that player is com and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is com
     there exists a position
        such that row is ?r2 and
                  col is ?c2 and
                  marker is com
        with (and (not (equal ?r1 ?r2))
                  (not (equal ?c1 ?c2))
                  (equal (abs (-  ?r1 ?r2))
                         (abs (-  ?c1 ?c2)))) and
     there exists a position called pos
        such that row is ?r3 and
                  col is ?c3 and
                  marker is free
        with (and (equal (abs (- ?r1 ?r3))
                         (abs (- ?c1 ?c3)))
                  (equal (abs (- ?r2 ?r3))
                         (abs (- ?c2 ?c3))))
     then
        forget pos and
        forget t1 and
        remember a position
           which inherits from pos
              except that marker is com)



(defrule blk_col states
  if there exists a turn called t1
        such that player is com and
                  count is ?x and
     there exists a position
```

```
      such that row is ?r1 and
                col is ?c1 and
                marker is opp
   there exists a position
      such that row is ?r2 and
                col is ?c2 and
                marker is opp
      with (and (not (equal ?r1 ?r2))
                (equal ?c1 ?c2)) and
   there exists a position called pos
      such that col is ?c3 and
                marker is free
      with (equal ?c2 ?c3)
   then
      forget pos and
      forget t1 and
      remember a position
         which inherits from pos
            except that marker is com and
      remember a turn
         such that player is opp and
                   count equals (1+ ?x))
```

```
(defrule blk_row states
  if there exists a turn called t1
        such that player is com and
                  count is ?x and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is opp
     there exists a position
        such that row is ?r2 and
                  col is ?c2 and
                  marker is opp
```

```
         with (and (equal ?r1 ?r2)
                   (not (equal ?c1 ?c2))) and
      there exists a position called pos
         such that row is ?r3 and
                   marker is free
         with (equal ?r2 ?r3)
      then
         forget pos and
         forget t1 and
         remember a position
            which inherits from pos
               except that marker is com and
         remember a turn
            such that player is opp and
                      count equals (1+ ?x))
```

```
(defrule blk_dia states
  if there exists a turn called t1
         such that player is com and
                   count is ?x and
      there exists a position
         such that row is ?r1 and
                   col is ?c1 and
                   marker is opp
      there exists a position
         such that row is ?r2 and
                   col is ?c2 and
                   marker is opp
         with (and (not (equal ?r1 ?r2))
                   (not (equal ?c1 ?c2))
                   (equal (abs (-  ?r1 ?r2))
                          (abs (-  ?c1 ?c2)))) and
      there exists a position called pos
         such that row is ?r3 and
                   col is ?c3 and
```

```
                  marker is free
        with (and (equal (abs (- ?r1 ?r3))
                         (abs (- ?c1 ?c3)))
                  (equal (abs (- ?r2 ?r3))
                         (abs (- ?c2 ?c3))))
     then
        forget pos and
        forget t1 and
        remember a position
           which inherits from pos
              except that marker is com and
        remember a turn
           such that player is opp and
                     count equals (1+ ?x))




(defrule see_won_game at rank 2 states
  if there exists a turn called t1 and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is ?m
           with (not (equal ?m 'free)) and
     there exists a position
        such that row is ?r2 and
                  col is ?c2 and
                  marker is ?m
           with (or (not (equal ?r2 ?r1))
                    (not (equal ?c2 ?c1))) and
     there exists a position
        such that row is ?r3 and
                  col is ?c3 and
                  marker is ?m and
           with (and (or (not (equal ?r3 ?r2))
                         (not (equal ?c3 ?c2)))
                     (or (not (equal ?r3 ?r1))
                        (not (equal ?c3 ?c1)))) and
    test (or (and (equal ?r1 ?r2)
                  (equal ?r1 ?r3))
             (and (equal ?c1 ?c2)
                  (equal ?c1 ?c3))
             (and (equal (abs (- ?r1 ?r3))
                         (abs (- ?c1 ?c3)))
                  (equal (abs (- ?r2 ?r3))
                         (abs (- ?c2 ?c3)))
                  (not (or (equal ?r1 ?r2)
                           (equal ?c1 ?c2))))))
  then forget t1 and
       execute (draw-board) and
       execute (format t "~%Player ~A Won The Game!!!~%" ?m))



(defrule see_tie_game at rank 2 states
  if there exists a turn called t1
        such that player is ?p and
     there does not exist a position
        such that marker is free
  then forget t1 and
       execute (format t "~%Tie Game!!!~%"))



(defrule make_random_move at rank -10 states
  if there exists a turn called t1
        such that player is com and
                  count is ?x and
     there exists a position called p1
        such that marker is free
  then forget p1 and
       forget t1 and
```

```
          remember a position
             which inherits from p1
                except that marker is com and
          remember a turn
             such that player is opp and
                       count equals (1+ ?x))



(defrule get_opp_move states
  if there exists a turn
        such that player is opp and
     there exists a position
        such that marker is free and
     there does not exist an opp_move
  then execute (draw-board) and
       remember an opp_move
          such that row equals (get-ans 'row) and
          such that col equals (get-ans 'col))



(defrule do_opp_move states
  if there exists a turn called t1
        such that player is opp and
                  count is ?x and
     there exists an opp_move called m1
        such that row is ?r1 and
                  col is ?c1 and
     there exists a position called p1
        such that row is ?r1 and
                  col is ?c1 and
                  marker is free
  then forget t1 and
       forget m1 and
       forget p1 and
       remember a position
          which inherits from p1
             except that marker is opp and
       remember a turn
          such that player is com and
                    count equals (1+ ?x))
```

```
(defrule bad_opp_move states
  if there exists a turn
        such that player is opp and
     there exists an opp_move called m1
        such that row is ?r1 and
                  col is ?c1 and
     there exists a position
        such that row is ?r1 and
                  col is ?c1 and
                  marker is ?m with (not (equal ?m 'free))
  then forget m1 and
       execute (format t "~%That position is already taken!~%"))
```

```
(defun init-ttt ()
  (progn
    (pest-reset)
    (pest-assert '(position 3 2 free))
    (pest-assert '(position 2 3 free))
    (pest-assert '(position 2 1 free))
    (pest-assert '(position 1 2 free))
    (pest-assert '(position 3 3 free))
    (pest-assert '(position 3 1 free))
    (pest-assert '(position 1 3 free))
    (pest-assert '(position 1 1 free))
    (pest-assert '(position 2 2 free))
    (pest-assert '(turn opp 0))))
```

```
(defun get-ans (label)
  (do ((x 0))
      ((and  (> x 0)(< x 4)) x)
      (format t "~%Enter ~A (1-3): " label)
      (setf x (read))))
```

```
(defun draw-board ()
  (format t "~%r
c: 1 : 2 : 3  ~%")
  (do ((r 1)(c 1 (1+ c)))
      ((and (= r 3)(> c 3)))
      (if (> c 3)
          (progn (setf c 1)
                 (incf r)
                 (format t "~%    ---+---+---~%")))
      (if (= c 1) (format t " ~A :" r))
      (format t " ~A " (print-marker r c))
      (if (< c 3) (format t "|"))))
```

```
(defun print-marker (row col)
  (dolist (f *facts*)
          (if (and (position-p (fact-body (eval f)))
                   (equal (position-row (fact-body (eval f))) row)
```

```
     (equal (position-col (fact-body (eval f))) col))
(if (equal (position-marker (fact-body (eval f))) 'opp)
    (return 'O)
  (if (equal (position-marker (fact-body (eval f))) 'com)
      (return 'X)
    (return #\Space)))))
```

Appendix D

Tic-Tac-Toe Specifications

arrays: **Module** [dtype: TYPE, undef_r: dtype]

Exporting array, newarray, assign, ⋆1[⋆1], length, dtype

Theory

array: TYPE
i, j: VAR nat
d: VAR dtype
a: VAR array
newarray: array
assign: function[array, nat, dtype $\rightarrow$ array]
⋆1[⋆1]: function[array, nat $\rightarrow$ dtype]
length: function[array $\rightarrow$ nat]

acessnew: **Axiom** newarray[i] = undef_r

selectassign: **Axiom** $a := i[j] =$ **if** $i = j$ **then** d **else** $a[j]$ **end if**

und_ax: **Axiom** ($i \leq 0 \vee i >$ length(a)) $\supset a[i] =$ undef_r

End arrays

data: **Module**

Exporting datum, undef, wild

Theory

datum: TYPE IS int
undef, wild: datum

dax1: **Axiom** undef $\neq$ wild

End data

diffs: **Module**

Exporting $| \star - \star |$, oddp, evenp

Theory

n, m: VAR nat
$| \star - \star |$: function[nat, nat $\rightarrow$ nat]
oddp: function[int $\rightarrow$ bool]
evenp: function[int $\rightarrow$ bool]

diff_ax: **Axiom** $| n - m | =$ **if** $n > m$ **then** $n - m$ **else** $m - n$ **end if**

```
oddp_ax: Axiom
    oddp(n) = if n = 1
        then true
       ELSIF n > 1 then oddp(n - 2) else false
        end if
```

```
evenp_ax: Axiom
    evenp(n) = if n = 0
        then true
       ELSIF n > 1 then evenp(n - 2) else false
        end if
```

nat_def_ax: **Axiom** $n \geq 0$

Proof

diff_lemma: **Lemma** $| n - m | \geq 0$

dlp: **Prove** diff_lemma **from**
    diff_ax {n ← n@cs, m ← m@cs},
    nat_def_ax {n ← n@CS},
    nat_def_ax {n ← m@cs}

End diffs

```
facts: Module
Using data, arrays[datum, undef]
Exporting fact with data, arrays[datum, undef]
Theory
fact: TYPE from array
End facts
```

sets: **Module** [t: TYPE]

Exporting set, $\emptyset$, UNIVERSE, $\{\star 1\}$, $\in$, $\subseteq$, $\subset$, $\cap$, $\cup$, $\setminus$, $\overline{\star 1}$,
sets.lesseq, sets.less, sets.times, sets.plus, sets.difference,
sets.minus, seteq

Theory

set: TYPE
$\emptyset$: set
UNIVERSE: set
S1, S2, S3: VAR set
x, y: VAR t
$\{\star 1\}$: function[t $\rightarrow$ set]
$\in$: function[t, set $\rightarrow$ bool]
$\subseteq$: function[set, set $\rightarrow$ bool]
$\subset$: function[set, set $\rightarrow$ bool]
$\cap$: function[set, set $\rightarrow$ set]
$\cup$: function[set, set $\rightarrow$ set]
$\setminus$: function[set, set $\rightarrow$ set]
$\overline{\star 1}$: function[set $\rightarrow$ set]
seteq: function[set, set $\rightarrow$ bool]

SETdef: **Axiom** $(x \in \{x\}) \wedge ((y \in \{x\}) \Leftrightarrow x = y)$

subdef: **Axiom** $(\mathrm{S1} \subseteq \mathrm{S2}) \Leftrightarrow (\forall \mathrm{x}: (x \in \mathrm{S1}) \supset (x \in \mathrm{S2}))$

psubdef: **Axiom**
$(\mathrm{S1} \subset \mathrm{S2}) \Leftrightarrow (((x \in \mathrm{S1}) \supset (x \in \mathrm{S2})) \wedge (\exists \mathrm{y}: (y \in \mathrm{S2}) \wedge \neg(y \in \mathrm{S1})))$

interdef: **Axiom** $(x \in (\mathrm{S1} \cap \mathrm{S2})) \Leftrightarrow ((x \in \mathrm{S1}) \wedge (x \in \mathrm{S2}))$

uniondef: **Axiom** $(x \in (\mathrm{S1} \cup \mathrm{S2})) \Leftrightarrow ((x \in \mathrm{S1}) \vee (x \in \mathrm{S2}))$

diffdef: **Axiom** $(x \in (\mathrm{S1} \setminus \mathrm{S2})) \Leftrightarrow ((x \in \mathrm{S1}) \wedge \neg(x \in \mathrm{S2}))$

compdef: **Axiom** $(x \in \overline{\mathrm{S1}}) \Leftrightarrow (\neg(x \in \mathrm{S1}))$

eqdef: **Axiom** $\mathrm{S1} = \mathrm{S2} \Leftrightarrow$ seteq(S1, S2)

eq_r: **Formula** seteq(S1, S1)

eq_s: **Formula** seteq(S1, S2) $\Leftrightarrow$ seteq(S2, S1)

eq_t: **Formula** (seteq(S1, S2) $\wedge$ seteq(S2, S3)) $\supset$ seteq(S1, S3) sb_t: **Formula** $((S1 \subseteq S2) \wedge (S2 \subseteq S3)) \supset (S1 \subseteq S3)$

seteqdef: **Axiom**
seteq(S1, S2) $\Leftrightarrow (\forall x: ((x \in S1) \supset (x \in S2)) \wedge ((x \in S2) \supset (x \in S1)))$

nulldef: **Axiom** $(x \in \emptyset) = \text{false}$

nulldef2: **Axiom** $(\forall x: \neg(x \in S1)) \Leftrightarrow S1 = \emptyset$

unidef: **Axiom** $(x \in \text{UNIVERSE}) = \text{true}$

sub_reflex: **Formula** $(S1 \subseteq S1) = \text{true}$

inter_identity: **Formula** $S1 = (S1 \cap S1)$

inter_commutative: **Formula** $(S1 \cap S2) = (S2 \cap S1)$

inter_null: **Formula** $(S1 \cap \emptyset) = \emptyset$

inter_assoc: **Formula** $(S1 \cap (S2 \cap S3)) = ((S1 \cap S2) \cap S3)$

union_commutative: **Formula** $(S1 \cup S2) = (S2 \cup S1)$

union_associative: **Formula** $(S1 \cup (S2 \cup S3)) = ((S1 \cup S2) \cup S3)$

union_identity: **Formula** $(S1 \cup S1) = S1$

union_null: **Formula** $(S1 \cup \emptyset) = S1$

double_neg: **Formula** $\overline{\overline{S1}} = S1$

neg_null: **Formula** $\overline{\emptyset} = \text{UNIVERSE}$

neg_uni: **Formula** $\overline{\text{UNIVERSE}} = \emptyset$

uni_union: **Formula** $(S1 \cup \overline{S1}) = \text{UNIVERSE}$

null_inter: **Formula** $(S1 \cap \overline{S1}) = \emptyset$

diff_inter: **Formula** $(S1 \setminus S2) = (S1 \cap \overline{S2})$

End sets

states: **Module**

Exporting state, s0, next

Theory

state: TYPE IS nat
s0: state $= 0$
s_1, s_2: VAR state
next: function[state $\rightarrow$ state] $= (\lambda\, s_1 \rightarrow \text{state} : (s_1 + 1))$

nonneg: **Axiom** $s_1 \geq \text{s0}$

succ: **Axiom** $s_1 > \text{s0} \supset (\exists\, s_2 : \text{next}(s_2) = s_1)$

End states

ttt_ptypes: **Module**

Using facts, states, kbs, sets[fact], diffs

Exporting row, col, marker, player, count, turn, opp_move, position, turn_p, opp_move_p, position_p, ttt_ptypes.equals, opp, com, free, match_turn, match_opp_move, match_position, akb, cornerp, middlep, centerp

Theory

```
row: nat = 1
col: nat = 2
marker: nat = 3
player: nat = 1
count: nat = 2
opp, com, free: datum
turn, opp_move, position: TYPE from fact
f: VAR fact
t, t1: VAR turn
o: VAR opp_move
p, p1, p2, p3, p4, p5, p6, p7, p8, p9: VAR position
d1, d2, d3: VAR datum
rx, cx, mx, px: VAR datum
kbx: VAR kb
sx: VAR state
akb: kb
turn_p: function[fact → bool]
match_turn: function[kb, state, fact, datum, datum → bool] =
    ( λ kbx, sx, f, d1, d2→ bool :
        (f ∈ kb_inst(kbx, sx))
          ∧ turn_p(f)
            ∧ (d1 = wild ∨ d1 = f[player]) ∧ (d2 = wild ∨ d2 = f[count]))

opp_move_p: function[fact → bool]
match_opp_move: function[kb, state, fact, datum, datum → bool] =
    ( λ kbx, sx, f, d1, d2→ bool :
        (f ∈ kb_inst(kbx, sx))
          ∧ opp_move_p(f)
            ∧ (d1 = wild ∨ d1 = f[row]) ∧ (d2 = wild ∨ d2 = f[col]))
```

```
position_p: function[fact → bool]
match_position: function[kb, state, fact, datum, datum, datum → bool]
        = ( λ kbx, sx, f, d1, d2, d3→ bool :
            (f ∈ kb_inst(kbx, sx))
              ∧ position_p(f)
                ∧ (d1 = wild ∨ d1 = f[row])
                  ∧ (d2 = wild ∨ d2 = f[col]) ∧ (d3 = wild ∨ d3 = f[marker]))

equals: function[position, position → bool] =
      ( λ p1 , p2 → bool :
          p1[row] = p2[row]
            ∧ p1[col] = p2[col] ∧ p1[marker] = p2[marker])

cornerp: function[fact → bool] =
      ( λ f→ bool : position_p(f) ∧ oddp(f[row]) ∧ oddp(f[col]))
centerp: function[fact → bool] =
      ( λ f→ bool : position_p(f) ∧ f[row] = 2 ∧ f[col] = 2)
middlep: function[fact → bool] =
      ( λ f→ bool :
          position_p(f)
            ∧ ((evenp(f[row]) ∧ oddp(f[col]))
                  ∨ (oddp(f[row]) ∧ evenp(f[col]))))
```

distinct_constants: **Axiom**
$\quad$ opp ≠ com ∧ com ≠ free
$\qquad$ ∧ opp ≠ free ∧ opp ≠ wild ∧ com ≠ wild ∧ free ≠ wild

trn_lth_ax: **Axiom** length(t) = 2

opp_lth_ax: **Axiom** length(o) = 2

pos_lth_ax: **Axiom** length(p) = 3

fact_type_1: **Axiom** turn_p(f) ∨ opp_move_p(f) ∨ position_p(f)

fact_type_2: **Axiom** turn_p(f) ⊃ ¬(opp_move_p(f) ∨ position_p(f))

fact_type_3: **Axiom** opp_move_p(f) ⊃ ¬(turn_p(f) ∨ position_p(f))

fact_type_4: **Axiom** position_p(f) ⊃ ¬(opp_move_p(f) ∨ turn_p(f))

fact_type_5: **Axiom** turn_p(t) $\wedge \neg$(turn_p(o) $\vee$ turn_p(p))

fact_type_6: **Axiom** opp_move_p(o) $\wedge \neg$(opp_move_p(t) $\vee$ opp_move_p(p))

fact_type_7: **Axiom** position_p(p) $\wedge \neg$(position_p(t) $\vee$ position_p(o))

```
possible_positions: Axiom
    ($p \in$ kb_inst(akb, sx))
      $\Leftrightarrow$ ($p$[row] $\geq 1$
            $\wedge$ $p$[row] $\leq 3$
              $\wedge$ $p$[col] $\geq 1$
                $\wedge$ $p$[col] $\leq 3$
                  $\wedge$ ($p$[marker] $=$ free $\vee$ $p$[marker] $=$ opp $\vee$ $p$[marker] $=$
com))
```

```
duplicate_positions: Axiom
    (($p \in$ kb_inst(akb, sx))
            $\wedge$ ($p_1 \in$ kb_inst(akb, sx)) $\wedge$ $p$[row] $=$ $p_1$[row] $\wedge$ $p$[col] $=$ $p_1$[col])
      $\supset$ $p$[marker] $=$ $p_1$[marker]
```

```
possible_turns: Axiom
    ($t \in$ kb_inst(akb, sx))
      $\Leftrightarrow$ (($t$[player] $=$ com $\vee$ $t$[player] $=$ opp) $\wedge$ $p$[count] $\geq 0$)
```

End ttt_ptypes

ttt_rls1: **Module**

Using ttt_ptypes, states, facts, kbs, sets[fact]

Exporting make_move, make_move_lhs

Theory

f: VAR fact
r, c: VAR nat
t_1, t2: VAR turn
p_1, p_2, p_3, p_4, p5, p6, p7, p8, p9, p0: VAR position
s_1, s_2: VAR state
make_move: function[kb, state $\rightarrow$ state]
make_move_lhs: function[kb, state $\rightarrow$ bool]

make_move_def: **Axiom**

```
    $(\exists\, t_1\ ,\, p_1 :$
        **if** match_turn(akb, $s_1, t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1, p_1$, wild, wild, free)
          **then** make_move(akb, $s_1$) = next($s_1$)
            $\wedge\ \neg(t_1 \in$ kb_inst(akb, next($s_1$)))
              $\wedge\ \neg(p_1 \in$ kb_inst(akb, next($s_1$)))
                $\wedge\ (\exists$ t2:
                        match_turn(akb, next($s_1$), t2, opp, wild)
                          $\wedge$ t2[count] = 1 + $t_1$[count])
                  $\wedge\ (\exists\, p_2 :$
                        match_position(akb, next($s_1$), $p_2$, wild, wild, com)
                          $\wedge\ p_2$[row] = $p_1$[row] $\wedge\ p_2$[col] = $p_1$[col])
          **else** make_move(akb, $s_1$) = $s_1$
          **end if** )
```

make_move_lhs_def: **Axiom**

```
    make_move_lhs(akb, $s_1$)
      $\Leftrightarrow (\exists\, t_1\ ,\, p_1 :$
          match_turn(akb, $s_1, t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1, p_1$, wild, wild, free))
```

move_order_1: **Axiom**

```
    make_move(akb, $s_1$) = next($s_1$)
```

$\Leftrightarrow (\exists p_1\ , p_2 :$
(match_position(akb, s_1, p_1, 2, 2, free)
$\supset$ match_position(akb, next(s_1), p_2, 2, 2, com)))

move_order_2: **Axiom**
make_move(akb, s_1) = next(s_1)
$\Leftrightarrow (\exists p_1\ , p_2\ , p_3 :$
(($\neg$match_position(akb, s_1, p_1, 2, 2, free)
$\wedge$ match_position(akb, s_1, p_2, 1, 1, free))
$\supset$ match_position(akb, next(s_1), p_3, 1, 1, com)))

move_order_3: **Axiom**
make_move(akb, s_1) = next(s_1)
$\Leftrightarrow (\exists p_1\ , p_2\ , p_3\ , p_4 :$
(($\neg$match_position(akb, s_1, p_1, 2, 2, free)
$\wedge\ \neg$match_position(akb, s_1, p_2, 1, 1, free)
$\wedge$ match_position(akb, s_1, p_3, 1, 3, free))
$\supset$ match_position(akb, next(s_1), p_4, 1, 3, com)))

move_order_4: **Axiom**
make_move(akb, s_1) = next(s_1)
$\Leftrightarrow (\exists p_1\ , p_2\ , p_3\ , p_4$, p5:
(($\neg$match_position(akb, s_1, p_1, 2, 2, free)
$\wedge\ \neg$match_position(akb, s_1, p_2, 1, 1, free)
$\wedge\ \neg$match_position(akb, s_1, p_3, 1, 3, free)
$\wedge$ match_position(akb, s_1, p_4, 3, 1, free))
$\supset$ match_position(akb, next(s_1), p5, 3, 1, com)))

move_order_5: **Axiom**
make_move(akb, s_1) = next(s_1)
$\Leftrightarrow (\exists p_1\ , p_2\ , p_3\ , p_4$, p5, p6:
(($\neg$match_position(akb, s_1, p_1, 2, 2, free)
$\wedge\ \neg$match_position(akb, s_1, p_2, 1, 1, free)
$\wedge\ \neg$match_position(akb, s_1, p_3, 1, 3, free)
$\wedge\ \neg$match_position(akb, s_1, p_4, 3, 1, free)
$\wedge$ match_position(akb, s_1, p5, 3, 3, free))
$\supset$ match_position(akb, next(s_1), p6, 3, 3, com)))

move_order_6: **Axiom**

$$\begin{aligned}
&\text{make\_move}(\text{akb}, s_1) = \text{next}(s_1)\\
&\quad \Leftrightarrow (\exists\, p_1, p_2, p_3, p_4, \text{p5}, \text{p6}, \text{p7}:\\
&\qquad ((\neg\text{match\_position}(\text{akb}, s_1, p_1, 2, 2, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_2, 1, 1, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_3, 1, 3, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_4, 3, 1, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, \text{p5}, 3, 3, \text{free})\\
&\qquad\quad \wedge \text{match\_position}(\text{akb}, s_1, \text{p6}, 1, 2, \text{free}))\\
&\qquad \supset \text{match\_position}(\text{akb}, \text{next}(s_1), \text{p7}, 1, 2, \text{com})))
\end{aligned}$$

move_order_7: **Axiom**

$$\begin{aligned}
&\text{make\_move}(\text{akb}, s_1) = \text{next}(s_1)\\
&\quad \Leftrightarrow (\exists\, p_1, p_2, p_3, p_4, \text{p5}, \text{p6}, \text{p7}, \text{p8}:\\
&\qquad ((\neg\text{match\_position}(\text{akb}, s_1, p_1, 2, 2, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_2, 1, 1, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_3, 1, 3, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_4, 3, 1, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, \text{p5}, 3, 3, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, \text{p6}, 1, 2, \text{free})\\
&\qquad\quad \wedge \text{match\_position}(\text{akb}, s_1, \text{p7}, 2, 1, \text{free}))\\
&\qquad \supset \text{match\_position}(\text{akb}, \text{next}(s_1), \text{p8}, 2, 1, \text{com})))
\end{aligned}$$

move_order_8: **Axiom**

$$\begin{aligned}
&\text{make\_move}(\text{akb}, s_1) = \text{next}(s_1)\\
&\quad \Leftrightarrow (\exists\, p_1, p_2, p_3, p_4, \text{p5}, \text{p6}, \text{p7}, \text{p8}, \text{p9}:\\
&\qquad ((\neg\text{match\_position}(\text{akb}, s_1, p_1, 2, 2, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_2, 1, 1, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_3, 1, 3, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, p_4, 3, 1, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, \text{p5}, 3, 3, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, \text{p6}, 1, 2, \text{free})\\
&\qquad\quad \wedge \neg\text{match\_position}(\text{akb}, s_1, \text{p7}, 2, 1, \text{free})\\
&\qquad\quad \wedge \text{match\_position}(\text{akb}, s_1, \text{p8}, 2, 3, \text{free}))
\end{aligned}$$

$\supset$ match_position(akb, next(s_1), p9, 2, 3, com)))

move_order_9: **Axiom**

make_move(akb, s_1) = next(s_1)

$\Leftrightarrow$ ($\exists p_1$, p_2 , p_3 , p_4 , p5, p6, p7, p8, p9, p0:

((¬match_position(akb, s_1, p_1, 2, 2, free)

$\wedge$ ¬match_position(akb, s_1, p_2, 1, 1, free)

$\wedge$ ¬match_position(akb, s_1, p_3, 1, 3, free)

$\wedge$ ¬match_position(akb, s_1, p_4, 3, 1, free)

$\wedge$ ¬match_position(akb, s_1, p5, 3, 3, free)

$\wedge$ ¬match_position(akb, s_1, p6, 1, 2, free)

$\wedge$¬match_position(akb, s_1, p7, 2, 1, free)

$\wedge$¬match_position(akb, s_1, p8, 2, 3, free)

$\wedge$match_position(akb, s_1, p9, 3, 2, free))

$\supset$ match_position(akb, next(s_1), p0, 3, 2, com)))

End ttt_rls1

ttt_rls2: **Module**

Using ttt_ptypes, states, facts, kbs, sets[fact]

Exporting win_col, win_row, win_dia, win_col_lhs, win_row_lhs, win_dia_lhs

Theory

```
f: VAR fact
t, t1, t2: VAR turn
p, c1, c2, p1, p2, p3, p4: VAR position
s1, s2: VAR state
win_col: function[kb, state → state]
win_row: function[kb, state → state]
win_dia: function[kb, state → state]
win_col_lhs: function[kb, state → bool]
win_row_lhs: function[kb, state → bool]
win_dia_lhs: function[kb, state → bool]
```

win_col_def: **Axiom**

```
    (∃ $t_1$ , $p_1$ , $p_2$ , $p_3$ :
        if  match_turn(akb, $s_1$, $t_1$, com, wild)
            ∧ match_position(akb, $s_1$, $p_1$, wild, wild, com)
              ∧ match_position(akb, $s_1$, $p_2$, wild, wild, com)
                ∧ match_position(akb, $s_1$, $p_3$, wild, wild, free)
                  ∧ $p_1$[col] = $p_2$[col]
                    ∧ $p_1$[row] ≠ $p_2$[row]
                      ∧ $p_3$[col] = $p_2$[col]
                        ∧ $p_3$[row] ≠ $p_1$[row] ∧ $p_3$[row] ≠ $p_2$[row]
          then (∃ $p_4$ :
            win_col(akb, $s_1$) = next($s_1$)
              ∧ ¬($t_1$ ∈ kb_inst(akb, next($s_1$)))
                ∧ ¬($p_3$ ∈ kb_inst(akb, next($s_1$)))
                  ∧ match_position(akb, next($s_1$), $p_4$, wild, wild, com)
                    ∧ $p_4$[row] = $p_3$[row]
                      ∧ $p_4$[col] = $p_3$[col]
                        ∧ (∀ p:
                            $p \neq p_3$
                              ⊃ (($p$ ∈ kb_inst(akb, $s_1$))
                                ⇔ ($p$ ∈ kb_inst(akb, next($s_1$))))))
        else  $\text{win\_col}(\text{akb}, s_1) = s_1$
        end if )

win_row_def: Axiom
    ( $\exists\, t_1, p_1, p_2, p_3$ :
        if  $\text{match\_turn}(\text{akb}, s_1, t_1, \text{com}, \text{wild})$
            $\wedge$ $\text{match\_position}(\text{akb}, s_1, p_1, \text{wild}, \text{wild}, \text{com})$
              $\wedge$ $\text{match\_position}(\text{akb}, s_1, p_2, \text{wild}, \text{wild}, \text{com})$
                $\wedge$ $\text{match\_position}(\text{akb}, s_1, p_3, \text{wild}, \text{wild}, \text{free})$
                  $\wedge\, p_1[\text{col}] \neq p_2[\text{col}]$
                    $\wedge\, p_1[\text{row}] = p_2[\text{row}]$
                      $\wedge\, p_2[\text{row}] = p_3[\text{row}]$
                        $\wedge\, p_3[\text{col}] \neq p_1[\text{col}] \wedge p_3[\text{col}] \neq p_2[\text{col}]$
        then ( $\exists\, p_4$ :
            $\text{win\_row}(\text{akb}, s_1) = \text{next}(s_1)$
              $\wedge\, \neg(t_1 \in \text{kb\_inst}(\text{akb}, \text{next}(s_1)))$
                $\wedge\, \neg(p_3 \in \text{kb\_inst}(\text{akb}, \text{next}(s_1)))$
                  $\wedge$ $\text{match\_position}(\text{akb}, \text{next}(s_1), p_4, \text{wild}, \text{wild}, \text{com})$
                    $\wedge\, p_4[\text{row}] = p_3[\text{row}]$
                      $\wedge\, p_4[\text{col}] = p_3[\text{col}]$
                        $\wedge$ ( $\forall$ p:
                            $p \neq p_3$
                              $\supset ((p \in \text{kb\_inst}(\text{akb}, s_1))$
                                $\Leftrightarrow (p \in \text{kb\_inst}(\text{akb}, \text{next}(s_1))))))$
        else  $\text{win\_row}(\text{akb}, s_1) = s_1$
        end if )

win_dia_def: Axiom
    ( $\exists\, t_1, p_1, p_2, p_3$ :
        if  $\text{match\_turn}(\text{akb}, s_1, t_1, \text{com}, \text{wild})$
            $\wedge$ $\text{match\_position}(\text{akb}, s_1, p_1, \text{wild}, \text{wild}, \text{com})$
              $\wedge$ $\text{match\_position}(\text{akb}, s_1, p_2, \text{wild}, \text{wild}, \text{com})$
                $\wedge$ $\text{match\_position}(\text{akb}, s_1, p_3, \text{wild}, \text{wild}, \text{free})$
                  $\wedge\, p_1[\text{row}] \neq p_2[\text{row}]$
                    $\wedge\, p_1[\text{col}] \neq p_2[\text{col}]$
                      $\wedge\, p_1[\text{row}] \neq p_3[\text{row}]$
                        $\wedge\, p_1[\text{col}] \neq p_3[\text{col}]$
                          $\wedge\, p_2[\text{row}] \neq p_3[\text{row}]$
```

```
                                  $\land p_2[\text{col}] \neq p_3[\text{col}]$
                                    $\land ((\text{cornerp}(p_1)$
                                      $\land \text{centerp}(p_2) \land \text{centerp}(p_3))$
                                        $\lor (\text{centerp}(p_1)$
                                          $\land \text{cornerp}(p_2) \land \text{cornerp}(p_3)))$
          **then** $(\exists\, p_4 :$
              $\text{win\_dia}(\text{akb}, s_1) = \text{next}(s_1)$
                $\land \neg(t_1 \in \text{kb\_inst}(\text{akb}, \text{next}(s_1)))$
                  $\land \neg(p_3 \in \text{kb\_inst}(\text{akb}, \text{next}(s_1)))$
                    $\land \text{match\_position}(\text{akb}, \text{next}(s_1), p_4, \text{wild}, \text{wild}, \text{com})$
                      $\land p_4[\text{row}] = p_3[\text{row}]$
                        $\land p_4[\text{col}] = p_3[\text{col}]$
                          $\land (\forall\, \text{p}:$
                              $p \neq p_3$
                                $\supset ((p \in \text{kb\_inst}(\text{akb}, s_1))$
                                  $\Leftrightarrow (p \in \text{kb\_inst}(\text{akb}, \text{next}(s_1))))))$
          **else** $\text{win\_dia}(\text{akb}, s_1) = s_1$
          **end if** )
```

win_col_lhs_def: **Axiom**

```
    $\text{win\_col\_lhs}(\text{akb}, s_1)$
      $\Leftrightarrow (\exists\, t_1, p_1, p_2, p_3 :$
          $\text{match\_turn}(\text{akb}, s_1, t_1, \text{com}, \text{wild})$
            $\land \text{match\_position}(\text{akb}, s_1, p_1, \text{wild}, \text{wild}, \text{com})$
              $\land \text{match\_position}(\text{akb}, s_1, p_2, \text{wild}, \text{wild}, \text{com})$
                $\land p_1[\text{col}] = p_2[\text{col}]$
                  $\land p_1[\text{row}] \neq p_2[\text{row}]$
                    $\land \text{match\_position}(\text{akb}, s_1, p_3, \text{wild}, \text{wild}, \text{free})$
                      $\land p_3[\text{col}] = p_2[\text{col}])$
```

win_row_lhs_def: **Axiom**

```
    $\text{win\_row\_lhs}(\text{akb}, s_1)$
      $\Leftrightarrow (\exists\, t_1, p_1, p_2, p_3 :$
          $\text{match\_turn}(\text{akb}, s_1, t_1, \text{com}, \text{wild})$
            $\land \text{match\_position}(\text{akb}, s_1, p_1, \text{wild}, \text{wild}, \text{com})$
              $\land \text{match\_position}(\text{akb}, s_1, p_2, \text{wild}, \text{wild}, \text{com})$
                $\land \text{match\_position}(\text{akb}, s_1, p_3, \text{wild}, \text{wild}, \text{free})$
                  $\land p_1[\text{col}] \neq p_2[\text{col}]$
```

```
                    $\wedge\ p_1[\text{row}] = p_2[\text{row}] \wedge p_2[\text{row}] = p_3[\text{row}])$

**win_dia_lhs_def: Axiom**
    win_dia_lhs(akb, $s_1$)
      $\Leftrightarrow$ ( $\exists\, t_1$ , $p_1$ , $p_2$ , $p_3$ :
          match_turn(akb, $s_1$, $t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1$, $p_1$, wild, wild, com)
              $\wedge$ match_position(akb, $s_1$, $p_2$, wild, wild, com)
                $\wedge$ match_position(akb, $s_1$, $p_3$, wild, wild, free)
                  $\wedge\ p_1[\text{row}] \neq p_2[\text{row}]$
                    $\wedge\ p_1[\text{col}] \neq p_2[\text{col}]$
                      $\wedge\ p_1[\text{row}] \neq p_3[\text{row}]$
                        $\wedge\ p_1[\text{col}] \neq p_3[\text{col}]$
                          $\wedge\ p_2[\text{row}] \neq p_3[\text{row}]$
                            $\wedge\ p_2[\text{col}] \neq p_3[\text{col}]$
                              $\wedge$ ((cornerp($p_1$)
                                $\wedge$ centerp($p_2$) $\wedge$ centerp($p_3$))
                                  $\vee$ (centerp($p_1$)
                                    $\wedge$ cornerp($p_2$) $\wedge$ cornerp($p_3$))))

**End** ttt_rls2
```

C-2

ttt_rls3: **Module**

Using ttt_ptypes, states, facts, kbs, sets[fact], diffs

Exporting blk_col, blk_row, blk_dia, blk_col_lhs, blk_row_lhs, blk_dia_lhs

Theory

```
$f$: VAR fact
$t_1$, t2: VAR turn
c1, c2, $p_1, p_2, p_3, p_4$: VAR position
$s_1, s_2$: VAR state
blk_col: function[kb, state → state]
blk_row: function[kb, state → state]
blk_dia: function[kb, state → state]
blk_col_lhs: function[kb, state → bool]
blk_row_lhs: function[kb, state → bool]
blk_dia_lhs: function[kb, state → bool]

blk_col_def: Axiom
    ($\exists t_1$ , $p_1$ , $p_2$ , $p_3$ :
        if match_turn(akb, $s_1$, $t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1$, $p_1$, wild, wild, opp)
              $\wedge$ match_position(akb, $s_1$, $p_2$, wild, wild, opp)
                $\wedge$ $p_1$[col] = $p_2$[col]
                  $\wedge$ $p_1$[row] $\neq$ $p_2$[row]
                    $\wedge$ match_position(akb, $s_1$, $p_3$, wild, wild, free)
                      $\wedge$ $p_2$[col] = $p_3$[col]
          then ($\exists p_4$ , t2:
              blk_col(akb, $s_1$) = next($s_1$)
                $\wedge$ $\neg$($t_1$ $\in$ kb_inst(akb, next($s_1$)))
                  $\wedge$ $\neg$($p_3$ $\in$ kb_inst(akb, next($s_1$)))
                    $\wedge$match_position(akb, next($s_1$), $p_4$, $p_3$[row], $p_3$[col], com)

                      $\wedge$ match_turn(akb, next($s_1$), t2, opp, wild)
                        $\wedge$ t2[count] = 1 + $t_1$[count])
          else blk_col(akb, $s_1$) = $s_1$
          end if )

blk_row_def: Axiom
    ($\exists t_1$ , $p_1$ , $p_2$ , $p_3$ :
        if  match_turn(akb, $s_1, t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1, p_1$, wild, wild, opp)
              $\wedge$ match_position(akb, $s_1, p_2$, wild, wild, opp)
                $\wedge$ $p_1$[row] $=$ $p_2$[row]
                  $\wedge$ $p_1$[col] $\neq$ $p_2$[col]
                    $\wedge$ match_position(akb, $s_1, p_3$, wild, wild, free)
                      $\wedge$ $p_2$[row] $=$ $p_3$[row]
          then ( $\exists\, p_4$ , t2:
              blk_row(akb, $s_1$) $=$ next($s_1$)
                $\wedge$ $\neg(t_1 \in$ kb_inst(akb, next($s_1$)))
                  $\wedge$ $\neg(p_3 \in$ kb_inst(akb, next($s_1$)))
                    $\wedge$match_position(akb, next($s_1$), $p_4$, $p_3$[row], $p_3$[col], com)

                      $\wedge$ match_turn(akb, next($s_1$), t2, opp, wild)
                        $\wedge$ t2[count] $= 1 +$ $t_1$[count])
          else  blk_row(akb, $s_1$) $=$ $s_1$
          end if  )

blk_dia_def: Axiom
    ( $\exists\, t_1$ , $p_1$ , $p_2$ , $p_3$ :
        if  match_turn(akb, $s_1, t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1, p_1$, wild, wild, opp)
              $\wedge$ match_position(akb, $s_1, p_2$, wild, wild, opp)
                $\wedge$ match_position(akb, $s_1, p_3$, wild, wild, free)
                  $\wedge$ $p_1$[row] $\neq$ $p_2$[row]
                    $\wedge$ $p_1$[col] $\neq$ $p_2$[col]
                      $\wedge$ $\mid$ $p_1$[row] $-$ $p_2$[row] $\mid=\mid$ $p_1$[col] $-$ $p_2$[col] $\mid$
                        $\wedge$ $\mid$ $p_1$[row] $-$ $p_3$[row] $\mid=\mid$ $p_1$[col] $-$ $p_3$[col] $\mid$
                          $\wedge$ $\mid$ $p_2$[row] $-$ $p_3$[row] $\mid=\mid$ $p_2$[col] $-$ $p_3$[col] $\mid$
          then ( $\exists\, p_4$ , t2:
              blk_dia(akb, $s_1$) $=$ next($s_1$)
                $\wedge$ $\neg(t_1 \in$ kb_inst(akb, $s_2$))
                  $\wedge$ $\neg(p_3 \in$ kb_inst(akb, $s_2$))
                    $\wedge$ match_position(akb, $s_2, p_4$, wild, wild, com)
                      $\wedge$ $p_4$[row] $=$ $p_3$[row]
                        $\wedge$ $p_4$[col] $=$ $p_3$[col]
                          $\wedge$ match_turn(akb, $s_2$, t2, opp, wild)
                            $\wedge$ t2[count] $= 1 +$ $t_1$[count])
          else  blk_dia(akb, $s_1$) $=$ $s_1$
        end if )
blk_col_lhs_def: Axiom
    blk_col_lhs(akb, $s_1$)
      $\Leftrightarrow$ ( $\exists\, t_1$ , $p_1$ , $p_2$ , $p_3$ :
          match_turn(akb, $s_1$, $t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1$, $p_1$, wild, wild, opp)
              $\wedge$ match_position(akb, $s_1$, $p_2$, wild, $p_1$[col], opp)
                $\wedge$ $p_1$[row] $\neq$ $p_2$[row]
                  $\wedge$ match_position(akb, $s_1$, $p_3$, wild, $p_2$[col], free))

blk_row_lhs_def: Axiom
    blk_row_lhs(akb, $s_1$)
      $\Leftrightarrow$ ( $\exists\, t_1$ , $p_1$ , $p_2$ , $p_3$ :
          match_turn(akb, $s_1$, $t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1$, $p_1$, wild, wild, opp)
              $\wedge$ match_position(akb, $s_1$, $p_2$, $p_1$[row], wild, opp)
                $\wedge$ $p_1$[col] $\neq$ $p_2$[col]
                  $\wedge$ match_position(akb, $s_1$, $p_3$, $p_2$[row], wild, free))

blk_dia_lhs_def: Axiom
    blk_dia_lhs(akb, $s_1$)
      $\Leftrightarrow$ ( $\exists\, t_1$ , $p_1$ , $p_2$ , $p_3$ :
          match_turn(akb, $s_1$, $t_1$, com, wild)
            $\wedge$ match_position(akb, $s_1$, $p_1$, wild, wild, opp)
              $\wedge$ match_position(akb, $s_1$, $p_2$, $p_1$[row], wild, opp)
                $\wedge$ match_position(akb, $s_1$, $p_3$, $p_2$[row], wild, free)
                  $\wedge$ $p_1$[row] $\neq$ $p_2$[row]
                    $\wedge$ $p_1$[col] $\neq$ $p_2$[col]
                      $\wedge$ | $p_1$[row] − $p_2$[row] |=| $p_1$[col] − $p_2$[col] |
                        $\wedge$ | $p_1$[row] − $p_3$[row] |=| $p_1$[col] − $p_3$[col] |
                          $\wedge$ | $p_2$[row] − $p_3$[row] |=| $p_2$[col] − $p_3$[col] |)

End ttt_rls3
```

ttt_rls4: **Module**

Using ttt_ptypes, states, facts, kbs, sets[fact]

Exporting blk_cc, blk_cm, blk_cc_lhs, blk_cm_lhs

Theory

```
f: VAR fact
r, c: VAR nat
t1, t2: VAR turn
p, px, p1, p2, p3, p4, p5: VAR position
s1, s2: VAR state
blk_cc: function[kb, state → state]
blk_cm: function[kb, state → state]
blk_cc_lhs: function[kb, state → bool]
blk_cm_lhs: function[kb, state → bool]

blk_cc_def: Axiom
    (∃ t1 , p1 , p2 , p3 , p4 :
        if match_turn(akb, s1, t1, com, 3)
            ∧ match_position(akb, s1, p1, wild, wild, opp)
              ∧ match_position(akb, s1, p2, wild, wild, opp)
                ∧ match_position(akb, s1, p3, 2, 2, com)
                  ∧ match_position(akb, s1, p4, wild, wild, free)
                    ∧ p1[row] ≠ p2[row]
                      ∧ p1[col] ≠ p2[col]
                        ∧ cornerp(p1)
                          ∧ cornerp(p2)
                            ∧ centerp(p3)
                              ∧ middlep(p4)
                                ∧ (∀ px:
                                    ((px ∈ kb_inst(akb, s1))
                                      ∧ px[marker] ≠ free)
                                      ⇔ (px = p1
                                        ∨ px = p2 ∨ px = p3))
          then blk_cc(akb, s1) = next(s1)
            ∧ ¬(t1 ∈ kb_inst(akb, next(s1)))
              ∧ ¬(p4 ∈ kb_inst(akb, next(s1)))
                ∧ (∃ t2:
                      match_turn(akb, next(s1), t2, opp, wild)
                    $\wedge$ t2[count] $= 1 + t_1$[count])
                $\wedge$ ( $\exists$ p5:
                    match_position(akb, next($s_1$), p5, wild, wild, com)
                        $\wedge$ p5[row] $= p_4$[row]
                            $\wedge$ p5[col] $= p_4$[col]
                                $\wedge$ ( $\forall$ p:
                                    ($p \neq p_4 \wedge p \neq$ p5)
                                        $\Leftrightarrow$ (($p \in$ kb_inst(akb, $s_1$))
                                            $\Leftrightarrow$ ($p \in$ kb_inst(akb, next($s_1$))))))

        else  blk_cc(akb, $s_1$) $= s_1$
        end if  )

blk_cm_def: Axiom
    ( $\exists$ $t_1$ , $p_1$ , $p_2$ , $p_3$ , $p_4$ :
        if  match_turn(akb, $s_1$, $t_1$, com, 3)
            $\wedge$ match_position(akb, $s_1$, $p_1$, wild, wild, opp)
                $\wedge$ match_position(akb, $s_1$, $p_2$, wild, wild, opp)
                    $\wedge$ match_position(akb, $s_1$, $p_3$, 2, 2, com)
                        $\wedge$ match_position(akb, $s_1$, $p_4$, wild, wild, free)
                            $\wedge$ $p_1$[row] $\neq p_2$[row]
                                $\wedge$ $p_1$[col] $\neq p_2$[col]
                                    $\wedge$ cornerp($p_1$)
                                        $\wedge$ middlep($p_2$)
                                            $\wedge$ cornerp($p_4$)
                                                $\wedge$ (($p_4$[row] $= p_1$[row] $\wedge$ $p_4$[col] $= p_2$[col])
                                                    $\vee$ ($p_4$[row] $= p_2$[row]
                                                        $\wedge$ $p_4$[col] $= p_1$[col]))
        then blk_cm(akb, $s_1$) $=$ next($s_1$)
            $\wedge$ $\neg$($t_1 \in$ kb_inst(akb, next($s_1$)))
                $\wedge$ $\neg$($p_4 \in$ kb_inst(akb, next($s_1$)))
                    $\wedge$ ( $\exists$ t2:
                        match_turn(akb, next($s_1$), t2, opp, wild)
                            $\wedge$ t2[count] $= 1 + t_1$[count])
                $\wedge$ ( $\exists$ p5:
                    match_position(akb, next($s_1$), p5, wild, wild, com)
                        $\wedge$ p5[row] $= p_4$[row]
                            $\wedge$ p5[col] $= p_4$[col]
                                $\wedge$ ( $\forall$ p:
                                    $p \neq p_4$
                                        $\Leftrightarrow (p \in \text{kb\_inst}(\text{akb}, s_1)) \supset (p \in$
$\text{kb\_inst}(\text{akb}, \text{next}(s_1)))))$
        else  $\text{blk\_cm}(\text{akb}, s_1) = s_1$
        end if  )
```

blk_cc_lhs_def: **Axiom**

```
    $\text{blk\_cc\_lhs}(\text{akb}, s_1)$
        $\Leftrightarrow (\exists t_1, p_1, p_2, p_3, p_4:$
            $\text{match\_turn}(\text{akb}, s_1, t_1, \text{com}, 3)$
              $\wedge\ \text{match\_position}(\text{akb}, s_1, p_1, \text{wild}, \text{wild}, \text{opp})$
                $\wedge\ \text{match\_position}(\text{akb}, s_1, p_2, \text{wild}, \text{wild}, \text{opp})$
                  $\wedge\ \text{match\_position}(\text{akb}, s_1, p_3, 2, 2, \text{com})$
                    $\wedge\ \text{match\_position}(\text{akb}, s_1, p_4, \text{wild}, \text{wild}, \text{free})$
                      $\wedge\ p_1[\text{row}] \neq p_2[\text{row}]$
                        $\wedge\ p_1[\text{col}] \neq p_2[\text{col}] \wedge \text{cornerp}(p_1) \wedge \text{cornerp}(p_2) \wedge$
$\text{middlep}(p_4))$
```

blk_cm_lhs_def: **Axiom**

```
    $\text{blk\_cc\_lhs}(\text{akb}, s_1)$
        $\Leftrightarrow (\exists t_1, p_1, p_2, p_3, p_4:$
            $\text{match\_turn}(\text{akb}, s_1, t_1, \text{com}, 3)$
              $\wedge\ \text{match\_position}(\text{akb}, s_1, p_1, \text{wild}, \text{wild}, \text{opp})$
                $\wedge\ \text{match\_position}(\text{akb}, s_1, p_2, \text{wild}, \text{wild}, \text{opp})$
                  $\wedge\ \text{match\_position}(\text{akb}, s_1, p_3, 2, 2, \text{com})$
                    $\wedge\ \text{match\_position}(\text{akb}, s_1, p_4, \text{wild}, \text{wild}, \text{free})$
                      $\wedge\ p_1[\text{row}] \neq p_2[\text{row}]$
                        $\wedge\ p_1[\text{col}] \neq p_2[\text{col}]$
                          $\wedge\ \text{cornerp}(p_1)$
                            $\wedge\ \text{middlep}(p_2)$
                              $\wedge\ \text{cornerp}(p_4)$
                                $\wedge\ ((p_4[\text{row}] = p_1[\text{row}] \wedge p_4[\text{col}] = p_2[\text{col}])$
                                    $\vee\ (p_4[\text{row}] = p_2[\text{row}] \wedge p_4[\text{col}] =$
$p_1[\text{col}])))$
```

End ttt_rls4

```
guarded_ops: Module

Using ttt_ptypes, ttt_rls1, ttt_rls2, ttt_rls3, ttt_rls4, states, kbs

Exporting ttt_op, g_win_col, g_win_row, g_win_dia, g_blk_cc,
    g_blk_cm, g_blk_col, g_blk_row, g_blk_dia, g_mke_mve

Theory

ttt_op: TYPE from function[kb, state → state]
g_win_col: ttt_op
g_win_row: ttt_op
g_win_dia: ttt_op
g_blk_cc: ttt_op
g_blk_cm: ttt_op
g_blk_col: ttt_op
g_blk_row: ttt_op
g_blk_dia: ttt_op
g_mke_mve: ttt_op
gop: VAR ttt_op
s_n: VAR state

g_win_col_def: Axiom
    g_win_col(akb, s_n)
      = if  win_col_lhs(akb, s_n) then win_col(akb, s_n) else  s_n end if


g_win_row_def: Axiom
    g_win_row(akb, s_n)
      = if  ((¬win_col_lhs(akb, s_n)) ∧ win_row_lhs(akb, s_n))
        then win_row(akb, s_n)
        else  s_n
        end if


g_win_dia_def: Axiom
    g_win_dia(akb, s_n)
      = if  ((¬(win_col_lhs(akb, s_n) ∨ win_row_lhs(akb, s_n)))
               ∧ win_dia_lhs(akb, s_n))
        then win_dia(akb, s_n)
        else  s_n
        end if
```

```
g_blk_cc_def: Axiom
    g_blk_cc(akb, $s_n$)
      = if  (($\neg$(win_col_lhs(akb, $s_n$)
                         $\vee$ win_row_lhs(akb, $s_n$) $\vee$ win_dia_lhs(akb, $s_n$)))
              $\wedge$ blk_cc_lhs(akb, $s_n$))
        then blk_cc(akb, $s_n$)
        else  $s_n$
        end if

g_blk_cm_def: Axiom
    g_blk_cm(akb, $s_n$)
      = if  (($\neg$(win_col_lhs(akb, $s_n$)
                         $\vee$ win_row_lhs(akb, $s_n$)
                           $\vee$ win_dia_lhs(akb, $s_n$) $\vee$ blk_cc_lhs(akb, $s_n$)))
              $\wedge$ blk_col_lhs(akb, $s_n$))
        then blk_col(akb, $s_n$)
        else  $s_n$
        end if

g_blk_col_def: Axiom
    g_blk_col(akb, $s_n$)
      = if  (($\neg$(win_col_lhs(akb, $s_n$)
                         $\vee$ win_row_lhs(akb, $s_n$)
                           $\vee$ win_dia_lhs(akb, $s_n$)
                             $\vee$blk_cc_lhs(akb, $s_n$)$\vee$blk_cm_lhs(akb, $s_n$)))
              $\wedge$ blk_col_lhs(akb, $s_n$))
        then blk_col(akb, $s_n$)
        else  $s_n$
        end if

g_blk_row_def: Axiom
    g_blk_row(akb, $s_n$)
      = if  (($\neg$(win_col_lhs(akb, $s_n$)
                         $\vee$ win_row_lhs(akb, $s_n$)
                           $\vee$ win_dia_lhs(akb, $s_n$)
                             $\vee$ blk_cc_lhs(akb, $s_n$)
                               $\vee$blk_cm_lhs(akb, $s_n$)$\vee$blk_col_lhs(akb, $s_n$)))
```

```
                $\wedge$ blk_row_lhs(akb, $s_n$))
            **then** blk_row(akb, $s_n$)
            **else** $s_n$
            **end if**

g_blk_dia_def: **Axiom**
    g_blk_dia(akb, $s_n$)
        = **if** ((¬(win_col_lhs(akb, $s_n$)
                              $\vee$ win_row_lhs(akb, $s_n$)
                                $\vee$ win_dia_lhs(akb, $s_n$)
                                  $\vee$ blk_cc_lhs(akb, $s_n$)
                                    $\vee$ blk_cm_lhs(akb, $s_n$)
                                      $\vee$blk_col_lhs(akb, $s_n$)$\vee$blk_row_lhs(akb, $s_n$)))

                $\wedge$ blk_dia_lhs(akb, $s_n$))
            **then** blk_dia(akb, $s_n$)
            **else** $s_n$
            **end if**

g_mke_mve_def: **Axiom**
    g_mke_mve(akb, $s_n$)
        = **if** ((¬(win_col_lhs(akb, $s_n$)
                              $\vee$ win_row_lhs(akb, $s_n$)
                                $\vee$ win_dia_lhs(akb, $s_n$)
                                  $\vee$ blk_cc_lhs(akb, $s_n$)
                                    $\vee$ blk_cm_lhs(akb, $s_n$)
                                      $\vee$ blk_col_lhs(akb, $s_n$)
                                        $\vee$blk_row_lhs(akb, $s_n$)$\vee$blk_dia_lhs(akb, $s_n$)))

                $\wedge$ make_move_lhs(akb, $s_n$))
            **then** make_move(akb, $s_n$)
            **else** $s_n$
            **end if**

gop_def: **Axiom**
    gop = g_win_col
      $\vee$ gop = g_win_row
```

```
            ∨ gop = g_win_dia
              ∨ gop = g_blk_cc
                ∨ gop = g_blk_cm
                  ∨ gop = g_blk_col
                    ∨ gop = g_blk_row ∨ gop = g_blk_dia ∨ gop = g_mke_mve

gmm_ax: Axiom true
End guarded_ops
```

safe_ttt: **Module**

Using ttt_ptypes, kbs, states, facts, diffs

Exporting safe, won_game, lose_next_move, unsafe_config_1,
unsafe_config_2, unsafe_config_3, unsafe_config_4

Theory

s_n, sm: VAR state
t_1, t2: VAR turn
p_1, p_2, p_3, p_4: VAR position
kbx: VAR kb
won_game: function[kb, state → bool]
lose_next_move: function[kb, state → bool]
unsafe_config_1: function[kb, state → bool]
unsafe_config_2: function[kb, state → bool]
unsafe_config_3: function[kb, state → bool]
unsafe_config_4: function[kb, state → bool]
safe: function[kb, state → bool]

safe_def: **Axiom**

```
    safe(kbx, $s_n$)
        $\Leftrightarrow$ won_game(kbx, $s_n$)
            $\vee \neg$(lose_next_move(kbx, $s_n$)
                $\vee$ unsafe_config_1(kbx, $s_n$)
                    $\vee$ unsafe_config_2(kbx, $s_n$)
                        $\vee$ unsafe_config_3(kbx, $s_n$) $\vee$ unsafe_config_4(kbx, $s_n$))
```

won_game_def: **Axiom**

```
    won_game(kbx, $s_n$)
        $\Leftrightarrow$ ($\exists p_1, p_2, p_3$ :
            match_position(kbx, $s_n$, $p_1$, wild, wild, com)
                $\wedge$ match_position(kbx, $s_n$, $p_2$, wild, wild, com)
                    $\wedge$ match_position(kbx, $s_n$, $p_3$, wild, wild, com)
                        $\wedge p_1 \neq p_2$
                            $\wedge p_2 \neq p_3$
                                $\wedge p_1 \neq p_3$
                                    $\wedge$ (($p_1$[col] = $p_2$[col] $\wedge$ $p_2$[col] = $p_3$[col])
                                        $\vee$ ($p_1$[row] = $p_2$[row] $\wedge$ $p_2$[row] = $p_3$[row])
                                            $\vee$ (cornerp($p_1$)
                                    $\land$ cornerp($p_2$)
                                      $\land$ centerp($p_3$)
                                        $\land p_1$[row] $\neq p_2$[row]$\land p_1$[col] $\neq$
$p_2$[col])))
```

lose_next_move_def: **Axiom**

```
    lose_next_move(kbx, $s_n$)
      $\Leftrightarrow$ ( $\exists t_1$ , $p_1$ , $p_2$ , $p_3$ :
          match_turn(kbx, $s_n$, $t_1$, opp, wild)
            $\land$ match_position(kbx, $s_n$, $p_1$, wild, wild, opp)
              $\land$ match_position(kbx, $s_n$, $p_2$, wild, wild, opp)
                $\land$ match_position(kbx, $s_n$, $p_3$, wild, wild, free)
                  $\land p_1 \neq p_2$
                    $\land$ ((cornerp($p_1$)
                                $\land$ cornerp($p_2$)
                                  $\land$ centerp($p_3$)
                                    $\land p_1$[row] $\neq p_2$[row]
                                      $\land p_1$[col] $\neq p_2$[col])
                          $\lor$ ($p_1$[row] $= p_2$[row] $\land p_2$[row] $= p_3$[row])
                            $\lor$ ($p_1$[col] $= p_2$[col] $\land p_2$[col] $= p_3$[col])))
```

unsafe_config_1_def: **Axiom**

```
    unsafe_config_1(kbx, $s_n$)
      $\Leftrightarrow$ ( $\exists t_1$ , $p_1$ , $p_2$ , $p_3$ , $p_4$ :
          match_turn(kbx, $s_n$, $t_1$, opp, 4)
            $\land$ match_position(kbx, $s_n$, $p_1$, wild, wild, com)
              $\land$ match_position(kbx, $s_n$, $p_2$, wild, wild, com)
                $\land$ match_position(kbx, $s_n$, $p_3$, wild, wild, opp)
                  $\land$ match_position(kbx, $s_n$, $p_4$, wild, wild, opp)
                    $\land p_1 \neq p_2$
                      $\land p_3 \neq p_4$
                        $\land$centerp($p_1$)$\land$cornerp($p_2$)$\land$cornerp($p_3$)$\land$cornerp($p_4$))
```

unsafe_config_2_def: **Axiom**

```
    unsafe_config_2(kbx, $s_n$)
      $\Leftrightarrow$ ( $\exists t_1$ , $p_1$ , $p_2$ , $p_3$ , $p_4$ :
          match_turn(kbx, $s_n$, $t_1$, opp, 4)
```

$\wedge$ match_position(kbx, s_n, p_1, wild, wild, com)
$\wedge$ match_position(kbx, s_n, p_2, wild, wild, com)
$\wedge$ match_position(kbx, s_n, p_3, wild, wild, opp)
$\wedge$ match_position(kbx, s_n, p_4, wild, wild, opp)
$\wedge\, p_1 \neq p_2$
$\wedge\, p_3 \neq p_4$
$\wedge$middlep(p_1)$\wedge$middlep(p_2)$\wedge$cornerp(p_3)$\wedge$cornerp(p_4))

unsafe_config_3_def: Axiom
unsafe_config_3(kbx, s_n)
$\Leftrightarrow$ ($\exists\, t_1\,, p_1\,, p_2\,, p_3\,, p_4$:
match_turn(kbx, s_n, t_1, opp, 4)
$\wedge$ match_position(kbx, s_n, p_1, wild, wild, com)
$\wedge$ match_position(kbx, s_n, p_2, wild, wild, com)
$\wedge$ match_position(kbx, s_n, p_3, wild, wild, opp)
$\wedge$ match_position(kbx, s_n, p_4, wild, wild, opp)
$\wedge\, p_1 \neq p_2$
$\wedge\, p_3 \neq p_4$
$\wedge$cornerp(p_1)$\wedge$cornerp(p_2)$\wedge$middlep(p_3)$\wedge$middlep(p_4))

unsafe_config_4_def: Axiom
unsafe_config_4(kbx, s_n)
$\Leftrightarrow$ ($\exists\, t_1\,, p_1\,, p_2\,, p_3\,, p_4$:
match_turn(kbx, s_n, t_1, opp, 4)
$\wedge$ match_position(kbx, s_n, p_1, wild, wild, com)
$\wedge$ match_position(kbx, s_n, p_2, wild, wild, com)
$\wedge$ match_position(kbx, s_n, p_3, wild, wild, opp)
$\wedge$ match_position(kbx, s_n, p_4, wild, wild, opp)
$\wedge\, p_1 \neq p_2$
$\wedge\, p_3 \neq p_4$
$\wedge$ centerp(p_1)
$\wedge$ cornerp(p_2)
$\wedge$ cornerp(p_3)
$\wedge$ middlep(p_4)
$\wedge\neg$((p_2[row] = p_3[row]$\wedge p_2$[col] = p_4[col])

$\vee$ (p_2[row] = p_4[row]
$\wedge\, p_2$[col] = p_3[col])))

End safe_ttt

initial_state: **Module**

Using ttt_ptypes, kbs, states, facts, sets[fact]

Theory

p, p_1: VAR position
t, t_1: VAR turn
sx: VAR state

initial_positions: **Axiom** $(p \in \text{kb\_inst}(\text{akb}, \text{s0})) \Leftrightarrow p[\text{marker}] = \text{free}$

initial_turn: **Axiom**
$(t \in \text{kb\_inst}(\text{akb}, \text{s0})) \Leftrightarrow t[\text{player}] = \text{opp} \wedge t[\text{count}] = 0$

End initial_state

initial_proof: **Module**

Using ttt_ptypes, safe_ttt, states, kbs, initial_state

Theory

sis: **Lemma** safe(akb, s0)

Proof

p: VAR position
rx, cx, mx: VAR datum

next_move: **Lemma** ¬lose_next_move(akb, s0)

nmp: **Prove** next_move **from**
  lose_next_move_def {kbx ← akb, s_n ← s0},
  initial_positions {p ← p_1@p1s},
  distinct_constants

config1: **Lemma** ¬unsafe_config_1(akb, s0)

c1p: **Prove** config1 **from**
  unsafe_config_1_def {kbx ← akb, s_n ← s0},
  initial_positions {p ← p_1@p1s},
  distinct_constants

config2: **Lemma** ¬unsafe_config_2(akb, s0)

c2p: **Prove** config2 **from**
  unsafe_config_2_def {kbx ← akb, s_n ← s0},
  initial_positions {p ← p_1@p1s},
  distinct_constants

config3: **Lemma** ¬unsafe_config_3(akb, s0)

c3p: **Prove** config3 **from**
  unsafe_config_3_def {kbx ← akb, s_n ← s0},
  initial_positions {p ← p_1@p1s},
  distinct_constants

config4: **Lemma** ¬unsafe_config_4(akb, s0)

c4p: **Prove** config4 **from**
  unsafe_config_4_def {kbx ← akb, s_n ← s0},
  initial_positions {p ← p_1@p1s},

```
    distinct_constants

sisp: **Prove** sis **from**
    safe_def {kbx ← akb, $s_n$ ← s0},
    next_move,
    config1,
    config2,
    config3,
    config4

**End** initial_proof
```

gwr_proof: **Module**

Using guarded_ops, safe_ttt, ttt_ptypes, states, ttt_rls2

Theory

s_n: VAR state

safe_gwr_step: **Lemma** safe(akb, s_n) $\supset$ safe(akb, g_win_row(akb, s_n))

Proof

no_transition: **Lemma**
    (safe(akb, s_n)$\wedge$g_win_row(akb, s_n) = s_n) $\supset$ safe(akb, g_win_row(akb, s_n))

ntp: **Prove** no_transition

safe_trans: **Lemma**
    (safe(akb, s_n) $\wedge$ g_win_row(akb, s_n) = next(s_n))
        $\supset$ safe(akb, g_win_row(akb, s_n))

stl1: **Lemma**
    g_win_row(akb, s_n) = next(s_n)
        $\supset$ (win_row_lhs(akb, s_n) $\wedge$ win_row(akb, s_n) = next(s_n))

stl1p: **Prove** stl1 **from** g_win_row_def {$s_n \leftarrow s_n$@cs}

stl2: **Lemma** win_row(akb, s_n) = next(s_n) $\supset$ won_game(akb, win_row(akb, s_n))

stl2p: **Prove** stl2 **from**
    win_row_def {$s_1 \leftarrow s_n$@cs, p $\leftarrow p_1$@pls},
    win_row_def {$s_1 \leftarrow s_n$@cs, p $\leftarrow p_2$@pls},
    won_game_def {kbx $\leftarrow$ akb,
        $s_n \leftarrow$ next(s_n@cs),
        $p_1 \leftarrow p_1$@pls,
        $p_2 \leftarrow p_2$@pls,
        $p_3 \leftarrow p_4$@pls},
    distinct_constants

stpp: **Prove** safe_trans **from**
    stl1 {$s_n \leftarrow s_n$@CS},

```
    stl2 {$s_n \leftarrow s_n$@CS},
    safe_def {kbx ← akb, $s_n \leftarrow$ win_row(akb, $s_n$@cs)}
sgsl1: **Lemma** win_row(akb, $s_n$) = $s_n$ ∨ win_row(akb, $s_n$) = next($s_n$)
sgsl1p: **Prove** sgsl1 **from** win_row_def {$s_1 \leftarrow s_n$@cs}
safe_gwr_step_proof: **Prove** safe_gwr_step **from**
    no_transition {$s_n \leftarrow s_n$@cs},
    safe_trans {$s_n \leftarrow s_n$@cs},
    sgsl1 {$s_n \leftarrow s_n$@cs},
    g_win_row_def {$s_n \leftarrow s_n$@cs}
**End** gwr_proof
```

gwc_proof: **Module**

Using guarded_ops, safe_ttt, ttt_ptypes, states, ttt_rls2

Theory

s_n: VAR state

safe_gwc_step: **Lemma** safe(akb, s_n) $\supset$ safe(akb, g_win_col(akb, s_n))

Proof

no_transition: **Lemma**
    (safe(akb, s_n) $\wedge$ g_win_col(akb, s_n) = s_n) $\supset$ safe(akb, g_win_col(akb, s_n))

ntp: **Prove** no_transition

safe_trans: **Lemma**
    (safe(akb, s_n) $\wedge$ g_win_col(akb, s_n) = next(s_n))
        $\supset$ safe(akb, g_win_col(akb, s_n))

stl1: **Lemma**
    g_win_col(akb, s_n) = next(s_n)
        $\supset$ (win_col_lhs(akb, s_n) $\wedge$ win_col(akb, s_n) = next(s_n))

stl1p: **Prove** stl1 **from** g_win_col_def {$s_n \leftarrow s_n$@cs}

stl2: **Lemma** win_col(akb, s_n) = next(s_n) $\supset$ won_game(akb, win_col(akb, s_n))

stl2p: **Prove** stl2 **from**
    win_col_def {$s_1 \leftarrow s_n$@cs, p $\leftarrow p_1$@pls},
    win_col_def {$s_1 \leftarrow s_n$@cs, p $\leftarrow p_2$@pls},
    won_game_def {kbx $\leftarrow$ akb,
        $s_n \leftarrow$ next(s_n@cs),
        $p_1 \leftarrow p_1$@pls,
        $p_2 \leftarrow p_2$@pls,
        $p_3 \leftarrow p_4$@pls},
    distinct_constants

stpp: **Prove** safe_trans **from**
    stl1 {$s_n \leftarrow s_n$@CS},

stl2 $\{s_n \leftarrow s_n@\mathrm{CS}\}$,
safe_def $\{\mathrm{kbx} \leftarrow \mathrm{akb}, s_n \leftarrow \mathrm{win\_col}(\mathrm{akb}, s_n@\mathrm{cs})\}$

sgsl1: **Lemma** $\mathrm{win\_col}(\mathrm{akb}, s_n) = s_n \vee \mathrm{win\_col}(\mathrm{akb}, s_n) = \mathrm{next}(s_n)$

sgsl1p: **Prove** sgsl1 **from** win_col_def $\{s_1 \leftarrow s_n@\mathrm{cs}\}$

safe_gwc_step_proof: **Prove** safe_gwc_step **from**
no_transition $\{s_n \leftarrow s_n@\mathrm{cs}\}$,
safe_trans $\{s_n \leftarrow s_n@\mathrm{cs}\}$,
sgsl1 $\{s_n \leftarrow s_n@\mathrm{cs}\}$,
g_win_col_def $\{s_n \leftarrow s_n@\mathrm{cs}\}$

End gwc_proof

gwd_proof: **Module**

Using guarded_ops, safe_ttt, ttt_ptypes, states, ttt_rls2

Theory

s_n: VAR state

safe_gwd_step: **Lemma** safe(akb, s_n) $\supset$ safe(akb, g_win_dia(akb, s_n))

Proof

no_transition: **Lemma**
    (safe(akb, s_n)$\wedge$g_win_dia(akb, s_n) = s_n) $\supset$ safe(akb, g_win_dia(akb, s_n))

ntp: **Prove** no_transition

safe_trans: **Lemma**
    (safe(akb, s_n) $\wedge$ g_win_dia(akb, s_n) = next(s_n))
        $\supset$ safe(akb, g_win_dia(akb, s_n))

stl1: **Lemma**
    g_win_dia(akb, s_n) = next(s_n)
        $\supset$ (win_dia_lhs(akb, s_n) $\wedge$ win_dia(akb, s_n) = next(s_n))

stl1p: **Prove** stl1 **from** g_win_dia_def {$s_n \leftarrow s_n$@cs}

stl2: **Lemma** win_dia(akb, s_n) = next(s_n) $\supset$ won_game(akb, win_dia(akb, s_n))

stl2p: **Prove** stl2 **from**
    win_dia_def {$s_1 \leftarrow s_n$@cs, p $\leftarrow p_1$@pls},
    win_dia_def {$s_1 \leftarrow s_n$@cs, p $\leftarrow p_2$@pls},
    won_game_def {kbx $\leftarrow$ akb,
      $s_n \leftarrow$ next(s_n@cs),
      $p_1 \leftarrow p_1$@pls,
      $p_2 \leftarrow p_2$@pls,
      $p_3 \leftarrow p_4$@pls},
    distinct_constants

stpp: **Prove** safe_trans **from**
    stl1 {$s_n \leftarrow s_n$@CS},

stl2 $\{s_n \leftarrow s_n@CS\}$,
safe_def $\{kbx \leftarrow akb, s_n \leftarrow win\_dia(akb, s_n@cs)\}$

sgsl1: **Lemma** win_dia(akb, s_n) $= s_n \vee$ win_dia(akb, s_n) $=$ next(s_n)

sgsl1p: **Prove** sgsl1 **from** win_dia_def $\{s_1 \leftarrow s_n@cs\}$

safe_gwd_step_proof: **Prove** safe_gwd_step **from**
no_transition $\{s_n \leftarrow s_n@cs\}$,
safe_trans $\{s_n \leftarrow s_n@cs\}$,
sgsl1 $\{s_n \leftarrow s_n@cs\}$,
g_win_dia_def $\{s_n \leftarrow s_n@cs\}$

End gwd_proof

Appendix E

Backward-Chaining System

```
(defun remember (new)
  "Places a fact on the *facts* list if it is not already there."
  (cond ((member new *facts* :test #'equal) nil)
(t (setf *facts* (cons new *facts*))
   new)))

(defun recall (fact)
  "Tests to see if a fact is already asserted on the *facts* list."
  (cond ((member fact *facts* :test #'equal) fact)
(t nil)))

(defun failed (fact)
  "Tests to see if a previous attempt to assert the fact failed."
  (cond ((and (not (recall fact))
      (member fact *asked* :test #'equal))
 t)
(t nil)))

(defun testif+ (rule)
  "Attempts to verify if each condition in the ifs clause
 can be satisfied."
  (prog (ifs)
(setf ifs (cdaddr rule))
(return
 (loop
  (cond ((null ifs)(return t))
((if (eql (caar ifs) 'not)
     (not (verify (cadar ifs)))
   (verify (car ifs)))
 (setf ifs (cdr ifs)))
(t (return nil)))))))

(defun testif (rule)
  "Tests to see if each conditions in the ifs clause
 has already been asserted."
  (prog (ifs)
(setf ifs (cdaddr rule))
(return
 (loop
  (cond ((null ifs)(return t))
((if (eql (caar ifs) 'not)
     (failed (cadar ifs))
   (recall (car ifs)))
 (setf ifs (cdr ifs)))
(t (return nil)))))))

(defun inthen (fact)
  "Returns a list of rules for which fact occurs on the RHS."
  (mapcan #'(lambda (r)
      (cond ((thenp fact r)
     (list r))))
  *rules*))

(defun thenp (fact rule)
  "Test to see if a fact is in the RHS of a rule."
  (member fact (cadddr rule) :test #'equal))

(defun usethen (rule)
  "Asserts the RHS of a rule whose LHS has been justified."
  (let ((thens (cdr (cadddr rule)))(success nil))
    (loop
     (cond ((null thens)(return success))
   ((remember (car thens))
    (format t "~%Rule <~A>~%~tdeduces: ~A"
    (cadr rule)(car thens))
    (setf success t)
    (setf thens (cdr thens))))))

(defun tryrule (rule)
  "Attempts to use a rule."
  (and (testif rule)(usethen rule)))

(defun tryrule+ (rule)
  "Attempts to use a rule.  Uses a recursive call to verify."
  (and (testif+ rule)(usethen rule)))

(defmacro get-question (fact)
  '(cadr (assoc fact *questions* :test #'equal)))

(defun verify (fact)
  "Main procedure which attempts to establish a fact."
  (prog (r1 r2)
(cond ((recall fact)(return t)))
(setf r1 (inthen fact))
(setf r2 r1)
(if (null r1)
    (cond ((member fact *asked* :test #'equal)(return nil))
  ((yes-or-no-p "~%~A?" (get-question fact))
   (remember fact)
   (return t))
  (t (setf *asked* (cons fact *asked*))
(return nil))))
loop1
(cond ((null r1)
       (go loop2))
      ((tryrule (car r1))
       (return t)))
(setf r1 (cdr r1))
(go loop1)
loop2
(cond ((null r2)(go exit))
      ((tryrule+ (car r2))
       (return t)))
(setf r2 (cdr r2))
(go loop2)
exit
(return nil)))

(defun diagnose ()
  "Top level function which conducts diagnosis."
  (let ((pos *hyps*)(*asked* nil)(*facts* nil))
    (declare (special *asked* *facts*))
    (loop
     (cond ((null pos)
    (format t "~%No diagnosis available.")
    (return nil))
   ((verify (car pos))
    (format t "~%Diagnosis:  ~A." (car pos))
    (return (car pos))))
     (setf pos (cdr pos)))))

(defvar *rules* nil
  "The list of current rules in the system.")

(defmacro define-rule (&rest rulebody)
  "A macro to make typing in rules more intuitive --Avoids hav-
ing to type in one long list of rules."
  `(setf *rules*
 (append *rules*
 (list (cons 'rule ',rulebody)))))

(defvar *questions* nil
  "The list of queries to establish ground facts.")

(defmacro phrase-question (fact question)
  "A macro to make typing in questions to be used to substanti-
ate facts more intuitive."
  `(setf *questions*
 (cons (list ',fact ,question)
       *questions*)))
```

```
(defun initialize ()
  (setf *rules* nil)
  (setf *questions* nil))

(defmacro hypothesis (&rest hlist)
  "A macro to make entering hypothesis more intuitive."
  `(setf *hyps* ',hlist))
```

Appendix F

Heuristics for Electrical System Diagnosis

```
;;;
;;; small rule base for diagnosis of problems in an automobile
;;; electrical system.
;;;

(initialize)

(define-rule r1
  (if (engine wont start)
      (starter doesnt turn)
      (bad battery))
  (then (replace or repair battery)))

(phrase-question (engine wont start)
"Does the engine FAIL to start")
(phrase-question (starter doesnt turn)
"Does the starter motor FAIL to turn")

(define-rule r2
  (if (engine wont start)
      (starter doesnt turn)
      (not (bad battery))
      (jump solenoid starter turns)
```

```
        (bad ignition))
    (then (replace or repair ignition)))

(phrase-question (jump solenoid starter turns)
"Does the starter turn normally when a jumper is connected
 across the battery and starter posts of the solenoid")

(define-rule r3
    (if (engine wont start)
        (starter doesnt turn)
        (not (bad battery))
        (not (bad ignition))
        (jump solenoid no response))
    (then (replace or repair solenoid)))

(phrase-question (jump solenoid no response)
"Does the starter FAIL to turn at all when a jumper is
 connected across the battery and starter posts of the solenoid")

(define-rule r4
    (if (engine wont start)
        (starter doesnt turn)
        (not (bad battery))
        (jump solenoid starter turns))
    (then (replace or repair starter)))

(define-rule r5
    (if (engine wont start)
        (starter doesnt turn)
        (jump solenoid starter buzzes))
    (then (replace or repair starter)))

(phrase-question (jump solenoid starter buzzes)
"Does the starter buzz or turn very slowly when a jumper
 is connected across the battery and starter posts of the
 solenoid")


(define-rule r6
  (if (battery case cracked))
  (then (bad battery)))

(phrase-question (battery case cracked)
"Is the battery case cracked")

(define-rule r7
  (if (battery connections corroded))
  (then (bad battery)))

(phrase-question (battery connections corroded)
"Are the battery clamps and posts corroded")

(define-rule r8
  (if (battery cells low))
  (then (bad battery)))

(phrase-question (battery cells low)
"Is the state of charge in any of the cells lower
 than normal")

(define-rule r9
  (if (no power at ignition))
  (then (bad ignition)))

(phrase-question (no power at ignition)
"Does the needle of a voltmeter connected to the start
 post of the solenoid fail to move when the key is jiggled")

(define-rule r10
  (if (flickers at ignition))
  (then (bad ignition)))

(phrase-question (flickers at ignition)
"Does the needle of a voltmeter connected to the start post
 of the solenoid flicker when the key is jiggled")

(hypothesis
 (replace or repair battery)
```

```
(replace or repair ignition)
(replace or repair solenoid)
(replace or repair starter))
```

Appendix G

Derived Expert System Rules for Diagnosis

```
;;
;; simple expert system rule set for diagnosis of problems
;; in a car's electrical system.
;;

(initialize)

(define-rule d1
  (if (power at cells))
  (then (power at battery)))

(define-rule d1a
  (if (cell levels normal))
  (then (power at cells)))

(phrase-question (cell levels normal)
"Are all of the voltage levels of the individual battery
 cells normal")

(define-rule d2
  (if (power at battery)
      (good battery))
  (then (power at ignition)))
```

```
(define-rule d2a
  (if (not (bad battery case))
      (not (bad battery connections)))
  (then (good battery)))

(phrase-question (bad battery case)
"Are there any cracks or damage to the battery case")
(phrase-question (bad battery connections)
"Are the battery posts and cable clamps corroded")

(define-rule d3
  (if (power at ignition)
      (good ignition))
  (then (power at points)
(power at solenoid)))

(define-rule d3a
  (if (not (open ignition switch))
      (not (sporadic ignition)))
  (then (good ignition)))

(phrase-question (open ignition switch)
"Does a voltmeter connected to the starter post of the
 solenoid fail to move when the key is turned to the start position")
(phrase-question (sporadic ignition)
"Does the needle of a voltmeter flicker when the key is
 jiggled")

(define-rule d4
  (if (power at points)
      (good points))
  (then (power at coil)))

(define-rule d4a
  (if (not (point gap off))
      (not (damaged points)))
  (then (good points)))
```

```
(phrase-question (point gap off)
"Does a dwell meter indicate the point gap needs adjusting")
(phrase-question (damaged points)
"Do the points show pitting, excessive wear, or other signs
 of damage")

(define-rule d5
  (if (power at coil)
      (good coil))
  (then (power at distributor)))

(define-rule d5a
  (if (good primary coil resistance)
      (good secondary coil resistance))
  (then (good coil)))

(phrase-question (good primary coil resistance)
"Does an ohmmeter indicate from 1 to 4 ohms of resistance
 in the primary coil")
(phrase-question (good secondary coil resistance)
"Does an ohmmeter indicate from 4,000 to 10,000 ohms of
 resistance in the secondary coil")

(define-rule d6
  (if (power at distributor)
      (good distributor))
  (then (power at plugs)))

(define-rule d6a
  (if (not (bad distributor cap)))
  (then (good distributor)))

(phrase-question (bad distributor cap)
"Does the distributor cap and rotor show signs of burning
 or corrosion")

(define-rule d7
  (if (power at solenoid)
      (good solenoid))
  (then (power at starter)))

(define-rule d7a
  (if (jumped solenoid turns normal))
  (then (good solenoid)))

(phrase-question (jumped solenoid turns normal)
"Does the starter turn normally when a jumper is connected
 between the battery and starter posts of the solenoid")

(define-rule d7b
  (if (jumped solenoid starter buzz))
  (then (good solenoid)))

(phrase-question (jumped solenoid starter buzz)
"Does the starter buzz or turn the engine slowly when a jumper
 is connected between the battery and starter posts of the
 solenoid")

(define-rule d7c
  (if (not (jumped solenoid no response)))
  (then (good solenoid)))

(phrase-question (jumped solenoid no response)
"Does the starter show no response when a jumper is connected
 between the battery and starter posts of the solenoid")

(define-rule d8
  (if (car fails to start)
      (starter does not turn))
  (then (no start)))

(phrase-question (car fails to start)
"Does your car fail to start")
(phrase-question (starter does not turn)
"Does the starter fail to turn when the key is engaged")

(define-rule y1
  (if (car fails to start)
      (not (starter does not turn))
      (power at battery)
      (power at ignition)
      (power at points)
      (power at coil)
      (power at distributor)
      (power at plugs)
      (not (good plugs)))
  (then (replace plugs)))

(phrase-question (good plugs)
"Are the spark plugs undamaged")

(define-rule y2
  (if (car fails to start)
      (not (starter does not turn))
      (power at battery)
      (power at ignition)
      (power at points)
      (power at coil)
      (power at distributor)
      (not (good distributor)))
  (then (replace distributor)))

(define-rule y3
  (if (car fails to start)
      (not (starter does not turn))
      (power at battery)
      (power at ignition)
      (power at points)
      (power at coil)
      (not (good coil)))
  (then (replace coil)))

(define-rule y4
  (if (car fails to start)
      (not (starter does not turn))
      (power at battery)
      (power at ignition)
```

```
    (power at points)
    (not (good points)))
  (then (replace points)))

(define-rule y5
  (if (car fails to start)
      (starter does not turn)
      (power at battery)
      (power at ignition)
      (not (good ignition)))
  (then (replace ignition)))

(define-rule y6
  (if (car fails to start)
      (starter does not turn)
      (power at battery)
      (not (good battery)))
  (then (replace battery)))

(define-rule y7
  (if (car fails to start)
      (starter does not turn)
      (not (power at battery)))
  (then (replace battery)))

(define-rule y8
  (if (car fails to start)
      (starter does not turn)
      (power at battery)
      (power at ignition)
      (power at solenoid)
      (power at starter)
      (not (good starter)))
  (then (replace starter)))

(phrase-question (good starter)
"Does the starter meet specifications when removed
 and bench tested")
```

```
(define-rule y9
  (if (car fails to start)
      (starter does not turn)
      (power at battery)
      (power at ignition)
      (power at solenoid)
      (not (good solenoid)))
  (then (replace solenoid)))

(hypothesis
 (replace starter)
 (replace solenoid)
 (replace plugs)
 (replace distributor)
 (replace coil)
 (replace points)
 (replace ignition)
 (replace battery))
```

Appendix H

Electrical System Formal Specification

components: **Module**

Exporting battery, cells, ignition, points, coil, distributor,
plugs, solenoid, starter, component

Theory

component: TYPE
battery, cells, ignition, points, coil, distributor, plugs, solenoid,
starter: component
c: VAR component

enum: **Axiom**

```
($\forall$ c: $c$ = battery
        $\vee\, c$ = cells
          $\vee\, c$ = ignition
            $\vee\, c$ = points
              $\vee\, c$ = coil
                $\vee c$ = distributor$\vee c$ = plugs$\vee c$ = solenoid$\vee c$ = starter)
```

unique: **Axiom**

```
battery $\neq$ cells
  $\wedge$ battery $\neq$ ignition
  $\wedge$ battery $\neq$ points
  $\wedge$ battery $\neq$ coil
  $\wedge$ battery $\neq$ distributor
  $\wedge$ battery $\neq$ plugs
  $\wedge$ battery $\neq$ solenoid
  $\wedge$ battery $\neq$ starter
  $\wedge$ cells $\neq$ ignition
  $\wedge$ cells $\neq$ points
  $\wedge$ cells $\neq$ coil
  $\wedge$ cells $\neq$ distributor
  $\wedge$ cells $\neq$ plugs
  $\wedge$ cells $\neq$ solenoid
  $\wedge$ cells $\neq$ starter
  $\wedge$ ignition $\neq$ points
  $\wedge$ ignition $\neq$ coil
  $\wedge$ ignition $\neq$ distributor
  $\wedge$ ignition $\neq$ plugs
```

```
    ∧ ignition ≠ solenoid
    ∧ ignition ≠ starter
    ∧ points ≠ coil
    ∧ points ≠ distributor
    ∧ points ≠ plugs
    ∧ points ≠ solenoid
    ∧ points ≠ starter
    ∧ coil ≠ distributor
    ∧ coil ≠ plugs
    ∧ coil ≠ solenoid
    ∧ coil ≠ starter
    ∧ distributor ≠ plugs
    ∧ distributor ≠ solenoid
    ∧ distributor ≠ starter
    ∧ plugs ≠ solenoid
    ∧ plugs ≠ starter
    ∧ solenoid ≠ starter

End components
```

engine_model: **Module**

Using components

Exporting power_at, good

Theory

c1, c2: VAR component
power_at: function[component → bool]
good: function[component → bool]

electrical_system_model: **Axiom**

$$\begin{aligned}
&(\text{power\_at}(\text{battery}) \Leftrightarrow \text{good}(\text{cells}))\\
&\quad \wedge (\text{power\_at}(\text{ignition}) \Leftrightarrow \text{power\_at}(\text{battery}) \wedge \text{good}(\text{battery}))\\
&\quad\quad \wedge (\text{power\_at}(\text{points}) \Leftrightarrow \text{power\_at}(\text{ignition}) \wedge \text{good}(\text{ignition}))\\
&\quad\quad\quad \wedge (\text{power\_at}(\text{coil}) \Leftrightarrow \text{power\_at}(\text{points}) \wedge \text{good}(\text{points}))\\
&\quad\quad\quad\quad \wedge (\text{power\_at}(\text{distributor}) \Leftrightarrow \text{power\_at}(\text{coil}) \wedge \text{good}(\text{coil}))\\
&\quad\quad\quad\quad\quad \wedge(\text{power\_at}(\text{plugs}) \Leftrightarrow \text{power\_at}(\text{distributor})\wedge\text{good}(\text{distributor}))\\
&\quad\quad\quad\quad\quad\quad \wedge(\text{power\_at}(\text{solenoid}) \Leftrightarrow \text{power\_at}(\text{ignition})\wedge\text{good}(\text{ignition}))\\
&\quad\quad\quad\quad\quad\quad\quad \wedge(\text{power\_at}(\text{starter}) \Leftrightarrow \text{power\_at}(\text{solenoid})\wedge\text{good}(\text{solenoid}))
\end{aligned}$$

single_failure: **Axiom** $\neg$good(c1) $\supset$ ($\forall$ c2: c2 $\neq$ c1 $\supset$ good(c2))

Proof

test1: **Lemma** $\neg$good(coil) $\supset$ $\neg$power_at(distributor) $\wedge$ $\neg$power_at(plugs)

t1p: **Prove** test1 **from** electrical_system_model

test2: **Lemma** $\neg$power_at(ignition) $\supset$ $\neg$power_at(starter)

t2p: **Prove** test2 **from** electrical_system_model

test3: **Lemma** power_at(coil) $\supset$ power_at(battery) $\wedge$ good(ignition)

t3p: **Prove** test3 **from** electrical_system_model

End engine_model

test: **Module**

Using components, engine_model

Theory

batt: **Lemma** power_at(battery)

igni: **Lemma** power_at(ignition)

poin: **Lemma** power_at(points)

coil: **Lemma** power_at(coil)

dist: **Lemma** power_at(distributor)

plug: **Lemma** power_at(plugs)

sole: **Lemma** power_at(solenoid)

star: **Lemma** power_at(starter)

End test

predict: **Module**

Using components, engine_model, test

Theory

comp: component = solenoid

cause: **Axiom** $\neg$good(comp)

Proof

```
power_at_batt: Prove test.batt from
    cause,
    electrical_system_model,
    unique,
    single_failure {c1 ← comp, c2 ← cells},
    single_failure {c1 ← comp, c2 ← battery},
    single_failure {c1 ← comp, c2 ← ignition},
    single_failure {c1 ← comp, c2 ← points},
    single_failure {c1 ← comp, c2 ← components.coil},
    single_failure {c1 ← comp, c2 ← distributor},
    single_failure {c1 ← comp, c2 ← plugs},
    single_failure {c1 ← comp, c2 ← solenoid},
    single_failure {c1 ← comp, c2 ← starter}

power_at_igni: Prove test.igni from
    cause,
    electrical_system_model,
    unique,
    single_failure {c1 ← comp, c2 ← cells},
    single_failure {c1 ← comp, c2 ← battery},
    single_failure {c1 ← comp, c2 ← ignition},
    single_failure {c1 ← comp, c2 ← points},
    single_failure {c1 ← comp, c2 ← components.coil},
    single_failure {c1 ← comp, c2 ← distributor},
    single_failure {c1 ← comp, c2 ← plugs},
    single_failure {c1 ← comp, c2 ← solenoid},
    single_failure {c1 ← comp, c2 ← starter}

power_at_poin: Prove test.poin from
    cause,
    electrical_system_model,
      unique,
      single_failure {c1 ← comp, c2 ← cells},
      single_failure {c1 ← comp, c2 ← battery},
      single_failure {c1 ← comp, c2 ← ignition},
      single_failure {c1 ← comp, c2 ← points},
      single_failure {c1 ← comp, c2 ← components.coil},
      single_failure {c1 ← comp, c2 ← distributor},
      single_failure {c1 ← comp, c2 ← plugs},
      single_failure {c1 ← comp, c2 ← solenoid},
      single_failure {c1 ← comp, c2 ← starter}

power_at_coil: Prove test.coil from
      cause,
      electrical_system_model,
      unique,
      single_failure {c1 ← comp, c2 ← cells},
      single_failure {c1 ← comp, c2 ← battery},
      single_failure {c1 ← comp, c2 ← ignition},
      single_failure {c1 ← comp, c2 ← points},
      single_failure {c1 ← comp, c2 ← components.coil},
      single_failure {c1 ← comp, c2 ← distributor},
      single_failure {c1 ← comp, c2 ← plugs},
      single_failure {c1 ← comp, c2 ← solenoid},
      single_failure {c1 ← comp, c2 ← starter}

power_at_dist: Prove test.dist from
      cause,
      electrical_system_model,
      unique,
      single_failure {c1 ← comp, c2 ← cells},
      single_failure {c1 ← comp, c2 ← battery},
      single_failure {c1 ← comp, c2 ← ignition},
      single_failure {c1 ← comp, c2 ← points},
      single_failure {c1 ← comp, c2 ← components.coil},
      single_failure {c1 ← comp, c2 ← distributor},
      single_failure {c1 ← comp, c2 ← plugs},
      single_failure {c1 ← comp, c2 ← solenoid},
      single_failure {c1 ← comp, c2 ← starter}

power_at_plug: Prove test.plug from
```

```
    cause,
    electrical_system_model,
    unique,
    single_failure {c1 ← comp, c2 ← cells},
    single_failure {c1 ← comp, c2 ← battery},
    single_failure {c1 ← comp, c2 ← ignition},
    single_failure {c1 ← comp, c2 ← points},
    single_failure {c1 ← comp, c2 ← components.coil},
    single_failure {c1 ← comp, c2 ← distributor},
    single_failure {c1 ← comp, c2 ← plugs},
    single_failure {c1 ← comp, c2 ← solenoid},
    single_failure {c1 ← comp, c2 ← starter}

power_at_sole: Prove test.sole from
    cause,
    electrical_system_model,
    unique,
    single_failure {c1 ← comp, c2 ← cells},
    single_failure {c1 ← comp, c2 ← battery},
    single_failure {c1 ← comp, c2 ← ignition},
    single_failure {c1 ← comp, c2 ← points},
    single_failure {c1 ← comp, c2 ← components.coil},
    single_failure {c1 ← comp, c2 ← distributor},
    single_failure {c1 ← comp, c2 ← plugs},
    single_failure {c1 ← comp, c2 ← solenoid},
    single_failure {c1 ← comp, c2 ← starter}

power_at_star: Prove test.star from
    cause,
    electrical_system_model,
    unique,
    single_failure {c1 ← comp, c2 ← cells},
    single_failure {c1 ← comp, c2 ← battery},
    single_failure {c1 ← comp, c2 ← ignition},
    single_failure {c1 ← comp, c2 ← points},
    single_failure {c1 ← comp, c2 ← components.coil},
    single_failure {c1 ← comp, c2 ← distributor},
    single_failure {c1 ← comp, c2 ← plugs},
    single_failure {c1 ← comp, c2 ← solenoid},
    single_failure {c1 ← comp, c2 ← starter}
```

End predict

Bibliography

[1] *Introduction to* EHDM. Computer Science Laboratory, SRI International, Menlo Park, CA 94025, September 28, 1988.

[2] EHDM *Specification and Verification System Version 4.1—Preliminary Definition of the* EHDM *Specification Language.* Computer Science Laboratory, SRI International, Menlo Park, CA 94025, September 6, 1988.

[3] EHDM *Specification and Verification System Version 4.1—User's Guide.* Computer Science Laboratory, SRI International, Menlo Park, CA 94025, November 29, 1988.

[4] B. Alpern and F.B. Schneider. Defining liveness. *Information Processing Letters*, 21(4):181–185, October 1985.

[5] Debra Anderson and Charles Ortiz. AALPS: a knowledge-based system for aircraft loading. *IEEE Expert*, 2(4):71–79, Winter 1987.

[6] A. Avižienis and J.C. Laprie. Dependable computing: from concepts to design diversity. *Proceedings of the IEEE*, 74(6):629–638, May 1986.

[7] H. P. Barendregt. *The Lambda Calculus, its Syntax and Semantics.* North-Holland, Amsterdam, 1978.

[8] William R. Bevier. *Kit: A Study in Operating System Verification.* Technical Report 28, Computational Logic Incorporated, Austin, TX, August 1988.

[9] Marc Bezem. Consistency of rule-based expert systems. In *9th International Conference on Automated Deduction (CADE-9)*, pages 151–161, Springer-Verlag Lecture Notes in Computer Science Vol. 310, Argonne, IL, 1988.

[10] Daniel G. Bobrow, editor. ***Qualitative Reasoning about Physical Systems.*** The MIT Press, Cambridge, MA., 1986.

[11] Bruce G. Buchanan and Reid G. Smith. Fundamentals of expert systems. In Joseph F. Traub, Barbara J. Grosz, Butler W. Lampson, and Nils J. Nilsson, editors, ***Annual Review of Computer Science, Volume 3***, pages 23–58, Annual Reviews, Inc., Palo Alto, CA., 1988.

[12] S. Cha, N.G. Leveson, T.J. Shimeall, and J.C. Knight. An empirical study of software error detection using self-checks. In ***Digest of Papers, FTCS 17***, pages 156–161, IEEE Computer Society, Pittsburgh, PA., July 1987.

[13] William F. Clocksin and Christopher S. Mellish. ***Programming in Prolog.*** Springer-Verlag, New York, NY., 1981.

[14] A.J. Cohn. Correctness properties of the Viper block model: the second level. In G. Birtwistle, editor, ***Proceedings of the 1988 Design Verification Conference***, Springer-Verlag, 1989. Also published as University of Cambridge Computer Laboratory Technical Report No. 134.

[15] A.J. Cohn. A proof of correctness of the Viper microprocessor: the first level. In G. Birtwistle and P.A. Subrahmanyam, editors, ***VLSI Specification, Verification and Synthesis***, pages 27–72, Kluwer, 1988.

[16] P.T. Cox and T. Pietrzykowski. Causes for events: their computation and applications. In ***8th International Conference on Automated Deduction (CADE-8)***, pages 608–621, Springer-Verlag Lecture Notes in Computer Science Vol. 230, Oxford, England, 1986.

[17] P.T. Cox and T, Pietrzykowski. General diagnosis by abductive inference. In ***Proceedings 1987 Symposium on Logic Programming***, pages 183–189, IEEE Computer Society, San Francisco, CA., August 1987.

[18] Randall Davis and Walter Hamscher. Model-based reasoning: troubleshooting. In Howard E. Shrobe, editor, ***Exploring Artificial Intelligence: Survey Talks from the National Conferences on Artificial Intelligence***, chapter 8, pages 297–346, Morgan Kaufmann Publishers, Inc, San Mateo, CA., 1988.

[19] Saumya K. Debray and Prateek Mishra. Denotational and operation semantics for Prolog. *Journal of Logic Programming*, 5(1):61–91, March 1988.

[20] E.W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, Englewood Cliffs, NJ, 1976.

[21] Dave E. Eckhardt, Jr. and Larry D. Lee. A theoretical basis for the analysis of multiversion software subject to coincident errors. *IEEE Transactions on Software Engineering*, SE-11(12):1511–1517, December 1985.

[22] Im Flannagan. The consistency of negation as failure. *The Journal of Logic Programming*, 3(2):93–114, July 1986.

[23] Charles L. Forgy. *OPS5 User's Manual*. Technical Report CMU-CS-81-135, Department of Computer Science, Carnegie-Mellon University, Pittburgh, PA., July 1981.

[24] Michael R. Genesereth and Nils J. Nilsson. *Logical Foundations of Artificial Intelligence*. Morgan Kaufmann Publishers Inc., Los Altos, CA., 1987.

[25] Allen Ginsberg. Knowledge-base reduction: a new approach to checking knowledge-bases for inconsistency and redundancy. In *Proceedings, AAAI 88 (Volume 2)*, pages 585–589, Saint Paul, MN., August 1988.

[26] Joseph A. Goguen and Timothy Winkler. *Introducing OBJ*. Technical Report SRI-CSL-88-9, Computer Science Laboratory, SRI International, Menlo Park, CA., August 1988.

[27] M.J.C. Gordon. *Mechanizing Programming Logics in Higher Order Logic*. Technical Report CCSRC-006, SRI International, Cambridge Computer Science Research Centre, Suite 23, Millers Yard, Mill Lane, Cambridge CB2 1RQ, England, September 1988.

[28] Paul Harmon and David King. *Expert Systems: Artificial Intelligence in Business*. John Wiley and sons, New York, NY., 1985.

[29] C. A. R. Hoare. An axiomatic basis for computer programming. *Comm. ACM*, 12(10):576–583, 1969.

[30] Warren A. Hunt, Jr. *The Mechanical Verification of a Microprocessor Design.* Technical Report 6, Computational Logic Incorporated, Austin, TX, 1987.

[31] J.C. King. Symbolic execution and program testing. *Communications of the ACM*, 19(7):385–394, July 1976.

[32] J.C. Knight and N.G. Leveson. An experimental evaluation of the assumption of independence in multiversion programming. *IEEE Transactions on Software Engineering*, SE-12(1):96–109, January 1986.

[33] D.E. Knuth and P.B. Bendix. Simple word problems in universal algebras. In J. Leech, editor, *Computational Problems in Abstract Algebra*, pages 263–293, Pergamon, New York, NY., 1970.

[34] Wm Leler. *Constraint Programming Languages: Their Specification and Generation.* Addison-Wesley, Reading, MA., 1988.

[35] Nancy G. Leveson. Software safety: Why, what and how. *ACM Computing Surveys*, 18(2):125–163, June 1986.

[36] A. Lew. Proof of correctness of decision table programs. *Computer Journal*, 27(3):230–232, 1984.

[37] B. Littlewood and D.R. Miller. A conceptual model of multi-version software. In *Digest of Papers, FTCS 17*, pages 150–155, IEEE Computer Society, Pittsburgh, PA, July 1987.

[38] R. Maes and J.E.M. van Dijk. On the role of ambiguity and incompleteness in the design of decision tables and rule-based systems. *Computer Journal*, 31(6):481–489, 1988.

[39] J Strother Moore. *A Mechanically Verified Language Implementation.* Technical Report 30, Computational Logic Incorporated, Austin, TX, September 1988.

[40] L. Moser, P.M. Melliar-Smith, and R. Schwartz. *Design Verification of SIFT.* Contractor Report 4097, NASA Langley Research Center, Hampton, VA., September 1987.

[41] Dana S. Nau and James A. Reggia. Relationships between deductive and abductive inference in knowledge-based diagnostic problem solving. In Larry Kerschberg, editor, *Expert Database Systems*, pages 549–558, Benjamin Cummings, 1986.

[42] Tin A. Nguyen, Walton A. Perkins, Thomas J. Laffey, and Deanne Pecora. Knowledge base verification. *AI Magazine*, 8(2):65–79, Summer 1987.

[43] David L. Parnas. Software aspects of strategic defense systems. *American Scientist*, 73(5):432–440, September–October 1985.

[44] D.A. Pearce. The induction of fault diagnosis systems from qualitative models. In *Proceedings, AAAI 88 (Volume 1)*, pages 353–357, Saint Paul, MN., August 1988.

[45] Harry E. Pople, Jr. On the mechanization of abductive logic. In *Proceedings 3rd IJCAI*, pages 147–152, Stanford, CA., August 1973.

[46] Raymond Reiter. A theory of diagnosis from first principles. *Artificial Intelligence*, 32:57–95, 1987.

[47] Barry K. Rosen. Tree-manipulating systems and Church-Rosser theorems. *Journal of the ACM*, 20(1):160–187, January 1973.

[48] John Rushby. *Quality measures and Assurance for AI Software*. Technical Report SRI-CSL-88-7R, Computer Science Laboratory, SRI International, Menlo Park, CA., September 1988. (Final Report for SRI Project 4616, Task 5, NASA Contract NSA1 17067), also available as NASA Contractor Report 4187.

[49] Jospeh R. Shoenfield. *Mathematical Logic*. Addison-Wesley, Reading, MA., 1967.

[50] Robert E. Shostak. *An Efficient Decision Procedure for Arithmetic with Function Symbols*. Technical Report CSL–65, Computer Science Lab, SRI International, September 1977.

[51] Robert E. Shostak. On the SUP–INF method for proving Presburger formulas. *Journal of the ACM*, 24(4):529–543, October 1977.

[52] Mark E. Stickel. *A Prolog-like Inference System for Computing Minimum-Cost Abductive Explanations in Natural-Language Interpretation*. Technical Note 451, Artificial Intelligence Center, SRI International, Menlo Park, CA., September 1988. Also presented at the International Computer Science Conference '88, Hong Kong, December 1988.

[53] Motoi Suwa, A. Carlisle Scott, and Edward H. Shortliffe. An approach to verifying completeness and consistency in a rule-based expert system. *AI Magazine*, 3(4):16–21, Fall 1982.

[54] Alan F. Turner, editor. *Chilton's Import Car Repair Manual 1983.* Chilton Book Company, Chilton Way, Radnor, PA, 1982.

[55] F.W. von Henke, J.S. Crow, R. Lee, J.M. Rushby, and R.A. Whitehurst. The EHDM verification environment: an overview. In *Proceedings 11th National Computer Security Conference*, pages 147–155, NBS/NCSC, Baltimore, MD., October 1988.

[56] Patrick Henry Winston and Berthold Klaus Paul Horn. *LISP.* Addison Wesley, Reading, Massachusetts, 1 edition, 1981. ISBN 0-201-08329-9.

NASA National Aeronautics and Space Administration

Report Documentation Page

1. Report No. NASA CR-181827	2. Government Accession No.	3. Recipient's Catalog No.
4. Title and Subtitle Formal Verification of AI Software		5. Report Date February 28, 1989
		6. Performing Organization Code
7. Author(s) John Rushby, R. Alan Whitehurst		8. Performing Organization Report No.
		10. Work Unit No. 549-03-31-03
9. Performing Organization Name and Address SRI International 333 Ravenswood Ave. Menlo Park, CA 94025		11. Contract or Grant No. NAS1-18226
12. Sponsoring Agency Name and Address National Aeronautics and Space Administration Langley Research Center Hampton, VA 23665-5225		13. Type of Report and Period Covered Contractor Report
		14. Sponsoring Agency Code

15. Supplementary Notes

Technical Monitor: Sally C. Johnson, Langley Research Center
Task 5 Final Report

16. Abstract

The application of formal verification techniques to AI software, particularly expert systems, is investigated. Constraint satisfaction and model inversion are identified as two formal specification paradigms for different classes of expert systems. A formal definition of consistency is developed, and the notion of approximate semantics is introduced. Examples are given of how these ideas can be applied in both declarative and imperative forms.

17. Key Words (Suggested by Author(s)) Formal Verification, Artificial Intelligence, Expert Systems, Constraint Satisfaction, Model Inversion	18. Distribution Statement Unclassified-Unlimited Subject Category 61

19. Security Classif. (of this report)	20. Security Classif. (of this page)	21. No. of pages	22. Price
Unclassified	Unclassified	158	