An Inference Engine for Embedded Diagnostic Systems†

Barry R. Fox
Artificial Intelligence Group
McDonnell Douglas Research Laboratories
PO Box 516, St. Louis, MO 63166

Larry T. Brewster
Department of Computer Science
University of Missouri-Rolla
Rolla, MO 65401

Abstract. This paper describes the implementation of an inference engine for embedded diagnostic systems. This system consists of two distinct parts. The first is an off-line compiler which accepts a propositional logical statement of the relationship between facts and conclusions and produces the data structures required by the on-line inference engine. The second part consists of the inference engine and interface routines which accept assertions of fact and return the conclusions which necessarily follow. Three design goals of this inference engine are emphasized. First, it is logically sound. Given a set of assertions it will generate exactly the conclusions which logically follow. At the same time, it will detect any inconsistencies which may propagate from an inconsistent set of assertions or a poorly formulated set of rules. Second, the memory requirements are fixed and the worst-case execution times are bounded at compile time. Third, the data structures and inference algorithms are very simple and well understood. This system has been implemented in Lisp, Pascal, and Modula-2. The data structures and algorithms are described in detail.

Introduction

Advanced aircraft and spacecraft are becoming increasingly reliant on onboard electronic systems. At the same time, onboard electronic systems are becoming increasingly complex, interrelated, and interdependent. Because of this reliance, it is necessary to constantly validate the behavior of onboard electronic systems, and when errors are detected, to quickly identify and isolate the faulty subsystems. Advances in artificial intelligence technology make it possible to construct embedded diagnostic systems which can monitor, validate, diagnose, and when necessary, deactivate critical onboard systems.

Embedded diagnostic systems impose implementation constraints which eliminate the use of most of the commercially available artificial intelligence tools. The implementation language and target processor are often dictated by contract. The memory requirements must be bounded and modest and the inference cycle must be bounded and predictable. The reliance on such systems for mission safety and success dictates that the behavior of the inference engine be demonstrably correct.

Many validation and diagnostic problems can be represented and solved entirely within the framework of zero-order (propositional) logic. For example, those problems which have traditionally been solved using decisions tables, debugging flowcharts, or decision trees involve a finite number of facts having true or false values which in turn are logically related to a finite number of intermediate and final conclusions. Generic facilities for the construction of embedded diagnostic systems are provided in the Zero-Order Environment for Test and Analysis (*Zeta*) described below.

† Research supported in part by the McDonnell Douglas Independent Research and Development Program.

Design Goals

The architecture and implementation of *Zeta* were guided by several design goals. Emphasis was placed on achieving generality and simplicity without sacrificing correctness nor capability.

The first goal was to make the system independent of any specific programming language or computer architecture. This would allow the system to be implemented with familiar programming languages on conventional architectures. At the same time, specific onboard computer architectures would not be eliminated, nor would special Lisp or Prolog architectures be excluded. This goal further requires that the data structures and algorithms be documented in sufficient detail that a new implementation in a new environment can be produced quickly.

The second goal was that the inference engine be generic. It should be possible to adapt a working inference engine to new diagnostic problems simply by creating and compiling the knowledge base for those new problems. A working implementation of the inference engine should be available as re-usable, off-the-shelf software.

The third goal was to make the inference engine demonstrably correct. Given a knowledge base which defines the logical relationship between observations and conclusions and given a set of assertions the inference engine should generate exactly the conclusions which logically follow. At the same time it should detect any inconsistencies which may propagate from an inconsistent set of assertions or a poorly formulated set of rules. Moreover, it should be possible to establish these properties from an analysis of the data structures and algorithms.

The fourth goal was to make the inference engine operate successfully and correctly with partial information. It should be possible to assert facts one at a time. With each assertion, the inference engine should be able to derive exactly those conclusions which follow from the aggregate of the present and preceding assertions.

The final goal was that, given a fixed knowledge base, the inference engine should have time and memory requirements which are both bounded and modest. This goal strongly influenced the choice of data structures and algorithms; it required the introduction of certain optimizations; and it also placed some limitations on the acceptable forms for a knowledge base.

External Knowledge Representation

The input to this system is a formula composed of the logical operators **and**, **or**, **xor**, **not**, and **implies**, parentheses for constructing sub-expressions, and symbols, which denote propositional parameters of the system under consideration. There is no explicit distinction between observations and conclusions. The input formula simply identifies the relevant boolean parameters of the system and the logical relationships between them.

The input formula can be a conjunction of rules of the form (*antecedent* **implies** *consequent*) but the input can be considerably more general than that allowed by most familiar rule-based systems. Most rule-based systems require statements in the form of an implication with additional restrictions that disjunctions (or more complicated expressions)

are not allowed in the consequent. For example, the phrase **(alpha xor beta xor gamma)** is very hard to formulate in a traditional rule-based system without enumerating one rule for each relevant combination of alpha, beta, and gamma. The input to this system can be an arbitrary boolean formula (with some limitations imposed only to eliminate syntactic ambiguity).

The only significant restriction on the input is semantic. It is assumed that the inference engine will only be used for mapping observations into conclusions and will not be used to derive new rules or prove theorems about the interrelationships between rules. For example, given the rule **(((alpha implies beta) and gamma) implies delta)** and a separate rule **(alpha implies beta)** it is certainly true that **(gamma implies delta)**. However, the detection and resolution of all such inter-rule relationships would be too time consuming to perform on-line. To guarantee bounded time and space requirements for the inference cycle, it is assumed that all such combinations of rules have been identified and resolved beforehand.

Internal Knowledge Representation

While the input to this system may be an arbitrary propositional formula, a much more regular structure is required for an efficient on-line inference cycle. For that reason, ZETA is composed of two parts. An off-line compiler which normalizes the given propositional formula and an on-line inference engine which performs the deductive processes. This normalization proceeds in four stages. First, expressions involving **xor** and **implies** are mapped into expressions involving only the operators **and**, **or**, and **not**. Second, all negated sub-expressions are recursively rewritten using deMorgan's law, resulting in a formula composed only of **and, or**, and positive or negative propositional variables. (Hereafter, positive or negative propositional variables will be referred to as *literals*. Third, the given formula is converted to conjunctive normal form through applications of the boolean distributive law, resulting in a conjunction of disjunctive clauses which in turn are composed only of literals. Finally, each disjunctive clause is mapped into a sequent of the form (*conjunction* **implies** *disjunction*). The natural interpretation of a sequent is that the truth of each literal in the antecedent implies the truth of at least one literal in the consequent. By convention, an empty antecedent is implicitly true, while an empty consequent denotes a contradiction. The sequent constructed from a disjunctive clause consists of an empty antecedent and a consequent identically equal to the given clause. Deductive processes which may be applied to sequents are discussed in the next section.

Inference Algorithm

Activities of the on-line inference engine are driven by a series of assertions and retractions of fact. It is useful to allow both assertion and retraction for very practical reasons. During development of a knowledge base, the knowledge engineer may wish to establish a particular state of the inference engine and then explore situations which can emanate from that state. It would be laborious to repeatedly reset the inference engine and then carry it to each situation through a series of assertions. Instead, it is better to be able to assert a fact, to determine its effect, and then to retract that fact in order to explore the immediate effect of other faults or conditions. On-line, this ability can be used to ensure the integrity of the diagnostic process in time critical situations. For instance, some physical conditions are intermittent. In a time-critical situation it would be risky to reset

the inference engine and repeat the entire history of observations and assertions simply because an intermittent physical condition no longer holds. In other situations it may be discovered that a sensor itself is faulty. Again, time may not permit a complete reset of the inference engine just to remove the conclusions derived from that erroneous sensor.

The inference cycle begins by placing an assertion on an agenda of activities to be performed. On each inference cycle one item is removed from the agenda and processed, but additional items may be placed on the agenda as a result.

The first step in processing an assertion is to determine whether it is consistent with the present state of the inference engine. Three conditions may hold. An assertion is an assignment of a boolean value to one of the propositional variables. If the variable is presently undefined, then any assertion for that proposition is considered to be consistent. If the variable presently has a value, and the value to be bound to that proposition is the same, then the assertion is considered to be redundant. However, if the variable presently has a value but it differs from the value to be bound to that proposition, then the assertion is considered to be inconsistent.

The second step in processing an assertion is to derive any conclusions which necessarily follow. This step is unnecessary for redundant assertions and must not be performed for inconsistent assertions. Given the semantic restrictions on the knowledge base discussed above, the derivation of necessary conclusions is both correct and efficient. This is accomplished by two techniques. First, the inference engine makes use of an index, constructed when the knowledge base was compiled, and inspects only those sequents which can potentially produce a conclusion from the given assertion. The index does increase the memory required to store the knowledge base by approximately a factor of two. However, the additional memory requirement can be more than offset by the following guarantee. The time required to process an assertion is independent of the size of the knowledge base; instead, it is related to the number of sequents which contain a given literal, and upon the number of conclusions which depend upon it. Second, instead of evaluating or analyzing a sequent, the inference engine applies a very simple rewriting rule based upon the natural interpretation of a sequent. Given an assertion that a literal **L** should be true, and given a sequent which contains **not-L** in its consequent, then some other literal in the consequent must be true. Hence, remove **not-L** from the consequent and include **L** in the antecedent. If only one literal remains in the consequent after this rewriting, then that literal must be true and an appropriate assertion is placed on the agenda.

A significant advantage of this sequent representation and method of inference is that assertions are reversible by retraction. The inference cycle begins by placing a retraction on the agenda. On each inference cycle one item is removed from the agenda and processed. As before, other retractions may be placed on the agenda as a result.

Like an assertion, a retraction is consistent if the value to be removed is equal to the value which a variable presently holds; it is inconsistent if the value to be removed is the opposite of the value which the variable presently holds; and it is redundant if the variable is presently undefined. There is an additional case. The value to be removed may match the value which the variable presently holds, but that value may be required or supported by the consequent of some sequent. Therefore that retraction would introduce an inconsistency if performed and is considered to be impossible. As with an assertion,

a retraction is performed only if it is consistent with the present state of the inference engine.

The process of rewriting sequents under retraction is the reverse of the assertion process described above. Given a retraction of some literal **L**, and given a sequent which contains **L** in its antecedent, first inspect the consequent of that rule. If only one literal remains in the consequent before this rewriting, then this retraction may necessitate the retraction of the consequent as well. If no other sequent supports this consequent, then place the appropriate retraction on the agenda and perform the rewriting: remove **L** from the antecedent and include **not-L** in the consequent. If the consequent has other support before the rewriting, or if the consequent contains more than one literal, then perform only the rewriting step.

Architecture

The architecture of this system can be sketched at two levels. At the highest level the system consists of two separate programs. The first accepts a propositional formula which defines the logical relationship between the boolean parameters of the system to be monitored. The output of the first program is the program fragments necessary to declare and initialize the data structures for the second program. The second program is produced by combining these program fragments with the off-the-shelf code for the inference engine. This two-stage process removes any parsing and normalization costs from the on-line system and it produces a diagnostic program of minimal size.

The lower level structure of the first program is not significant. The program can be viewed as a black box which performs the compilation function. The structure of the second program is significant. It provides the interface to the inference engine. This interface includes an initializing procedure which resets all propositional values to undefined and rewrites every sequent to the form **(() implies disjunction)**. There is a procedure for making an assertion which requires two parameters, a propositional identifier and a value to be bound to that variable, and which initiates the inference cycle described above. There is a procedure for making a retraction which requires only a propositional identifier, assuming that the user wishes to retract the present value of the given proposition. All deductive results produced during an inference cycle are placed on a stack; these results can be retrieved one at a time by simple procedure calls. Other procedures exist which will return the present value of a proposition, its symbolic name, etc.

Conclusion

The Zero-Order Environment for Test and Analysis system has characteristics which make it suitable for embedded diagnostic systems. Notably, the representation and methods of inference are independent of any specific programming language or computer architecture; the time and memory requirements are modest and the upper bounds may be determined prior to run time; the inference engine is generic and can be adapted to new applications by introducing a new knowledge base; the inference engine is demonstrably correct generating exactly the conclusions which follow from a given set of assertions while detecting any inconsistency; the inference engine complements the facility for incremental assertion with a facility for incremental retraction.