The Correspondence Between Methods of Artificial Intelligence and the Production and Maintenance of Evolutionary Software

C.V. Ramamoorthy
EECS Department
U C Berkeley

Daniel E. Cooke
CS Department
U T El Paso

Abstract
In this paper we explore how methods of truth maintenance may be applied in the engineering of integrated, dynamic systems. In the life of a large system, it is not possible to foresee every significant event which may affect the behavior of the system. Truth maintenance techniques may prove useful in the production and maintenance of these large systems. We discuss how it is the decision making aspects of a system which change. Both the information on which we base our decisions and our strategies for decision making change. We discuss truth maintenance with respect to preserving the models of specifications and the models of programs.

1. Introduction
Even if our goals in life do not change, events occur which cause us to alter our plans. When this happens, we readjust our plans in order to resolve new information with respect to the static goals.

Compare a program to an organism. We do not plan the life of an organism, but we typically try to plan the life of a program in great detail. If a program is correct, the specifications are valid, and the program's environment remains consistent with assumptions followed in the program's development, the author of the program is capable of predicting with great accuracy, program behavior.

In a large integrated system, events which should impact the behavior of a program are not always predictable. A program needs to be capable of adjusting itself to changes in its environment while maintaining the goals and the intrinsic truth of the program. In other words, in most systems, we should carefully plan, not for every possible contingency, but instead for strategies for contending with change.

In the future, larger, more dynamic, and more integrated systems will be built. These systems will have goals and plans, and unforeseen events will occur which the system itself will have to resolve with respect to existing plans and goals. We see the need to maintain the truth of systems on a temporal basis. This paper explores the promise of truth maintenance with respect to dynamic software systems.

2. Background
Lehman gives definitions for two types of programs.[3] The types are based upon the nature of the program's specification. An S-type program is a program which is correct w.r.t. specifications which do not change over time; while an E-type program is one which is correct w.r.t. specifications which may indeed change over time, hence the reference to the evolution of software in Lehman's paper. A large system is typically comprised of some mixture of the two types of programs.

The reason an E-type program evolves is due to the number of assumptions coded into an E-type program. Unfortunately, these assumptions may change over time. Lehman [4] gives a dramatic example of a ship in the Falkland island war. The software system defending the ship was based, in part, on the assumption that an EXOCET missile is friendly. This assumption had been true until the Falkland island war in which a British ship was sunk by an EXOCET.

The greatest successes in automatic programming have occurred with S-type programs.[4] This and other points of this discussion suggest an important relationship between Lehman's S- and E-type programs and monotonic and nonmonotonic (NM) reasoning. In monotonic reasoning we see more of the traditional mathematical reasoning wherein once a truth is established it is forever true and we can build upon the truth. NM reasoning approximates human thinking much better. We have knowledge we know to be true and knowledge we believe to be true. In other words in NM reasoning, we have belief systems which are analogous to Lehman's assumptions. In order to have true automatic programming systems we must employ nonmonotonic and monotonic reasoning to accommodate both types of programs.

Lehman's types of programs are best described as a hierarchy of programs where E-type has precedence. In a program there may exist several functions and/or procedures. At this level of granularity we may allow for a stricter classification. In other words a given procedure may be S-type, but if the procedure is a part of a program containing an E-type procedure, then the program itself is E-type.

In addition, in order to clarify the nature of the program types we prefer calling S-type procedures computational and E-type procedures decision making. We feel that renaming the program types facilitates a more intuitive understanding of the nature of these types of programs. The focus of this paper is the evolutionary aspects of decision making programs. The goal of this paper is to demonstrate the correspondence between the problems addressed by techniques in nonmonotonic reasoning and the problems caused by...
evolutionary software, and hence we intend to demonstrate the promise NM reasoning holds as the basis for software engineering tools to assist in the production and maintenance of systems which are based on changing specifications. In NM reasoning we have ways of contending with inconsistencies which arise in an evolutionary environment. Thus, we believe that a specification language which serves as the input to an Automatic Programming tool must incorporate features of NM reasoning. Although we focus on NM reasoning here, fuzzy logic may apply as well. One final note: when we refer to evolutionary software, we are referring to the decision making units of software systems.

3. Truth Maintenance

One of the major problems with evolutionary software is truth maintenance. Suppose the life of a software system is fifteen years and when the software is first released, there is a great deal of confidence that the software is correct. Over the years a number of "improvements" are made to the system. There is a potential for the alterations to the software to actually erode its correctness. In other words, we may improve the functionality of the system while we erode the correctness of the system. If the erosion occurs in a non-critical aspect of the system, it may not be noticed. The best case results when changes are made which preserve the truth of the original system. We need to be sure that the changes to a system are consistent with the rest of the system. Such an assurance requires that we have the ability to detect inconsistencies where feasible. For greater detail concerning the area of truth maintenance see de Kleer. [1]

It is important to understand that the way we make decisions changes over time. In order to organize an approach to contend with the evolutionary aspects of software, it is appropriate to realize that there are two complementary elements involved in decision making. The first element is the knowledge we have in a domain where decisions are made. The second element is the rules we follow in making decisions.

Most of the work in truth maintenance has addressed the changing nature of the information used in decision making. Some of the efforts in this area will be reviewed in the next section. The knowledge is organized into two types: those items known to be true (or false) and those items believed to be true (or false). It is fundamental to realize that known facts are true forever from a logical standpoint. Therefore, the evolutionary aspects of knowledge have to do with beliefs or assumptions.

EXAMPLE 1. Known facts versus believed facts: It may be true that indeed Bill and Sam are brothers, but it may be believed that Bill and Sam are kind to each other. The known fact which is forever true is that Bill and Sam are brothers. They are now and will forever be brothers. However, it is a belief that Bill and Sam are kind to each other.

The belief given in example 1 may be true or false. In fact it may be true at one point in time, but false in another. Therefore, there is a temporal aspect to beliefs which must be considered. If a decision is to be based on whether Bill and Sam are kind to each other, then the nonmonotonic aspects of the relationship must be considered in order to achieve a good decision.

The second element, the rules used in decision making, has not received as much attention as the changing knowledge. Nevertheless, it is true that the rules we use in decision making indeed change as a result of new knowledge or new capability.

EXAMPLE 2. Changing Rules: Consider a defense system for a British ship. At one point in time we may decide to shoot down any incoming missile detected. Therefore, the rule is: if X is incoming then shoot down X. As our sophistication improves we may later alter the rule to: if X is incoming and X has the radar signature of an enemy missile then shoot down X.

An alteration to our knowledge or our rules for decision making can affect the truth of our decisions. Therefore both elements of decision making must be considered when it is desirable to maintain the truth of an evolutionary system.

4. Truth Maintenance of Knowledge in NM Reasoning Systems

In this section some of the efforts to maintain truth in NM systems is reviewed. In order to understand the work discussed, a few definitions are needed. Consider a language L consisting of predicates P, Q, ..., object variables; function symbols; and the logic operators ∧, →, ∨, ¬, and ∀. Formulas are defined in the usual way. For convenience we will often substitute α → β with β ← α.

Assume a set of well formed formulae T in language L. An interpretation of T is a domain (a set of objects) D; a function that assigns to each predicate of T an arity of corresponding arity; a function that assigns an n-ary operator for each n-ary function symbol of T where the operator maps from D to D; and a function that assigns to each constant of T an element of D. An interpretation satisfies T if and only if the assignment makes T true. A model of T is an interpretation which satisfies T. An alternate definition of a model is, given well-formed formulae of T, (F₁, F₂, ..., Fₙ), an interpretation of T is a model if the interpretation makes (F₁ ∧ F₂ ∧ ... ∧ Fₙ) true.

The semantics of a given logic program T are described in terms of models. Therefore, truth maintenance has to do with maintaining the models of a given program as it undergoes change. A given program may have more than one model and truth maintenance may involve the preservation of all of those models. In the worst case, a change to a program may eliminate all models of the program, rendering the program inconsistent.

In NML (Nonmonotonic logic) [5] an operator for consistency, M, is introduced.

EXAMPLE 3.
\[ ∀x∀y(\text{GetAlong}(x,y) \leftarrow \text{Related}(x,y) \land \text{MKind}(x,y) \land \text{MKind}(y,x)) \]
This statement reads: for any \( x \) and any \( y \) if \( x \) and \( y \) are related and it is consistent with the rest of our knowledge that \( x \) is kind to \( y \) and \( y \) is kind to \( x \) then we conclude that \( x \) and \( y \) get along. Since proving consistency is undecidable, this methodology follows the lead of resolution and attempts to show that literals augmented by \( M \) are not true. If we fail to show the \( M \)-literals false then we conclude that the \( M \)-literals are consistent with our knowledge.

In NML, then, the desire is to show that if we cannot prove that a belief is false, then we conclude that it is a consistent belief.

Default logic [7] is similar to NML. A basic difference between DL and NML is that when there are conflicting results from our knowledge (these results are called extensions - not models), NML determines theoremhood based on the intersection of extensions while DL admits a rule as a theorem if it is valid in some extension. DL does not, however, provide any mechanism for choosing among theorems.

Inheritance [8] attempts to accommodate changing information based on a simple concept. Many of our beliefs are generalizations. Beliefs concerning generalizations change as more precise information becomes available. For example, I may believe that all birds fly and that Sam is a bird. If I later discover that penguins are abnormal birds with respect to the property of flying and that Sam is a penguin, then I alter my belief concerning the bird generalization and the penguin exception. [1]

Autopoietic logic (AEL) employs the inheritance principles. [6] AEL introduces a belief operator which works on a predicate \( AB \) which means abnormal. For example, if \( x \) is a bird and I do not have reason to believe that \( x \) is abnormal with respect to flying, then I conclude that \( x \) can fly. Although the \( not \) in this example is based on the closed world assumption, AEL also provides for classical negation.

AEL can be augmented with temporal information. For example, I may have the belief that if someone shoots me with a gun I will die unless the gun is unloaded. A temporal aspect exists to this general belief. If I know that the gun has been previously loaded and not yet shot, then if someone shoots me I must conclude that I will die. Assuming that a gun can only hold one bullet: if the gun is loaded, shot, and then shot at me, I can conclude that I will not die. AEL provides a very robust way of dealing with changing facts in planning systems.

As stated previously, maintaining the truth of changing facts is not enough to preserve the truth of decision making programs. We must also employ methods to maintain the truth of such programs when the rules used for decision making are altered. This is the topic of the next section.

5. Changing Decision Making Rules

Clearly, truth maintenance involves the preservation of models of formulae. When the concern for truth maintenance is applied to the area of automatic programming, two concerns arise: the preservation of the models of specifications and the preservation of the models of programs. We term the preservation of the models of a specification the definitional concern and the preservation of the models of a program the operational concern.

The model of a program can be viewed as the output set of the program. Therefore, a concern for truth maintenance has to consider the changes in output sets of the program as the program undergoes changes.

The models of a specification are the correct programs implied by the specification. If specifications are given at a sufficiently high enough level of abstraction, several correct programs are implied by the specifications, and a subset of these programs may be nonequivalent. Therefore, due to the inherent ambiguity in high level specifications, many models of the specifications may exist. Truth maintenance involves the preservation of these models. Feasible methods which find the correct programs implied by a specification are of great importance when it comes to truth maintenance.

In fact, it seems that the problems inherent in specification languages (i.e., inconsistency, incompleteness, and ambiguity) have some impact on truth maintenance of specifications and/or programs. The implications of each will be discussed in this section. We shall focus on the problems of inconsistency and exceptions to general rules as they apply to the models of programs. We leave open the application of these problems to specifications. We then consider ambiguity resulting from incompleteness as it applies to both the models of specifications and the models of programs.

5.1 Inconsistency and its Impact on the Models of Programs

In this section, we consider an alteration in decision making rules which produces the worst case in a program: inconsistency. In other words, we have program \( T' \) and the models of \( T' \). We change \( T' \) to achieve \( T \) and there are no models of \( T \). In an inconsistent database \( \Delta \) we have the ability to derive literal \( p \) and \( \neg p \) for some literal \( p \in \Delta \).

Suppose the following consistent program is used to compute the tax an individual owes. An individual pays a ten percent tax if his/her income is more than 11,000 dollars:

**EXAMPLE 4.**

(1) \( \text{you_pay}(I,T) \leftarrow \text{tax_owed}(I,T) \).

(2) \( \text{you_pay}(I,T) \leftarrow \neg \text{tax_owed}(I,T) \).

(3) \( \text{tax_owed}(I,T) \leftarrow I > 11000 \land T = 1 * 0.10 \).

(4) \( \neg \text{tax_owed}(I,0) \leftarrow I * 0.10 =< 1100 \).

A change in the tax law occurs and someone must now make 11,000 or more dollars in order to pay ten percent. Suppose we change rule 3 but forget to change rule 4.

**EXAMPLE 5.**

(1) \( \text{you_pay}(I,T) \leftarrow \text{tax_owed}(I,T) \).

(2) \( \text{you_pay}(I,T) \leftarrow \neg (\text{tax_owed}(I,T)) \).

(3) \( \text{tax_owed}(I,T) \leftarrow I > 11000 \land T = 1 * 0.10 \).

(4) \( \neg (\text{tax_owed}(I,0)) \leftarrow I * 0.10 =< 1100 \).
As a result, for an individual making exactly 11 thousand dollars we can achieve both tax owed and not(tax owed). This example demonstrates a form of inconsistency which is prevalent in programming: the situation where we can achieve more than one answer for a given variable under non-mutually exclusive conditions.

Inconsistencies in a program are largely due to conflicting definitions. From an operational standpoint, we can at least detect these inconsistencies by distinguishing variables that are iteratively defined from variables which are defined noniteratively. For noniteratively defined variables we prevent multiple assignments. Thus it becomes a simple problem to detect when a variable has multiple definitions.

5.2 Exceptions to General Rules

From the work in NM reasoning, we can identify problems with truth maintenance in a software engineering environment. In this section we will relate the results from NM reasoning to evolutionary software. Consider a very simple tax calculation example in Prolog. First we see very general rules and deal with potential inconsistencies in dealing with exceptions to these rules:

**EXAMPLE 6.**

\[
\text{interest\_paid}(1987, X, \text{Interest\_Paid}, \text{Interest\_Paid}, \text{mortgage}) \leftarrow \text{not}(\text{ab}(X, \text{mortgage})).
\]

\[
\text{interest\_paid}(1987, X, \text{Interest\_Paid}, 0, \text{mortgage}) \leftarrow \text{not}(\text{ab}(X, \text{mortgage})).
\]

In these rules, one sees how we make decisions about deducting interest from our gross income. Notice that the tax year is given with these rules in order to respect the temporal nature of tax laws. Notice that in general we can deduct all of the interest we pay on our home mortgage. However, there are exceptions to this case; for example when an individual owns more than two homes. The \[\text{not}(\text{ab}(X, \text{mortgage}))\] reads "we do not have reason to believe that \(X\) is abnormal with respect to the property of deducting mortgage interest." The notl operator implements the Closed World Assumption in autoepistemic logic. Through this operator, inconsistencies arising from exceptions to general rules are obviated. Therefore, if we are synthesizing programs which make decisions based on general cases, our specification language needs to possess the features accommodating exceptions.

Concerning the temporal information in these rules, one should notice that we can go back to a previous year and see if the taxpayer paid the current year's taxes out of an overpayment in the previous year. Thus we can reduce the current year's taxes automatically based upon the temporal information we have maintained.

Reconsider the example 5 from the previous section. If we make a minor change to these rules using the notl operator, we can easily detect the inconsistency:

**EXAMPLE 7.**

(1) \[\text{you\_pay}(1, T) \leftarrow \text{tax\_owed}(1, T) \land \text{not}(\text{tax\_owed}(1, T)).\]

(2) \[\text{you\_pay}(1, T) \leftarrow \text{not}(\text{tax\_owed}(1, T)) \land \text{not}(\text{tax\_owed}(1, T)).\]

(3) \[\text{you\_pay}(1, \text{inconsistency}).\]

(4) \[\text{tax\_owed}(1, T) \leftarrow I := 11000 \land T = 1 \cdot 0.10.\]

(5) \[\text{not}(X) \leftarrow \text{call}(X) \land !.\]

(6) \[\text{not}(X).\]

In line 5 the exclamation point (!) serves as the standard cut symbol to control backtracking in Prolog. What we are actually saying in example 7 is that we pay taxes if we owe taxes and we do not have reason to believe that we do not owe taxes. In rule 2 we are saying that I do not pay taxes if I do not owe taxes and I do not have reason to believe that I owe taxes. Thus, we accommodate classical negation (as in rule 4) and negation as failure for our beliefs (as in rules 5 and 6). The inconsistency in example 5 is discovered as a result of augmenting the rules in this straightforward manner.

5.3 Incompleteness and Ambiguity

In this subsection, we consider the impact of incompleteness and ambiguity on truth maintenance. We consider both the operational and the definitional concerns.

5.3.1 Impact on the Models of Programs

As mentioned earlier, in an evolutionary system we may wish to preserve all of the models of an earlier version of the system. A modification to the system may not result in an inconsistency, but may cause us to lose models of an earlier version. Gelfond [2] has devised a way to syntactically detect alternations of logic programs which render the new version "unstable."

Consider the following specification, \(7D\):

**EXAMPLE 8.**

\[\forall x (\text{Citizen}(x) \lor \neg \text{Citizen}(x))\]

There are exactly two models of this formula: \(M_1(7D)\) contains all \(x\) who are citizens while \(M_2(7D)\) contains all \(x\) who are not citizens. Suppose this specification is to be improved to specify all people who pay taxes, yielding \(7T:\)

**EXAMPLE 9.**

\[\forall x ((\text{Citizen}(x) \land \text{PaysTaxes}(x)) \lor \text{PaysTaxes}(x) \land (\text{Citizen}(x) \lor \neg \text{Citizen}(x)))\]

There is exactly one model of \(7T\) and \(M_1(7D) \models M(7T)\). Thus, through an alteration in \(7D\) we have lost a model of \(7T\). It is important to note that although we do not have an inconsistency, we have not preserved the models (or truth) of the original specification.

The problem with truth maintenance in this example is that we began with a complete specification in \(7D\) and the alteration rendered an incomplete specification.
Completeness. Completeness of $T_1$ holds if for every term $t$ of $T_1$ and every predicate $p$ of $T_1, T_1\models p(t)$ or $T_1\models \neg p(t)$.

5.3.2 Impact on the Models of Specifications

When formulae are used to specify programs and the formulae are at a sufficiently high enough level of abstraction, ambiguity occurs. The models of a program specification are the programs which are correct with respect to the specification. Nonequivalent programs are programs which given the same input produce different output.

Def. Ambiguity. If there is more than one model of a specification and the programs of two or more models are nonequivalent, then the specification is ambiguous.

If we are deriving programs from high level specifications and we alter the specifications, we may wish to assure ourselves that all of the models of the original specification are preserved. Thus, truth maintenance in the derivation of programs involves maintaining the models of a specification.

Completeness presents an operational and a definitional problem. The operational aspect can be seen in example 9. In this case, we are interested in preserving the models of the program itself. The definitional problem is that we wish to preserve the models of the specification. In other words we want to be sure that as a result of an alteration to the specification we do not lose a candidate program from the derivation. If we do lose a candidate, we must be able to detect the loss and report it to the specifier. Methods to help accomplish this are beyond the scope of this paper.

6. Summary and Conclusions

There is a nonmonotonic aspect to the production and maintenance of decision making programs. Tools to assist in the production and maintenance of decision making programs must incorporate features of NM reasoning in order to accommodate the changing nature of software. If NM features are not incorporated in specification languages when specifications change, we have no way to know if the changes maintain the truth of the original system from both the definitional and operational standpoints.

An effort to determine what results from NM reasoning are applicable to software engineering is warranted as is an effort to translate these results into corresponding software engineering practices and tools. It is significant to note that, specification languages may require alterations which allow for the separation of specifications which are axiomatic versus those which are beliefs.

The importance of this effort is that it begins to address the problem of large systems operating in dynamic environments wherein it is not possible to predict every event or condition which may affect the system. Instead, we suggest the study of strategies for change which can be embedded in large systems.

REFERENCES


ACKNOWLEDGEMENT

Research sponsored by the Air Force Office of Scientific Research (AFSC), under contract F49620-89-C-0074. The United States Government is authorized to reproduce and distribute reprints for governmental purposes notwithstanding any copyright notation hereon.

118