REASONING WITH SIMPLIFYING ASSUMPTIONS: A METHODOLOGY AND EXAMPLE

Yishai A. Feldman and Charles Rich

The Artificial Intelligence Laboratory
Massachusetts Institute of Technology
545 Technology Square
Cambridge, Mass. 02139

ARPANET: Yishai@MC, Rich@MC

Abstract

Simplifying assumptions are a powerful technique for dealing with complexity, which is used in all branches of science and engineering. This work develops a formal account of this technique in the context of heuristic search and automated reasoning. We also present a methodology for choosing appropriate simplifying assumptions in specific domains, and demonstrate the use of this methodology with an example of reasoning about typed partial functions in an automated programming assistant.

1 Simplifying Assumptions

Simplifying assumptions are a powerful technique for dealing with complexity, which is used in all branches of science and engineering. Stated informally, the basic idea of using simplifying assumptions is: *Don't worry about the details until you have the main story straight.*

For example, in working towards the solution of a difficult physics problem, it is often a good idea to begin by assuming the absence of friction and gravity. Using this simplified world model, it is much easier to explore and evaluate alternative solution approaches. The full complexity of the problem can then be re-introduced later, when you think you have found a viable approach.

Similarly, if you are designing a complex software system, it makes sense to postpone consideration of issues like exception handling and round-off error until you have a design that is plausible with respect to the normal operation of the system.

The role of simplifying assumptions in various types of human problem solving has been studied in previous work [11,8,7]. The contribution of this work is to develop a formal account of this technique in the context of heuristic search and automated reasoning, and to present a methodology for choosing appropriate simplifying assumptions in specific domains.

The techniques discussed here are also closely related to techniques for reasoning with default assumptions, and non-monotonic reasoning generally. However, whereas most of the current work in this field (see [2,1]) focusses on the logical properties of these types of reasoning, this work emphasizes methodological and pragmatic issues. In particular, other current work does not address the questions of how to choose default assumptions and what specific control mechanisms are necessary to reason effectively with such assumptions.

1.1 Heuristic Search

Many types of problem solving can be viewed abstractly as search procedures in which part of the evaluation function involves proving that some logical condition follows from a set of premises defined by the current search state, using a set of axioms which embody the problem solver's "theory of the world." In such situations, the theorem-proving component of the evaluation function is often the dominant cost in the search.

For example, program synthesis can be viewed as searching the space of possible programs (or partial programs) for one that satisfies a given specification and scores well on other evaluation criteria, such as time, space, etc. The premises in each search state encode the structure of the current program candidate. The condition to be verified is the program's specification. The axioms used by the problem solver embody the theory of the various symbols used in defining programs and specifications.

If problem solving is viewed this way, the use of simplifying assumptions amounts to substituting a simplified world theory (set of axioms) for the "correct" one during the search process. When a promising candidate is found using the simplified theory, it is then checked using the full theory. The two key properties of a simplified theory are:

- Proving the relevant conditions from the given premises should be less expensive than in the full theory.
- The answers given by the simplified theory should be good predictors of answers in the full theory. (The formal logical relationship between simplified and full theories is discussed below.)

Simplifying assumptions are thus a kind of heuristic, i.e., task-dependent information which reduces search effort. As with many heuristic search methods, the use of simplifying assump-

This report describes research done at the Artificial Intelligence Laboratory of the Massachusetts Institute of Technology. Support for the laboratory's artificial intelligence research has been provided in part by the Advanced Research Projects Agency of the Department of Defense under Office of Naval Research contract N00014-80-C-0505, in part by National Science Foundation grant MCS-8117633, and in part by the IBM Corporation. Yishai Feldman was supported by a grant from the Bantrell Charitable Trust.

The views and conclusions contained in this document are those of the authors, and should not be interpreted as representing the policies, neither expressed nor implied, of the Department of Defense, of the National Science Foundation, nor of the IBM Corporation.

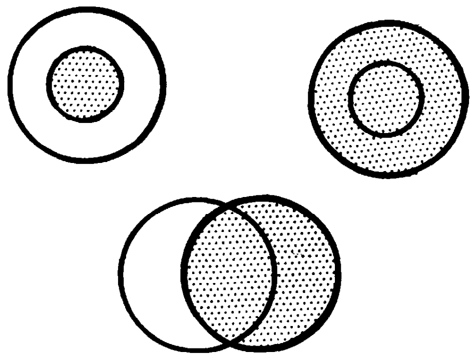

Figure 1: Logical relationship between simplified theory and full theory.

tions reduces search cost, but not without sacrificing the guarantee of finding an optimal solution. Furthermore, the savings in effort is usually seen only in the average over some class of problem instances.

For example, an important part of the theory underlying program synthesis is the theory of typed partial functions (this example is developed in detail in the second half of the paper). The full theory of typed partial functions is somewhat complex, involving the instantiation of two axiom schemas for each function application. Most of this complexity, however, has to do with worrying about what happens when one of the arguments to the function is outside the domain. A simplified theory assumes that the value of a function application is defined, and therefore is in the range of the function.

The methodology described in the next section addresses the key issue of how to derive such simplified theories in general.

1.2 A Methodology for Simplification

The essential basis of the simplification methodology is an analysis of the relationship between the full theory, the simplified theory, and problem being solved.

Figure 1 shows three possible logical relationships between a simplified theory and a full theory. The top left diagram illustrates the case of a simplified theory which is strictly weaker than the full theory (i.e., fewer things are true). This is the easiest case to deal with, corresponding to the common optimization of a two-stage evaluation function. The first stage is the simplified theory, which filters out many candidates at a low cost. The second stage is the more expensive full theory, which is applied only to those candidates that pass the cheaper test. Furthermore, using the dependency-directed techniques described in the following section, proofs developed in the simplified theory are reused in the full theory, if possible.

Unfortunately, in many situations, including the example of typed partial functions, it is not possible to strictly weaken the theory without losing the ability to prove anything useful at all. Also, there is no intrinsic correlation between the "size" of a theory and the cost of proving theorems in that theory (viz. the empty theory and the theory in which everything is true, both of which have trivial proof procedures). Thus in many situations a strictly weaker theory can be more expensive to compute with—for example, because all of the conclusions have more qualifications.

The top right diagram of Figure 1 illustrates a simplified theory which is strictly stronger than the full theory (i.e., more things are true). Intuition for this case can be gained by considering a theory embodied in a set of axioms, each of which is in the form of an implication. Suppose also that the consequent of each implication is a useful conclusion, i.e., a proposition which is likely to advance the reasoning process toward verifying typical search conditions. A simplified theory is one in which we replace each implication by its consequent. Clearly this theory is cheaper than the full theory. However, this strategy can easily lead to contradictions, i.e., an inconsistent theory in which everything is provable.

This brings us to the bottom diagram of Figure 1, in which the two theories mostly overlap, but sometimes differ. We believe this is the most typical case. (The two theories of typed partial functions have this relationship.) Our methodology in this case is based on classifying the propositions appearing in the axioms of the full theory into the following two categories:

- Propositions which, if true, are likely to advance the reasoning process further, e.g., by interaction with other theories. We call these the ***main conclusions*** of the theory.
- Propositions which are normally true, but concern details that can usually be ignored in the first-cut evaluation. We call these the ***default assumptions*** of the theory.

If it is not possible to make these distinctions with some confidence, then the methodology is not applicable to the particular axioms. In order to apply the methodology, it must also be the case that, in the full theory, the default assumptions imply the main conclusions. (Note that it sometimes helps to restate the axioms of a theory in logically equivalent forms in order to facilitate this analysis.)

The axioms of the simplified theory are taken to be the collection of main conclusions. This causes the simplified theory to be partly stronger than the full theory.

It is not necessary for all of the propositions to fall into either of two categories above. For example, there may be propositions which are implied by the default assumptions, but which are not likely to advance the reasoning process or may cause contradictions which do not exist in the original theory. The axioms containing these propositions are omitted from the simplified theory. This causes the simplified theory to be partly weaker than the full theory.

The simplified theory resulting from applying this methodology will be cheaper than the full theory because the deduction required to prove the main conclusions is saved. The simplified theory will be a good predictor of the full theory to the extent the intuitions about the "normal case" are sound.

1.3 Reasoning Facilities

The efficient implemention of the methodology above requires a reasoning system which provides several important control facilities. This section describes these facilities in the abstract. An example of such a reasoning system is described in more detail in the second half of the paper.

The most important control facility which the reasoning system must provide is ***retraction***. Once a condition has been proved using the simplified theory, the reasoning system must be able to undo the proof, and try it again using the full theory. It would also be beneficial if the retraction process was incremental (i.e., the system could use the full theory for some objects under discussion, but not necessarily all) and if the system could exploit parts of the proof which carry over from the simplified to the full theory.

A second important control facility is the ability to handle contradictions explicitly. Despite consideration in the methodology towards keeping a simplified theory internally consistent, it is still possible for interactions between different theories to cause a contradiction. In this situation, the problem solver needs to explicitly detect that the heuristic approach has failed and fall back to the full theories, as opposed to proving everything true.

A mechanism which supports both retraction and contradiction handling is the use of explicit *dependencies* [10]. Dependencies are relations between assertions, which encode proof trees. For each true assertion in the data base, the dependencies record the set of antecedents and the inference rule (axiom) used to deduce it. Premises have the empty set of dependencies.

Dependencies make it possible to retract the truth of an assertion whenever, due to changing circumstances or decisions, one of its antecedents is retracted. If the antecedents later become true again, the dependencies are used to reestablish the consequent assertion without having to rediscover the proof. Dependencies also provide a framework within which to analyze and respond to contradictions, such as by choosing a premise to retract. As an additional benefit, dependencies can also be used to help explain the reasons for the system's conclusions.

In reasoning with simplifying assumptions, the main conclusions of the simplified theory are initially installed as premises. When the axioms of the full theory are installed, these premises are retracted. If the main conclusions can be proved from the full axioms, then the dependencies will cause the previous proof between the main conclusions and the search condition to be reused. Furthermore, since most theories are in the form of universally quantified facts which are instantiated for each object under discussion, there is a separate set of premises for each object, which can be separately retracted.

2 An Example

The example we describe here takes place within the context of building an interactive programming aid, called the Programmer's Apprentice (PA) [5,12]. The overall philosophy of this project and some of the specific technical decisions are important to understanding and motivating the approach we have taken to automated reasoning.

A fundamental tenet of the PA project is that program development, like other engineering activities [6], is an evolutionary process. This means that *change* is the predominant feature of the process—specifications change, design decisions change, bugs are discovered and corrected, and so on. Furthermore, this evolutionary nature is an intrinsic property of large software systems. It is not possible for the designers or potential users of a large system to foresee all of the opportunities for the system's use. Also, the environment in which the system operates is itself subject to change. New regulations, business practices, and technology appear and force modifications to the system.

An implication of this view of the programming process are that, independent of the use of simplifying assumptions, the reasoning component of the PA must support retraction and must be tolerant of contradictions.

2.1 The Full Theory of Typed Partial Functions

Partial functions are an important mathematical construct used to model and reason about the behavior of programs. For example, one of the fundamental properties of computer programs is that sometimes they do not terminate. If we view a program as a function from the inputs to the outputs, such a function will be undefined on those inputs for which it never terminates. Another important use of partial functions is to represent errors, such as division by zero, or an array reference out of bounds.

For simplicity of presentation we will discuss a function of two arguments; extension to the general case will be obvious. As in the usual formulation of partial functions, we introduce an undefined value $\perp$. Let U be the set $\{\perp\}$, and D be the complement of U, namely the set of defined objects in the universe. Let f be a function from $A \times B$ to the range C. One implication of the functionality of f is that if its arguments are in the domains, then its value is in the range. Our first axiom is therefore

A1. $\quad f \in D \wedge x \in A \wedge y \in B \;\Rightarrow\; f(x,y) \in C.$

Note that this axiom includes as one of its antecedents the condition that f is defined. This is because we allow terms in the logic in which the operator is itself a function application and may therefore be undefined. The domains and range of a function term are a syntactic property of the term, interpreted to mean that if the symbol is defined, then it has that functionality.

In this formulation, a *total* function is a function whose range includes only defined objects, i.e., $C \subseteq D$; a *partial* function includes undefined values in its range, i.e. $U \subseteq C$. For example, the functionality of integer addition $(+)$ is Integer $\times$ Integer $\rightarrow$ Integer. The functionality of integer division $(/)$ is Integer $\times$ Integer $\rightarrow$ Integer $\cup\, U$, because the result of dividing by zero is undefined.

An application may also be undefined because one of its arguments is not an element of the corresponding domain, or because the operator is undefined. In some systems, these are treated as syntax errors. However, in our context, since decisions about the properties of objects may change over time, we need to treat these cases within the logic. Our second axiom is therefore

A2. $\quad f \notin D \vee x \notin A \vee y \notin B \;\Rightarrow\; f(x,y) \in U.$

This axiom may or may not be stronger then the converse of A1, depending on whether or not f is total.

2.2 Cake

In order to evaluate the cost benefit of simplifying the theory above, we first need to introduce some further specifics of the reasoning engine we are using. The reasoning component of the PA is called Cake [4]. It incorporates most of the algorithms of McAllester's Reasoning Utility Package [3], such as unit propositional resolution and congruence closure, plus additional decision procedures for some basic algebraic structures, such as partial orders, lattices, and boolean algebras.

The fundamental data structure in Cake is the *term*. Terms are composed of subterms in the usual recursive way. Non-atomic terms are called *applications*. Note that the operator of an application may also be an application. Terms are indexed into a data base which provides input canonicalization (as in a symbol table) and simple associative retrieval.

The basic inference mechanisms of Cake are propositional. Each boolean-valued term (*proposition*) in the data base is associated with a truth value, which is either *true*, *false*, or *unknown*. Propositions are connected into *clauses*, which are the axioms of the system. A clause is a list of *literals*, each of which contains a proposition and a sign specifying whether that proposition appears positively or negatively. A literal is said to be *satisfied* either if the proposition appears positively and its truth value is true, or if the proposition appears negatively and its truth value is false. A literal is *unsatisfiable* either if the proposition appears positively and its truth value is false, or if the proposition appears negatively and its truth value is true.

Deduction occurs when all but one of the literals in a clause are unsatisfiable and the proposition of the remaining literal has the unknown truth value. In this case, the truth value of this proposition is set to the value (true or false) which will satisfy the literal, with dependencies on the other propositions in the clause. If all the literals in a clause are unsatisfiable, a contradiction is signalled, invoking a higher level control structure to decide what to do.

For example, the axiom $P \wedge Q \Rightarrow R \wedge S$ would be installed in Cake by reducing it to conjunctive normal form, giving rise to the two clauses $(\overline{P}, \overline{Q}, R)$ and $(\overline{P}, \overline{Q}, S)$.

If P and Q are true, the system will deduce that R is true and S is true. If R is false and Q is true, the system will deduce that P is false, and so on. The system also supports retraction (setting the truth value of a proposition to unknown) using the dependencies. For example, if R is deduced from P and Q using the first clause above, then if either P or Q is retracted, the system will retract R.

Quantified knowledge is expressed in Cake using the technique of pattern-direction invocation of procedures (demons). Each term in the data base can have associated with it a procedure called a *noticer*. Whenever a new application is created, the noticer associated with the operator term is invoked with the application as its argument. A typical use of such a noticer is to instantiate an axiom schema using the arguments of the application.

The only additional mechanism of Cake that needs to be specified before building an implementation of partial functions is the type algebra. Since the notion of data types is ubiquitous in reasoning about programs, we decided to base Cake on a typed logic. Types in this logic are total functions from $D \cup U$ (the universe of all terms) to Boolean. Types form a boolean algebra with the usual operators of meet, join, and complement. There are special-purpose mechanisms in Cake for performing inferences based on this structure. For example, if T is a subtype of (subsumed by) T', then $T'(x)$ follows from $T(x)$.

2.3 The Cost of the Full Theory

An implementation of the full theory of typed partial functions can now be defined as follows. The basic idea is to instantiate axioms A1 and A2 for each application of f. The domains and range of f are implemented as type predicates. The set D is implemented as the type Defined. (All the usual data types such as Integer, Boolean, etc., are subtypes of Defined.) We install a noticer on the term f which, given a new application $f(x, y)$, creates the following clauses:

$$\left(\overline{\text{Defined}(f)}, \overline{A(x)}, \overline{B(y)}, C(f(x,y))\right),$$

$$\left(\overline{\text{Defined}(f(x,y))}, \text{Defined}(f)\right),$$

$$\left(\overline{\text{Defined}(f(x,y))}, A(x)\right),$$

$$\left(\overline{\text{Defined}(f(x,y))}, B(y)\right).$$

The cost of this implementation can be measured roughly as the number of new data structures created per new application, namely, five new terms and four new clauses, containing ten literals. These new data structures translate into a corrresponding computational cost because, generally speaking, the amount of computation in the reasoning component increases strongly with the number of terms and clauses. In particular, we have found from experience that, due to the activities of the congruence closure algorithm, it is particularly important to control the number of terms created in the system.

A striking feature of this straightforward implementation is that *half* the literals in the clauses above involve terms with the operator Defined. Thus we argue that, especially within the context of evolutionary design, the system is spending a disproportionate amount of its effort worrying about the details of whether things are defined or not.

2.4 Applying Simplifying Assumptions

The application of the methodology to the theory of typed partial functions breaks into two cases, corresponding to whether f is total or partial.

Let us first consider the case when f is total. In this case, the main conclusion of axioms A1 and A2 is $f(x, y) \in C$. This fact is likely to advance the reasoning by eliminating cases or triggering specialized information about the elements of C. For example, C might be the set of positive integers, and the term $f(x, y)$ might appear in a conditional expression of the form

if $f(x, y) > 0$ **then** ... **else** ....

The default assumption of the theory is $f(x, y) \in D$. The "normal" state of affairs in reasoning is that most expressions are defined. The cases wherein certain terms are undefined can safely be considered "detail to be treated later." Note that this default assumption does imply the main conclusion above, as required (this is easy to see by considering the contrapositive of A2).

The role of the remaining propositions in A1 and A2, namely $f \in D$, $x \in A$, and $y \in B$, is interesting to consider for a moment. Logically, these propositions are in fact implied by the default assumption. However, we have chosen not to consider them as main conclusions. The reason for this is that this information is not intrinsic to the form of the terms f, x, or y, but rather to their appearance in a certain context. For example, the same variable x may appear in two applications with different operators having disjoint domains. Although this may not be a contradiction once the details of the reasoning are considered (the two applications may be on opposite sides of a conditional expression which tests the type of x), making these propositions part of the simplified theory could force the system to immediately invoke the details to resolve the contradiction, thereby defeating the whole purpose of the strategy.

The case when f is partial has an additional wrinkle. As mentioned above, for partial functions $U \subseteq C$. In this case, the proposition $f(x,y) \in C$ is not likely to advance the reasoning process. For example, knowing that $f(x,y) \in \text{Integer} \cup U$ is not as useful as knowing that $f(x,y) \in \text{Integer}$. We therefore restate the theory in a logically equivalent form for this case, by replacing axiom A1 with the following simpler axiom, where C' is the set $C \cap D$ (i.e., subtracting out undefined):

A1'. $\quad f(x,y) \in D \Rightarrow f(x,y) \in C'$.

We now take the proposition $f(x,y) \in C'$ as the main conclusion in this case. Note that it is implied by the same default assumption as above, namely $f(x,y) \in D$.

2.5 Implementation

After the theory has been analyzed according to the methodology, an efficient implementation in Cake was achieved as follows.

We install a noticer on the term f which, given a new application $f(x,y)$, creates the term for the main conclusion and makes it a premise. If f is total, this premise is simply $C(f(x,y))$. If f is partial, the procedure computes the type C', obtained by intersecting C with Defined,[1] and installs the premise $C'(f(x,y))$. These premises are also marked by the system as being supported by (implicit) simplifying assumptions.

Thus in the first stages of reasoning, we create only a single term, as compared to the five terms and four clauses of the straightforward implementation. Furthermore, if the main conclusion was well chosen, this premise may advance the reasoning enough to decide to abandon this path regardless of the details.

We also define an operation on premises called *discharging*. When a premise is discharged, its truth value is retracted and, if it is marked as being supported by simplifying assumptions, a procedure is run to instantiate the rest of the underlying axioms.[2] In the case of total functions, discharging the premise causes the same five terms and four clauses to be created as described in the straightforward implementation. In the case of partial functions, A1' is instantiated instead of A1, giving rise to the following clause:

$$\left(\overline{\text{Defined}(f(x,y))}, C'(f(x,y))\right).$$

The total number of terms and clauses eventually created in this case is the same as in the total function case. Notice that the term $C(f(x,y))$ is never created in this case, since it is not usually a useful fact. If, however, this term is created by some other procedure, its truth is provable by the mechanisms of the type lattice from the axioms instantiated here.

[1]This computation is possible since the type hierarchy has been made non-retractable for efficiency reasons. We have implemented a special data structure in the type lattice to support this computation.

[2]Discharging also removes the simplifying assumptions mark, to avoid instantiating the same axioms twice.

Discharging of premises supported by simplifying assumptions can occur in a number of ways. First, the higher level control structure may decide, for its own reasons, that now is the time to pursue the details. For example, the current design may look good enough to warrant spending additional resources working it through. Alternatively, a contradiction may be detected involving some of the marked premises. Rather than simply abandoning one of the current set of premises, the contradiction handler may decide that the contradiction is only apparent and can be resolved by descending to the next level of detail. Finally, we install a noticer on the term Defined which, given a new application of the form Defined$(f(x,y))$, discharges the premise $C(f(x,y))$ or $C'(f(x,y))$, depending on whether f is total or partial. This noticer embodies the heuristic that when you actually create the term for the default assumption, it means you want to begin to consider the details.

We conclude this section with a brief example using the partial function $/$, with functionality $\text{Integer} \times \text{Integer} \rightarrow \text{Integer} \cup U$. In addition to knowing the functionality of $/$, let us assume the system also has the following axiom about the behavior of the function.

D1. $\quad i \in \text{Integer} \wedge j \in \text{Integer} \wedge j \neq 0 \Rightarrow i/j \in \text{Integer}$.

Applying the methodology of simplifying assumptions to this "theory", we decide that the main conclusion is $i/j \in$ Integer, and that the default assumptions are $i \in$ Integer, $j \in$ Integer, and $j \neq 0$. This is implemented by installing a noticer on $/$ which, given a new application i/j, creates the premise for the main conclusion and marks it with the axiom D1 to be instantiated when this premise is discharged. Notice that the same proposition can be the main conclusion of more than one theory. Thus when a premise is discharged, more than one group of underlying axioms may be triggered.

Now suppose that the term i/j is created. By the procedures described above, this will cause the term Integer(i/j) to be created and made a premise. Using this premise, the system may proceed, without stopping to prove that i and j are integers and that $j \neq 0$. For example, further reasoning may reveal that that the computation involving i/j is wrong and that this term should be $i/(j+1)$ instead.

If and when the premise Integer(i/j) is discharged, the following clauses will be installed due to the theory of partial functions:

$$(\overline{\text{Defined}(i/j)}, \text{Integer}(i/j)),$$

$$(\overline{\text{Defined}(i/j)}, \text{Defined}(/)),$$

$$(\overline{\text{Defined}(i/j)}, \text{Integer}(i)),$$

$$(\overline{\text{Defined}(i/j)}, \text{Integer}(j)),$$

and the following clause due to the theory of $/$:

$$(\overline{\text{Integer}(i)}, \overline{\text{Integer}(j)}, \overline{j \neq 0}, \text{Integer}(i/j)).$$

3 Conclusions

We have described a general methodology for using simplifying assumptions in automated reasoning, and have illustrated its application to the implemention of a theory of typed partial functions in the context of evolutionary program development. We believe this methodology can profitably be applied in other areas of reasoning.

The next area in which we plan to apply the methodology is reasoning about side effects. To simplify the first stages of reasoning in this context, it is important to make the default assumption that there is no aliasing (i.e., two variables do not hold pointers to the same data structure or parts of the same data structure). Shrobe [9] has taken a similar approach in this area.

As the reasoning component of the PA develops with many different kinds of simplifying assumptions for different purposes, we imagine the reasoning process will begin to resemble "peeling the layers of an onion." Discharging one level of premises will cause the next lower level of detail to be instantiated, which may have its own simplifying assumptions, and so on. For example, in the reasoning involving applications of / above, we might in fact want to install control mechanisms to allow instantiation of the details of the partial function theory, while keeping the assumption $j \neq 0$.

Another direction of future work we would like to mention here is to partition the undefined type into different sub-types to represent different kinds of exceptional conditions. For example, the term 5/0 is undefined for a different reason than 5/"hello" is undefined, which is different again from the reason that the output of an non-terminating computation is undefined. We expect that the PA will be able to take advantage of these distinctions. Note that this extension would require some modifications to the axioms presented in the paper and to the definitions of partial versus total functions.

Acknowledgements

The authors would like to thank David Chapman and Dick Waters for their help in working out some of the ideas in this paper.

References

[1] AAAI Workshop on Non-Monotonic Reasoning, New Paltz, NY, October 1984.

[2] *Artificial Intelligence*, Vol. 13, No. 1,2, Special Issue on Non-Monotonic Logic, April 1980.

[3] McAllester, D. A., "Reasoning Utility Package User's Manual", MIT Artificial Intelligence Lab. Memo 667, April 1982.

[4] Rich, C., "The Layered Architecture of a System for Reasoning about Programs", *Proc. of the 9th Int. Joint Conf. on Artificial Intelligence*, Los Angeles, CA, August 1985.

[5] Rich, C., and H. Shrobe, "Initial Report on a Lisp Programmer's Apprentice", *IEEE Trans. on Software Eng.*, Vol. 4, No. 6, November 1978.

[6] Rich, C., H. E. Shrobe, R. C. Waters, G. J. Sussman, and C. E. Hewitt, "Programming Viewed as an Engineering Activity", (NSF Proposal), MIT Artificial Intelligence Lab. Memo 459, January 1978.

[7] Rich, C., and R. C. Waters, "The Disciplined Use of Simplifying Assumptions", *Proc. of ACM SIGSOFT Second Software Engineering Symposium: Workshop on Rapid Prototyping, ACM SIGSOFT Software Engineering Notes*, Vol. 7, No. 5, December 1982.

[8] Sacerdoti, E. D., "Planning in a Hierarchy of Abstraction Spaces", *Artificial Intelligence*, Vol. 5, No. 2, 1974.

[9] Shrobe, H. E., "Common-Sense Reasoning About Side Effects to Complex Data Structures", *Proc. of 6th Int. Joint Conf. on Artificial Intelligence*, Tokyo, Japan, August 1979.

[10] Stallman, R. M., and G. J. Sussman, "Forward Reasoning and Dependency Directed Backtracking in a System for Computer-Aided Circuit Analysis", *Artificial Intelligence*, Vol. 9, October 1977, 135-196.

[11] Sussman, G. J., "The Virtuous Nature of Bugs", *Proc. Conf. on Artificial Intelligence and the Simulation of Behavior*, U. of Sussex, July 1974.

[12] Waters, R. C., "The Programmer's Apprentice: A Session with KBEmacs", *IEEE Trans. on Software Eng.*, Vol. 11, No. 11, November 1985.