RANDOMIZING THE KNOWLEDGE ACQUISITION BOTTLENECK

Stuart H. Rubin
Department of Computer Science
Central Michigan University
Mt. Pleasant, MI
rubin@cps.cmich.edu

Michael H. Smith
EECS Department
University of California
Berkeley, CA
mhs@mining.ubc.ca

Ljiljana Trajkovic
School of Engineering Science
Simon Fraser University
Burnaby, B.C., Canada
ljilja@cs.sfu.ca

ABSTRACT

This paper addresses the knowledge acquisition bottleneck. It first takes an information-theoretic view of knowledge acquisition as having a basis in randomization theory and subsequently gives practical examples. The examples are taken from the field of expert compiler technology. Such technology can be used to effect the realization of fourth generation languages. These languages have been shown to be among software engineering's greatest success stories. Expert compilers have the advantage of being easily maintainable and extensible. They randomize translational information in the form of rules. The capture of domain-specific knowledge allows for the construction of context-sensitive languages that are easy to work with. Of course, such languages are necessarily domain-specific; but even here expert compilers lend their advantage of promoting rapid prototyping within similar domains. It follows that if one builds a text-oriented domain-specific language that one can build a more complex expert compiler and so on. Clearly, randomization has application to not just the data, but to the representation of the data as well.

1. INTRODUCTION

Consider the theory of randomization [1], [2] that entails reducing information to its simplest form through the use of compression techniques. This results in a minimal image having maximal complexity since redundancy has, for the most part, been eliminated. The idea of constructing intelligent systems is not new and entails the design, implementation, and embedding of better expert systems. Recall that expert systems are the technologies that underpin methodology drivers. Indeed, they have found their way into software wizards and all manner of intelligent software. In the business world, they are referred to as decision support systems and have been used by DuPont Corporation to save on the order of one billion dollars.

This paper will cover expert systems in terms of improving their weakest link, the knowledge acquisition bottleneck. This weakness can be addressed through the randomization of information, which permits the subsequent reuse of both knowledge and data. Randomization lies at the heart of decreasing the entropy of any information-theoretic system -- just as in inorganic chemistry, the removal of water from an acid accelerates the rate of its participation in a chemical reaction.

2. THE KNOWLEDGE ACQUISITION BOTTLENECK

The knowledge acquisition bottleneck refers to the difficulty of capturing knowledge for use in the system. Whether the system is used for evaluating bank loans, intelligent tutoring, or prescribing medical treatments, one question is: How do we obtain the knowledge used by the system (i.e., code it) and how do we verify it for mission critical applications. For example, the medical rule, *IF the patient is coughing THEN prescribe cough syrup*; is usually true. However, an expert medical system having this limited medical knowledge would prescribe cough syrup to someone whose airway is obstructed during the course of a meal! It is lacking in the more specific rule: *IF the patient is coughing AND the patient was just eating THEN apply Dr. Heimlich's maneuver with certainty factor = 0.90.*

It can be seen from this small example that knowledge acquisition needs to be relatively complete lest there be potentially dire consequences. Now suppose that a knowledge engineer has a task of writing rulebases -- each containing thousands of rules for mission critical applications (e.g., flight-control systems, medical decision support systems, weapons systems, or vehicle guidance systems). It should be clear that the task is too difficult for human beings given the current state of the art in knowledge acquisition. It is only that this software is inherently difficult to test. Hence, in this paper, we propose a new method for cracking the knowledge acquisition bottleneck. Efurthermore, each of the following sections addresses fundamental issues pertinent to its design. Basically, this new method creates and uses second generation expert compilers which translate evolving representational formalisms. The knowledge bases for such compilers can be bootstrapped through the use of such formalisms for knowledge representation. That is, the higher the level of domain-specific language used for the representation of knowledge, the greater the efficacy of the resultant expert compiler. This circularly allows for higher levels of domain-specific language. These higher-level languages have evolved because they are more closely aligned with the way in which we think. For example, *Windows* (or other GUI systems) was developed as a metaphor for our spatial-temporal reasoning. After all, we cannot be easily taught to read machine code for we are not machines.

3. DECISION TREES

Decision trees make it easy for the user to see and modify the knowledge bases in an expert compiler. This is necessary to the evolutionary construction of an expert compiler. This is because decision trees capture knowledge in graphical form. Such knowledge is relatively easy to glean and thus modify. One tool, *XpertRule* by Attar Software Ltd., uses decision trees as its main front-end for solving the knowledge acquisition bottleneck. One of its decision trees is illustrated in Figure 1. The decision tree begins to grow in complexity as the application rules evolve to reflect the complexity of real-world modeling. The result is shown in Figure 2. Decision trees are visual representations of knowledge and, as a consequence, they serve to make it relatively easy to determine which predicates are in a rule and if the rule is valid in this sense. However, they do not facilitate the discovery of which predicates might be in the rule, or which ones are improperly missing. It turns out that the absence of necessary predicates is the surest way to invalidate an otherwise
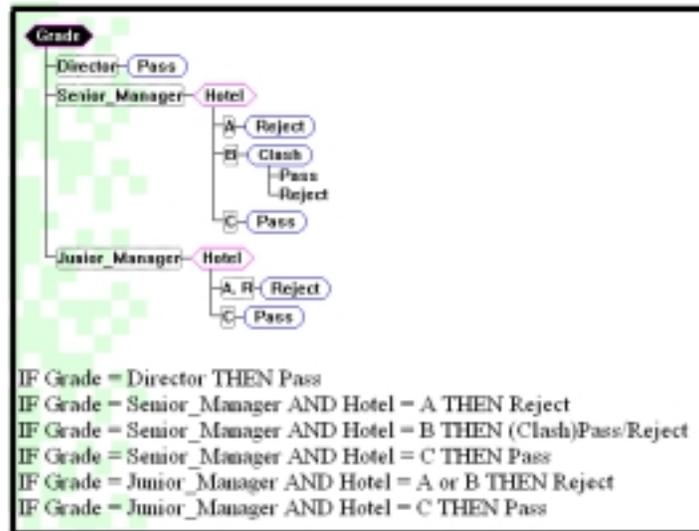
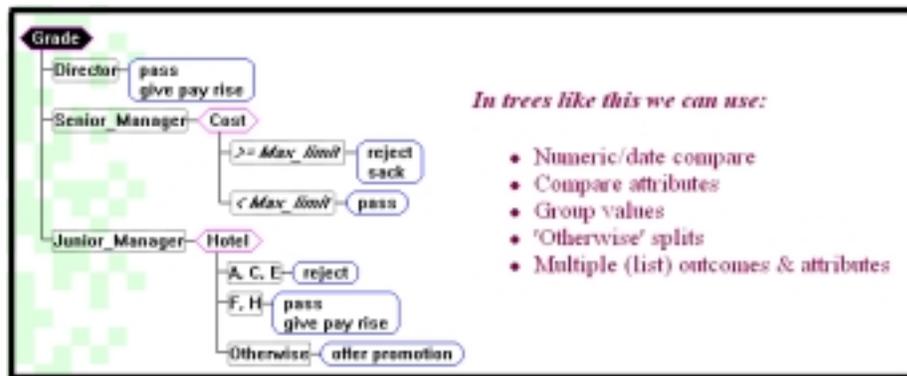**Figure 1.** A decision tee in *XpertRule*.

Figure 2. A more complex decision tree.

valid rule. The problem applies equally to the antecedent and the consequent predicates. Note it is comparatively easy to review the validity of what one has stated. It is much more arduous to find what one is missing -- until it may be too late and the damage is done (e.g., a medical expert system that neglects to ask if a patient is on a certain drug before prescribing another one, which may result in a fatal interaction). It follows that the discovery of what predicates may be missing is the single most significant facet of the knowledge acquisition bottleneck.

4. A THEORETICAL PERSPECTIVE

Computability theory provides us with a fundamental understanding of the nature of the problem [3]. Rule predicates are recursively enumerable, but not recursive. The predicates can only be discovered by search and cannot be functionally characterized. Any search process intrinsically takes time and that is the fundamental problem underlying the knowledge acquisition bottleneck. However, it is proven, through the use of computability theory, that non-trivial knowledge -- necessarily including any means for its acquisition, must be domain-specific, implying a knowledge-based expert approach. Thus, a second generation expert compiler cannot be realized by an algorithm -- rules are intrinsic to its success. However, the purported grammatical inference technique can suggest predicates, but only through the implied use of a heuristic methodology. Otherwise, there would be a contradiction on the non-recursiveness of the problem. However, heuristic search can be extremely powerful

in practice but can fail when the sought knowledge is random relative to that embodied by the grammar. This theoretical limitation is acceptable as the methodology is potentially extremely good at providing symmetric knowledge with little if any search. Again, that is what it offers us towards cracking the knowledge acquisition bottleneck. It can be seen in Figure 3 that one needs to "fall through" a lot of decision points to determine that the food is a vegetable. Nevertheless, this is easier for humans to do visually than if the same information were represented linearly in the form of production rules as the use of color and graphics appeals to our visual cortex. However, observe that the information is organized in the form of a hierarchy. Think of the superclass concept in object-oriented programming. Recall the basic Java syntax: apple extends fruit extends food. The concept of a hierarchy readily lends itself to a visual interpretation. However, it is more than visual -- it is mathematical and logical. It can be used to minimize search through the categorization of information. It can facilitate analogical reasoning. For example, if one needs and does not have a hammer, then one can take the superclass of hammer and instantiate one of its members to serve as a substitute hammer (e.g., a pipe wrench, gas pliers, or a heavy screwdriver). This capability will turn out to be useful when formalized.

5. VISUAL PROGRAMMING

Visual programming technologies represent the embodiment of human factors in expert compiler construction. That is, since
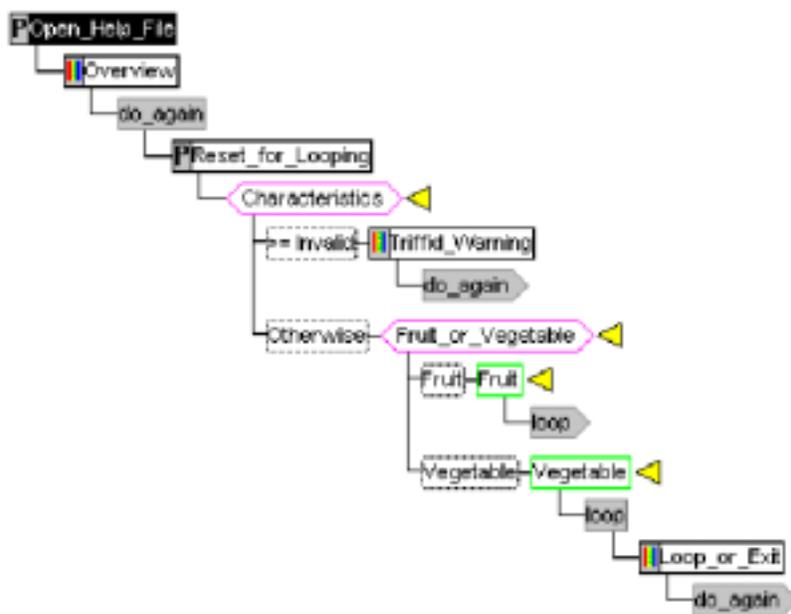
Figure 3. Visual programming in *XpertRule*

people build expert compilers, they need user-friendly interfaces. Figure 3 hints how rules can become so complex that their representation (i.e., regardless of the representational formalism used) approximates a computer program. This can be seen from the use of control constructs, i.e., the decision tree has evolved into a visual programming language.Experienced programmers will note that clarity of exposition is now lost as the tree approximates a flowchart. Of course, the tree may be hierarchical and object-oriented and this offsets the complexity of the view. However, hiding the complexity simultaneously limits its use in pattern matching. Pattern matching is what enables the discovery of symmetry in the tree. Thus, if for example one is trying to code a decreasing sort, one can find a pattern (or subpattern) for increasing sort to assist in one's efforts by way of symmetric reuse. Where does one look? What encapsulated objects does one inspect? Our approach will render the answers to these questions automatic.

6. PSEUDO-CODE

The following sections address the problem of how to effectively crack the knowledge acquisition bottleneck by exploiting the inherent symmetries in rule-base construction. First, pseudo code is discussed as a representational formalism: Pseudo-code can be conceptually thought of as a high-level language, such as structured English, which is used to represent a conceptual/rule-incorporating knowledge, which can be transformed through some effective process into lower-level rules/effective code. Expert compilers supply the knowledge, which drives the transformational process. Figure 3 represents a visual pseudo-code. The idea of automatically translating pseudo-code to effective code originated with the Sparks language to Fortran IV compiler and was later formalized into what is now termed an *expert compiler*. Expert compilers use rules (i.e., expert systems) to translate fourth generation languages and beyond. Fourth generation languages represent the best success story to date in the broad field of software engineering [4]. As a matter of historical record, the first compilers were rule-based. Table-driven compilers are faster for serial processors where a dynamic translation capability is not required. When massive parallelism makes its way to the desktop (as DARPA envisions), then rule-based compilers should once again prove to be competitive.

Consider the following translation from pseudo-code to effective code shown in Table 1. This transformation is realized by one of many schema rules, which collectively define a segment in an expert compiler. In Table 1, the schema is shown in upper case and the slots are defined using lower case. The left side of the table constitutes a transformation rule antecedent and the right side constitutes a transformation rule consequent. Of course, a conflict resolution strategy is needed for this system as it is for any other expert system. Here, the most-specific matching rule should be first to be fired. Note that schemas have a certain duality, i.e., they can be interpreted as rules or cases. The rule interpretation is favored, as it is consistent with universality and grammatical inference.

Table 1. The makings of an expert compiler.

PSEUDO-CODE	EFFECTIVE CODE
CONTINUE TO ADD	REPEAT
y TO x AND	x := x + y
STOP WHEN YOU	UNTIL
GET TO z;	x $\geq$ z;

7. EXPERT COMPILERS

Expert compilers apply domain-specific knowledge bases to the transformation of pseudo-code to render it effective. Such pseudo code can boostrap the expert compiler. Notice that the expert rule shown in Table 1 is of the form, *IF A1 and A2 THEN C*, where *A1* and *A2* are antecedents in pseudo-code and *C* is an effective consequent. A simple knowledge acquisition bottleneck here is given *A1* or *A2*, how does one come up with *A2* or *A1* respectively? The answer lies in the symmetry of our grammatical approach. Any scheme for suggesting an appropriate predicate contributes to the quality of the rule as well as to the quantity of such rules that can be produced. Quantity is increased by reducing the rule design time through the suggestive process. Any methodology for cracking the knowledge acquisition bottleneck in expert systems qualifies as a CASE tool. This is because CASE tools, by definition, serve to

increase programmer productivity. Clearly, expert systems (e.g., helpdesks) fall into this category. That is, if one can more or less automate the process of knowledge acquisition, then one will have commensurately reduced the overhead associated with CASE-tool construction. The utility of such tools, once constructed, will also be increased.

Expert compilers or systems have control mechanisms (i.e., the inference engine), explanation facilities (i.e., the explanation subsystem), and of course at least one knowledge representation language with which to express the rules. The higher the level of the knowledge representation language, the easier it will be to specify complex rules. It then follows that the rules, control mechanism, explanation subsystem, etc. can be bootstrapped (i.e., more rapidly programmed) using the now effective pseudo-code! The more complex the expert compiler becomes, the higher the level of pseudo code that can be realized, which of course implies that a more complex expert compiler can be created and so on. Following this chain of reasoning, one will see that it realizes the ultimate CASE tool. It potentially contributes the most to the twin fields of artificial intelligence and software engineering.

Improve the ability to code and one improves the expert compiler. Improve the expert compiler and one improves the ability to code. This idea is achievable, at least in theory, through the randomization of software patterns in the form of a grammar and the use of the evolving grammar for assisting in the specification of symmetric rules. A two-level expert compiler is depicted in Figure 4:
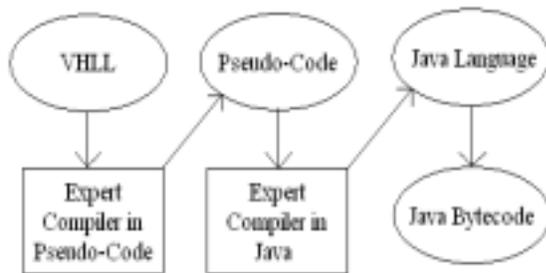

Figure 4. A two-level expert compiler.

Here, a Very High-Level Language (VHLL) is transformed by an expert compiler, written in pseudo-code, to pseudo-code. The pseudo-code is then transformed by an expert compiler, written in Java, to the Java language, which is then converted to bytecode and executed. There are two levels of bootstrapping here because there are two levels of expert compilers. The use of multiple levels of expert compilers is a randomization. It is similar to the way in which we view fuzzy logic as a randomization of crispy rules [5]. That is, it represents a segmentation of knowledge bases from the standpoint of functionality, removing symmetry, and thus randomizing. The same pseudo-code compiler can be reused in the expert compilation of many different VHLL's. The Layout software CASE tool by Objects Inc. (Danvers, MA) was built through the realization of such a scheme.

Expert compilers are uniquely suited to the realization of higher-level languages. This stems in part because they are readily maintained and updated -- making for an extensible compiler, which in turn makes for an extensible language. Expert compilers can also be networked for the translation of heterogeneous languages (i.e., languages whose semantics are difficult to define). Such languages are extremely high level and

suffer from ambiguity in their use. The network serves to bring different domain-specific expert compilers to bear on their translation. Java is well suited to many web-based applications because it is threaded.

If one should choose to question the role of artificial intelligence in software engineering, just remember that the most powerful CASE tool to date is a human being. The one thing that differentiates humans from any conventional compiler in the present context is that humans are intelligent -- they can transform solutions from the abstract to the concrete (i.e., from the concept to the code). Indeed, the brain is thought by Minsky to consist of a heterogeneous network of communicating expert systems [6]. Human brain also has its own extremely high-level effective representation (or equivalently effective pseudo-code).

8. THE TARGET CODE

Figure 4 shows, among other things, that expert compilers can compile psuedo-code to any number of target languages (e.g., Java) simply by substituting appropriate knowledge bases for the last stage of compilation. This means that expert compilers can be quite flexible in the choice of target language. First, a target language must be selected. This is code for which there already exists a compiler or interpreter. Having used JBuilder 2, we choose Java as our target language. Furthermore, we restrict ourselves to that subset of Java defined by the following pair of symmetric programs -- selection sort (Figure 5) and insertion sort (Figure 6).

```
public class Selection_Sort_Test
{
public static void main (String[] args)
{
int[] numbers = {3, 9, 6, 1, 2};
Selection_Sort.sort (numbers);
for (int index = 0;
     index < numbers.length; index++)
System.out.println (numbers [index]);
}
}
class Selection_Sort
{
public static void sort (int[] numbers)
{
int min, temp;
for (int index = 0;
     index < numbers.length-1; index++)
{
min = index;
for (int scan = index+1;
     scan < numbers.length; scan++)
if (numbers[scan] < numbers[min])
min = scan;
// swap the values
temp = numbers[min];
numbers[min] = numbers[index];
numbers[index] = temp;
}
}
}
```

Figure 5. The target code for selection sort.

The task is to define another high-level code, complete with an expert compiler, to make it easier (i.e., with respect to software

quality and quantity) to write the above programs. That is, one sequence of high-level instructions and one sequence of expert rules should be capable of realizing either program (and others) depending on the program written. Conflict resolution is to be in order from most specific to least specific rule. More specific rules will have longer antecedent schemas to be matched.

```
public class Insertion_Sort_Test
{
public static void main (String[] args)
{
int[] numbers = {3, 6, 1, 9, 2};
Insertion_Sort.sort (numbers);
for (int index = 0;
     index < numbers.length; index++)
System.out.println (numbers[index]);
}
}
class Insertion_Sort
{
public static void sort (int[] numbers)
{
for (int index = 1;
     index < numbers.length; index++)
{
int key = numbers[index];
int position = index;
// shift larger values to the right
while (position > 0 &&
       numbers[position-1] > key)
{
numbers[position] = numbers[position-1];
position--;
}
numbers[position] = key;
}
}
}
```

Figure 6. The target code for insertion sort.

9. CHURCH'S THESIS

A scientist named Alfonso Church purported his thesis (i.e., Church's Thesis) around the turn of the twentieth century, which states that if a concept can be clearly and unambiguously represented as an algorithm, then it can indeed be coded [3]. One need not code the algorithm to prove that it can be coded -- only if one intends to use it. We appeal to Church's Thesis to defray the cost of model-building for the purpose of convincing the reader of the feasibility and efficacy of our new method. For example, expert compilers may have upwards of 500,000 lines of code. Hence, the cost of construction needs to be justified.

10. SYMMETRY

The final issue to be addressed in the design of our new method, defined through the use of two-level expert compilers, is the extraction of symmetry, or the reduction to a random representation. In order to accomplish this, we will need to find symmetry in the above examples and design our pseudo-code to capture that symmetry (i.e., orthogonal programming, which is easier to read and maintain). An example should serve to clarify this. Notice that there are two types of loop constructs used -- *for* and *while*. They are symmetric in meaning, but not in structure. Can we develop a pseudo-coded structure to capture each? We will actually attempt to do this. Here is an example of pseudo-code. First, the schema:

Repeat forever
(**Initialize** *variable* **to** *value*;)
// Parenthesized expressions are optional.
Break if *variable* {<, >, =, <>} *value*;
(**Body;**)
{**Increment**, **Decrement**} *variable*;
(**Body;**)
End
Thus, the pseudo code:

Repeat forever
Initialize *index* = 0;
// There would be a rule that sees no decimal and
// thus knows that it is an int.
Break if *index* >= *numbers.length*;
// The expert compiler will transform the relation.
Increment *index*;
System.out.println (numbers [index]);
End

would be compiled into:

```
for (int index = 0; index < numbers.length; index++)
System.out.println (numbers [index]);
```

Now, compare this pseudo code with similar pseudo-code, which compiles into a while loop:

Repeat forever
Break if *position <= 0 |*
       numbers[position-1] <= key;
// The expert compiler can translate the logic.
numbers[position] = numbers[position-1];
Decrement *position*;
End

which would be compiled into:

```
while (position > 0 && numbers[position-1] > key)
{
numbers[position] = numbers[position-1];
position--;
}
```

It is clear that a human can do the compilations. Next, one should observe how an expert compiler could be made to do the same. Tables 2 and 3 illustrate two sample schema-based rules that determine if a *for* or a *while* loop is called for. Note that they are only based on the previous examples. We try to make the rules more general if not too difficult. Also, it is not shown here that rules will apply in sequence. Different schemes, including a tagging scheme, can be used to process the partially transformed code. The partial transformation serves as a context to enable subsequent transformations. By tagging the transformed code one insures that it will not itself be subjected to subsequent transformations. The tags are all removed upon completion of the transformation.

Next, observe several items. First, the capability to pattern match the rule antecedent schemas is facilitated through the use of a very high-level language. That is why it helps to bootstrap the expert compiler in at least two levels. The capability to compile pseudo-code facilitates writing the rules. Here, the pseudo-code should be designed to facilitate high-level textual comparisons and substitutions. Conversely, observe how difficult it would be to test if "variable1 = value1" in assembler. One also needs a symbol table to track variable definitions.

Table 2. A sample for-loop rule.

PSEUDO-CODE	EFFECTIVE CODE
Repeat forever **Initialize** *variable1=value1*; **Break if** *variable2 R1 value2*; **Increment** *variable3*; **Body**; **End**	**for**(*type variable1=value1*; *variable2 !R1 value2*; *variable3++*) Body;

Table 3. A sample while-loop rule.

PSEUDO-CODE	EFFECTIVE CODE
Repeat forever **Break if** *variable1 R1 value1	variable2 R2 value2;* **Body**; **Decrement** *variable3*; **End**	**while** (*variable1 !R1 value1 && variable2 !R2 value2*) **Body**; *variable3--*;

Having the proper pseudo-code to work with would facilitate this. Once facilitated, one could more easily write rules, leading to higher-level languages, which facilitate writing rules, and so on. Remember that a small gain in the level of the language will result in an enormous improvement in programmer productivity -- not only in time saved writing code, but in time saved in debugging it and reusing it! Think of an atomic pile where if the language reaches a certain very-high level (e.g., English) and one has a knowledge explosion.

Symmetry is a powerful ally in reducing the programmers' overhead. Its use enables software reuse, which in turn serves to greatly reduce the incidence of programmer error. Symmetry can be captured through the reuse of syntax (e.g., converting an increasing sort to a decreasing sort), or through the reuse of semantics (e.g., an expert compiler). Higher-level languages make it easier to program in ever-greater complexity. In the information-theoretic sense, *complexity* is the distillate of what remains in a program, once its symmetric components are extracted. For example, a program is not more complex simply because one chooses not to use do loops in its realization.

Patterns exist at many levels. Indeed, the patterns of greatest significance are domain specific. For example, the functional relationship between a gas pedal and a brake pedal is in some sense symmetric. However, the relative location of the two is domain specific. Also, the relationship is domain-specific and differs from a gasoline-powered engine to an electric engine. In particular, the latter allows for practical regenerative braking.

Taking the components in all of the preceding sections together, we are able to implement a second-generation expert compiler, which again is the key to higher-level programming, broadly defined. Thus, higher-level programming moves us closer to the brains language (and possible representation) using metaphors. Programming thus becomes more of a creative activity. This is in sharp contract with detail-oriented and error-prone machine language programming. In part, this explains the rising success of the Unified Modeling Language (UML), which is a semantic net-based software modeling and design tool [7].

11. CONCLUSION

Randomization can be viewed as a self-referential process. This means that Gödel's Incompleteness Theorem applies. This in turn implies that domain-specific languages can be extended with a rapidity and/or scope that cannot be formally predicted. In a practical sense the implication is that we need to build networks of expert compilers -- networks capable of self-referential transformation. As these networks scale up (e.g., through the construction of textual transformation languages, which in turn support the construction of higher-level rule bases), increases in utility should occur, which cannot be foreseen or subsequently given formal explanation.

Programming languages need to be designed for metaphorical expression to a much greater degree than is presently the case. Such a representational formalism facilitates reuse and reduces the incidence of error. It is also more readable than otherwise. The recent and growing success of the UML serves as a testimonial to this advocacy. This paper has presented a prescription for randomizing languages through the construction of effective pseudo-code. It was shown that such an approach has strong information-theoretic underpinnings in addition to being immanently practical.

This paper has demonstrated that issues of representation and language play a key role in the capture of intelligent processes. In other words, the randomization of information must recursively reference the representation of that information itself. This implies that knowledge engineering and software engineering are symbiotic. We need to develop expert compilers for ever-higher-levels of representation if we are to ever program intelligent machines. For example, one type of expert compiler could mine rules for patterns and if found, convert the applicable rules into a fuzzy representation together with a fuzzy inference mechanism. There is much worthy of exploration here. The time is rapidly approaching when we must view computers as capable of performing supra-formalistic computations. This is in keeping with Einstein's dictum, "Everything should be as simple as possible, but no simpler." After all, this sentiment captures the very meaning of randomization!

12. ACKNOWLEDGMENTS

13. REFERENCES

[1] G.J. Chaitin, "Randomness and Mathematical Proof," *Sci. Amer.*, vol. 232, no. 5, 1975, pp. 47-52.
[2] S.H. Rubin, "New Knowledge for Old Using the Crystal Learning Lamp," *Proc. 1993 IEEE Int. Conf. Syst., Man, Cybern.*, 1993, pp. 119-124.
[3] A.J. Kfoury, R.N. Moll, and M.A. Arbib, *A Programming Approach to Computability*, New York, NY: Springer-Verlag Inc., 1982.
[4] R.L. Glass, "The Realities of Software Technology Payoffs," *Comm. ACM*, vol. 42, 1999, pp. 74-79.
[5] L.A. Zadeh, "Fuzzy Logic, Neural Networks, and Soft Computing," *Comm. ACM*, vol. 37, 1994, pp. 77-84.
[6] M. Minsky, *The Society of Mind*, New York, NY: Simon and Schuster, Inc., 1987.
[7] H.E. Eriksson and M. Penker, *UML Toolkit*, New York, NY: John Wiley & Sons, Inc., 1998.