ON THE ROLE OF INFORMED SEARCH IN VERISTIC COMPUTING

STUART H. RUBIN, ROBERT J. RUSH JR., and JAMES BOERKE / LJILJANA TRAJKOVIC
SPAWARS SYSTEMS CENTER
53560 Hull Street
San Diego CA 92152-5001
{srubin, rushr, jboerke}@spawar.navy.mil

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

Abstract

Veristic computing is defined to mean computing with words. It necessarily entails the use of informed search in the solution of qualitatively constrained equations. Its use does not preclude computing with numbers. Versitic computing allows for the specification of higher-level programming languages, which can evolve domain-specific knowledge bases. The knowledge is evolved on a high-end computer for subsequent porting to a PC. The application of that knowledge to the translation of a higher-level program is termed, expert compilation. This paper serves to make clear the ubiquitous role assumed by randomization in all aspects of software engineering – from programming language design to program design to program testing to knowledge transference.

Keywords
Data Mining, Expert Compilers, Knowledge Discovery, Randomization, Transference.

1 Introduction to Randomization

The fundamental need for domain-specific knowledge is in keeping with Rubin's proof of the Unsolvability of the Randomization Problem [1]. A second application of the Unsolvability of the Randomization Problem reduces the capability for forming domain-specific generalizations to the capability for forming domain-specific representations. But then, this capability is dependent on previous structure. All that matters is to minimize the entropy of the information-theoretic model using randomization [2]. Of course, there is also a practical dimension that involves issues pertaining to realization on a supercomputer. However, even here the Unsolvability of the Randomization Problem implies the necessary search for ever-better domain-specific hardware.

2 Knowledge Acquisition

The acknowledged key to breakthroughs in the creation of intelligent software is cracking the knowledge acquisition bottleneck [3]. This project benefits from the use of high-speed computers to study deductive and inductive processes. Formerly, such studies relied upon the use of heuristics, or rules of thumb, to delimit the search space. While such "best-first" methods are of practical utility, they limit machine learning to that of improving the heuristic base because this aspect must be separated from the organizational representation of the knowledge, which is to be searched.

The chosen representational formalism must also include all manner of features. Data mining is critically dependent on the quality of the feature space. To better see this, consider mining a chess game. If all that is recorded about a chess game are the pieces and their sequence of positions, then this highly specialized knowledge would prove to be of little utility. What is needed is a generalization of this declarative knowledge into procedural knowledge. Here, such features as nearness of the king to the center of the board, whether or not the rook can be castled, or even if a particular depth-first search procedure should be applied are clearly representative of procedural knowledge.

3 Software Debugging

In addition to software reuse and retrieval [4], software debugging also has strong ties to randomization. For example, consider the debugging of any sort program. It follows from the Unsolvability of the Randomization Problem [1] that any program sufficiently complex to be capable of self-reference can never be assured to be totally bug-free. Rather, the best that can be done is to test it to satisfaction. There are two separate, but related issues here. First, how does one develop a repository of test cases for maximal coverage and second, what principles does one employ to construct software so that it is relatively bug free?

Test cases should cover as many execution paths as practical; yet, be minimal in number so as to render the testing process practical. For example, when testing a sort program, one does not want test cases such as: (1) (2 1) (3 2 1) (4 3 2 1) … because the test vectors are mutually symmetric. Rather, one needs to develop an approximation of a basis (e.g., akin to the basis of a matrix). For example: () (1) (2 1) (1 3) (3 1 2) … Furthermore, Rubin [5, 6] has developed a fuzzy programming language that accepts a generalized program description and uses an orthogonal sequence of test cases to instantiate a working program, where possible. What is interesting here is that the induced program is no better nor worse than allowed by the supplied test sequence. This is testing in reverse and serves to underscore the critical role played by test case selection in determining the overall quality of the resulting software produced. One could argue that the aforementioned process works well for functional programming, but what about say for visual programming? How does one map test cases to form? The answer is that visual programming (e.g., creating GUIs) requires testing as does any other program; although, here an expert or expert system is required to provide the feedback that determines the success or failure of each test. Notice that the process of randomization is pervasive: It exists in the generation of a minimal set of test cases; it exists in the specification of a fuzzy program (i.e., a minimal representation of a program space whose instances approximate useful working programs); it exists in the reuse of expert systems for GUI design; and, it surely exists in all related software processes too detailed to be described here. Fig. 1 presents a depiction of a fuzzy program written in an extended version of the LISP language [7, 8].

To construct software that is relatively bug-free, one needs to maximize the testability of every included software unit. This is necessary because, in keeping with the above arguments, software can only be certified in proportion to its having been tested using mutually random test cases that attempt to approximate the environment in which it will be used. For example, consider the development of the Fortran subroutine. Instead of transferring to different sections of possibly erroneous code, all these would-be calls are instead transferred to a parametized randomization of their semantics. The greater number of calls here serves to increase the number of tests made of the subroutine with the result that the overall quality of the program is improved through the use of subroutines. The same argument extends to the use of objects and components. Thus, we see that what may be termed, "randomized programming practices" results in improved code quality. A new view of software reuse will evolve in the form of a taxonomy:

- Level 5: Reuse requires a dynamic domain-specific representation of knowledge.
- Level 4: Reuse requires the application of knowledge bases (e.g., expert compilers).
- Level 3: Reuse occurs in the form of objects and components.
- Level 2: Reuse occurs at the code level and allows for parametization.
- Level 1: Reuse occurs at the level of immutable code.

4 On Feature Acquisition

The discovery of features represents a search process involving test and evaluation. Indeed, the search space can rapidly become intractable, which serves to underscore the need for a supercomputer. The term, *veristic computing* was introduced by Zadeh [9]. The term refers to computing with words in much the same manner as we presently compute with numbers. In particular, veristic computing provides for a qualitatively fuzzy computational capability [1].

The feature space here consists of an object space of LISP functions together with an operant search methodology. Notice the key role assumed by the representation in the synthesis of the LISP functions. For example, the conditions (Null S) and Nil are instances of the empty list, S. These instances may also be fuzzy function calls as in the definition of MYSORT below.

There are three ways to speedup veristic computations, which are not mutually exclusive. First and foremost, one can apply massively parallel processing. Second, one can break an X * X problem into an Y + Y problem (i.e., apply the triangle inequality). Finally, one can find heuristics, where possible, to partially order the search space. In particular, these heuristics can serve to realize the triangle inequality through the application of domain knowledge.

Higher-level functions (e.g., MYSORT as compared to NULL) require the application of more processing power to test. Many such tests may proceed in parallel. The need for a supercomputer at once follows. Indeed, this need can evolve to that for distributed clusters of supercomputers running in MIMD mode.

These experiments promise to deliver veristic programming languages, which require fast computing facilities for compilation in the absence of domain-specific knowledge. Furthermore, this domain-specific knowledge can be represented in and generated through the use of veristic computation. This means that while a high-end computer is needed to create the language and attendant knowledge bases in the first place, the results can be ported to an ordinary high-end PC.

5 Expert Compilers

Expert compilers were first defined by Geoffrey Hindin [10]. Now, it is well-known that programmer productivity bears proportion to the level of the programming language in which a program may be expressed. Thus, we saw a six-fold jump in productivity in the 70s when assembly-language programming gave way to Fortran (i.e., 3d generation language) programming. Smaller gains are reported for object and component-based programming due to complexity issues. Simply put, the new problem in increasing programmer productivity is that 3d generation (universal) languages do not in effect substitute for user domain knowledge. One still needs to program every domain detail to get the program to work as desired. Expert compilers relax this stipulation by providing for rule-based knowledge insertion. For example, consider the following 3d generation vs. expert-compiled program.

In Fig. 2, the expert rules act on the expert specification to produce the third generation code. The rules have been oversimplified for purposes of illustration. Notice that the expert specification is at a higher level and is thus easier to debug and be more productive in through the use of the expert compiler. In fact, the separation of the domain knowledge from the program knowledge may be viewed as an extension of the traditional separation of the knowledge base from the inference engine in an expert system. Notice the parallels with the introduction of the Fortran subroutine described above. That is, an expert compiler is yet another form of randomization!

In traditional software engineering practice, an extensible language (e.g., LISP) serves many of the same requirements for randomization. However, the difference between an extensible language and expert compilation is that the knowledge is embodied in the object in an extensible language, which tends to delimit its reusability. Such is not the case with an expert compiler. Now, as the size of the object gets smaller, its reusability increases until in the limit it offers the same advantages offered by an equivalent expert compiler, or so it would seem. What's missing from this argument is any notion of execution efficiency. That is, as the objects get smaller, the programs that they detail tend to loose efficiency.

An expert compiler can optimize code and thereby offers the better of two approaches. On the one hand, it frees the user to express a program in relatively simple and cognitively straightforward terms. On the other hand, the resulting sub-optimal program can then be automatically transformed into a more efficient form. For example, the typical computer scientist will find it far easier to write an $O(n^2)$ Bubble or Insertion sort than he/she will to write an $O(n \log n)$ Quicksort. They may have the same test suite and the expert compiler can incrementally transform one into the other given an economy of scale. Note that the details pertaining how to accomplish this could occupy an entire volume. Thus, we necessarily concern ourselves with the concept for now.

A simple expert compiler can become more complex by way of fusing a network of domain-specific knowledge bases. Notice that as more and more domain-knowledge can be brought to bear on the compilation, the level of the effectively transformed language can increase. All this may be succinctly stated to be a consequence of randomization. Moreover, the language in which the rule predicates are represented in each rule base can also be subject to expert compilation. We call this a knowledge-based bootstrap. Observe that it too is recognizable as being a higher-level randomization. Informally, we call this a capability for the dynamic domain-specific representation of knowledge.

Any network of expert compilers can grow to be exceedingly complex. After all, given that there will be errors, how does one trace a resultant error back to its source? This problem can be accentuated through the use of asynchronous MIMD architectures. The solution is to keep the human in the loop. Moreover, it is key to note that the more reusable the representation of knowledge (and software), the greater will be the propagation of repairs. This follows because highly reusable components are invoked by many other components. Correcting one error then serves to correct many errors. Think of this as being the inverse of testing, where the higher the degree of reusability the more likely the program will be valid.

What emerges from the previous discussion is the notion that higher-level software is going to be more structured in all its salient aspects. Instead of being hand-crafted, it will be assembled – not by humans, but by machines that were programmed for its assembly. The central thesis of this section is that higher structures necessarily go beyond the lax bounds imposed by domain-independent representation. While such representations account for objects and components, they place the entire burden of assembly upon the user. Here, the knowledge source is a sole source – namely, the human.

```
((DEFUN MYSORT (S)
   (COND ((NULL S) NIL)
     (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))
? io
((((1 3 2)) (1 2 3)) (((3 2 1)) (1 2 3)) (((1 2 3)) (1 2 3)))
? (pprint (setq frepos '((CRISPY'
          (DEFUN MYSORT (S)
            (COND
             (FUZZY
              ((NULL S) NIL)
              ((ATOM (FUZZY S ((FUZZY CAR CDR) S))) NIL))
             (T (CONS (MYMIN S (CAR S))
             (MYSORT (REMOVE (MYMIN S (CAR S)) S))))))))))


((CRISPY '(DEFUN MYSORT (S)
    (COND (FUZZY ((NULL S) NIL) ((ATOM (FUZZY S ((FUZZY CAR CDR) S))) NIL))
      (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S))))))))

; Note that (ATOM S) was automatically programmed using the large fuzzy function space.

? (pprint (auto frepos io))

((DEFUN MYSORT (S)
   (COND ((ATOM S) NIL)
     (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))

; Note that each run may create syntactically different, but semantically equivalent
; functions:

? (pprint (auto frepos io))

((DEFUN MYSORT (S)
   (COND ((NULL S) NIL)
     (T (CONS (MYMIN S (CAR S)) (MYSORT (REMOVE (MYMIN S (CAR S)) S)))))))
```

Fig. 1 A Sample Veristic Computation for Supercomputing

To climb above the third-generation plateau, one needs to capture domain-specific knowledge for reuse. Such is accomplished by an expert compiler, which effects in theoretical terms, semantic randomization. Fourth-generation languages may appear to get around this problem without resorting to expert compilation, but this is a convenient illusion. The reason for the illusion is that such languages are not universal. In a practical sense, this means that they are not flexible or conveniently extensible. Also, fifth generation languages (e.g., LISP, PROLOG, *et al.*) provide tools for the construction of expert compilers, but are not expert compiled per se.

6 Veristic Predicate Calculus

Fuzzy logic is a logic of imprecision. Predicate calculus is a formalism that enables qualitative search. If both methodologies are properly combined, then the result is a knowledge-based mechanism for commonsense reasoning that can learn by way of an object-oriented approach.

The situations, or states, and the goals of many types of problems can be described by predicate calculus wffs. In Fig. 3, for example, a situation is depicted in which there are three blocks, *A*, *B*, and *C*, on a table [11].

Third Generation:

```
Repeat
  Read x, y;
Until
  y <> 0;
ratio = x/y;
If ratio > 0 then
  Print ratio
Else
  Print –ratio;
```

Expert Specification:

```
Read x, y;
ratio = x/y;
Print ratio;
```

Expert Rules:

```
If "/" Then denom <> 0
If "/" Then ratio +
```

Fig. 2 An Expert Compiler Running on a PC

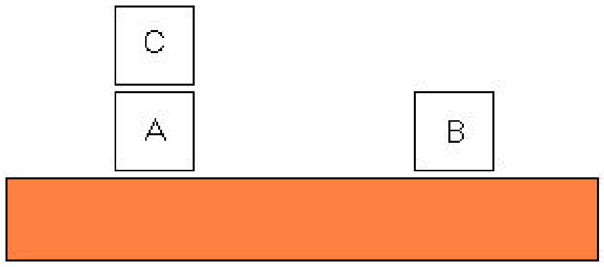

Fig. 3 A Situation with Three Blocks on a Table

The following formulas formalize the situation.

ON (C, A)
ONTABLE (A)
ONTABLE (B)
CLEAR (C)
CLEAR (B)
$(\forall x)[CLEAR(x) \Rightarrow \neg (\exists y)ON(y,x)]$

The formula *CLEAR (B)* means that block *B* has a clear top. That is, no other block is on top of it. The *ON* predicate is used to describe which blocks are immediately (i.e., non-transitivity) on top of other blocks. The formula *ONTABLE (B)* is intended to mean that *B* is somewhere on the table. The last formula means that if a block has a clear top, then there does not exist a block on top of it.

Formulas, like the relatively simple ones given above, can be used as a global database in an expert system. The way in which we can use these formulas depends upon the problem and its representation. For example, suppose that we wish to prove that there is nothing on block *C* in Fig. 3. We can prove this fact by showing that the formula $\neg (\exists y)ON(y,C)$ logically follows from the state description for Fig. 3. Equivalently, we could show that this formula is a theorem derived from the state description by the application of sound rules of inference.

Expert or production systems can be used to show that a given formula, called the *goal wff*, is a theorem derivable from a set of formulas (the state description). Such theorem-proving systems operate using deduction. We will see that generalizing such systems for the use of induction will allow for heuristic reasoning capabilities.

In forward-chaining expert systems, the state description serves as the start state. Here, production rules are applied until a state description is produced that either includes the goal formula or unifies with it in some appropriate fashion. In backward-chaining expert systems, the goal description serves as the start state. Here, production rules are applied until a subgoal is produced that unifies with formulas in the state description. Combined forward-backward chaining is also possible [11].

Let us use the following two rules and forward chain.

R1: [ON (y, x) $\wedge$ CLEAR (y)] $\Rightarrow$ ONTABLE (y) $\wedge$ CLEAR (x)]
R2: [ONTABLE (x) $\wedge$ CLEAR (x)] $\wedge$ CLEAR (y) $\Rightarrow$ ON (x, y)]

R1 means that if one block is on top of another and the higher block has nothing on top of it, then we can place the higher block on the table, which clears the top of the block immediately beneath it.

R2 means that if a block is on the table and has a clear top, then that block can be placed on top of any other block having a clear top.

Observe that one can apply *R1* to move block *C* to the tabletop, followed by an application of *R2* to move block *C* on top of block *B*. Similarly, one can apply these rules in different sequences and with different arguments to achieve any desired configuration of blocks. The thing to note is that search in this deductive system needs to be controlled. Just being able to solve a problem is not enough. One must also be able to find a (near-optimal) solution, where one exists, in tractable time.

It follows that two types of rules are needed to augment the rule set. The two types are not mutually exclusive. First, are search control rules and second are induced rules (i.e., generalizations and analogs).

Next, let us use the predicate calculus to negate the expression,
$(\forall x)[CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x)]$:

$$\neg(\forall x)[CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x)] =$$

$$(\exists x)[\neg(CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x))] =$$

$$(\exists x)[\neg(\neg CLEAR(x) \vee \neg(\exists y)ON(y,x))] =$$

$$(\exists x)[CLEAR(x) \wedge (\exists y)ON(y,x))] =$$

$$(\exists x,y)[CLEAR(x) \wedge ON(y,x))]$$

The deduced expression states that there exists a pair of blocks such that the bottom one has a clear top *and* it also has a block on top of it. Clearly, this is a contradiction having the value False, which is the negation of the truth-value for the starting formula, $(\forall x)[CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x)]$.

Suppose now that we would like to amplify the knowledge base beyond the limits provided by classical deductive inference. The purported solution is fractal-like in nature. That is, just as variables *x, y* may be instantiated, so too may the object predicates that reference them be instantiated. The following formulas formalize the situation.

OFF (A, B)
OFF (C, B)
LOADED (A)
OFFTABLE (C)

The *OFF* predicate is true just in case neither *ON (x, y)*, nor *ON (y, x)*. The *LOADED* predicate means that the *CLEAR* predicate is false. The *OFFTABLE* predicate means that the *ONTABLE* predicate is false.

Next, let us negate the expression: $(\forall x)[CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x)]$ again – only this time through the use of predicate negation:

$$\neg(\forall x)[CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x)] =$$

$$(\forall x)[\neg CLEAR(x) \Rightarrow \neg(\exists y)\neg ON(y,x)] =$$

$$(\forall x)[LOADED(x) \Rightarrow \neg(\exists y)OFF(y,x)] =$$

$$(\forall x \exists y)[LOADED(x) \Rightarrow \neg\, OFF(y,x)]$$

The induced expression correctly informs us that for every loaded block there is at least one block that is not off it. Observe that predicate negation is not the same as ordinary negation. It will only hold if all of the relevant formulas and their negations are symmetric. The new rule is acquired and serves to amplify the existing knowledge base.

Next, consider the case where the relevant formulas are mostly symmetric, but only in an implicit sense. That is, for example the rules pertaining to automotive repair are similar to those pertaining to truck repair. They are mostly symmetric, but of course there are differences. Here, we will consider the similar problem of stacking glasses. Only empty glasses may be stacked and the task is to induce rules analogous to *R1* and *R2*. First, the following formulas detail the situation.

EMPTY (G)
STACK (G1, G2)
ONTABLE (G)
$(\forall x,y)[EMPTY(x) \Rightarrow STACK(y,x)]$

The formula *EMPTY (G)* means that a glass is empty and thus can be placed at the bottom of a stack. The *STACK* predicate means that you can stack glass *G1* on top of another, *G2*. The *ONTABLE* predicate is as before. The last formula means that if a glass is empty, then you can always stack a glass on top of it.

Next, an attempt is made to establish an isomorphism. The actual details of doing so in a large-scale system requires some form of learning to correct errors (i.e., random points) as well as prevent them from recurring [1]. Consider:

$$(\forall x)[CLEAR(x) \Rightarrow \neg(\exists y)ON(y,x)]$$

$$(\forall x)[EMPTY(x) \Rightarrow (\forall y)STACK(y,x)]$$

Here, *CLEAR* and *EMPTY* are in bijective correspondence. Similarly, *ON* and *STACK* are in bijective correspondence because the negation of the existential qualifier is the universal qualifier. Applying these transformations to *R1* and *R2* yields:

R3: *[STACK (y, x)* $\wedge$ *EMPTY (y)]* $\Rightarrow$ *ONTABLE (y)* $\wedge$ *EMPTY (x)]*
R4: *[ONTABLE (x)* $\wedge$ *EMPTY (x)]* $\wedge$ *EMPTY (y)* $\Rightarrow$ *STACK (x, y)]*

R3 means that if two glasses are stacked and the top glass is empty, then we can place the top glass on the table, which implies that the glass beneath it is empty.

R4 means that if a glass is on the table and it is empty, then that glass can be placed on top of any other empty glass.

As noted above, cross-domain transformations need not preserve validity. A special case of validity is the preservation of generality. Here, we know from the physics of the glass domain that there is no need to check the contents of the topmost glass. That is, while we may not be able to move a block if there exists another block on top of it, we can move a filled glass almost as easily as an empty one. A single meta-rule will effect the desired transformation to *R3* and *R4*.

R5: $META\ (x, y): [STACK\ (y, x) \wedge EMPTY\ (y)] \Rightarrow STACK\ (y, x)]$

The results of applying *R5* to *R3* and *R4* follow.

R6: $[STACK\ (y, x)] \Rightarrow ONTABLE\ (y) \wedge EMPTY\ (x)]$
R7: $[ONTABLE\ (x)] \wedge EMPTY\ (y) \Rightarrow STACK\ (x, y)]$

R6 means that if two glasses are stacked, then we can place the top glass on the table, which implies that the glass beneath it is empty.

R7 means that if a glass is on the table, then it can be placed on top of an empty glass.

Notice that a glass can be filled or empty, while a block (i.e., as presented here) cannot. This is said to be a random point of variation between the two domains (i.e., Blocks World and Glasses World). The method for achieving veristic knowledge acquisition, which is beyond the scope of this paper, is to effectively accomplish the following five steps.

1. Identify symmetric (random) domains.
2. Create a knowledge base for at least one domain.
3. Create a base of transformation rules, mapping one domain to the other.
4. Create a base of meta-rules for resolving random differences between domains.
5. Develop a set of tools, expert systems, domain experts, etc. for quality assurance.

7 Conclusions

Randomization theory holds that the human should supply novel knowledge exactly once (i.e., random input) and the machine extend that knowledge by way of capitalizing on domain symmetries (i.e., expert compilation). This means that in the future, programming will become more creative and less detailed and thus the cost per line of code will rapidly decrease.

One of the basic tenets of randomization theory pertains to porting knowledge from one base to another. We have shown that not only is this possible, but that knowledge transference is itself a byproduct of randomization. We have also seen that the knowledge needed to effect transference is more complex than is the image of transformation and outlined a five-step program for achieving veristic knowledge acquisition as a consequence.

We have learned from various sources that the White House has chosen LISP for programming some server-based applications. It is claimed that they experienced a 500 percent improvement in productivity as a result of the extensible features imbued in this language. Again, this success story does not begin to touch on the possibilities offered by networked expert compilers of scale. According to Bob Manning [7],

> Processing knowledge is abstract and dynamic. As future knowledge management applications attempt to mimic the human decision-making process, a language is needed which can provide developers with the tools to achieve these goals. LISP enables programmers to provide a level of intelligence to knowledge management applications, thus enabling ongoing learning and adaptation similar to the actual thought patterns of the human mind.

In conclusion, the solution to the software bottleneck will be cracking the knowledge acquisition bottleneck in expert systems (compilers). We need to study knowledge representation and learning, rule-based compilers, and associated architectures. For example, it is possible that knowledge-based segments can be retrieved on demand over the Internet, which can provide the necessary economy of scale required for the successful implementation of networked expert compilers [12]. Finally, according to Zadeh [13], improved methods for soft computing will have impact on qualitative and approximate reasoning, computing with words, manipulation of perceptions, intelligent control, intelligent information systems, expert systems, chaotic systems, image analysis and image understanding, speech and natural language processing, planning, learning, search, data mining, and decision analysis.

Acknowledgments

This paper includes the work of U.S. Government employees performed in the course of their employment and is not subject to copyright. It is approved for public release with an unlimited distribution. This work was supported in part by the Office of Naval Research, NSERC, and the BC Advanced Systems Institute Fellowship.

References

[1] S.H. Rubin, "Computing with Words," *IEEE Trans. Syst. Man, Cybern.*, vol. 29, no. 4, pp. 518-524, 1999.
[2] S.H. Rubin, "On Knowledge Amplification by Structured Expert Randomization (Kaser)," *SSC San Diego Biennial Review*, to appear in 2001.
[3] E.A. Feigenbaum and P. McCorduck, *The Fifth Generation*. Reading, MA: Addison-Wesley Publishing Co., 1983.
[4] S.H. Rubin and L. Trajkovic, "On the Role of Randomization in Software Engineering," *The Intern. Conf. on Comp. & Indus. Engr. (28th ICC&IE)*, Cocoa Beach, FL, to appear in 2001.
[5] S.H. Rubin, "A Fuzzy Approach Towards Inferential Data Mining," *Computers and Industrial Engineering*, vol. 35, nos. 1-2, pp. 267-270, 1998.
[6] S.H. Rubin, "A Heuristic Logic for Randomization in Fuzzy Mining," *J. Control and Intell. Systems*, vol. 27, no. 1, pp. 26-39, 1999.
[7] B. Manning, "Smarter Knowledge Management Applications: LISP," *PC AI*, vol. 14, no. 4, pp. 28-31, 2000.
[8] E. Gat, "LISP as an Alternative to Java," *Intelligence*, pp. 21-24, Winter 2000.
[9] L.A. Zadeh, "From Computing with Numbers to Computing with Words – From Manipulation of Measurements to Manipulation of Perceptions," *IEEE Trans. Ckt. and Systems*, vol. 45, no. 1, pp. 105-119, 1999.
[10] J. Hindin, "Intelligent Tools Automate High-Level Language Programming," *Computer Design*, vol. 25, pp. 45-56, 1986.
[11] N.J. Nilsson, *Principles of Artificial Intelligence*. Mountain View, CA: Morgan Kaufmann Publishers Inc., 1980.
[12] I. Ben-Shaul and G. Kaiser, "Coordinating Distributed Components over the Internet," *IEEE Internet Computing*, vol. 2, pp. 83-86, 1998.
[13] L.A. Zadeh, *BISC* Electronic Communique, Jan. 2001.