Intelligent Guidance of an Unmanned Helicopter

Stuart H. Rubin[1] and Gordon Lee[2]

[1]SPAWAR – SSC Pacific
53560 Hull Street
San Diego, CA 92152-5001

[2]Dept. of Electrical and Computer Engineering
San Diego State University
San Diego, CA 92182

Executive Summary

The objective of this project is to define a science which allows for the evolution of complex software systems that can fully utilize massively parallel computers of ever-greater capability (initially dual quad cores running fewer higher-level sensors for tractability) and employ this design in human-robot interaction with a final goal of full autonomy of the robot (in this case, an unmanned helicopter). Briefly, the greater the number of processors, the less the chance of error and the better the analogical match that can be found in the processing time allowed. Clearly, any practical science of design should allow the human in the loop to do what he/she does best while at the same time utilize the robot to do what it does best. Such an eminently practical symbiosis is based on the science of randomization as defined by Chaitin, Kolmogorov, Rubin, Solomonoff, and Uspenskii [1], [2], [3], [4], [5]. Here, repetitive or symmetric actions such as testing evolutionary alternatives are carried out by a fast computer, while novel or ~~"r~~andom" actions such as that of providing a model definition are realized by the human.

Autonomous behavior may be determined by the interaction of simple heuristics, rather than through the design of complex software that is difficult to maintain. The emergent behavior of simple heuristics is not only universal, but potentially at least as complex as that of any other design for the realization of software control systems.

While any autonomous vehicle equipped with suitable sensors can be autonomously guided by an appropriate set of heuristics, the acquisition of such a base has proven to be elusive to date. This document provides a forum for some preliminary results in the area of knowledge amplification as well as a setting for projected future research in the area of intelligence.

The problems encountered in developing a set of heuristics to guide an autonomous vehicle include:

- Rules are too crisp and not sufficiently flexible to cover real-world situations.
- No one knows all of the rules needed in advance.
- Fuzzy logic only addresses quantitative aspects of the rules; whereas, the qualitative aspects are at least as important.
- Conventional rule acquisition is slow and otherwise impractical.
- Neural networks cannot plan effectively.
- Neural networks cannot learn nearly fast enough to be used in conjunction with a human trainer.
- Cases cannot be autonomously adapted, which limits the applicability of CBR here.
- Systems do not grow their knowledge bases by sharing, which limits the acquisition of knowledge by swarms of Unmanned Aerial Vehicles (UAVs).
- Algorithmic control aspects need to be processed separately and locally.

No one would be a pilot if they lacked common sense. They can learn how to fly, but without common sense, they are preordained to fail. Yet, our predecessors have sought to build UAVs that are lacking in this most critical capability – presumably because they are of the mistaken belief that computers can never exhibit commonsense reasoning beyond that for which they are explicitly programmed. However, knowledge amplifiers, known as KASERs, have been designed and patented by one of the authors [6] and elsewhere, which allow for the symmetric expansion of a knowledge base – enabling heuristic programs to reason by analogy – just like human pilots.

Applications for autonomous or semi-autonomous vehicles such as the UAVs that support the US military in reconnaissance missions (Predator and the GlobalHawk, for example) or could support our civilian forces using autonomous robot colonies (in rescue missions such as the events surrounding the World Trade Center and the

Dagstuhl Seminar Proceedings 10401
Learning, Planning and Sharing Robot Knowledge for Human-Robot Interaction
http://drops.dagstuhl.de/opus/volltexte/2011/2930

Oklahoma City bombing, or in homeland security, for example) continue to spurn new research and development efforts in intelligent agents, light-weight material, fuel-cell based propulsion, hybrid engine designs, smart sensor networks, secure wireless communication networks, and energy-efficient computing architectures. Further, with the advent of advances in nanotechnology and microsystems, several research teams continue to investigate the integration of such technologies for small swarms of AVs or SAVs for military, commercial, and civilian applications.

SSC-PAC has designed, developed, patented, and in some cases transitioned intelligent systems, which serve to automate the creation, testing, and maintenance of complex software systems. Subsequently, the co-authors have collaborated on the construction of a Cognitive KASER for learning and have investigated its application to such areas as web searching and decision support architectures. A brief summary of the work in knowledge amplification is provided in Section III. In summary, in order to test a new learning approach that can bestow intelligence on a system, based upon human (tutor) – robot (student) interaction, we are working jointly to design an intelligent system architecture, based on the KASER, and plan to implement the methodology on an aerial vehicle (a helicopter), which will execute several maneuvers such as basic hovering, steep approach, confined area approach, and basic altitude flying. The software to perform these maneuvers will be developed using two methods (teach mode with a human-in-the-loop) and classical control. Then, the knowledge amplification system will autonomously generate another set of software, which will then be tested and compared to the execution of the baseline two software codes using the same helicopter flight scenarios. We anticipate that such complex functional real-time software can be most cost-effectively written through the use of a software-writing system embodying knowledge amplification.

Keywords—**Evolution, Heuristics, KASER, System of Systems, Unmanned Helicopter**

I. INTRODUCTION

Intelligence is generally defined as the ability to complete a task or achieve a goal in an uncertain environment. Hence, some characteristics of intelligence include adaptability, capability for self-optimization based upon some goal or goals, ability for performing self-diagnostics and self-maintenance, and the ability to learn and reason. Currently, there are many systems which can play chess, perform character/image recognition, have decision-making capabilities, and do control/compensation. These systems may be characterized as having some form of intelligence. The idea of intelligence has been applied to robotics and automation to enhance the applicability to a variety of problems.

Some systems try to mimic human intelligence (e.g., understand speech or recognize handwriting). New technologies have served to advance the field of artificial intelligence – including greater computational power, new algorithms based upon cognitive science, better sensors, and smaller devices requiring less power. The field of intelligent systems has also advanced because of the integration of several disciplines in neuro-computing, evolutionary computing, probabilistic algorithms, fuzzy, neural architectures, and machine learning techniques. Research has progressed from simple symbolic manipulation to information fusion, syntactic ontologies, smart multi-agents, information reuse, and embedded intelligent systems.

One of the obstacles in implementing intelligence methods is the need for more computational power. However, the number of computational devices using embedded software is rapidly increasing and the embedded software's functional capabilities are becoming increasingly complex each year. These are predictable trends for industries such as aerospace and defense, which depend upon highly complex products that require systems engineering techniques to create. We also see consumer products as increasingly relying upon embedded software – such as automobiles, cell phones, PDAs, HDTVs, etc.

Embedded software often substitutes for functions previously realized in hardware such as custom ICs or the more economical gate arrays; for example, digital fly-by-wire flight control systems have superseded mechanical control systems in aircraft. Software also increasingly enables new functions, such as the intelligent cruise control, driver assistance, and collision avoidance systems in high-end automobiles. Indeed, the average car now contains roughly seventy computer chips and 500,000 lines of code – more software than it took to get Apollo 11 to the Moon and

back. In the upper-end cars, in which embedded software delivers many innovative and unique features, there can be far more code.

However, the great number of source lines of code (SLOC) itself is not a fundamental problem. The main difficulty stems from the ever-more complex interactions across software components and subsystems. All too often, coding errors only emerge after use. The software testing process must be integrated within the software creation process – including the creation of systems of systems in a spiral development. This follows because in theory, whenever software becomes complex enough to be capable of self-reference it can no longer be formally proven to be valid [5].

The door is thus opened for heuristic validation and this in turn implies software testing as a bona fide method for software synthesis. Unlike the conventional software development cycle, which may differ from the waterfall model because an allowance must be made for parallel development, heuristic validation can be modeled as shown in Figure 1.

Here, searching a transformational space of basis maps is an activity that is under computer control. It constitutes a boundary value problem, where the search technique(s) serve to relax the constraints. This is a fundamentally different approach than that currently followed using such modeling languages as UML (or IBM's RUP-SE) or even deductive techniques based on the Prologue language. The user need only enter and dynamically maintain a basis for the rule set and the machine effectively extrapolates it. Using the UML, there is no automated extrapolation of knowledge. Unlike the predicate calculus (Prologue), candidate induced knowledge need not be valid – only locally valid, which is ascertained by testing. In either case, a system for autonomously writing control software is realized, which substitutes the science of randomization testing for the art of software creation. This difference is profound as will be seen in the next section.

- Design the control variables (sensor variables)
- Design the actuator variables, which may be macros
- Design the local actuator subsystems, which are invoked by the actuator variables
- Repeat
  Acquire a basis of the form, control state → actuator variable, as necessary
  Sense a possibly unique control state
  If a basis map does not apply, then do
  Select a non-empty subset of basis maps
  Search a transformational space of basis maps
  Test the effects of the resulting actuator variable
- Until
  The actuator variable is deemed to be correct.

Figure 1: Component Synthesis using Heuristic Validation

Here, searching a transformational space of basis maps is an activity that is under computer control. It constitutes a boundary value problem, where the search technique(s) serve to relax the constraints. This is a fundamentally different approach than that currently followed using such modeling languages as UML (or IBM's RUP-SE) or even deductive techniques based on the Prologue language. The user need only enter and dynamically maintain a basis for the rule set and the machine effectively extrapolates it. Using the UML, there is no automated extrapolation of knowledge. Unlike the predicate calculus (Prologue), candidate induced knowledge need not be valid – only locally valid, which is ascertained by testing. In either case, a system for autonomously writing control software is realized, which substitutes the science of randomization testing for the art of software creation. This difference is profound as will be seen in the next section.

II. RANDOMIZATION

As software gets more complex, one might logically expect the number of electronic components to grow with it. Actually, the exact opposite is true. Engineers are required to obtain tighter integration among components in an effort to address cost, reliability, and packaging considerations, so they are constantly working to decrease the number of software components but deliver an ever-expanding range of capabilities (Figure 2). Such *randomization* (compression) [1], [3] has an attendant advantage in that it allows for more thorough testing of software components by virtue of their lying on a greater number of execution paths. For example, suppose that one were to use the methodology depicted in Figure 1 to synthesize a sort function. This has actually been done by us and appears in Figure 3. Rather than concern ourselves with such details as, —should I use For i = 1 to n-1, or is it n, or is it n+1" and the like, we set up a transformational space of alternative function configurations and enable the computer to uniformly cover the search space subject to such I/O constraints as (((3 2 1) (1 2 3)) ((3 1 2) (1 2 3))). That is, when I input (3 2 1) to the sort function, it is required to output (1 2 3). Similarly, when I input (3 1 2) to it, it is required to output the same.

The goal here is to cover the maximum number of execution paths using the fewest I/O tests (i.e., heuristic validation). Clearly, there is little value is using a test set such as (((1) (1)) ((2 1) (1 2)) ((3 2 1) (1 2 3)) ((4 3 2 1) (1 2 3 4)) …). The problem here is that this test set is relatively symmetric or compressible into a compact generating function. A fixed-point or random test set is required instead and the use of such relatively random test sets is called, *random-basis testing* [7]. For example, such a test set is (((1) (1)) ((2 1) (1 2)) ((3 1 2) (1 2 3)) ((1 2 3) (1 2 3))). Many similar ones exist. Notice that the human specifies a schema that can be as qualitatively fuzzy as the computational horsepower will permit. It is certainly easier to specify say, For i = 1 to [n-1, n+1] than to be exact (see Figure 3). Besides, the validity of the complex software, under any design methodology is only as good as the testing that it undergoes. It's just that using random-basis testing, while the need for functional decomposition remains, the complexity for the designer is shifted from writing qualitatively crispy codes to writing relatively random tests. The former grows non-linearly with scale; whereas, the latter is essentially linear with scale. Furthermore, as indicated in Figure 1, the test vectors can be dynamically configured by the context of the problem – leading to the automatic synthesis of control rules. Here, testing takes the simpler form of correcting the flight control of an otherwise autonomous helicopter.

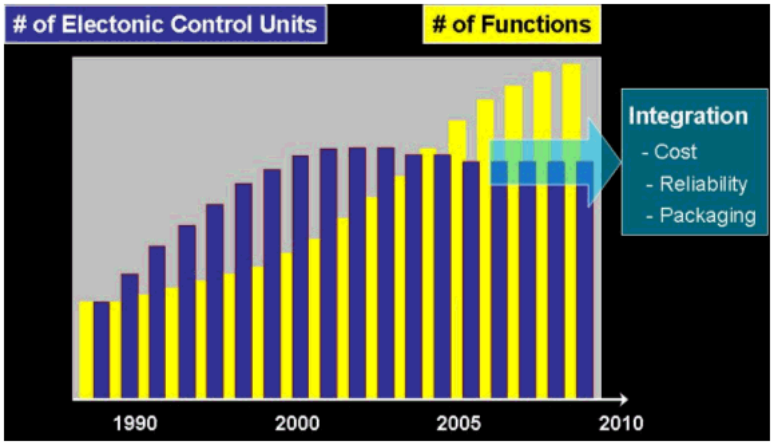

Figure 2: Randomization of Software Components

It is to the robot's advantage to minimize the number of alternatives in any one function (recursively defined). Of course, to do so requires more knowledge be supplied in the form of schemas and associated codes a priori. The more powerful and numerous the processors, the less this is a concern all else being held equal. This process is formally referred to in numerical analysis as the *triangle inequality*.

Given multiple functions and functions of functions, the time-savings can mount up exponentially. Such heuristics need not be perfect to create perfectly viable software. Certainly, the human programmers they replace are not.

Moreover, according to Nilsson [8], such inadmissible heuristics can solve much more complex problems than their perfectionist counterparts. Furthermore, according to an article that made the front page of the Wall Street Journal several years ago, the difference for the Traveling Salesman Problem can be at least as extreme as a century on a supercomputer for admissible (i.e., optimal) heuristics to a second on a PC for inadmissible heuristics at a cost of only 1 percent of optimality!

The methodology presented in Figure 1 has the advantages that (1) it maximally respects the triangle inequality and (2) it maximally reuses its control rules, which serves to minimize the use of admissible heuristics. Of course, the power of heuristics is of little value if we cannot find and incorporate them into the solution. Here is where dimensionality reduction comes into play.

In the context of the autonomous helicopter, dimensionality reduction implies that local functions be off-loaded to embedded controllers. For example, when the control rules actuate the landing signal, the special-purpose landing algorithm takes control and adjusts the necessary servos to insure a reliable and soft landing. While such local control can be acquired by the control rules, this is not appropriate. The reasons are (1) an algorithm can be written for the final stage of landing, (2) the more that the control rules need to survey, the more powerful the host computer needs be, (3) the use of assisted local embedded controllers implies faster learning – all things being held equal, and (4) the control rules are more appropriately used for planning purposes, where computational creativity is truly needed.

```
((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)))))))
```

Figure 3: Component Synthesis by Way of Testing

III. THE KASER PLATFORM FOR INTELLIGENCE

The concept of randomization has previously been shown to reduce the effort required to write software (or generate high-level designs), while concomitantly improving the quality of the resulting code. Context-free grammars are inherently capable of randomizing, but only within the confines of a certain logical regimen. To achieve a greater degree of randomization, one must step outside of those confines. It is clear that when one 'steps outside' of these confines, one is dealing with randomizing the representational formalism itself – not merely what is represented in that formalism.

If a component is capable of transforming other components, it is said to be *active*. The same component can be passive in one situation and active in another. Active components can also transform each other and even themselves. In fact, two interesting observations can be made at this point. First, DNA-based strands and the enzymes that are spun off of them, which of course are basic to living organisms, have been playing the active/passive component game for eons. Such examples define randomization operations at ever-increasing levels of complexity. It also follows that since these enzymes, or active components, are capable of self-reference, that a countably infinite number of properties hold true for the system – all of which are true, but none of which can be proven. In the case of biologically-based organisms, the definition of life is one such property. It used to be held that something was alive if it was capable of reproduction.

Grammars that consist entirely of passive components allow for the design of relatively inefficient components. This follows because the user may select a locally correct component definition in a pull-down menu, but do so unaware that the resulting global design will be inefficient if that component is used in the local context. The inclusion of active components provides for the capture and reuse of the users expressed knowledge of optimization. Such optimizations can be applied to active components – including, at least in theory, self-referential optimization. Thus, there need be no attendant inefficiency of scale if the grammar includes active components.

Clearly, randomization is ubiquitous. It must then be the case that the proper space-time tradeoff is defined by the domain to which the system is to be applied. Again, such systems are necessarily domain-specific in keeping with the dictates of the halting problem in computability theory. For example, a web address repeater should be composed entirely of passive components. On the other hand, a search for the deepest most random knowledge must be conducted by pure chance, strange though as it must sound. The great majority of other systems fall between these extremes and it is for those that these randomization techniques need to be detailed.

Transformational components need to be represented in a context-free grammar in such a manner as to be context sensitive in their use. For example, one software design may require a quicksort component because it involves a large data set, while another may require an insertion sort because it does not. Clearly, the suggested component can never converge in the absence of context. Furthermore, it is only appropriate to substitute one component for its siblings when that component is provably better. Such is the case when, for example, one component has been run through an optimizing compiler and the other has not.

Again, without breaking boundaries, the existential optimization is the same as the initial amended grammar. Interaction with the user will serve to further randomize this model. Randomization involves the substitution of existing component definitions into grammar strings. Clearly, this technique involves a heuristic search because the order of substitutions is critical.

Associative Memory allows for the partial syntactic specification to induce retrieval of the full syntactic specification. This is implemented in hardware at the level of bit-pattern matching. The brain may use such an underlying substrate to realize *semantic associative memory* (SAM). Here, a partial set or sequence is associatively filled in and the result may trigger a mnemonic representation, recursively defined.

A realization of component transformation is provided by the KASER. The KASER is a knowledge amplifier (the acronym stands for *K*nowledge *A*mplification by *S*tructural *E*xpert *R*andomization) based on the principle of randomization. This principle refers to the use of fundamental knowledge in the capture and reduction of a larger, dependent space of knowledge (not excluding self-reference). In a KASER system, the user supplies declarative

knowledge in the form of a semantic tree using single inheritance. Unlike conventional intelligent systems, however, KASERs are capable of accelerated learning in symmetric domains [9-10].

Conventional expert systems generate cost curves below the breakeven line. In conventional expert systems, cost increases with scale and the increase is never better than linear. In the case of KASER systems, the cost *decreases* with scale and is always better than linear, unless the domain is asymmetric (random). Perfectly (asymmetric) random domains are trivial constructs and are not encountered in the construction of practical applications [1].

Conversely, perfectly symmetric (non-random) domains are also trivial and are also not found in practice [1]. In other words, a perfectly random domain would have no embedded patterns (true random numbers), while a perfectly symmetric domain would be infinitely compressible (free of information content). Clearly, such constructs are strictly artificial. The more symmetric is the operational domain, the less the cost of knowledge acquisition.

As a synopsis of the KASER, a production rule is defined to be an ordered pair whose first member is a set of antecedent predicates and whose second member is an ordered list of consequent predicates. Predicates can be numbers or words [11-12]. The linking of the two members forms rules or courses of action.

KASER systems can be classified as different types, depending on their characteristics. In a Type I KASER, words and phrases are entered through the pull-down menus. The user is not allowed to enter new words or phrases if an equivalent semantics already exists in the menu. In a Type II KASER, distinct syntax may be equated to yield the equivalent normalized semantics. The idea in a Type II KASER is to ameliorate the inconvenience of using a data entry menu with scale. In a Type II KASER, selection lists are replaced with semantic equations from which the list problem is automatically solved. Other more enhanced versions of the KASER have been developed [see 13-14, for example].

Thus a KASER system can amplify a knowledge base. It represents an advance in the design of intelligent systems because of its capability for symbolic learning and qualitative fuzziness. In a conventional expert system, the context may cover the candidate rule antecedent, in which case an agenda mechanism is used to decide which matched rule to fire (most-specific match, first to match, chance match.). The KASER system follows the same rule-firing principle – only the pattern-matching algorithm is necessarily more complex and embeds the conventional approach as its degenerate case.

IV. TESTING THE APPROACH: THE AIRWOLF PROJECT

Adaptive Intelligence using Randomization with Optimal Learning Functions (Airwolf) is a small-scale helicopter meant to demonstrate but one credible transition path for our approach to the creation of autonomous software. A successful result here will not only allow for the subsequent construction of more capable UAVs, which may incorporate advanced vision sensors for reconnaissance work, but will allow for all-manner of control software to be written through testing as opposed to the design and test methodology currently practiced. Moreover, our iterative approach to control will allow for the capture of commonsense reasoning [6], [13]. This is critical to the emulation of a pilot's decision-making process as well as in other domains, where software must emulate aspects of human creativity.

Airwolf will test our concept in a hard real-time environment and will allow for a comparison of its learning capabilities with that of a custom neural network. In fact, neural networks have been used to learn to write chess software [15]. The problem is that they are extremely slow. Speed is certainly important in autonomous and necessarily creative control of a helicopter. Of course, this is not to say that they are lacking in utility. Neural networks make for excellent local preprocessors of sub-symbolic data (e.g., for imagery) to feed-in to a higher-level software synthesis system. Just as the human cortex is organized into areas of specialization, the same is the case for systems of systems for autonomous software control.

Small scale helicopters are low cost and allow reasonable payloads (sensors, power supplies and embedded controllers) for acquiring signals and data. Given the limited amount of computing power on-board, these systems also challenge the software community in designing efficient yet reliable codes. Further, systems engineers are challenged by the integration requirements, given the low power and small footprint requirements of small-scale helicopters. With these constraints in mind, the following tasks will be performed for the proposed effort.

V. CONCLUDING REMARKS

Features are based on a composition of functions, which are subject to realization on parallel/distributed processors. Each function is said to be immutable because it may embody the full range of features in the programming language used for realization; yet, once specified, it exists in a library of functions and may only be subject to manual update. Such functions may be iteratively composed using one or more pairs of nested braces, {}.

Selections are made at uniform chance. The search space here can rapidly grow to be intractable if certain precautions are not taken. The complexity of definition here usually results in features being ascribed a heuristic definition. Notice how symmetry provides direct support for heuristic programming.

A full schema has a search space of $\prod_{i=1}^{m}(X_{FN1_i}-1)\prod_{i=1}^{n}(X_{FN2_i}-1)$. However, by careful pruning, this search space is reduced to $\sum_{i=1}^{m}(X_{FN1_i}-1)(X_{FN2_i}-1)$, where $m = n$ and FN1 and FN2 are over-parametized functions. Here, $\sum_{i=1}^{m}(X_{FN1_i}-1)(X_{FN2_i}-1) << \prod_{i=1}^{m}(X_{FN1_i}-1)\prod_{i=1}^{n}(X_{FN2_i}-1)$ with scale. Thus, knowledge should be added in this manner whenever possible to minimize the implicit search space. Here, optimizations may be realized by an optimizing compiler; although, the continual evolution of the features makes this less necessary than with conventional programs, since instancing the schemas will be the main time sink. These schemas may be manually optimized by making use of the *triangle inequality*.

The triangle inequality tells us that simply put, the less that is left to chance the more efficient the model will be at producing viable features. The process of manually and iteratively removing chance is referred to as the triangle inequality (e.g., $|A| + |B| \geq |A + B|$). This means that it is better to have several program schemata with few articulation points than to place all of the articulation points in one schema, where the resultant complexity can easily overwhelm at least a low-end computer. A key concept in writing the features is that instead of the user searching for the correct code construct or function at various points in the program, the user specifies a space of alternative constructs at a limited number of articulation points over a maximal number of schemata. In any case, the user specifies reasonable alternatives, which are captured in the form of a set – possibly tagged with a pneumonic id. Here, the user need not contemplate the details – details that would detract from his/her capability to be an efficient programmer/debugger. The computer will find for programs that satisfy all of the test vectors, if possible. As a consequence, the automatic programming of UAVs becomes distinctly practical.

VI. REFERENCES

[1] G.J. Chaitin, "Randomness and Mathematical Proof," *Scientific American*, vol. 232, no. 5, pp. 47-52, 1975.

[2] A.N. Kolmogorov and V.A. Uspenskii, "On the Definition of an Algorithm," In Russian, English Translation: *Amer. Math. Soc.* Translation, vol. 29, no. 2, pp. 217-245, 1963.

[3] S.H. Rubin, "On Randomization and Discovery," *Information Sciences*, vol. 177, no. 1, pp. 170-191, 2007.

[4] R. Solomonoff, "A Formal Theory of Inductive Inference," *Inform. Contr.*, vol. 7, pp. 1-22 and 224-254, 1964.

[5] V.A. Uspenskii, *Gödel's Incompleteness Theorem*, Translated from Russian. Moscow: Ves Mir Publishers, 1987.

[6] S.H. Rubin, S.N.J. Murthy, M.H. Smith, and L. Trajkovic, "KASER: Knowledge Amplification by Structured Expert Randomization," *IEEE Transactions on Systems, Man, and Cybernetics – Part B: Cybernetics*, vol. 34, no. 6, pp. 2317-2329, 2004.

[7] Q. Liang and S.H. Rubin, "Randomization for Testing Systems of Systems," *Proc. 10th IEEE Intern. Conf. Info. Reuse & Integration*, Las Vegas, NV, 2009.

[8] N.J. Nilsson, *Principles of Artificial Intelligence*, Mountain View, CA: Morgan Kaufmann Publishers, Inc., 1980.

[9] S. Rubin and G. Lee, "Learning Using an Information Fusion Approach", Proc. of the ISCA Int'l Conference on Intelligent and Adaptive Systems, Nice (2004)

[10] S. Rubin and G. Lee, ―On the Use of Randomization for System of Systems (SoS) Design of Intelligent Machines", Proc. of the World Automation Congress, ISSCI, Budapest (2006)
[11] L.A. Zadeh, ―From Computing with Numbers to Computing with Words – From Manipulation of Measurements to Manipulation of Perceptions," IEEE Trans. Circuits and Systems, vol. 45, no. 1, pp. 105—119 (1999)
[12] S. Rubin, R. Rush, Jr., J. Boerke, and Lj. Trajkovic, ―On the Role of Informed Search in Veristic Computing," Proc. 2001 IEEE Int. Conf. Syst., Man, Cybern., pp. 2301--2308 (2001)
[13] S.H. Rubin, G. Lee, W. Pedrycz, and S.C. Chen, ―Modeling Human Cognition Using a Transformational Knowledge Architecture", *Proc. of the IEEE Intl. Conference on System of Systems Engineering (SoSE)*, Monterey, CA, 2008, IEEE 978-1-4244-2173-2.
[14] S. Rubin and G. Lee, ―Requirements for an Intelligent System for Experts" Proc. of the ISCA International Conference on Computers and Their Applications, Honolulu, 2010.
[15] D.B. Fogel, *Blondie24: Playing at the Edge of AI*, Mountain View, CA: Morgan Kaufmann Publishers, Inc., 2001.