Test Program Generation for Functional Verification of PowePC Processors in IBM

Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein,
Yossi Malka, Charlotte Metzger, Moshe Molcho, Gil Shurek

IBM Israel – Haifa Research Lab
IBM AS/400 Division, Rochester, Minnesota
yossi@haifasc3.vnet.ibm.com

Abstract

A new methodology and test program generator have been used for the functional verification of six IBM PowerPC processors. The generator contains a formal model of the PowerPC architecture and a heuristic data-base of testing expertise. It has been used on daily basis for two years by about a hundred designers and testing engineers in four IBM sites. The new methodology reduced significantly the functional verification period and time to market of the PowerPC processors. Despite the complexity of the PowerPC architecture, the three processors verified so far had fully functional first silicon.

1 Introduction

A new methodology and tool for functional test program generation has been used for several IBM PowerPC processors. The functional verification period and the processors time to market were reduced. Only one fifth of the simulation cycles needed to verify a RISC System/6000 processor with a previous generator was needed with the new methodology for a PowerPC processor.

The new generator is an expert system that contains a formal model of a processor architecture and a heuristic data base of testing knowledge. The heuristic data-base allows testing engineers to add knowledge to the tool throughout the verification process. The formal architectural model enables test program generation for different processor architectures and gives the generator its name – *Model Based Test Generator*.

32nd ACM/IEEE Design Automation Conference ®

IBM has invested about three million dollars in developing the methodology. About two million dollars went into testing-knowledge acquisition and development; the rest was put in developing the test program generator itself.

Automatic test program generators for functional verification of processors were previously reported ([1], [5], [3], [4], [2]). The assumption behind these tools is that only a small subset of possible tests are simulated during functional verification. These tests are run through the design simulation model (the HDL model), and the results are compared with those predicted by the architecture specification represented by a behavioral simulator. Namely, processor functional verification consists of two steps: 1) Generation of tests – for complex processors, this is done by automatic test generators which supply better productivity and quality than manual tests. 2) Comparison between results computed by the two levels of simulation – both simulators are provided with the generated tests as stimuli, and the simulators final states are compared.

The rest of this paper is organized as follows. We describe the problematic aspects of functional test program generators and the motivation for the new approach (section 2). Then, the new test program generation system (section 3) and its generation scheme (section 4) are described. The results of using the methodology (section 5) and a brief description of further work to be done (section 6) conclude this paper.

2 Motivation

The goal of functional verification is to achieve a first logically operational silicon. The direct cost of silicon realization of a processor design is between $150K to $1M depending on the number of chips and technology. On top of that, the costs of bringing up the hardware and debugging it are also high. For complex processors and manual tests, first operational

silicon was a practical impossibility. With automatic test generation, this goal is still difficult to achieve. For example, the verification of a RISC System/6000 processor necessitated fifteen billion simulation cycles, large engineering teams and hundreds of computers during a year. The *first* motivation of the new methodology is to produce better quality tests. Better quality will allow smaller test sets, lower simulation costs and shorter verification periods.

In order to generate better quality tests, it is needed to capture test engineers expertise in the test generator. The *second* motivation is to allow test engineers to add knowledge in a visible and systematic way to the test generator. This will allow immediate usage of the testing expertise developed during the verification process. Furthermore, it will allow re-usage of this knowledge for future designs of the same architecture. Previous functional test generators did include testing knowledge. However, it was usually coded by the tool's developers as part of the generator; it lagged after the test engineers expertise, and was difficult to re-use.

The *third* motivation is to re-use functional test generators across different architectures. Previous generators were custom made for specific architectures and cost millions of dollars per tool. Separating the architectural details from the architecture independent generation will allow re-usage of the generation core for different architectures. It will reduce the tool development costs, and allow its easy update when the architecture evolves during the verification process.

The *fourth* and last motivation is to reduce the complexity of functional test generators. Previous tools incorporated the architecture, testing knowledge, generation procedures and behavioral simulator in one software system. Separating these components and generalizing the generation procedures to operate on different architectures and testing knowledge will reduce the overall complexity of the generator. It will also make architecture and testing knowledge more visible and allow easier tool adjustment when they change.

3 The System

The Model Based Test Generator comprises an architectural model, a testing knowledge data-base, a behavioral simulator, architecture independent generator and a Graphical User Interface (figure 1). The generator and GUI have been implemented in C, spanning about 75,000 lines of code.

The architectural model contains a specification of instructions, resources and data types as described in section 4.1. The heuristic knowledge base includes generation and validation functions implemented in C (section 4.3). Both architectural specification and testing knowledge are stored in an object oriented data-base; its class hierarchy is given in figure 2.

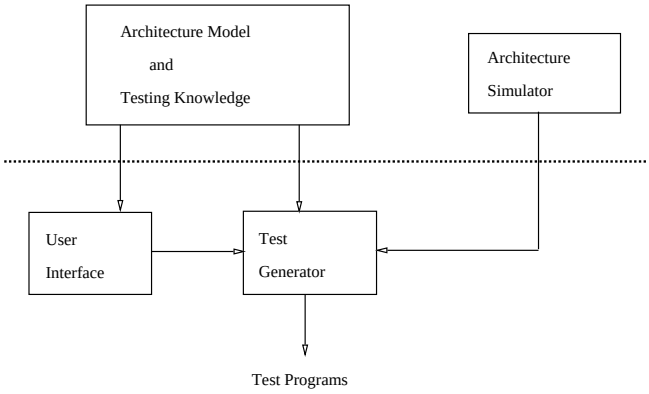

Figure 1: System components and interrelations

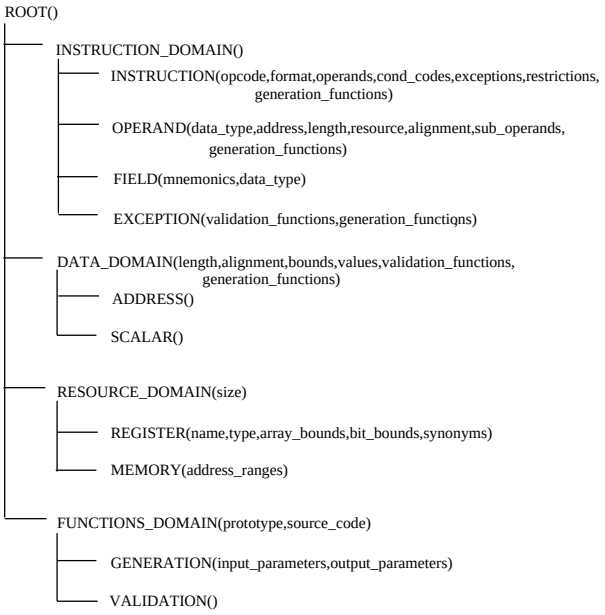

Figure 2: A class hierarchy skeleton

The population of the architectural model is carried out by an engineer familiar with the architecture books and the test program generation methodology. The heuristic testing knowledge is populated by test experts. The testing knowledge accumulated during the verification of the first PowerPC design includes about a 1,000 generation and validation functions (see 4.3) totalling 120,000 lines of C code. All subsequent PowerPC processors have been verified using this knowledge, avoiding duplication of effort and reducing costs.

The system employs a separate architecture simulator which is built to allow operating system development before the processor is available in hardware. A PowerPC behavioral simulator spans about 40,000 of C++ code.

The Motif based GUI offers extensive control over the generation process. Apart from the ability to determine the number of instructions in each test and to initialize resources, the user can direct the generation at three levels: 1) *Global* selections pertain to the generation process as a whole. 2) *Local* selections apply to every instance of a particular instruction whenever it is generated. 3) *Specific* selections bear on particular instances of generated instructions. The generator interprets many user directives as selection of generation functions which reside on the testing knowledge base.

The system is used on a daily basis by more than a hundred designers and test engineers in four IBM sites. It is used under different verification methodologies: from massive test program generation and simulation with little user direction to specific test generation directed carefully to cover detailed test plans.

4 Modelling and Generation

This section describes by example the architectural model, the testing knowledge base and the generation scheme. The expert-system perspective of the Model Based Test Generator are detailed in [6].

4.1 Architectural Model

Instructions are modeled as trees at the semantic level of the processor architecture. Generation of instruction instances is done by traversing the instruction tree. An instruction tree includes a format and a semantic procedure at the root, operands and sub-operands as internal nodes and length, address and data expressions as leaves. The expressions use the instruction's format as alphabet and represent the *static* relations between operands. These relations do not change; they are the same before and after the execution of an instruction. Thus, they are central to the automatic generation of tests and are modelled declaratively. Address expressions denote immediate fields, registers, and memory storage in various addressing modes. Length expressions denote the size of storage used by the operand. Data expressions are just literals denoting datatypes. Declarative modelling of the full semantics would have made automatic generation too complex to be practical. The approach employed here gives enough power to generate useful and revealing test programs whilst keeping the complexity of the generator and the model reasonable. Moreover, the time needed for generation is kept within acceptable limits.

4.2 Example: An `Add Word` Instruction Tree

Although the `Add Word` is not a PowerPC instruction, it is given as an example because it refers to both memory and register resources and has clear semantics. Informally, `Add Word` adds the second operand to the first one and place the sum at the first operand's location. The first operand is both a source and a target; it resides in a memory storage and is pointed to by a base register and displacement. The second operand is used only as a source and is the contents of a word register. The instruction tree is depicted by figure 3. The resources assumed are a main memory, base-registers and word-registers.

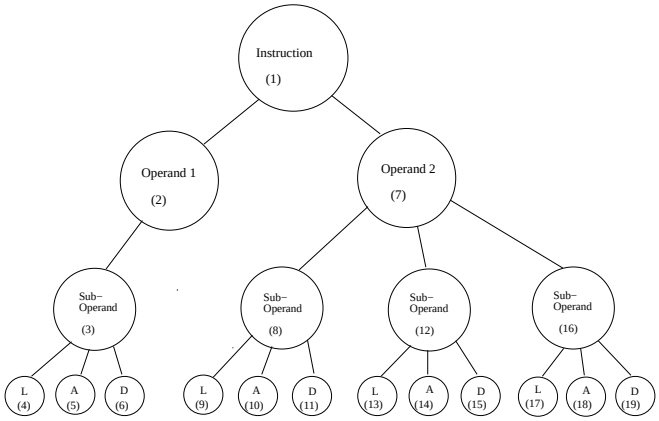

Figure 3: Add-Word

- Instruction: Semantic procedure: `Add-Word()`.
  Format: `AW-OPCODE W1, D2, B2`. Where the fields are:
  `< AW-OPCODE, (AW), No-Resource >`,
  `< W1, (0,1,2, ...,E,F), Word-Register >`,
  `< D2, (00,01,02,, ...,FE,FF), No-Resource >`, and
  `< B2, (0,1,2, ...,E,F), Base-Register >`.

- First operand (represents the register used both as a source and target):

Sub-operand: Length: 4; Address: `register(W1)`; Data: `Signed-Binary`.

- Second operand (represents the memory storage, base register and displacement comprising the source operand):
  Sub-operand: Length: `4`; Address: `contents(register(B2))+value(D2)`; Data: `Signed-Binary`.
  Sub-operand: Length: `2`; Address: `in-field(D2)`; Data: `Displacement-Data-Type`.
  Sub-operand: Length: `6`; Address: `register(B2)`; Data: `Address-Data-Type`.

4.3 Testing Knowledge

The heuristic testing knowledge is represented by generation and validation functions coded in C by test engineers. The functions are invoked while the generator traverses an instruction tree. Traversing a node involves invoking all the generation functions associated with it. The outputs of these functions are used to generate the instances of the current sub-tree. Generation functions serve various purposes:

- Modelling Condition Codes (inter-operand verification tasks):
  An instruction execution may result in the setting of condition code bits. This effect is part of the instruction's specification and is modelled by the semantic procedure. Moreover, the condition codes partition the input domain of the instruction. As it is a common testing knowledge to use this input partitioning, a generation function may bias the data of operands to exercise all condition codes. Program Exceptions are modeled in the same manner.

- Modelling Procedural Aspects of Resources (inter-instruction):
  Address translation and cache mechanisms are common in computer architectures and are not directly modelled in the instruction trees. Generation functions are used to incorporate inputs which test these mechanisms into test programs.

- Data Type Special Values (within operand):
  The domain of (typed) data instances may also be partitioned. Again, it is common to require that representatives of all data-type partitions be tested.

- Modelling Design Implementation:
  Various aspects of the hardware design are usually taken into consideration in the verification process. Although these aspects are not considered part of the architecture, their testing is considered essential.

Validation functions are also used by the generation scheme. After generating a sub-instance-tree, the validation functions associated with the corresponding sub-instruction-tree are invoked. If any of them returns a REJECT answer, the generation results of the sub-tree are retracted and the sub-tree is traversed again. Validation functions serve different purposes: 1) Imposing restrictions that are not modeled by the length, address and data expressions on instruction instances. 2) Preventing instruction instances from causing program exceptions (when they are not desired). 3) Validating typed data instances. Validation functions allow to use relatively simple generation functions; their cost, in terms of REJECT answers and backtracking, has been acceptable.

The fact that generation functions are allowed to produce only simple data-types (i.e., length-instance, address-instance, data-instance), enables a knowledge engineer to express his (or her) testing knowledge in a natural and local manner. Yet, the ability to generate sets of such instances and to associate functions with instructions, operands and sub-operands gives these functions the desired expressiveness. Had generation functions been allowed to create full instruction-instances, they would have been too complex to be written by users. Their simplicity allows openness and make it possible to model the evolving testing knowledge.

4.4 Example: `Add Word` Generation Functions

The `Add Word` instruction tree is augmented with generation functions. This should illustrate the various objectives which may be achieved by generation functions; for example:

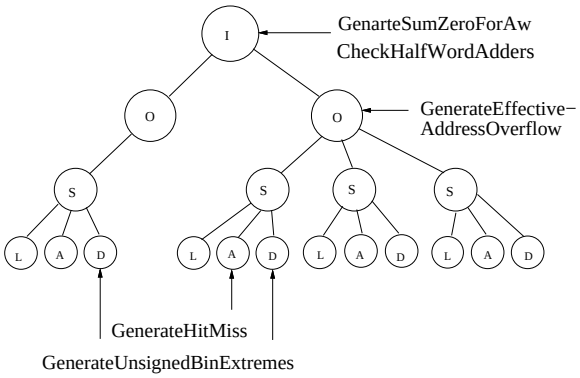

Figure 4: Generation Functions for Add-Word

- Modelling Condition Codes:
  The execution of ADD WORD sets the condition code to SUM IS ZERO, SUM IS LESS THAN ZERO or SUM IS GREATER

THAN ZERO. The function GENERATE SUM ZERO FOR AW is associated with the root of the instruction tree. It generates two (as random as possible) data-instances for the appropriate sub-operands, such that their sum is zero.

- Modelling Procedural Aspects of Resources:
  An address-instance may either be resident in the cache (HIT) or not (MISS). Likewise, the address and length instances of a sub-operand instance may render its least-significant byte as either HIT or MISS. The function GENERATE HIT MISS includes knowledge about the cache mechanism and is associated with the memory address of the second operand. It generates address and length instances which randomly exercise one of the hit/miss combinations.

4.5 Generation

The generation scheme traverses the instruction tree in a depth first order and generates instruction instances. The recursive procedure GENERATE accepts two inputs: the current node to be traversed (NODE) and instances already generated (KEPT-OUTPUTS). At the root and internal nodes generation and validation functions are invoked to produce length address and data instances for the corresponding subtrees. At the leaves, length, address and data instances are either already set by previous selections or are produced using generation and validation functions. This scheme ensures consistency of the generated instruction instances – namely, the values selected for fields and resources that are shared by several sub-operands are identical.

Tests are generated by the procedure GENERATE-TEST; it accepts the required number of instruction instances (N) as input and activates the instruction tree traversal. It also interleaves instruction generation with instruction execution: the intermediate processor state is checked after the new instruction instance has been executed and a decision is taken if to include the new instance in the test.

A resource manager exists in the background of the generation process. It manages the processor's state which is essential for the dynamic GENERATE-TEST algorithm. It is also essential in GENERATE-TEST for resolving address expressions. Generation and validation functions query the resource manager about the allocation and contents of resources. This information is used to select resources and validate the instruction tree expressions.

Great importance is attributed to the efficiency of automatic test generation. Constraint solvers have been introduced to avoid superfluous backtracking, due to violation of relations between values of leaves in the instruction tree (as specified by length and address expressions). A solver is activated at internal nodes of the instruction tree and simultaneously assigns values to the leaves such that the relations are not violated.

```
Initialize the minimal processor state
WHILE Number-of-instructions < N
  Select an instruction
  Denote its model by Instruction-Tree
  GEN:
  Instance = Generate(Instruction-Tree,Empty)
  Simulate Instance by its Semantic-Procedure
  IF Instance is executable
     THEN
      Write it to the test file
      Increment number-of-instructions
     ELSE
      Retract Instance
      Restore the processor's previous state
     IF retry-limit not exceeded
        THEN go-to GEN
        ELSE Abort
Return Success
```

Figure 5: Generate-Test(N)

4.6 Example: An `Add Word` Instruction Instance

The instruction tree (given in section 4.2) is traversed in depth first order; the node labels of figure 3 denote this generation order. An instance of this `Add Word` instruction is depicted by figure 7.

This instruction instance sets both the syntax and the semantic entities of the `Add Word` instruction. The syntax is a format instance (`AW 7, 0100, 9`). The semantic domain includes the contents of word register number 7 (`5F93A16B`), the contents of base register number 9, (`000010008000`), and the contents of the main memory word `000010008100` (`15EA917C`).

5 Results

The Model Based Test Generator has been used in the functional verification of six designs for different derivatives of the PowerPC architecture. Three additional, Complex Instruction Set Computers have been modelled. Table 1 summarizes the current verification experience.

The *Bugs* and *Stage* columns indicate the actual results.

Table 1: Results

	System	Processor	Bits	Bugs	Stage
1	AS/400	PowerPC 1	64	450	First silicon fully functional
2	AS/400	PowerPC 2	64	480	First silicon fully functional
3	AS/400	PowerPC 3	64	600	First silicon fully functional
4	S/6000	PowerPC 3	64	–	Verification in process
5	PC	PowerPC 4	32	–	Verification in process
6	403	PowerPC 5	32	–	Verification in process
7	X86	CISC	32	–	Verification in process
8	S/390	FPU	–	–	Modelling only
9	AS/400	CISC	–	–	Modelling only

```
Invoke Node's generation functions
Add their outputs to Kept-Outputs

IF Node is internal
 THEN
 FOR each of Node's immediate descendants
  Generate(Descendant, Kept-Outputs)
  IF Reject is returned
   THEN
   Restore Kept-Outputs
   IF retry limit not exceeded
    THEN
     Generate(Descendant, Kept-Outputs)
    ELSE Return Reject
   ELSE Return Accept

 ELSE (Node is a leaf)
  Select one of Node's Kept-Outputs
  IF there is no such output
   THEN
    Select randomly an instance from
     the Node's expression semantics
    IF the instance does not
       satisfy this expression
     THEN Return Reject
     ELSE Return Accept

Invoke Node's validation functions
IF any of them returns Reject
 THEN Return Reject
 ELSE Return Accept
```

Figure 6: Generate(Node, Kept-Outputs)

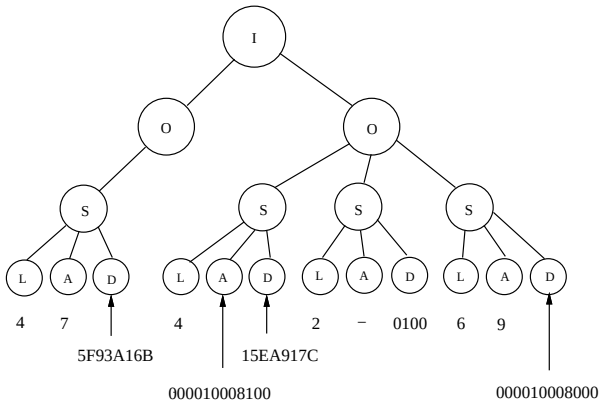

Figure 7: Add-Word Instance

The number of design bugs is a very rough measure of the complexity of the design and its functional verification. The ultimate measure of success is the number of silicon realizations needed to achieve a fully functional processor. For the three verification processes which have been already completed, the *first* silicon realizations are fully operational.

An analysis of the return on investment of the new methodology is yet to be performed. However, preliminary results indicate that it reduces the functional verification costs. Figure 8 provides the bug distributions for two designs during the functional verification of the processors. One verification process used a previous test program generation technology (Processor-A) and the other utilized the new generator (Processor-B).

The number of design bugs is plotted against the number of billions of simulation cycles which correspond to the number

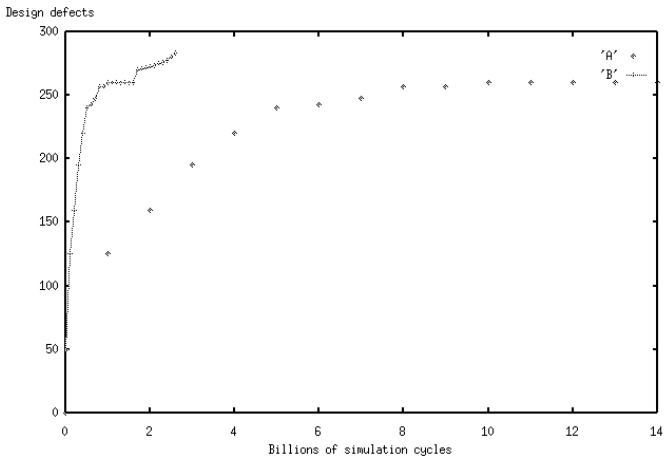

Figure 8: Bug Distribution

and size of generated tests. Processor-B is a high-end 64-bit PowerPC processor (number 3 in table 1); Processor-A is one of the RISC System/6000 earlier 32-bit processors. The functional verification of Processor-A required five times as much simulation cycles as needed for Processor-B. The simulation period has been reduced from fifteen months (Processor-A) to six months (Processor-B). These factors translate into cost: simulation of a huge number of cycles requires hundreds of computers running continuously; tieing design and testing teams for long periods is expensive.

The above numbers and graph indicate that testing knowledge usage gives better quality tests; fewer simulation cycles are then needed to uncover the design bugs. The verification of Processor-A used a close generation system, modelling only some testing knowledge. It relied heavily on random test program generation and giga-cycles of simulation. In contrast, the possibility to direct the generation by user-defined testing knowledge emphasized quality throughout the functional verification of Processor-B.

An additional benefit of the new methodology is the reduction of the effort required to adapt the tool to a new architecture. In case of close architectures, the cost was two man-months: Moving from the 64-bit AS/400 PowerPC processor to the 403 32 bit PowerPC micro-controller entailed updating the data bases only. Enhancing the generator to support X86 CISC architectures took five calendar months; it is considerably less then building a new generator.

A major weakness of the new tool is its performance. A previous generator produces about 50 instructions per second on an S/6000 workstation. The new test program generator produces about 10 instructions per second in similar conditions. We feel that this slow-down factor is compensated for by the better quality of the tests.

To conclude, the available indications show that the new test generation methodology obtain better quality and reduce time to market.

6 Further Work

We are conducting a broad bug classification work across different architectures, design process and verification methodologies. We hope that new testing knowledge will be acquired through this work and better quality tests will be achieved. In particular, we study the nature of instruction sequences involved in uncovering bugs.

Coverage achieved by automatically generated test cases is also studied. We are investigating different coverage criteria over the architectural specification and the HDL implementation. We hope to use architectural coverage measures as basis to architectural conformance test sets. We are investigating a hierarchy of criteria that will allow to label different levels of testing quality from low level sets used for frequent regression testing to high quality tests used for full functional verification.

References

[1] A. Aharon, A. Bar-David, B. Dorfman, E. Gofman, M.; Leibowitz, and V. Schwatzburd. Verification of the ibm risc system/6000 by a dynamic biased pseudo-random test program generator. *IBM Systems Journal*, 30(4), April 1991.

[2] A. M. Ahi, G. D. Burroughs, A. B. Gore, S. W. LaMar, C. R.; Lin, and Wiemann A. L. Design verification of the hp 9000 series 700 pa-risc workstations. *Hewlett-Packard Journal*, 14(8), August 1992.

[3] J. I. Alter. Dacct – dynamic access testing of ibm large systems. In *International Conference on Computer Design (ICCD)*, 1992.

[4] W. Anderson. Logical verification of the nvax cpu chip design. In *International Conference on Computer Design (ICCD)*, 1992.

[5] A. Chandra and V. Iyengar. Constraint solving for test case generation – a technique of high level design verification. In *IEEE International Conference on Computer Design (ICCD)*, 1992.

[6] Y. Lichtenstein, Y. Malka, and A. Aharon. Model-based test generation for processor design verification. In *Innovative Applications of Artificial Intelligence (IAAI)*. AAAI Press, 1994.