Verification and Validation and Artificial Intelligence

Tim Menzies
Lane Department of Computer Science,
University of West Virginia,
PO Box 6109, Morgantown,
WV, 26506-6109, USA;
http://tim.menzies.com
tim@menzies.com

September 15, 2002

Abstract

AI software is often a *declarative model-based executable knowledge-level nondeterministic complex adaptive* system. These article describes all these features as well as their implications for verification and validation.

To appear, Foundations 02: A V&V Workshop; October 22-23, 2002Kossiakoff Conference & Education Center, Johns Hopkins U. Applied Physics Lab Laurel, Maryland USA. Home page: http://www.sisostds.org/webletter/siso/iss_86/art_493.htm.

Download this paper at http://tim.menzies.com/pdf/02vvai.pdf.

WP ref: wp/02/aivv/aivvis-v2.

Contents

List of Figures

About the author

Former nurse/taxi-driver. Failed hippy (a watch needs tension in the spring to make it go). Forever Australian but currently on extended loan to the USA.

Dr. Menzies is the Software Engineering Research Chair at the Lane Department of Computer Science and Electrical Engineering, West Virginia University. In that position he consults nearly fulltime with the NASA IV&V facility Fairmont West Virginia on applying advanced software engineering techniques to NASA problems.

Dr. Menzies has a long background in practical applications of artificial intelligence. For example, he was the author of Australia's first ever exported expert system in 1987.

Dr. Menzies holds a Ph.D. in artificial intelligence (1995), a masters of cognitive science (1988) and a computer science undergrad degree (1985), all from the University of New South Wales, Sydney, Australia. When studying scientific models, Dr. Menzies routinely uses learners to condense the output of those models into a succinct form. Apart from machine learning, has has explored other automatic generalization tools (e.g. formal concept analysis and rough sets) and their utility in requirements engineering.

He published over 115 published papers in venues such as the IEEE Transactions of Knowledge and Data Engineering [84], IEEE Software [89], the AI in Medicine journal [85], the ACM IEEE International Symposium on Requirements Engineering [47, 91], the IEEE International Symposium on Reliability Engineering [86, 97], the ACM IEEE Automated Software Engineering Conference [99, 101], IEEE International Conference on Tools with Artificial Intelligence [87], and the International Journal on Artificial Intelligence Tools [88].

Active in the research community, Dr. Menzies has organized workshops in the past for ICSE (2000), IJCAI (1991,2001), AAAI (1999), numerous workshops in knowledge acquisition (1996-2000), and the recent model-based requirements engineering workshop (2001). He has also served as guest editor for the Requirements Engineering Journal, the International Journal of Human Computer Studies (twice). Currently, he is special guest editor and organizer of a special issue on Empirical AI for the IEEE journal of Intelligent Systems.

1 Introduction

Imagine you are a verfication and validation (V&V) analyst asked to review some artificial intelligence (AI) software. Would you know what to do? How should you modify your approach from regular V&V? What are the traps of V&V of AI software? What leverage for V&V can be gained from the nature of AI software?

This article offers an overview of the six features of AI systems that a V&V analysts must understand. An AI system is often some combination of:

1. A *declarative model-based* system;
2. A system that is *executable* very early in its development.
3. A *knowledge-level* system.
4. An *nondeterministic* system
5. *Complex software*.
6. *Adaptive software*.

Fortunately, not all AI systems have all the above features since each can come with a *significant cost*. However, each of these features grants *significant benefits* that can make the costs acceptable.

The rest of this article is structured around this list of features of an AI system. The features will be discussed in the order above. Each feature will be defined and its costs and benefits summarized. Our discussion will use the abbreviations shown in Figure 1.

AI artificial intelligence

KB knowledge base

KE knowledge engineering

KBS knowledge based system

RAX remote agent experiment

V&V verfication and validation

Figure 1: Abbreviations used in this article.

Note that the approach of this review is different to the traditional reviews of AI verification [3, 24, 45, 55, 105, 106, 114, 119–122, 127, 143] or AI validation [21, 58, 74, 107, 115, 126, 128, 147, 156]. Much has changed since the early days of AI and the field has moved on to more than just simple rule-based systems. While other articles offer success stories with that representation (e.g. [4, 43, 80, 139] and [18, chpt8,30,31,34]), this review focuses on the features of modern AI that distinguishes it from conventional procedural software; e.g. non-deterministic adaptive knowledge-level systems. If the reader is interested in that traditional view, then they might care to read the references in this paragraph (in particular [4, 18, 122, 156] or one of the many excellent on-line bibliographies on V&V of AI systems[1].

2 Model-based AI Systems

Every V&V analysts knows that reading and understanding code is much harder than reading and understanding high-level descriptions of a system. For example, before reading the "C" code, an analyst might first study some high-level design documents. The problem with conventional software is that there is no guarantee that the high-level description actually corresponds to the low-level details of the system. For example, *after* the high-level block diagram is designed, a programmer might make a call between blocks and forget to update the high-level block diagram.

A distinct advantage of *model-based* AI systems is that the high-level description *is* the system. A common technique used in AI is to define a specialized, succinct, high-level modelling language for some domain. This high-level language is then used to model the domain. If another automatic tool is used to directly execute that notation, then we can *guarantee* that the high-level model has a correspondence to the low-level execution details.

These models are often *declarative* and V&V analysts can exploit such declarative knowledge for their analysis[2]. Declarative representations can best be understood by comparing them to *procedural representations* used in standard procedural languages such as "C". Procedural representations encode the precise ordering

[1] E.g. `http://www.csd.abdn.ac.uk/~apreece/Research/vvbiblio.html`

[2] Logic programming theorists distinguish between "theories" and "models" where the latter is an instance of the former and is generated automatically at runtime. This article will use "model" in its more common usage; i.e. the thing that is generated by analysts when they record information about their domain.

require to complete some task. Such procedures store knowledge about *how* to do something. This knowledge is held by individuals in a way which does not allow it to be communicated directly to other individuals. Procedural knowledge often manifests itself in the *doing of something*. Much human knowledge, such as riding a bicycle, is procedural. Anyone who has tried to train another in riding a bike knows that it is difficult to reduce to words that which we obviously know or know how to do. In fact, attempts to do so are often recognized as little more than after-the-fact rationalizations (a phenomenon well known to all requirement engineers).

On the other hand, declarative representations describes facts and relationships within a domain. Declarative knowledge is often statements about *what* is true in a domain. Such knowledge can takes the form of relatively simple and clear statements which can be added and modified without difficulty. Due to its simplicity, declarative knowledge can be easily communicated to others and used in different ways.

For example, consider the following piece of procedural knowledge. This sample of code reports that you have a kind of "X" disease if symptoms can be found for any sub-type of that disease.

```
if ((record.disease(X)==found) &&
    (diseases = record.disease(X).subtypes)
   ) { for(disease in diseases) {
          for(symptom in disease.symptoms) {
              for (observation in observations) {
                 if symptom == observation {
                    printf(
                      "You have %s which is a type of %s!\n",
                      disease,X);
                    return 1 }}}}}
```

This implementation reports that you have "X" if it finds any evidence for any of the sub-types of "X". Suppose we wanted to report the `disease` that we have the *most* evidence for; i.e. the `disease` that has the most `symptoms` amongst the available `observations`. In this procedural representation, this change would imply extensive modification to the code.

A declarative representation of the above might look like this:

```
subtype(bacterial,   measles).
subtype(bacterial,   gastro).
subtype(injury,      carAccident).

symptom(measles,     temperature).
symptom(measles,     spots).
symptom(gastro,      temperature).
symptom(gastro,      dehydration).
symptom(carAccident, wounds).
```

Declarative representations free the analyst from specifying tedious procedural details. For example, the above procedural code could be reproduced as follows:

```
evidence(Disease,SubType,Evidence) :-
    subtype(Disease,SubType),
    symptom(SubType,Evidence),
    observation(Evidence).
```

Of course this declarative representation is useless without some procedure that interprets it. Our example here uses the syntax of the Prolog logic programming language [13]. In that language, upper case words are variables and lower case words are constants. The above definition of `evidence` supplies all the details Prolog needs to specify our search through the `symptoms` and `subtypes`.

Procedural knowledge is opaque and relatively inflexible. Declarative knowledge is far more flexible since the knowledge of *what* is separated from the *how*, This means that the *what* can be used in many ways. For example, suppose we want to drive the diagnosis *backwards* and find what might cause spots. To do this, we first must *trick* Prolog into believing that all observations are possible. This is easily done as follows:

```
observation(_).
```

Here, the "_" is an *anonymous variable* that matches anything at all. In the language of Prolog, this means that we will assume any observation at all. With this trick in place, we can now drive the `evidence` rule backwards to find that `spots` can be explained via a `bacterial` infection.

```
?- evidence(Disease,SubType,spots).

Disease = bacterial
SubType = measles
```

A more complicated query might be to find evidence that *disproves* some current hypothesis. For example, suppose we believe the last query; i.e. the observed `spots` can be explained via a `bacterial` infection. Before we commence treatment, however, it might be wise to first check for evidence of other diseases that share some of the `evidence` for `measles`. Since our knowledge is declarative, we need not change *any* of the `evidence` rule. Instead, we just reuse it in a special way:

```
differentialDiagnosis(Disease,Old,Since,New,If) :-
    evidence(Disease,Old,Since),
    evidence(Disease,New,Since),  % Old and New share some
                                  % evidence
    evidence(Disease,New,If),
    not evidence(Disease,Old,If). % New has some evidence
                                  % not seen in Old
```

With this in place, we can run the following query to learn that `measles` can

be distinguished from `gastro` if `dehydration` can be detected:

```
?- differentialDiagnosis(bacterial,Old,Since,New,If).

Old = measles
Since = temperature
New = gastro
If = dehydration
```

2.1 Declarative Models and V&V

The ability to build simple queries for a declarative model greatly reduces the effort required for V&V. For example, one method for V&V of model-based systems is to build a profile of an *average model*. The TEIREISIAS [36] rule editor applied a clustering analysis to its models to determine what parameters where *related*; i.e. are often mentioned together. If proposed rules referred to a parameter, but not its related parameters, then TEIREISIAS would point out a possible error.

Many declarative modelling languages only use a small number of modelling constructs. This simplifies the construction of translators from one modelling language to another. This can be a useful V&V tool. For example, model checking is a powerful method for automatically exploring all pathways within a program (model checking is discussed in §6.1.3). However, model checkers have to run over a declarative representation of the program. Typically, this must be hand-coded. However, for model-based AI systems, simple automatic tools [118] can convert the syntax of the model to the input syntax of the model checker.

Model checking represents the state-of-the-art in automatic model-based analysis. The technique is powerful, but can be complicated. Feather and Smith report that a much simpler model-based technique can still be very insightful [48]. When asked to check the planner module of NASA's Remote Agent Experiment, they developed the architecture of Figure 2. RAX1's planner automatically generated plans that responded to environmental conditions while maintaining the constraints and type rules specified by human analysts. An important feature of the planner was the declarative nature of the constraints being feed into the planner *and* the plans being generated. Feather and Smith found these plans could be easily and automatically converted into the rows of a database. Further, the constraints could also be easily and automatically converted to queries over the database. As a result, given the same input as the planner, they could build a simple test oracle that could check if the planner was building faulty plans.

The Feather and Smith method can be very cost-effective and applied quite widely:

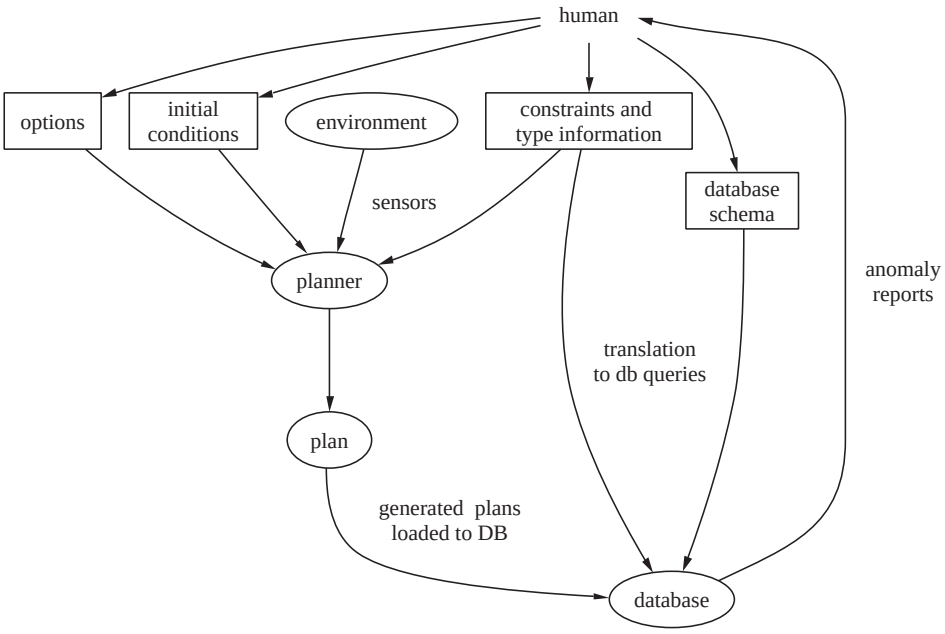

Figure 2: A framework for model-based V&V.

- The rectangles in Figure 2 denoting the sections that must be built manually. Once these sections are built, they can be reused to check any number of plans.
- The architecture of Figure 2 could be generalized to any device that accepts declarative constraints as inputs and generates declarative structures as output.

Preece reports other simple but effective V&V tools that utilize the model-based nature of AI systems [122]. Preece studied rule-based models which are lists of statements of the following form:

$$if \overbrace{L_a \wedge L_b \wedge L_c \wedge \ldots}^{premise} then \overbrace{L_x \wedge L_y \wedge L_z \wedge \ldots}^{conclusion}$$

The Preece analysis defined a taxonomy of verification issues for rule-based models (see Figure 3) and argued that a variety of AI model-based verification tools target different subsets of these issues (perhaps using different terminology).

$$
Anomaly \leftarrow \begin{cases} Redundanct \leftarrow \begin{cases} Unusable \\ Redundant \leftarrow \begin{cases} Duplicate \\ Subsumed \end{cases} \end{cases} \\ Ambivalence \leftarrow Conflict \\ Circularity \\ Deficiency \leftarrow \begin{cases} Missing\ rules \\ Missing\ values \end{cases} \end{cases}
$$

Figure 3: The Preece hierarchy of verification errors.

	Application					
	mmu	tapes	neuron	displan	dms1	ratio errors to anomalies
subsumed	0	5/5	0	4/9	5/59	14/73 = 19%
missing rules	0	16/16	0	17/59	0	33/75 = 44%
circularity	0	0	0	20/24	0	20/24 = 83%

Figure 4: Ratio of $^{\text{errors}}/_{\text{anomalies}}$ seen in real-world expert systems. From [121]. "Subsumed" reports repeated rule conditions. "Missing rules" denote combination of attribute ranges not seen in any rule. "Circularity" reports loops found in the dependency graph between variables in the system.

The Preece taxonomy require meta-knowledge about the terms found within a knowledge base (which Preece et.al. call *literals*):

- A literal L_i is *askable* if it represents a datum that the rule-base can request from the outside world.
- A literal L_i is a *final hypothesis* if it is declared to be so by the rule-base's author and only appears in a rule conclusion.
- A rule is *redundant* if the same final hypotheses are reachable if that rule was removed. An *unusable* redundant rule has some impossible premise. A rule-base is *deficient* if a consistent subset of *askables* leads to no final hypotheses. A *duplicate redundant rule* has a premise that is a subset of another rule premise.
- Preece defined *duplicate rules* for the propositional case and *subsumed redundant rules* for the first-order case. In the first-order case, instantiations have to be made to rule premise variables prior to testing for subsets.
- Preece defined *ambivalence* as the case where, given different consistent subset of askables, a rule-base can infer the same final hypotheses.

Preece stresses that the entries in their taxonomy of rule-base anomalies may not be true errors. For example, the dependency network from a rule-base may show a circularity anomaly between literals. However, this may not be a true error. Such *circularities* occur in (e.g.) user input routines that only terminate after the user has supplied valid input.

More generally, Preece argued convincingly that automatic verification tools can never find "errors". Instead, they can only find "anomalies" which must be checked manually. The percentage of true errors, i.e. $^{\text{errors}}/_{\text{anomalies}}$ can be quite small. For example, Figure 4 shows the $^{\text{errors}}/_{\text{anomalies}}$ ratios seen in five knowledge based system (KBS)s. Note that not all anomalies are errors.

3 AI Models are Executable

Model-based systems offer both a notation *and* an interpreter of that notation. For example, the rule-based systems studied by Preece are often interpreted using a MATCH-SELECT-ACT cycle:

MATCH: Find all the rules with satisfied conditions.

SELECT: Cull that list of rules down to one item. For example, we might favor rules that have worked best in the past.

```
cylinder = object +
[height=
   [range      = number(new value) and new value > 0
   ,help       = print("Height must be a positive number")
   ,if_needed  = ask
   ,if_removed = remove volume from this cylinder
   ,cache      = yes
   ]
,radius=
   [range      = number(new value) and new value > 0
   ,help       = print("Radius must be a positive number")
   ,if_needed  = ask
   ,if_removed = remove cross_section from this cylinder
   ,cache      = yes
   ]
,cross_section=
   [if_needed  = pi * radius of this cylinder ^ 2
   ,if_removed = remove volume from this cylinder
   ,cache      = yes
   ]
,volume=
   [if_needed  = cross_section of this cylinder
                 * height of this cylinder
   ,cache      = yes
   ]
].
```

Figure 5: An example frame. Adapted from `http://www.cse.unsw.edu.au/~billw/cs9414/notes/kr/frames/frames.html`.

ACT: Apply the action of the selected rule.

Another commonly used inference procedure is *frame-based demons*. Frames are like objects except frame-based languages come with built-in mechanisms for searching over all the frames. Frames have *slots* and slots can have *demons*. Demons cause side effects when the slot is accessed. Standard demons are:

if_added: triggered when a new value is put into a slot.

if_removed: triggered when a value is removed from a slot.

if_replaced: triggered when a slot value is replaced.

if_needed: triggered when there is no value present in an instance frame and a value must be computed from a generic frame.

if_new: triggered when a new frame is created.

```
rule332 = rule +
[description           = "Tell the user to hold
                           his breath if the chemical
                           is toxic"
,ifWorkingOnTask       = ascertainImminentDanger
,ifPotentiallyRelevant = toxicity of chemical  is high
,ifTrulyRelevant       = location of chemical is
                               (nearby of user )
,thenTellUser          = "Do not breathe this chemical!"
,thenAddToAgenda       = [summonAmbulances,warnOthers]
,priority              = high
,worth                 = 900
,avgRunningTime        = seconds of 0.1
,frequencyOfUse        = [considered=985, used=4]
,generalizations       = [rule899, rule45]
,specializations       = [rule336]
,justification         = "Breathe&DieScenario"
,author                = johnson
,creationDate          = 81/July/9/17/30
].
```

Figure 6: Rules as frames. Modified from Buch83.

Slots can also have `facets` which contain meta-knowledge about the slot. These facets are not demons themselves, but contain the information or procedures used by some demos. For example, in the frame example of Figure 5:

- `Range` is a facet triggered by the `if-added` demon a new value is added. The new value must satisfy the range and, if not, the procedure in the `help` facet is triggered.
- `Cache` is a boolean facet that means that when a value is computed it is stored in the instance frame.
- If the boolean `multi_valued` facet is true, then the slot may contain more than one value.

Aikins [1] and Lenat [51] noted that the MATCH-SELECT-ACT cycle was often modified to the particulars of a rule-base. The modifications were typically made to control the order in which rules were searched. These authors proposed using frames to model that control knowledge. For example, `rule332` in Figure 6 describes a rule that should be tested if the current task is `ascertainImminentDanger` (note that MATCH can ignore rules outside of the current task- this can speed up the MATCH process). For rules of this form, MATCH first makes a set of quick tests shown in `ifPotentiallyRelevant` which can cull rules that are not truly relevant. The surviving rules are then studied in more detail using the

`ifTrulyRelevant` test. The remaining rules are then sorted according to numerous criteria such as the `worth` of their action, the `priority` of using this rule, and the `frequencyOfUse`. This last criteria might cull this rule since we see that is often considered, but rarely used. The `specializations` of the top ranked remaining rules are then tested to see if some other specialized knowledge should be applied instead of this rule. If not, this rule adds two items to the list of current actions `summonAmbulances` and `warnOthers`.

3.1 RUDE Models

For less-defined tasks, a traditional waterfall development process can stagnate in the analysis stage since not enough is known about the domain. An alternative approach is to use model-based methods to generate an executable version of the current conceptualization of a system. Since the model is high-level and succinct, it should be easy to quickly add more knowledge. On execution, the interaction of that model can lead to surprising results that prompt clarifications and extensions of the model. This approach has been called various things including "knowledge elicitation via irritation" or the RUDE model; i.e.

$$RUDE \ = \ \underline{R}un \rightarrow \underline{U}nderstand \rightarrow \underline{D}ebug \rightarrow \underline{E}dit$$

Rule-based RUDE methods resulted in the "AI spring" of the 1980s. Many well-documented, mature, and optimized rule-based systems were developed such as ART [3], CLIPS[4], and OPS5 [16] (just to name a few). Numerous significant rule-based systems were developed including the commercially successful XCON computer configuration system [81].

3.2 V&V of Executable Models

The ability to directly execute the specification tempts the developer not to produce explanatory documentation about the system. Imagine a V&V analyst asked to review a complex model-based system developed using RUDE. That RUDE system may have no other documentation than the model itself. Without that supporting documentation, it can become difficult to assess the model.

[3] From Inference Corporation

[4] The "C" Language Integrated Production System, developed by NASA [111]

The core of the problem is the *operationalization assumption* [103] which is the mistaken belief that if we can watch a program execute, then we can understand it. The flaw with this assumption is two-fold. Firstly, all too frequently, no other V&V procedure is proposed other than "let's watch it run". That is, the operationalization assumption deludes the developers into doing less testing. Second, usually, it is the creators of the program who are watching it run. We should not ask the creators of a program to evaluate that program by merely watching it run. The "halo effect" prevents a developer from looking at a program and assessing its value. Cohen likens the halo effect to parents gushing over the achievements of their children and comments that...

> What we need is not opinions or impressions, but relatively objective measures of performance. [32, p74].

Consider the case where a program runs and generates 10MB of output. How can a human review all that output? Unless we have (a) some expectation of appropriate behavior and (b) analysis tools for the resulting behavior then we cannot assess if the runtime behavior of a system is adequate. Methods for analyzing that output discussed in this article include the Feather and Smith method of Figure 2; TAR2 (§5.2); runtime verification (§6.1.2); and model checking (§6.1.3).

4 AI and the Knowledge-Level

The blossoming of model-based methods in the AI spring was not followed by an AI summer. Most of those models were rule-based and a assumption underlying the RUDE approach for such systems was the *RUDE rules assumption*; i.e:

> *Rules are independent chunks of knowledge which can be easily added or changed or removed.*

This proved not to be the case. For example, once XCON grew to 10,000 rules, the developers of XCON had a RUDE awakening: maintaining XCON's rules had become fiendishly complicated. To some extent, this was due to the density of knowledge within XCON:

- The expertise within XCON's rules reflected DEC's state-of-the-art knowledge in configuring computers.
- Such a rich library of knowledge will be intricate to maintain, no matter how it is expressed.

However, another factor that complicated XCON's maintenance was that its rules violated the RUDE assumption. Real-world rule bases often contained groups of rules with significant interactions. For example a careful reverse engineering of XCON showed that the system executed though several *operator spaces* where methods for improving the design of a computer were carefully collected, rejected, elaborated, or assessed, before the appropriate best *operator* was finally selected [4]. The use of such coordinating rules violates the RUDE assumption since every addition to the rule base has to be assessed with respect to its effect on the rest of the rules.

Many other researchers argued that rules were not the appropriate primitive construct for AI. Despite careful attempts to generalize the early rule-based work (e.g. [142]), the construction of rule bases remained a somewhat hit-and-miss process. By the end of the 1980s, it was recognized that design concepts such as rule-based methods were incomplete [19]. For example, Bobrow's reverse engineering of real-world AI systems [8] found that numerous paradigms were being employed including rule-based, logic-based, functional, object-oriented, and "access-based" (which, these days, we might call implicit invocation [136]).

The dominant paradigm for the 1990s AI research into better methods for model-based AI was the *knowledge-level* proposal of Allen Newell [112, 113]. In this proposal, intelligence is modelled as:

> A search for appropriate *operators* that convert some *current state* to a *goal state*. Domain-specific knowledge in used to select the operators according to *the principle of rationality*; i.e. an intelligent agent will select an operator which its knowledge tells it will lead the achievement of some of its goals.

For example, the above analysis of XCON that divided its processing into *operator spaces* is a Newell-style knowledge-level analysis.

We can divide research into knowledge level modelling into two broad camps- a majority view and a minority view:

1. In the majority view, a KB should be divided into domain-specific facts and libraries of domain-independent problem-solving methods (PSMs). For example, Clancey argues that AI models should separate heuristics like Figure 7 into domain-specific knowledge about the terminology (see Figure 8) and the meta-knowledge which controls the application of the knowledge (see Figure 9) and true domain-specific heuristic knowledge (see Figure 10) [26]. In the KADS approach, PSMs may be expressed graphically

```
if    the infection is meningitis and
      the infection is bacterial and
      the patient has undergone surgery and
      the patient has undergone neurosurgery and
      the neurosurgery-time was < 2 months ago and
      the patient received a ventricular-urethral-shunt
then  infection = eColi (.8) or klebsiella (.75)
```

Figure 7: A rule.

```
subtype(meningitis,
        bacteriaMenigitis).
subtype(bacteriaMenigitis,
        eColi).
subtype(bacteriaMenigitis,
        klebsiella).

subsumes(surgery,
         neurosurgery).
subsumes(neurosurgery,
         recentNeurosurgery).
subsumes(recentNeurosurgery,
         ventricularUrethralShunt).

causalEvidence(bacteriaMenigitis,
               exposure).
circumstantialEvidence(bacteriaMenigitis,
                       neurosurgery).
```

Figure 8: Some sentences from Figure 7.

Strategy	*Description*
exploreAndRefine	Explore super-types before sub-types.
findOut	If an hypothesis is subsumed by other findings which are not present in this case then that hypothesis is wrong.
testHypothesis	Test causal connections before mere circumstantial evidence.

Figure 9: PSMs identified by Clancey [26] within Figure 7.

```
if     the patient received a
       ventricular-urethral-shunt
then   infection = e.coli (.8) or
       klebsiella (.75)
```

Figure 10: The real heuristic knowledge within Figure 7.

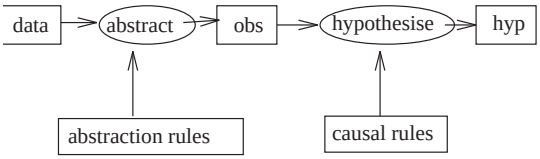

Figure 11: Explicit problem solving (PSM) meta-knowledge: A simple KADS-style PSM for diagnosis. `Abstract` and `hypothesis` are primitive inferences which may appear in other PSMs. From [148].

such as in Figure 11 (ovals are functions, rectangles are data structures).

2. In the minority view, knowledge models use a single PSM. In Newell's operationalisation of KL, this single PSM is the problem-space computational model (PSCM) [113, 154, 155]. Programming the PSCM involves the consideration of multiple, nested problem spaces. Whenever a "don't know what to do" state is reached, a new problem space is forked to solve that problem. The observation that a PSCM system is performing (e.g.) classification is a user-interpretation of a single lower-level inference (operator selection over a problem space traversal) [154].

That is, both the majority and the minority views use PSMs. The difference is that the minority view only uses 1 PSM while the majority approach uses N PSMs. In this, the majority approach, each PSM combines a common set of underlying inference mechanisms (called various terms like "knowledge sources" [151], "mechanisms" [78], etc; e.g. `abstract` and `hypothesis` in Figure 11).

Currently, the major focus of the AI modelling community is on the majority approach: e.g.

- cognitive patterns [50];
- CommonKADS [132, 133, 151];
- configurable role-limiting methods [53, 144];
- MIKE [2];
- the Method-To-Task approach [44];
- generic tasks [23];

- SPARK/BURN/FIREFIGHTER- hereafter, SBF [78];
- model construction operators [26];
- components of expertise [141];

Libraries of PSMs are described in [7, 15, 23, 50, 110, 133, 145]. See the *Related Work* section of [151] for a discussion of the differences in some of these techniques.

A halfway position between the majority and minority views is offered by Chandrasekaran et.al. [23] in which:

- PSMs describe *tasks* (e.g. diagnosis) which can be implemented by...
- *Methods* (e.g. classification, simulation) which can be specified in a generic way using the PSCM. Note that methods may be implemented as tasks (which may recursively contain methods).

However, even though Chandrasekaran et.al. use the PSCM internally, they argue that the task-level is the best view for understanding the system without using too much low-level detail. We have some sympathy with this view. Our biggest criticism of the PSCM is that there is nowhere to model cliched sets of operators which have proved useful in previous applications.

4.1 Knowledge-Level V&V

The knowledge level offers a rich meta-level description of a system. A V&V analyst can use this knowledge to both assess and fix an AI system. For example, van Harmelen & Aben [148] discuss formal methods for repairing KADS-style PSMs such as Figure 11. For example, that figure can be formally represented as a mapping from data d to an hypothesis h via intermediaries Z and other data R_i:.

$$abstract(data(d), R_1, obs(Z)) \bigwedge$$
$$hypothesize(obs(Z), R_2, hyp(h))$$

V&V analysts can restrict their analysis of this model to the three ways this process can fail:

1. It can fail to prove $abstract(data(d), R_1, obs(Z))$; i.e. it is missing abstraction rules that map d to observations.
2. It can fail to prove $hypothesize(obs(Z'), R_2, hyp(h))$; i.e. it is missing causal rules that map Z' to an hypothesis h.

3. It can prove either subgoal of the above process, but not the entire conjunction; i.e. there is no overlap in the vocabulary of Z and Z' such that $Z = Z'$.

Case #1 and #2 can be fixed by adding rules of the missing type. Case #3 can be fixed by adding rules which contain the overlap of the vocabulary of the possible Z values and the possible Z' values. More generally, given a conjunction of sub-goals representing a PSM, fixes can be proposed for any sub-goal or any variable that is used by > 1 sub-goal.

Another knowledge-level V&V technique is to audit how the PSMs are built. Knowledge not required for the PSM of the application is superfluous and can be rejected. In fact, numerous AI editors are *PSM-aware* and auto-configure their input screens from the PSM such that only PSM-relevant knowledge can be entered by the user. For example:

- RIME's rule editor [39, 139] acquired parts of the KB minority-type meta-knowledge for the XCON computer configuration system [4]. RIME assumed that the KB comprised operator selection knowledge which controlled the exploration of a set of problem spaces. After asking a few questions, RIME could auto-generate complex executable rules.
- SALT's rule editor interface only collected information relating directly to its propose-and-revise inference strategy. Most of the SALT rules ($^{2130}/_{3062} \approx 70\%$) were auto-generated by SALT.
- Users of the SPARK/ BURN/ FIREFIGHTER (SBF) [78] can enter their knowledge of computer hardware configuration via a click-and-point editor of business process graphs. SBF reflects over this entered knowledge, then reflects over its library of PSMs. When more than one PSM can be selected by the entered knowledge, SBF automatically generates and asks a question that most distinguishes competing PSMs.

PSM-aware editors can not only assist in entering knowledge, but also in testing and automatically fixing the entered data. For example, in the case where numerous changes have to be made to a PSM, if the user does not complete all those changes, then the PSM may be broken. Gil & Tallis [54] use a scripting language to control the modification of a multi-PSM to prevent broken knowledge. These *KA scripts* are controlled by the EXPECT TRANSACTION MANAGER (ETM) which is triggered when EXPECT's partial evaluation strategy detects a fault. Figure 12 shows some speed up in maintenance times for two change tasks for EXPECT KBS, with and without ETM. Note that ETM performed some automatic changes (last row of Figure 12).

	Simple task #1				Harder task #2			
	no ETM		with ETM		no ETM		with ETM	
	S4	S1	S2	S3	S2	S 3	S1	S2
Total time (min)	25	22	19	15	74	53	40	41
Time completing transactions	16	11	9	9	53	32	17	20
Total changes	3	3	3	3	7	8	10	9
Changes made automatically	n/a	n/a	2	2	n/a	n/a	7	8

Figure 12: Change times for ETM with four subjects: S1. . . S4. From [54]

5 AI Software can be Nondeterministic

Conventional software is *deterministic* since, usually, it contain hard-wired decision paths that are a one-way function for converting inputs to outputs. A *nondeterministic* product lacks such hard-wired paths and, hence, the same inputs can give rise to different outputs. Nondeterminism is mistrusted and misunderstood by test engineers. For example, the guru of software safety Nancy Leveson cautions that "nondeterminacy is the enemy of reliability" [75]. However, as we shall see, nondeterminacy can be a powerful tool; can enable solutions that are otherwise impossible; and can be a useful tool for a test engineer.

5.1 Random SELECT and Scalability

MATCH-SELECT-ACT systems can exhibit nondeterminism. In the case where MATCH returns more than one possible current action, and SELECT can't pick between them, then a random SELECT operator might be applied. Random SELECT seems a perverse method of processing a system. However, there are many situations where nondeterminism is a valid inference procedure.

A repeated and surprising empirical result is that a randomized SELECT can often solve larger problems faster than traditional complete methods. For example, random SELECT has been observed to generate plans in AI systems one to two orders of magnitude bigger than ever done before with complete search [70, 134].

Other work has compared complete to random SELECT through all the what-ifs generated from requirements models from different stakeholders. The HT0 nondeterministic inference engine [96] uses a random SELECT technique to return the first what-if found using a random walk ordering around the model. HT0 can be compared to HT4 [102]- which is the same algorithm with the random

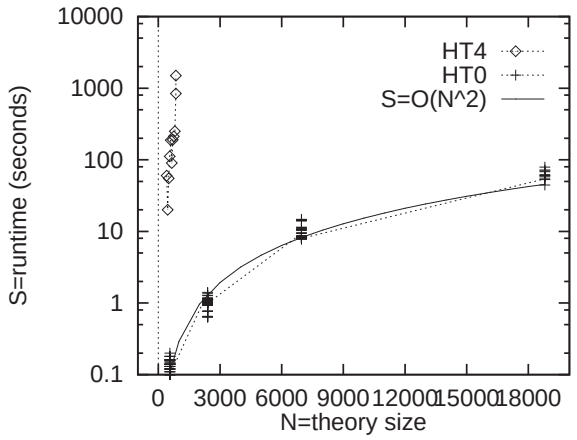

Figure 13: Runtimes

search replaced by a complete search. The runtimes of the two algorithms are shown in Figure 13. Note that experimentally, HT0 takes $O(N^2)$ time to terminate and hence is much faster than the exponential time HT4 algorithm. Note shown in Figure 13 is the fact that HT0's search is nearly as good as HT4: the random SELECT of HT0 finds 98% of the goals found by non-random SELECT.

The success of random SELECT can be explained in terms of the *phase-transition* effect. Traditional theoretical computer science declares a problem impractical if it can be shown to map to a known NP-hard task. Recent empirical studies have challenged that traditional theoretical view. In thousands of studies, conducted by hundreds of researchers around the world (e.g. [25, 52, 138]), the same result appears:

> *Theoretically slow NP-hard tasks are only truly slow in very narrow zones.*

For example, Figure 14 shows the number of times a particular NP-hard algorithm hits a dead-end and has to backtrack. Note that outside of a narrow zone, the algorithm could terminate quickly without extensive backtracking. The slow zone corresponds to the *phase transition* between under-constrained and over-constrained problems:

- In an over-constrained problem, the odds of finding a solution are very low. Further, if the over-constraints are very tight, we can quickly discover that

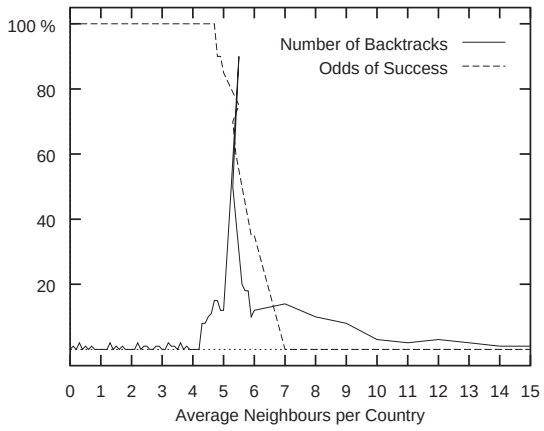

Figure 14: The phase-transition effect.

no solution exists since our searches are all quickly blocked. Figure 14 is over-constrained above $X = 6$.

- In an under-constrained problem, the odds of find a solution is very high since many solutions exist. Figure 14 is under-constrained below $X = 5$.

5.2 Surveying the Nondeterministic Space

The fear with nondeterminism is that the variance in the system's output will be so wild that little can be predicted or guaranteed about the system's performance at runtime. The previous section argued that this fear is not totally founded since recent results offer an empirically-based guarantee that nondeterministic search will find good solutions quicker than complete methods.

Those empirical result may not satisfy safety-focused V&V analysts since the nondeterministic search could blunder into some unsafe mode of operation. However, other empirical results suggest that the space of possible modes from a nondeterministic device can be quickly sampled. The *narrow funnel effect* is the observation that what happens in the total space of a system can be controlled by a small critical region. If nondeterministic systems contain narrow funnels, then the space of possible behaviors reduces the space of possibilities within the funnel. The funnel effect has been reported in many domains (under different names):

- *Master-variables* in scheduling [34];

- *Prime-implicants* in model-based diagnosis [130] or machine learning [129], or fault-tree analysis [76].
- *Backbones* in satisfiability [116, 137];
- *The dominance filtering* used in Pareto optimization of designs [68];
- *Minimal environments* in the ATMS [38];
- Recall from the previous section that HT4's complete search yielded little more than HT0's random search. This result is consistent with the space searched by HT0/HT4 having narrow funnels that constrained both the complete search and random search to similar regions.
- There is some theoretical grounds for believing that narrow funnels are an emergent property of certain systems [100].

To see the effect of narrow funnels, consider Menzies et.al. [83] study of simulations based on a *select* and *cache* strategy. In *select and cache*, if a value for some uncertain parameter is required during execution, that value is *selected* at random based known ranges for that value. The selected value is then *cached* and if that parameter is required again, the cached value is used. The model executions are re-run the model many times, taking care to clear the cache between each run[5].

As opponents of nondeterminism might predict, this approach generates an overwhelming amount of data that clouds and confuses the issues. For example, Figure 15.i and Figure 15.ii show output values generated from two software cost prediction models executed thousands of times using *select and cache*. In these figures, each mark represents the *cost* and *benefits* associated with a set of decisions about the structure of a software project. Note the large variance in the possible cost and benefits from the different possible decisions. Faced with such a large variance in the possible behavior, it is hard to demonstrate that any particular decision leads to a particular outcome.

If these models contained narrow funnels, then the space of possible behaviors should be easily reduced by constraining a small number of variables. And in fact, this turns out to be the case. Figure 15.iii and Figure 15.iv show simulation results from the same models, with a small number of key parameters constrained. The constraints were learnt by the TAR2 *treatment learner* [47,65,92–95]. A treatment learner seeks the *least number* of attribute ranges that *most differentiate* between

[5]Many variants on this scheme have been discussed in the literature. For example this scheme is the same as Monte Carlo simulations when uncertain parameters are just system inputs. Also, this scheme is the same as abductive inference [69] where the uncertain parameters are truth assignments to assumptions within the model, and some global invariant checking executes before a new value is assigned.

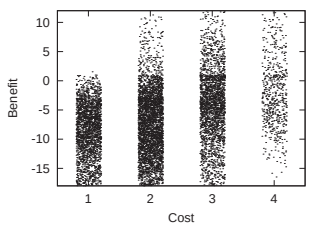

Figure 15.i: Cloud1

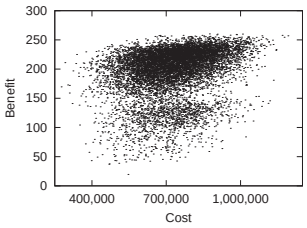

Figure 15.ii: Cloud2

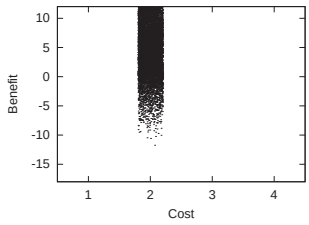

Figure 15.iii: Cloud1, condensed

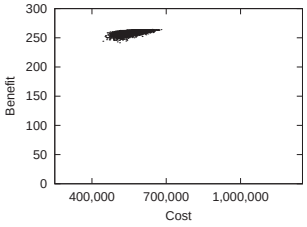

Figure 15.iv:Cloud2, condensed

Figure 15: Examples of condensing clouds. The right-hand model's cost values are continuous while the left-hand model has discrete costs.

desired and undesired behavior. Conceptually, given a set of input attributes and output classifications, the algorithm searches through all combinations of attribute ranges that are under consideration to find those which lead to the *most* desired outputs and the *least* undesirable outputs. This search is clearly intractable since a complete search of all subsets of the input attribute ranges would take exponential time. TAR2 only works if the space being explored contains funnels. If the space does not contain narrow funnels then many variables would be required to control a space and TAR2's run times would be too slow.

TAR2 worked in this domain. Figure 15.iii and Figure 15.iv showed a marked reduction in the variance of the system output and a marked increase in the mean behavior (less cost, more benefit). Note also that this effect is consistent with narrow funnels since the restrictions were achieved by only constraining a few variables while leaving the rest to be chosen nondeterministically during *select and cache*.

TAR2 is both the test and the application of funnel theory. If a model contains funnels then TAR2's runtimes won't be too slow. Further, for domains with narrow funnels, TAR2 can be used to find controllers that improve the mean and reduce the variance of the systems behavior.

In summary, for models containing narrow funnels, nondeterminism can be tamed. Hence, before rejecting nondeterminism, it may be wise to perform some experimentation and test for funnels (perhaps using TAR2).

5.3 Random SELECT and Options Analysis

In the previous section, we have argued that a nondeterministic search can run fast. We have also argued that the space of options generated via nondeterminism can be very simply constrained to manageable proportions. This section discusses another feature of nondeterminism; namely, it enables a style of discussion not available for purely deterministic systems.

Software design is a search through a space of options:

- That one requirement can be implemented many different ways and software engineers may try and assess several alternatives before finding a good choice.
- Different stakeholders have different views and different goals, and software engineers have to navigate through this space of competing requirements.
- That the same specification may have incomplete sections representing choices that have to be explored as the software matures.

How can nondeterminism help analysts explore this space of options? In its simplest form, the answer is this: you can't reason about options unless you can access them. Deterministic systems generate one output and only offer one option. Nondeterministic systems generate multiple alternative outputs, and we can learn much by debating the differences between those options. To illustrate this point, we will consider the extreme case where SELECT has been completely relaxed. In this case, no MATCHed action is culled and *every possible action* is allowed.

Figure 16 illustrates this technique. That figure represents some knowledge about economics. In that figure, the thin lines were created from the knowledge of one expert, Dr. Thin, and the thick lines were created from the knowledge of another expert, Dr. Thick. Also, squares denote *and-nodes*: i.e. conjunctions that follow some premise. Each variable of that figure has three states: *up*, *down* or *steady*. These values model the sign of the first derivative of these variables. Edges represent dependencies between variables, of which there are two types:

- Dr. Thin's *direct* connection between *flexibility* and *flexible work patterns* (denoted with plus signs) means Dr. Thin would explain *flexible work patterns* being *up* or *down* using *flexibility* being *up* or *down* respectively.

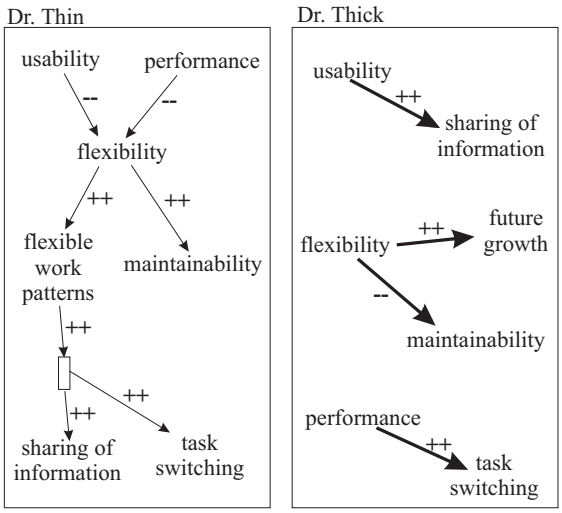

Figure 16: Viewpoints from two experts.

- Dr. Thick's *inverse* connection between *flexibility* and *maintainability* (denoted with minus signs) means that Dr. Thick would explain *maintainability* being *up* or *down* using *flexibility* being *down* or *up* respectively.

Note that our experts disagree on certain points:

- Dr. Thin holds the standard view that future change requests are best managed via a flexible system; i.e.

$$flexibility \overset{++}{\rightarrow} maintainability$$

- Dr. Thick takes the opposite view saying that when developers work in very flexible environments, their bizarre alterations confuse the maintenance team; i.e.

$$flexibility \overset{--}{\rightarrow} maintainability$$

To better support a dialogue between these two feuding experts, we first combine the two viewpoints into Figure 17. Next, we take some known input and output conditions and see what can be consistently inferred. In the case where the inputs are *IN1=* $\{$*performance=up, usability=down*$\}$, and the goals are *GOALS1=* $\{$*task switching=up, future growth=up, sharing of information=down*$\}$, there are five proofs `P` across figure 17 that can reach these goals, from those inputs:

- `P.1`: *performance=up, task switching=up*

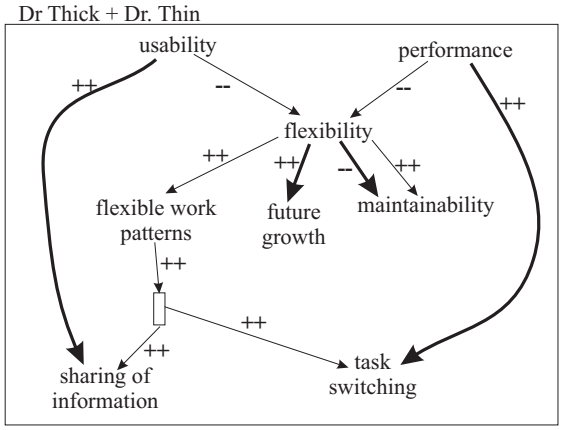

Figure 17: Union of the viewpoints of figure 16.

- P.2: *usability=down, flexibility=up, flexible work patterns=up, task switching=up*
- P.3: *usability=down, flexibility=up, future growth=up*
- P.4: *usability=down, sharing of information=down*
- P.5: *performance=up, flexibility=down, flexible work patterns=down, sharing of information=down*

Note that these proofs for $< IN1, GOALS1 >$ contain contradictory assumptions; e.g. *flexibility=up* in P.2 and *flexibility=down* in P.5. Classical logic would have us stop right here since, in the classical view, models that generate contradictions are incoherent. However, using a technique called *graph-based abductive inference* [85, 102], we can still make interesting inferences by sorting these proofs into maximal subsets that contain no contradictory assumptions. This process generates the *worlds of belief* shown in Figure 18.

In Figure 18, each world is one what-if scenario. Note that world #1 covers all our output goals while world #2 only covers two-thirds of our outputs. These worlds tell us that in the case of $< IN1, GOALS1 >$, the dispute over influences on *maintainability* don't matter since that part of the theory is not required to reach $GOALS1$. Also, recalling World #1, we see that neither expert can ignore the other. World #1 says that both Thick and Thin knowledge is required to reach all of $GOALS1$. With this knowledge, we can encourage the experts away from irrelevant disagreements (e.g. the influences on *maintainability*) and

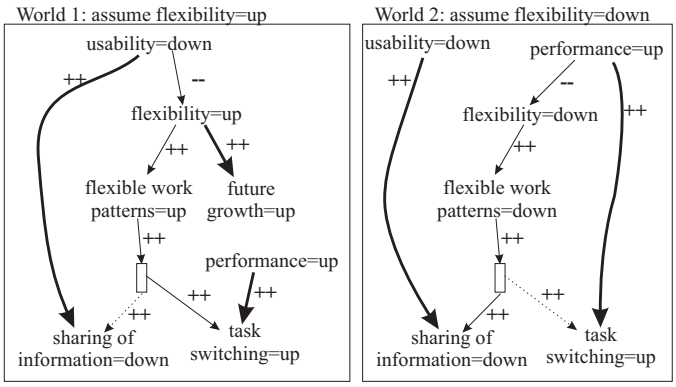

Figure 18: Worlds from figure 17. Ellipses denote the key assumptions that define a world. World #1 contains the proofs that do not contradict *flexibility=up*; i.e. `P.1, P.2, P.3, p.4`. World #2 contains the proofs that do not contradict *flexibility=down*; i.e. `P.1, P.4, P.5`. Dashed lines denote inferences that should be supported by the original model, but which are contradicted by the current set of inputs and goals.

towards relevant agreements (e.g. knowledge from both experts is required to reach $GOALS1$).

Note how this nondeterministic contradiction-tolerant multiple world approach has guided our doctors to a point of collaboration, despite conflicting viewpoints. Rather than focus on their obvious dispute (the effects of flexibility on maintenance), we can show our doctors how to work together using other portions of their viewpoints.

5.4 Nondeterminism and V&V

A common view is that nondeterminism is undesirable and must be avoided. We disagree. V&V analysts will see more nondeterministic systems since nondeterministic search can find solutions when complete search fails. Lest our V&V analyst panics at the prospect of testing a nondeterministic system, we pointed out that narrow funnels promise that the range of possible outputs of a system can be easily constrained. Further, the key settings to those funnel variables can be found very simply using methods like TAR2. Given that nondeterminism is fast, and can be constrained, then this lets a V&V analyst explore the space of options within a program using the multiple worlds reasoning discussed in the previous section.

6 AI Software can be Complex

AI software solves hard problems. AI has always been at the forefront of computer science research. Many hard tasks were first tackled and solved by AI researchers before they transitioned to standard practice. Those examples include time-sharing operating systems, automatic garbage collection, distributed processing, automatic programming, agent systems, reflective programming and object-oriented programming.

This tradition of AI leading the charge and solving the hard problems continues to this day. NASA's deep space missions require *autonomous satellites* that don't rely on ground control. For distant satellites or rovers such ground control would be impractically slow.

To test their autonomy technology, NASA flew the Remote Agent Experiment (RAX1) for two days using on-board AI control while that satellite was 60,000,000 miles (96,500,000 kilometers) from Earth. This trial was a step toward robotic explorers of the 21^{st} century that are less costly, more capable and more independent from ground control. RAX1 continually monitored the on-hard hardware and readjusted the mission tactical goals in line the available hardware and the mission's strategic goals. RAX1 has three components:

- *The planner* took general goals and determined detailed activities needed to achieve the goals. The RAX1 trial included asking the planner to achieve broad goals such as, "Find your position, and fire your ion engine whenever practical." If a hardware problem developed that prevents execution of the plan, the planner made a new plan, taking into account degraded capabilities.
- *The executive* interpreted the plans and added more detail to them, then issued commands to the flight software, coordinating the three parts of RAX1. Some commands turned the spacecraft to point in a different direction. Other commands asked the onboard camera to take pictures of asteroids and stars for navigation purposes.
- The LIVINGSTONE *diagnosis* module acted like a doctor, monitoring the spacecraft's health. If something went wrong, LIVINGTSTONE told the Executive there is a problem and then the executive consults the "doctor" for simple procedures that may quickly remedy the problem. For example, if a camera does not respond, a quick fix would be to turn the camera off and then on again. If this didn't work, the executive asked the planner for a new plan that still achieves mission goals.

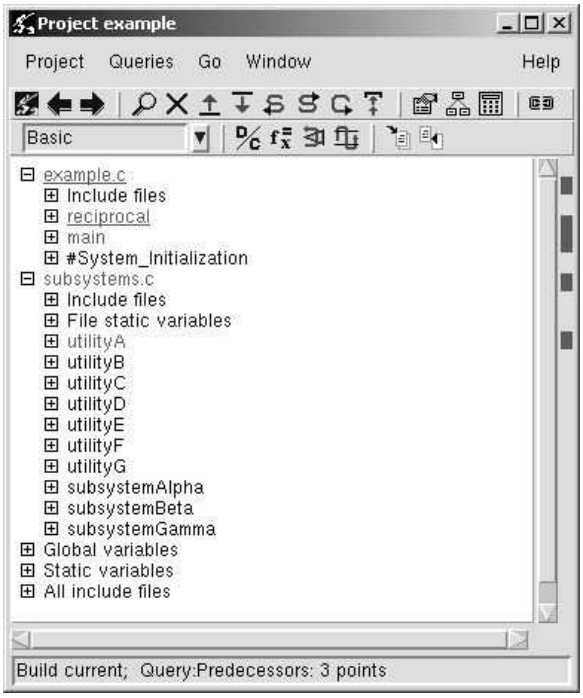

Figure 19: Slicing in Grammatech's CodeSurfer tool (see `http://www.grammatech.com/products/codesurfer/example.html`).

6.1 V&V of Complex Software

The *benefits* of complexity are clear. Some problems such as RAX1 are inherently complex and require an extension to existing technology. The *cost* of complexity is that complex systems are harder to understand and hence harder to test. Complex systems like RAX1 could hide intricate interactions which, if they happen during flight, could compromise the mission. Three tools for revealing hidden intricate interactions are discussed below: *static analysis*, *runtime verification*, and *model checking*.

6.1.1 Static Analysis

In *static analysis*, compiler technology is used to trace through the possible program pathways. One static analysis technique is *code slicing* which is used to reveal the code portions that are relevant to a particular set of variables or a func-

tion. For example, Grammatech's CODESURFER® tool can find the code *not* reachable from the `main` function[6]. Such code is dead code and represents either over-specification (i.e. analysts exploring too many special cases) or a code defects (e.g. the wrong items are present in a conditional). In Figure 19 the code sections reached by `main` are shown with colored marks in the right-hand-side of the display. Note that in this case, most of the code is *not* reachable from the `main` program!

There are several problems with static analysis:

- Manually exploring the reachable zone looking for problems is a tedious and time-consuming task. Ideally, we should be able to offer our analysis tool a succinct description of *features of interest* and then let it look for those features.
- Static analysis reveals the superset of all possible program sections that can be reached from some zone in a program. For any particular invocation of a program, only a small subset of those regions are actually reached. Hence conclusions about incorrect operations within the reachable zone are hard to make.
- The problem with reasoning about incorrectness in the reachable zone is even worse for programs supporting concurrency. Static analysis offers little assistance in checking that within the reachable zones there does *not* exist (e.g.) deadlocks.

6.1.2 Runtime Verification

Runtime verification tools such as the JAVA PATH FINDER [60] and the JAVA PATH EXPLORER [59], resolve some of the problems of static analysis. These tools work in tandem with a special version of the JAVA virtual machine to instrument JAVA programs to automatically find (e.g.) concurrency errors such as deadlock.

Runtime verification tools simplify the examination of the details of a running system. For example, during the May 1999 RAX1 mission, the satellite deadlocked in space, causing the ground crew to put the spacecraft on standby. The ground crew located the error using data from the spacecraft. The JAVA PATH FINDER used the same example to test runtime verification. The Lisp code of RAX1 was coded up in JAVA, a portion of which is shown in Figure 20. The

[6] `http://cayuga.grammatech.com/products/codesurfer/`

```
01 class Event {}
02   int count = 0;
03
04   public synchronized void wait_for_event()  {
05        try{wait();\}catch(InterruptedException e){};
06   }
07
08 public synchronized void signal_event() {
09     count = (count + 1) % 3;
10     notifyAll();
11 } }
12
13 class Planner extends Thread{
14 Event event1,event2;
15   int count = 0;
16
17   public void run(){
18     while(true){
19       if (count == event1.count)
20          event1.wait_for_event();
21       count = event1.count;
22      /* Generate plan */
23       event2.signal_event();
24 }  }  }
```

Figure 20: The RAX error. From [60].

resulting JAVA program had 40 threads, each with 10,000 states. When combined with the planner and the executive, this resulted in a search space too large for most automatic V&V tools: 10^{160} states in total. JAVA PATH FINDER found the defect that caused the deadlock: a race condition at line 19 (the exact problem is described in Figure 21). Note that a human code reader *might* have found the same error, but not in the 25 seconds (!!!) that it took JAVA PATH FINDER.

JAVA PATH EXPLORER is a generalization of Java PATH FINDER that automatically searches from the *features of interest* specified by V&V analysts. These features are expressed succinctly using a *temporal logic notation*. Temporal logic is classical logic augmented with temporal operators such as $\Box X$ (always X is true), $\Diamond X$ (eventually X is true), $\bigcirc X$ (X is true at the next time point), $X \,\mathcal{U}\, Y$: (X is true until Y is true). For example, consider the following constraint on an elevator:

> *Always, the elevator door never opens more than twice between the source floor and the destination floor.*

If P is the elevator doors opening and Q is the arrival at the source floor and R arrival at the destination floor, then the temporal logic expression of Figure 22

Figure 20 shows two JAVA classes. The `Event` class has a local counter and two synchronized methods, one for waiting on the event and one for signaling the event, releasing all threads having called wait for event. In order to catch events that occur while tasks are executing, each event has an associated event counter that is increased whenever the event is signaled. A task then only calls wait for event in case this counter has not changed, hence, there have been no new events since it was last restarted from a call of wait for event. The body of the `Planner`'s `run` method contains an infinite loop, where in each iteration a conditional call of wait for event is executed. The condition is that no new events have arrived, hence the event counter is unchanged.
When examining the RAX1 deadlock defect, the error trace found by JAVA PATH FINDER showed that if the RAX1 planner first evaluates the test`count == event1.count` at line 19 to true then, before the call of `event1.wait` for `event()`, the executive signals the event, the event counter is increased notifying all waiting threads. However, since there are no waiting threads yet, the planner now unconditionally waits and misses the signal.

Figure 21: The deadlock error of Figure 20. Text adapted from [60].

models our constraint of the elevator doors. Such a formulae looks intimidating but can be easy to write. Dwyer, Avrunin & Corbett [41, 42] have identified *temporal logic patterns* within the constraints seen in many real-world properties models. For each pattern, they have defined an expansion from the intuitive pseudo-English form of the pattern to a formal temporal logic formula. In this way, analysts are shielded from the complexity of formal logics. For example, writing the above query is a simple matter: just look up the "bounded existence" temporal logic pattern at `http://www.cis.ksu.edu/santos/spec-patterns/ltl.html` and extract the expression associated with *transitions to P-states occur at most 2 times between Q and R*.

Standard practice is to take the temporal formulae, negate it, then look for evidence that the negated formulae can be satisfied. A runtime trace that satisfies the negated formulae is a *counter-example* that offers a specific example of how a program can fail.

Standard practice is also to store the negated counter example as a as a directed

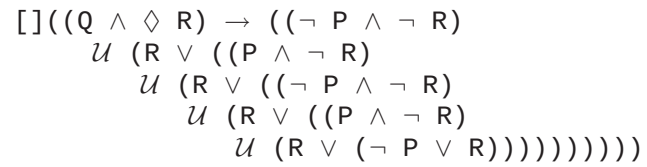

Figure 22: Always, the elevator door never opens more than twice between the source floor and the destination floor.

graph called a *Bucci automaton*. Within the automaton, the current state of the program is indicated by a single pointer to one of the nodes in that graph. New output from the program can be passed to the current state, and that state can check if any of its *out-edges* can use that output. If so, the current state jumps to the state at the other end of the usable out-edge. A few of the nodes in this graph are marked as "accept states" and if the current state ever arrives at an accept state, a counter example can be reported.

Note that without a tool like the PathEXPLORER, a V&V analyst might need to build a very large log tracing all the variable changes within a system, then explore that log manually using (e.g.) SQL. With PathEXPLORER, the disk memory required for that log can be reclaimed. Further, the exploration of the log with SQL is superfluous since the output assessment process has been automated into an incremental monitoring device (the Bucci automaton).

Runtime verification tools such as those discussed above are very new, not widely used, and not fully documented. Also, the current generation of runtime verification tools require instrumentation of some virtual machine. A more general and mature approach, with more support tools and documentation is *model checking*. Model checking makes no assumptions of an instrumented virtual machine.

6.1.3 Model Checking

Model checking is a method invented by Edward Clarke in the 1980s to test asynchronous protocols [28]. Model checking requires the construction of two models:

1. The *systems model* which is some abstract description of the essential characteristics of a program.

2. The *properties model* which is a set of constraints that should hold across the systems model. The properties model is often expressed as a temporal logic constraint.

A model checker searches all pathways of the systems model looking for ways to violate the properties model. If successful, the model checker returns the counter examples showing exactly how an invariant is violated. Such counter examples are useful in localizing and repairing faults.

The benefits of model checking come at a cost that is often prohibitively expensive. These costs including the *writing cost*, the *running cost* and the *re-writing cost*. Scarce and expensive PhD-level mathematical expertise may be required to *write* the formal temporal logic of the properties model. Once expressed, automatic methods such as the SPIN model checker [64] can search an formal representation of a program to find counter-examples to the supplied constraint. This search explores all the interactions within the program. In the worst case, the number of such interactions is exponential on the number of different assignments to variables in the system. Hence, the *running cost* of this query can be excessive. This large *running cost* often forces analysts to rewrite and shorten the formal models or formal constraints. Such a rewrite incurs the *rewrite cost* and may ignore some, potentially significant, system detail.

We will discuss below the declarative model-based nature of AI systems. Such declarative models are simpler to analyze than (e.g.) the quirkier constructs in the "C" language. Hence, it can be significantly easier to write the systems models. Automatic tools can usually be quickly written to convert the AI model into the format required for a model checker. However, the writing cost of the properties model remains, as does the running and rewrite cost.

Much research has tried to reduce these costs. For example:

- The temporal logic patterns work of Dwyer, Avrunin & Corbett [41, 42] discussed above.
- Simplified modelling environments have been developed such as SCR [35, 61,62] or simple influence diagrams [98] where users can express their models in simple and intuitive framework. Models written in these toolkits can be mapped into model checkers such as SPIN, thus combining easy model specification with exhaustive formal verification.
- The writing cost of the systems model can disappear to zero if that model can be auto-extracted from the source code. For example, the BANDERA system extracts systems models from JAVA source code via a consideration of the JAVA virtual machine [33].

These tools reduce the *writing cost* but don't necessary reduce the *running cost* or the *rewriting cost*. The *rewrite cost* is incurred only when the *running cost* is too high and the models or constraints must be abbreviated. There is no guarantee that model checking is tractable over the constraints and models built quickly using temporal logic patterns and tools like SCR. Restricted modelling languages *may* generate models simple enough to be explored with model checking-like approaches, but the restrictions on the language can be excessive. For example, checking temporal properties within simple influence diagrams can take merely linear time [98], but such a language can't model common constructs such sequences of actions or recursion. Hence, analysts may be forced back to using more general model checking languages. Unfortunately, despite decades of work (see Figure 23), the high *running cost* of such general model checking persists.

Despite these limitations, model checking is a widely used tool for AI systems at institutions like NASA. For a sample of such applications, see the proceedings of the recent AAAI 2001 Spring Symposium on Model-Based Validation of Intelligence[7].

7 Adaptive AI Systems

Not only can AI systems be complex, they can be *adaptive*; i.e. they can change their own internal logic at runtime. Adaption can complicate V&V. For example, suppose a test engineer takes two months to certify version 1.0 of a device. To her surprise and alarm, just before it is used, version 1.1 is released. Our test engineer, who is a dedicated professional, hurriedly studies version 1.1. One month into her work on version 1.1, version 1.2 is released. Our test engineer further discovers that version 1.3 is due, any week now. She contacts her management and complains "I can't certify a device that is constantly changing".

This is the dilemma of testing *adaptive systems*. AI systems containing a *machine learning* module can adjust themselves at runtime. In a way, adaptation is an extreme form of nondeterminism discussed in §5. A nondeterministic system can generate *different* outputs from the *same* input. An adaptive system that can generate different outputs after each adaptation.

Adaptive systems have the benefit that:

- The software can significantly tune its behavior. For example, a planning system might find a new method to generate better plans in less time.

[7] `http://ase.arc.nasa.gov/mvi/`

Each of the following techniques has proven to be useful in reducing the runtime cost of model checking or variations on model checking. However,

- *Abstraction or partial ordering*: use only the part of the space required for a particular counter-example. Implementations exploiting this technique can constrain how the space is traversed [56], or constructed in the first place [131].
- *Clustering*: divide the systems model into subsystems which can be reasoned about separately [27, 30].
- *Meta-knowledge*: study only succinct meta-knowledge of the space. One example used an eigenvector analysis of the long-term properties of the systems model under study [66].
- *Exploit symmetry*: find properties in some part of the systems model, then reuse those counter-examples if ever those parts are found elsewhere in the systems model [29].
- *Semantic minimization*: replace the space with some smaller, equivalent space. For example, the internal memory requirements of a model checker can be reduced if the state space is compacted using binary decision diagrams [17]. Another system of interest is the BANDERA system [33] that reduces both the systems modelling cost and the execution cost by automatically extracting (slicing) the minimum portions of a JAVA program's bytecodes which are relevant to particular properties models.

While the above techniques have all been useful in their test domains, they may not be universally applicable. Certain optimizations, such as those of [66], require expensive pre-processing. Also, these methods may rely on certain combinatorial features of the system being studied. Exploiting symmetry is only useful if the system under study is highly symmetric. Clustering generally fails for tightly connected models. Hence, in the general case, it seems that only small models can be assessed using model checking techniques.

Figure 23: Experiments in reducing the state-space explosion problem in model checking.

- The software can adapt to new situations not seen during design time. For example, a robot that moves an object around might learn to push objects with its torso if its arms malfunction.

However, the cost of adaptive systems is that the adaption might render obsolete any pre-adaption certification.

This section focuses on V&V of adaptive systems[8]. In essence, the argument will be that V&V of adaptive systems should *not* try to certify the latest product of the latest adaption. Instead, the V&V of adaptive systems must *certify the adaptive process*. Our proposed certification process will be to check if the test engineer can trust that the adaption process will produce adequate new models. That checking process will be summarized below, after an introduction to learning algorithms and some case studies using different learners.

7.1 Introduction to Learning

Two properties of AI system make them particular suitable to adaption. Firstly, *knowledge-level* AI systems can reflect over their goals at runtime. This means that if a knowledge-level system adapts itself, it can use its knowledge of system goals to automatically evaluate the adaptation; e.g. by checking if more goals can be reached using fewer resources.

Secondly, *model-based* AI systems often use a simple uniform declarative semantics for their models. This simplifies the adaptation problem since only a small number of adaptive operators are required to modify a system. For example, consider a logic program of the form

$$Goal\ if\ PreCondition1\ and PreCondition2\ and \ldots$$

Two operators are enough to extensively adapt this program: *specialization* and *generalization*:

- Undesired behaviors can be removed by *specializing* the preconditions; i.e. increasing the number of preconditions.
- Desired behavior which was not achieved can be reached via *generalizing* a pre-condition; i.e. removing one or more of the preconditions.

[8]For a broader review of the general field of machine learning methods, see [40, 82, 104, 109, 153]

Operator	Preconditions	Postconditions
PUSH(obj,loc)	At(robot,obj) AND Clear(obj) AND ArmEmpty	At(obj,loc) AND At(robot,loc)
CARRY(obj,loc)	At(robot,obj)	At(obj,loc) AND At(robot,loc)
WALK(loc)	None	At(robot,loc)
PICKUP(obj)	At(robot,obj)	Holding(obj)
PUTDOWN(obj)	Holding(obj)	NOT Holding(obj)
PLACE(obj1,obj2)	At(robot,obj2) AND Holding(obj2)	On(obj1,obj2)

Figure 24: Operators available to our adaptive robot.

	Operators used for each macro					
macro	push	carry	walk	pickup	putdown	place
Move object	★	★				
Move robot			★			
Clear object				★		
Object on object						★
Get arm empty					★	★
Hold an object				★		

Figure 25: Planning macros.

Such a logic-based adaptation framework dates back to at least Shapiro [135].

The following example illustrates adaption via specialization. Consider an AI planning robot manipulating objects in a room containing a table holding up two objects[9]. For argument's sake, we will say that we want to move the table and the objects to another room. Before we can move the table we need to move the two objects. Our robot can access a set of *operators* that can change state in a room. Our robot operates via MATCH-SELECT-ACT cycle which reflects over the operators of Figure 24. Note that each operator has *preconditions* describing when that operator is applicable and *postconditions* that describe the effects of the operator. At runtime, our AI system must build a *sequence* of operators which take us from some initial *start state* (e.g. objects on table) to some *goal state* (e.g. objects on table in another room). At each step of the sequence, the *preconditions* must be satisfied to enable the next step.

These operators might be too low-level to explain to human programmers, so we might define a macro language like Figure 25 that maps higher-level constructs into the operators. Note that there are two ways to *move* and object: we can *push*

[9]This example is adapted from [71]

or *carry* it. Suppose we originally designed the planning robot for houses where all objects have the same weight. Suppose further the environment changes and now the robot faces an environment where objects have very different weights.

After a few days in the new environment, our robot notices that that *carrying* heavy objects results in more wear and tear that *pushing* heavy objects. It would sensible to let our robot automatically *specialize* its operators with the new preconditions shown here in italic:

Operator	New preconditions
PUSH(obj,loc)=	At(robot,obj) AND Clear(obj) AND ArmEmpty AND NOT *light(obj)*
CARRY(obj,loc)=	At(robot,obj) AND *light(obj)*

The above example demonstrated small-scale adaptation where only a small part of a current model was changed. Such small scale adaptations are typical in *deductive learning* schemes such as EBG [108, 149] and "chunking" [73]:

- EBG is short for *explanation-based generalization* where program traces are studied for short-cuts. For example, if a program always calls six operators in the order O_1, O_2,...O_6, then EBG might replace these with one new operator O_7 with the preconditions of O_1 and the postconditions of O_6.
- "Chunking" is a rule-based programming technique that seeks patterns of rules commonly fired in a certain sequence. The rules are then "chunked" into a super rule that leaps the inference from the first rule to the last rule in the sequence.

The opposite of deductive learning is *inductive learning*. Inductive learners such as genetic algorithms [37,57], neural nets [63], and decision tree learners [31, 124] create a summary model from input training examples. Inductive learning schemes are typically characterized by large-scale adaptation of a model since these learners ignore pre-existing knowledge and build an entirely new model from scratch.

For example, the decision tree of Figure 26 was automatically built by the C4.5 inductive decision tree learner [123] from a log of golf playing behavior. We see that we do not play golf on high-wind days when it might rain. This learner has its own internal measures of the information content of each attribute in the log; i.e. `outlook, temp, humidity` and `windy`. If an attribute does not satisfy these internal measures it is discarded; e.g. `temp` does not appear in the generated decision tree. The problem with (e.g.) ignoring an attribute is that the generated model may disagree with established wisdom in a field; e.g. that heat is an important factor in determining golf playing behavior.

INPUT:

```
#outlook,    temp, humidity, windy,   class
#-------     ----  --------  -----    -----
sunny,       85,   85,       false,   dont_play
sunny,       80,   90,       true,    dont_play
overcast,    83,   88,       false,   play
rain,        70,   96,       false,   play
rain,        68,   80,       false,   play
rain,        65,   70,       true,    dont_play
overcast,    64,   65,       true,    play
sunny,       72,   95,       false,   dont_play
sunny,       69,   70,       false,   play
rain,        75,   80,       false,   play
sunny,       75,   70,       true,    play
overcast,    72,   90,       true,    play
overcast,    81,   75,       false,   play
rain,        71,   96,       true,    dont_play
```

OUTPUT (estimated error=38.5%):

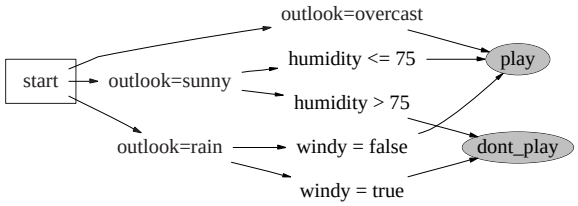

Figure 26: Decision-tree learning. Classified examples (above) generate the decision tree (below).

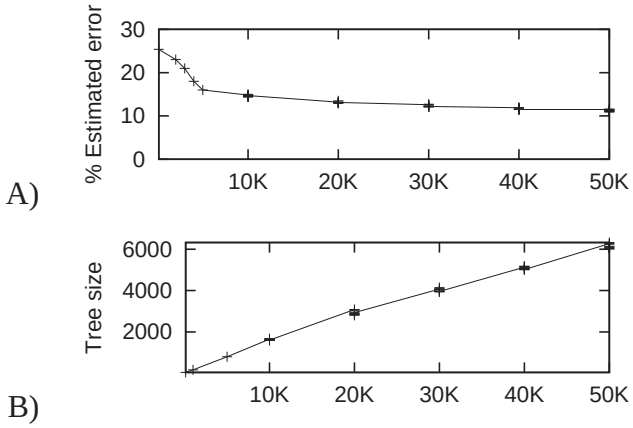

Figure 27: Error rates (top) and number of nodes in the learnt decision tree (bottom) seen in one decision tree learning study [101].

Newer styles of inductive learning accept as input some background knowledge, which the learner can extensively modify based on the available training data. For example, *bayesian tuning* [49] and *inductive logic programming* [12] let users initialize a model with a simple causal network or intricate horn clause expressions.

7.2 Adaption and V&V

7.2.1 Do you Have Enough Data?

One way to assess an adaptive system is to ask if the system is seeing enough data to make reasonable adaptations. Inductive techniques are typically very data hungry: the efficacy of the inductively learnt model can never be better than the quality of the input examples. Hence, the more examples, the better the learnt model.

We have seen the results of learning on too little data above. C4.5 estimated that the decision tree shown in Figure 26 will have a 38.5% error rate of future cases. We should expect such a large errors when learning from only 15 examples. In general, C4.5 needs hundreds to thousands of examples before it can produce trees with low errors.

When we build models from data, we should increase the sample size till the estimated error drops to an acceptable level. For example, Figure 27.A shows

	Changes in the learnt tree's...	
domain	... size	...classification error
demon	0.97	0.51
wave	1.91	0.95
diff	1.46	0.69
othello	1.68	0.8
heart	1.61	0.65
sleep	1.73	0.91
hyper	1.74	0.83
hypo	1.45	0.85
binding	1.51	0.82
replace	1.38	0.8
euthy	1.33	0.61
mean	1.52	0.77

Figure 28: Impact of learning a decision tree from N or $2N$ examples. From [22].

the error rate of decision trees generated using C4.5 after learning from 10,000 examples, then 20,000 examples, and all the way up to 50,000 examples. Such curves can be used to determine when enough data is enough. Figure 27 suggests that in that domain, the benefits of using more than 10,000 examples was marginal.

On the other hand, Catlett cautions that there is always a case for using more examples [22]. In his study, he gave a machine learner N or 2N examples from different domains. He found that you should never disable adaptation since, in nearly all cases, more experience lead to much better and bigger models with fewer errors (see Figure 28).

While we don't dispute Catlett's findings, pragmatic considerations often impose a restriction the size of the data given to a machine learner:

- If the example set grows very large (i.e. thousands of examples or more), then considerable manual knowledge engineering effort may be required to prepare the data for the inductive learner [152].
- In many domains, the available data sets can be very small. For example, in software engineering effort estimation, data set sizes are usually dozens, not hundreds, of examples [20, 140].
- Sometimes, large data sets generate awkwardly large models. Figure 27.B shows what happened in one case study when decision trees were built from tens of thousands of examples. While an automatic program can process a decision tree with (e.g.) 4000 nodes, such a tree is too large for a human to read. We will return to this issue of *readability* later in this article

7.2.2 External Validity

Another issue with the size of a data set is how much is used for *training* and how much is used for *testing*. One goal of machine learning is to generate models that have some useful future validity. To test if that goal has been meet, the generated model should be tested on data not used to built the model. Failing to do so can result in a excessive over-estimate of the learnt model. For example, Srinivasan and Fisher report an 0.82 correlation (R^2) between the predictions generated by their learnt decision tree and the actual software development effort seen in their training set [140]. However, when that data was applied to data from another project, that correlation fell to under 0.25. The conclusion from their work is that a learnt model that works fine in one domain may not apply to another.

One standard method for testing how widely we might apply a learnt model is *N-way cross validation*:

- The training set is divided into N buckets. Often, N=10.
- For each bucket in turn, a treatment is learned on the other $N - 1$ buckets then tested on the bucket put aside.
- The prediction for the error rate of the learnt model is the *average* of the error rate seen during the N-way study.

In essence, N-way cross validation is orchestrating experiments in which the learnt model is tested ten times against data not seen during training.

A learnt model can pass a N-way cross validation test, but sill not be externally valid. For example, Figure 29 and Figure 30 show decision trees learnt in different domains that predict fault software modules. While the *task* in each domain is different, the *available attributes* in each domain was different. Hence, the trees are of different shapes and use different attributes. The lesson here is the same as above: analysts should take care when importing models learnt in other domains.

7.3 Case Study

The detailed case study of Burgess and Lefley [20] illustrates many other issues in evaluating adaptive systems. In their work, they applied different learners to the same data set and assessed the merits of each. Their test domain was software cost estimation: given particulars of different projects, they asked their learners to produce an estimate of project development time. The learners in that study were:

NN: A neural network algorithm that adjusts weights between a network of simple nodes which "fire" if the sum of the input weights pass some threshold.

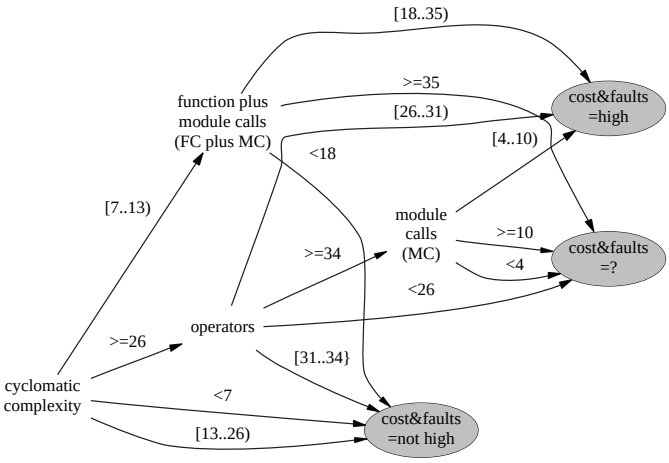

Figure 29: Predicting modules with high cost modules and many faults. Data from 16 NASA ground support software for unmanned spacecraft control [146]. These systems were of size 3,000 to 112,000 lines of FORTRAN and contained 4,700 modules.

GP: A genetic programm algorithm which tried many generations of combinations and mutations of a mathematical equation that computes project development time.

LSR: A linear standard regression algorithm. Normally, regression is not usually termed a "machine learner" (exception: [14]) but Burgess and Lefley included it in their study for comparison purposes.

Burgess and Lefley offered several assessment criteria for their studied learners: *accuracy, readability, consistent convergence* and *ease of configuration*. These criteria, and how they were applied in the Burgess and Lefley study, are discussed below.

7.3.1 Accuracy of the Learnt Model

Burgess and Lefley collected the statistics shown in Figure 31 by averaging values seen 10 runs of the system. In that figure, correlation was was the standard R^2

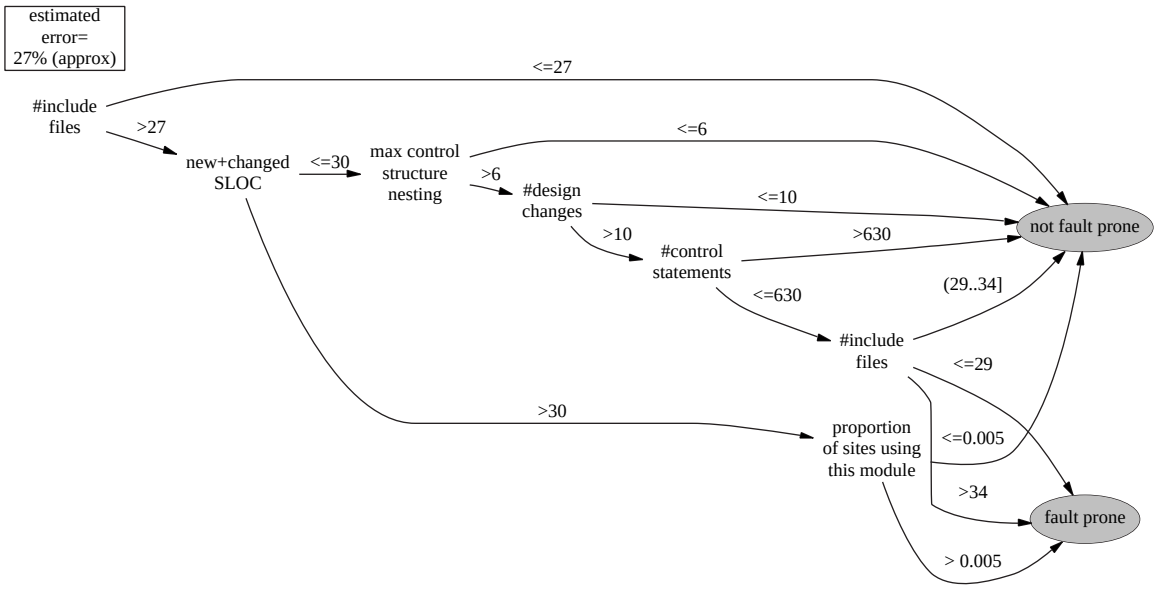

Figure 30: Predicting fault-prone modules [72]. Learnt by the CART tree learner [14] from data collected from a telecommunications system with > 10 million lines of code containing a few thousand modules (exact details are proprietary confidential). Estimated error comes from 10-way cross-validation studies.

statistic and $Pred(25)$ was the percentage of predictions that fell within 25% of the actual value.

Figure 31 shows that genetic algorithms produced the *best* fit to the data $R^2 = 0.75$ while were *worst* at coming near the actual figures ($Pred(25) = 23.6\%$). Such a strange result may result when the target model has discontinuities and does not fit a standard regression model.

Note that *how* a learnt model is assessed can crucially effect the evaluation. For example, if we desired to generate a detailed expression for the connection of project factors and development time, the equation learnt by the GP might be

Metric	LSR	NN	GP
Correlation	0.56	0.64	0.75
Pred(25)%	55.6	55.6	23.6

Figure 31: Accuracy assessment data on models learnt by Burgess and Lefley [20].

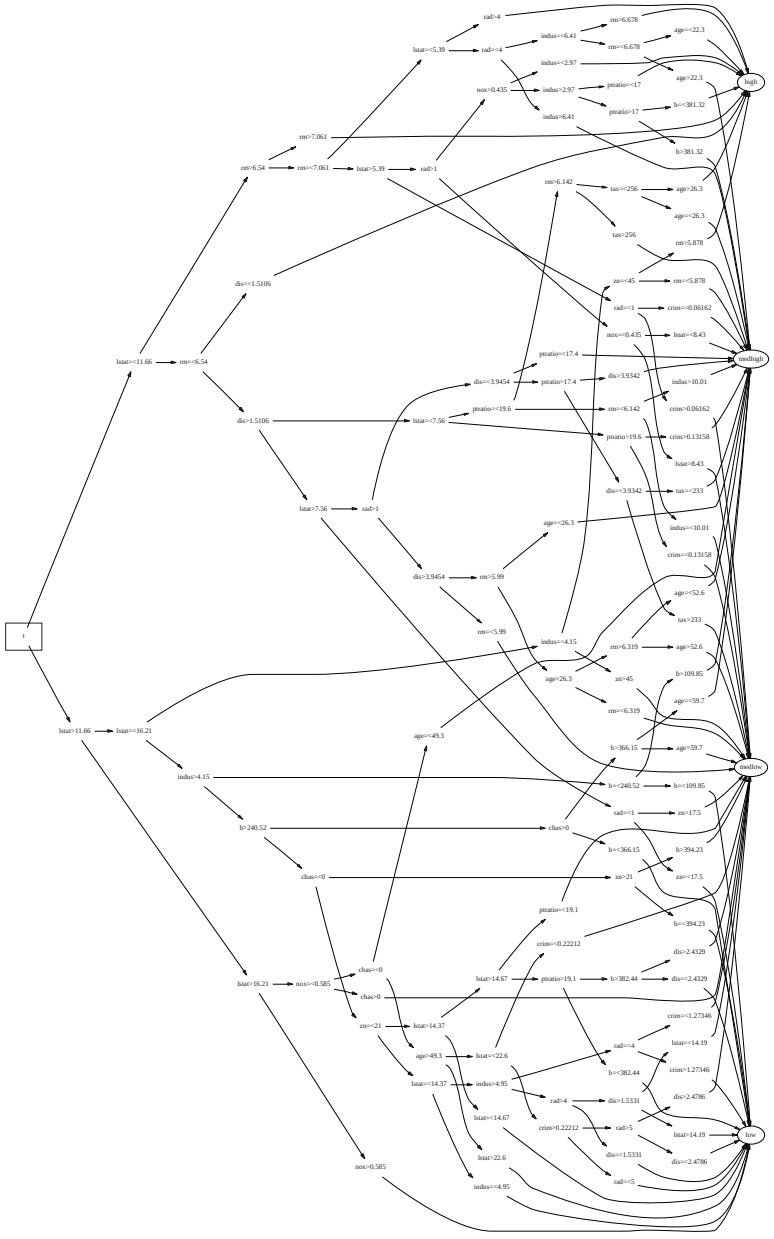

Figure 32: A learnt decision tree. Classes (right-hand-side), top-to-bottom, are "high", "medhigh", "medlow", and "low" This indicates median value of owner-occupied homes in $1000's. Decision tree learnt from the 506 cases in HOUSING example set from the UC Irvine repository (`http://www.ics.uci.edu/~mlearn/`).

the best. However, if we sought an oracle that offered the best approximation, the LSR or NN might be best. Analysts should therefore develop and audit a range of assessment criteria with their target user community before applying some criteria to a learnt model.

7.3.2 Learning and Consistent Convergence

Suppose we are monitoring the progress of a learner and we see that the error rate of the current version of a learnt model is converging gracefully towards some plateau. Observing such a convergence, we might have some confidence that the learner's conclusions are stable and can be trusted.

Not all learners exhibit graceful convergence:

- Learners that use random search can leap around within the learning process. For example, the GP methods used in the Burgess and Lefley study randomly *mutate* a small portion of each generation of their models.
- The gradient descent method used by NN can "shoot passed" the correct model if the steps of the gradient descent are too enthusiastic.

Burgess and Lefley report that in ten runs of their NN and GP systems, the former usually converged to the same conclusion while the GP could generate different answers with each run.

7.3.3 Learning and Ease of Configuration

Learners are programs and programs often require configuration settings. A complaint against NN learning is that the configuration can be complicated and may include issues such as:

- What is the best size of steps in the gradient descent?
- How many layers in the network?
- How many nodes in each node?
- What is shape of the threshold function used in each node?
- etc.

Burgess and Lefley commented that their GP learner was easier to configure than their NN learner, while the regression system (LSR) was easiest of all to configure.

7.3.4 Learning and Readability

Different learners report their learnt models in different ways. Burgess and Lefley's GP system outputted an equation that predicted software development effort from domain attributes. If is fundamentally very difficult to extract such a high-level description of a NN's knowledge since domain concepts are spread out across the network on the edge weights. Hence, Burgess and Lefley comment that the GP output was more readable than NN.

7.4 Readability and Treatment Learners

Learners that optimize for readability can take a very different form to traditional learners that optimize for expressiveness in the output model. Menzies et.al. have found that (e.g.) decision tree learners can generate overly-complex models. In their observations, business users often request summaries of even small-sized decision trees. For example, when faced with a C4.5 decision tree such as Figure 32, business users may look puzzled and ask for an executive summary of the essential features of the learnt tree. The TAR2 *treatment learner* [47, 65, 83, 92–95] generates such summaries and seeks the *smallest* number of attribute ranges that *most* select for preferred classes and *least* select for undesired classes. As we shall see, this output may be very brief indeed.

For example, the decision tree of Figure 32 was learnt from 506 examples of houses in Boston. Each house was rated into one of four value ratings. The distribution of these ratings are shown on the left of Figure 33: 21% of the houses were in the *lowest* class; 29% were in the *highest* class; and the other houses fell in-between. Treatment learners seek ways to change the ratios of classes. Technically, a *treatment* is a constraint which, if applied to the training set, selects a subset of the training examples. A treatment is *best* if it *most improves* the distribution of classes seen in the selected examples. For example, the middle histogram of Figure 33 shows the ratio of housing types after applying the best treatment (Equation 1) found by TAR2 from the housing dataset:

$$\begin{aligned} &(6.7 \leq RM < 9.8) \wedge \\ &(12.6 \leq PTRATION < 15.9) \end{aligned} \tag{1}$$

This best treatment advises us to favor houses with 7 to 9 rooms in suburbs with a parent-teacher ratio in the local schools of 12.6 to 15.9. Note that under that best treatment, none of the 38 selected houses fall into the lowest class while 97% fall into the highest class.

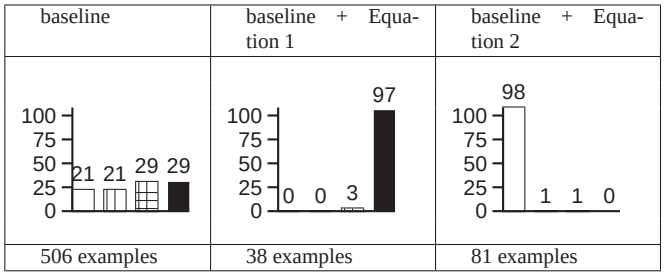

Figure 33: Treatments learnt by TAR2 using the data used in Figure 32. That dataset had the class distribution shown left-hand-side. Actions that most increase/decrease housing values are shown in the middle/right columns (respectively).

By reversing the internal scoring of the classes, a treatment learner can also find the *worst* treatment; i.e. the constraint that *most degrades* the distribution of classes seen in the selected examples. For example, the right histogram of Figure 33 shows the ratio of housing types after applying the worst treatment (Equation 2) found by TAR2 from the housing dataset:

$$\begin{aligned}&(0.6 \leq NOX < 1.9) \wedge \\ &(17.16 \leq LSTAT < 39.0)\end{aligned} \tag{2}$$

This worst treatment advises us to avoid houses in suburbs with an atmospheric nitrous oxide levels of 0.6 to 1.9 parts-per-10-million and suburbs with a living standard of 17.6 to 39.0. Note that under that worst treatment, none of the 81 selected houses fall into the highest class while 98% fall into the lowest class.

TAR2 scales well. The technique was developed in order to let humans browse the trees learnt in Figure 27.B. TAR2 could reduce those trees to a few histograms like Figure 33 and show them to users in under one page [101].

The central claim of TAR2 is that treatments are easier to read and understand than decision trees. While decision trees offer elaborate and detailed descriptions of different classes, treatments offer succinct summaries of the best and worse situation within a training set. The reader may care to pause and consider whether they prefer to read the complex decision tree of Figure 32 or the tiny treatments shown above as Equation 1 and Equation 2.

7.5 Learning to Trust Adaption

The goal of this section on V&V and adaptive systems was to outline the circumstances when a test engineer might trust that an adaptive system was trustable. There are several questions a V&V analyst can ask to ensure that trust:

- What is decay rate of the errors seen in the learnt model viewed as a function of number of examples (e.g. Figure 27.A)? If error rate is high and not decaying, then the learner is not seeing enough examples to learn effectively.
- What is the external validity of the learnt model? If the N-way cross-validation results are not promising, then the learnt model may fail in future cases.
- How might different users view the results of the learner? This will influence the accuracy criteria used when measuring the model.
- How well does learner converge? If the learning curve is not graceful, then the learner's results might be unpredictable.
- How easy is it to configure the learner?
- Is the output of the learner readable enough for the target audience? If the learnt model is to be auto-converted to (e.g.) "C" code, then the model can be very complicated indeed. However, if a human is to read the learnt model, then perhaps techniques like treatment learning should be employed.

In summary, a test engineer might decide to trust an adaptive system when:

- It can be demonstrated they they are getting enough examples to learn adequate models;
- The N-way cross validation results are good;
- The learner is being assessed according to a criteria endorsed by the user;
- The learner converges gracefully to its conclusions;
- The user can read and understand the learnt model.

8 AI is Software

Having stressed the unique features of AI systems, it is wise to recall that AI software is still software and much is known about V&V of software. Any software project that violates known norms in the software development process is at risk and requires elaborate V&V.

Previous sections in this article focused on the special features of AI. This section discusses the use of some norms and their implications on the V&V process. Note that there is *nothing special* in this section about AI systems and that is the point: many established software assessment practices can and should be applied to AI systems.

There are several such oracles for checking a project against established norms. For example, Steve McConnell organized and documented a debate on the best influences on software engineering [79]. From that dialogue, we can see that a project is more likely to have problems if it ignores the following accepted best practices:

Reviews and inspections: Unstructured readers seeking all types of bugs in source code can get distracted and confused by the open-ended nature of their quest. On the other hand, if different readers focus on different classes of bugs, then such *structured reading methods* can result in very high defect detection rates (defect detection capability is the percentage of defects that are latent in the artifact that is being inspected that are detected) [5, 46].

Information hiding: Good designs don't show everyone every detail. Rather, they only present information on a *need to know basis* [117]. Whereas other design approaches focus on notations for expressing design ideas, information hiding provides insight into how to come up with good design ideas in the first place.

Incremental development: Often users can't see that a project is running off the rails unless they can see it running. Software projects that never generate interim deliverables run the risk of delivering the wrong product. Any number of standard methods exist for encouraging incremental development including incremental compilers; the spiral model of software development [9]; and use-case driven development [67]. More recently, other methods have began to be emphasized such as daily builds, open source methods [125], agile methods [11] in general, or extreme programming [6] in particular.

User involvement: Users find problems and the budget to solve those problems. Developers spend that budget to build software solutions to those problems. Software development that ignores users can ignore the real problems. There are many ways to ensure user involvement such as user-interface prototyping, co-locating users with the developers and use-cases based development.

	rely= very low	rely= low	rely= nominal	rely= high	rely= very high
sced= very low	0	0	0	1	2
sced= low	0	0	0	0	1
sced= nominal	0	0	0	0	0
sced= high	0	0	0	0	0
sced= very high	0	0	0	0	0

Figure 34: A Madachy table. From [77]. This table reads as follows. In the exceptional case of high reliability systems and very tight schedule pressure (i.e. *sced=low or very low* and *rely= high or very high*), add some increments to the built-in parameters (increments shown top-right). Otherwise, in the non-exceptional case, add nothing to the built-in parameters.

Barry Boehm's COCOMO project has generate several other oracles for checking a project against those norms. Two such oracles are the COCOMO multipliers and the Madachy risk model:

The COCOMO multipliers: For example, based on regression models generated from 161 software projects, Boehm's has identified a set of *multipliers* that capture the impact of some factors on software development; e.g. losing highly productive programmers can increase development effort by a factor of up to 3.53. These multipliers can be used to test if a currently successful software team will have problems in some new project [10, 150]. For example, say the budget for the new project is be based on prior successful experience. This budget is then *multiplied* by the Boehm factors representing the changes from the old project to the new project. For example, if the most productive team members are not available for the new project, the new budget could be 3.53 times as large as the initial budget prediction. More V&V effort is required for projects that fall below the multiplied budget.

The Madachy Model: The Madachy model of software project management issues [77]. This model outputs a numeric index representing how concerned an experienced analyst might be about a particular software project. The

model contains 94 tables that implement a context-dependent modification to internal COCOMO parameters. Figure 34 operationalizes one of the 94 Madachy heuristics: i.e. software that must be highly reliable should not be developed under excessive schedule pressure[10]. The model can be accessed on-line at `http://sunset.usc.edu/research/COCOMOII/expert_cocomo/expert_cocomo2000.html`.

The problem with using established norms from other projects is that they may not apply to the current project [99]. However, that should not stop developers seeking such norms. As well as studying the above oracles, developers should keep their own metrics. Such incrementally acquired project-level metrics can reduce the cost of V&V. For example, Menzies & diStefeno found that in one project, if V&V analysts restricted their code reviews to modules of more than 18 lines of code, then they would be 4.5 times more likely to be reading code containing defects [90].

9 Summary and Challenges

Now that this review is nearly finished, what has been achieved? What have we learnt about the current practice and future challenges for V&V for AI?

The premise of this review was that AI systems may contain certain special features as:

1. A declarative model;
2. Very early life cycle executables;
3. Knowledge-level content;
4. Nondeterminism;
5. Great complexity
6. Adaptivity.

This section discusses the costs, benefits, and pressing challenges for each feature.

9.1 Benefits and Costs of Declarative Models

Declarative model-based software reduces the distance between conceptualization and implementation. Such declarative representations are simple to process and a

[10]While Madachy calls his work a "risk" model, his definition is so different to the standard definition of $risk = severity * frequency$ that we rename it to a "worries" model.

wide range of time-consuming tasks can be automated. These tasks include code generation and, as seen in the Feather and Smith work of Figure 2, certain classes of V&V queries.

The use of such declarative model-based methods is increasing. For example, the North American commercial aviation industry and the European VLSI community make extensive use of these techniques. The recent sudden growth in model-based methods, languages, tools and techniques has a downside. Since there are no widely-accepted standards for this technology, users of model-based systems often must make a considerable investment in learning the quirks of their particular model-based system. This effort may be required every time staff changes and new staff must learn the local model-based method.

Hence, our first challenge is:

Challenge 1 *Either train developers to jump between modelling frameworks or developing standards for model-based development.*

In this author's opinion, it is too early to standardize model-based development and we should accept at least another decade of a tower of model-based Babel.

9.2 Benefits and Costs of Very Early Life Cycle Executables

AI offers the SE community general and powerful interpreters of declarative models for a wide range of tasks. The benefit of this approach is that it is trivial to generate a running version of a model. The cost here is that this ease-of-implementation can lead to sloppy development practices. What is hard to convince developers is that if it is easy to run, it can also be easy to run into trouble. Practitioners using model interpreters often stop generating standard software documentation (e.g. requirement documents) since the high-level model *IS* the executable system.

The open issue in this area is the simplification of not only the construction of a running system, but its assessment as well (perhaps using the Feather & Smith framework of Figure 2). Fundamentally, this means increasing the modelling effort of a system; i.e.

Challenge 2 *Not only must developers supply enough information for execution, they must also describe the constraints and assessment criteria for their system.*

If that challenge is meet then this enables the next challenge:

Challenge 3 *Extend standard model-based methods such that executables and test-harnesses are automatically generated from declarative models.*

9.3 Benefits and Costs of Knowledge-level Content

The knowledge-level stores and reflects over the goals of a program. Access to these goals at V&V time permits the automatic development of assessment oracles for a system.

The benefits of the knowledge-level come at a cost. Knowledge-level modelling is still an arcane art. Most developers know how to model the core data structures of their implementation. However, most developers are not trained in how to add goal and goal processing knowledge to their system. Therefore:

Challenge 4 *Train developers in goal-oriented thinking for their modelling.*

In this author's opinion, such a change would represent a major paradigm change. Hence:

- It may be too late to do this for the *current generation* of developers.
- Educators should should focus on goal-based methods for the *next generation* of developers.

9.4 Benefits and Costs of Nondeterminism

The costs and benefits of nondeterminism are simple to state. If software is free to widely explore options at runtime, then it is free to find options that might have escaped the notice of the developers. However, such freedom means that it is also free to run to some undesirable state.

While nondeterminism is normally depreciated, the view of this review was that nondeterminism has significant benefits. Nondeterminism is a source of options which developers can tap to explore options or repair to the current system. Empirically, it is known that nondeterministic methods can solve some problems orders-of-magnitude larger than complete methods. Hence, the challenge of nondeterminism is:

Challenge 5 *Understand the range of possible behaviors that arise from a nondeterministic system.*

The work above on TAR2 is the author's response to this challenge. Recalling §5.2, if we survey the space of options generated by a nondeterministic device, we can often find ways to collapse that space towards a more desirable mode of operation.

9.5 Benefits and Costs of Great Complexity

Alan Kay once said *simple things should be simple, complex things should be possible*. Complex tasks face us everyday and complex tasks require complex solutions. The benefits of accepting such complexity is that we can solve harder problems. The cost is obvious: complex software is harder to understand and evaluate.

Nevertheless, this article has explored several promising methods of evaluating complex systems and the open challenge in this area is to:

Challenge 6 *Improve our automatic program understanding methods such as static analysis, runtime verification, and model checking.*

9.6 Benefits and Costs of Adaptivity

Adaptive systems can change themselves via experience. This has the benefit that programmers have to code less. However, V&V analysts know the cost of this approach: as the system changes, then any previous assessment of the system may become out-dated.

The challenges here are two-fold:

Challenge 7 *MICRO-CHANGES: in systems where the adaptive portion is very small, then the challenge is to understand the space of possible future behaviors that could arose from adaption.*

Note that the answer to this challenge may very well be the same as **Challenge 5**. That is, if we treat the outputs from the adaptive portion as nondeterministic events, then we may be able to understand the range of possible behaviors that arise from future adaptations.

The other challenge rejects the MICRO-CHANGE assumption:

Challenge 8 *MACRO-CHANGES: in systems where the adaptive portion can make changes all over a system, then the challenge is to validated the adaptive process and not the results of adaption.*

That is, if we can learn to trust the changes made by a machine learner, system, we would have less need to constantly re-evaluated a adapting system.

Acknowledgements

Dale Pace and Steve Stevenson offered excellent guidance on the content of this piece. Lisa Montgomery provided invaluable editorial advice.

This research was conducted at West Virginia University under NASA contract NCC2-0979. The work was sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the NASA IV&V Facility.

Disclaimer

Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government.

References

[1] J. S. Aikins. Prototypical knowledge for expert systems. *Artificial Intelligence*, 20(2):163–210, 1983.

[2] J. Angele, D. Fensel, and R. Studer. Domain and task modelling in mike. In A. S. et.al., editor, *Domain Knowledge for Interactive System Design*. Chapman & Hall, 1996.

[3] M. Ayel. Protocols for consistency checking in expert system knowledge bases. In *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, pages 220–225, 1988.

[4] J. Bachant and J. McDermott. R1 Revisited: Four Years in the Trenches. *AI Magazine*, pages 21–32, Fall 1984.

[5] V. Basili, F. McGarry, R. Pajerski, and M. Zelkowitz. Lessons learned from 25 years of process improvement: The rise and fall of the nasa software engineering laboratory. In *Proceedings of the 24th International Conference on Software Engineering (ICSE) 2002, Orlando, Florida*, 2002.

[6] K. Beck. *Extreme Programming Xplained*. Addison-Wesley, 1999.

[7] R. Benjamins. Problem-solving methods for diagnosis and their role in knowledge acquisition. *International Journal of Expert Systems: Research & Applications*, 8(2):93–120, 1995.

[8] D. Bobrow. If prolog is the answer, what is the question? or what it takes to support ai programming paradigms. *IEEE Transactions on Software Engineering*, 11(11):1401–1408, November 1985.

[9] B. Boehm. A spiral model of software development and enhancement. *Software Engineering Notes*, 11(4):22, 1986.

[10] B. Boehm. Safe and simple software cost analysis. *IEEE Software*, pages 14–17, September/October 2000.

[11] B. Boehm. Get ready for agile methods. *IEEE Computer*, pages 2–7, 2002.

[12] I. Bratko and S. Muggleton. Applications of inductive logic programming. *Communications of the ACM*, 38(11):65–70, 1995. Available from `ftp://ftp.cs.york.ac.uk/pub/ML_GROUP/Papers/cacm.ps.gz`.

[13] I. Bratko. *Prolog Programming for Artificial Intelligence. (third edition)*. Addison-Wesley, 2001.

[14] L. Breiman, J. H. Friedman, R. A. Olshen, and C. J. Stone. Classification and regression trees. Technical report, Wadsworth International, Monterey, CA, 1984.

[15] J. Breuker and W. V. de Velde (eds). *The CommonKADS Library for Expertise Modelling*. IOS Press, Netherlands, 1994.

[16] L. Brownston, R. Farell, E. Kant, and N. martin. *Programming Expert Systems in OPS5: An Introduction to Rule-Based Programming*. Addison-Wesley, 1985.

[17] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 24(3), September 1992.

[18] B. Buchanan and E. Shortliffe. *Rule-Based Expert Systems: The MYCIN Experiments of the Stanford Heuristic Programming Project*. Addison-Wesley, 1984.

[19] B. Buchanan and R. Smith. Fundamentals of Expert Systems. In P. C. A. Barr and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence, Volume 4*, volume 4, pages 149–192. Addison-Wesley, 1989.

[20] C. Burgess and M. Lefley. Can genetic programming improve software effort estimation? a comparative evaluation. *Information and Software Technology*, 43(14):863–873, December 2001.

[21] E. P. B. Lopez, P. Meseguer. Knowledge based systems validation: A state of the art. *AI Communications*, 5(3):119–135, 1990.

[22] J. Catlett. Inductive learning from subsets or disposal of excess training data considered harmful. In *Australian Workshop on Knowledge Acqusition for Knowledge-Based Systems, Pokolbin*, pages 53–67, 1991.

[23] B. Chandrasekaran, T. Johnson, and J. W. Smith. Task structure analysis for knowledge modeling. *Communications of the ACM*, 35(9):124–137, 1992.

[24] C. Chang, J. Combs, and R. Stachowitz. Report on the expert systems validation associate (eva). *Expert Systems with Applications*, 1(3):217–230, 1990.

[25] P. Cheeseman, B. Kanefsky, and W. Taylor. Where the really hard problems are. In *Proceedings of IJCAI-91*, pages 331–337, 1991.

[26] W. Clancey. Model Construction Operators. *Artificial Intelligence*, 53:1–115, 1992.

[27] D. Clancy and B. Kuipers. Model decomposition and simulation: A component based qualitative simulation algorithm. In *AAAI-97*, 1997.

[28] E. Clarke, E. Emerson, and A. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[29] E. Clark and T. Filkorn. Exploiting symmetry in temporal logic model checking. In *Fifth International Conference on Computer Aided Verification*. Springer-Verlag, 1993.

[30] E. Clark and D. E. Long. Compositional model checking. In *Fourth Annual Symposium on Logic in Computer Science*, 1989.

[31] P. Clark and T. Ng. The cn2 induction algorithm. *Machine Learning*, 3:261–283, 1989.

[32] P. Cohen. *Empirical Methods for Artificial Intelligence*. MIT Press, 1995.

[33] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasarenu, Robby, and H. Zheng. Bandera: Extracting finite-state models from java source code. In *Proceedings ICSE2000, Limerick, Ireland*, pages 439–448, 2000.

[34] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.

[35] B. L. C. Heitmeyer and D. Kiskis. Consistency checking of scr-style requirements specifications. In *International Symposium on Requirements Engineering, York, England , March 26-27*, 1995.

[36] R. Davis. Interactive transfer of expertise: Acqusiition of new inference rules. *Artificial Intelligence*, 12(2):121–157, 1979.

[37] K. DeJong and W. Spears. An analysis of the interacting roles of population size and crossover in genetic algorithms. In *Proc. First Workshop Parallel Problem Solving from Nature*. Springer-Verlag, 1990.

[38] J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.

[39] A. V. de Brug, J. Bachant, and J. McDermott. The Taming of R1. *IEEE Expert*, pages 33–39, Fall 1986.

[40] T. Dietterich. Machine learning research: Four current directions. *AI Magazine*, 18(4):97–136, 1997.

[41] M. B. Dwyer, G. S. Avrunin, and J. Corbett. A system specification of patterns. `http://www.cis.ksu.edu/santos/spec-patterns/`, 1997.

[42] M. Dwyer, G. Avrunin, and J. Corbett. Patterns in property specifications for finite-state verification. In *ICSE98: Proceedings of the 21st International Conference on Software Engineering*, May 1998.

[43] C. C. D. Hamilton, K. Kelley. State-of-the-practice in knowledge-based system verification and validation. *Expert Systems with Applications*, 3:403–410, 1991.

[44] H. Eriksson, Y. Shahar, S. W. Tu, A. R. Puerta, and M. A. Musen. Task modeling with reusable problem-solving methods. *Artificial Intelligence*, 79(2):293–326, 1995.

[45] R. Evertsz. The automatic analysis of rule-based system based on their procedural semantics. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 22–27, 1991.

[46] M. Fagan. Advances in software inspections. *IEEE Trans. on Software Engineering*, pages 744–751, July 1986.

[47] M. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002. Available from `http://tim.menzies.com/pdf/02re02.pdf`.

[48] M. Feather and B. Smith. Automatic generation of test oracles: From pilot studies to applications. In *Proceedings of the Fourteenth IEEE International Conference on Automated Software Engineering (ASE-99), Cocoa Beach, Florida*, pages 63–72, October 1999. Available from `http://www-aig.jpl.nasa.gov/public/planning/papers/oracles-ase.pdf`.

[49] N. E. Fenton and M. Neil. A critique of software defect prediction models. *IEEE Transactions on Software Engineering*, 25(5):675–689, 1999. Available from `http://citeseer.nj.nec.com/fenton99critique.html`.

[50] K. M. Gardner, A. R. Rush, M. Crist, R. Konitzer, J. J. Odell, B. Teegarden, and R. Konitzer. *Cognitive Patterns: Problem-Solving Frameworks for Object Technology*. Cambridge University Press, June 1998.

[51] J. Gaschnig, P. Klahr, H. Pople, E. Shortliffe, and A. Terry. Evaluation of expert systems: Issues and case studies. In F. Hayes-Roth, D. Waterman, and D. Lenat, editors, *Building Expert Systems*, chapter 8, pages 241–280. Addison-Wesley, 1983.

[52] I. Gent, E. MacIntyre, P. Prosser, and T. Walsh. Scaling effects in csp phase transistion. In *International Conference on Principles and Practice of Constraint Programming*, 1995.

[53] Y. Gil and E. Melz. Explicit representations of problem-soving strategies to support knowledge acquisition. In *Proceedings AAAI' 96*, 1996.

[54] Y. Gil and M. Tallis. A script-based approach to modifying knowledge bases. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 1997.

[55] A. Ginsberg, S. Weiss, and P. Politakis. Automatic knowledge base refinement for classification systems. *Artificial Intelligence*, 35:197–226, 1988.

[56] P. Godefroid. On the costs and benefits of using partial-order methods for the verificiation of concurrent systems (invited papers). In *The 1996 DIMACS workshop on Partial Order Methods in Verificaition, July 24-26, 1996*, pages 289–303, 1997.

[57] D. Goldberg. *Genetic Algorithms in Search, Optimization, and Machine Learning*. Addison-Wesley, 1989.

[58] P. Grogono, A. Batarekh, A. Preece, R. Shinghal, and C. Suen. Expert system evaluation techniques: A selected bibliography. *Expert Systems*, pages 227–239, 1992.

[59] K. Havelund and G. Rosu. Java pathexplorer - a runtime verification tool. In *The 6th International Symposium on AI, Robotics and Automation in Space*, May 2001. Available from `http://ase.arc.nasa.gov/havelund/Publications/pax-overview.ps`.

[60] K. Havelund. Using runtime analysis to guide model checking of java programs. In *SPIN Model Checking and Software Verification*, pages 245–264. Springer-Verlag, 2000.

[61] C. Heitmeyer, R. Jeffords, and B. Labaw. Automated consistency checking of requirements specifications. *ACM Transactions on Software Engineering and Methodology*, 5(3):231–261, July 1996. Available from `http://citeseer.nj.nec.com/heitmeyer96automated.html`.

[62] C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, January 2002. Available from `http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.p%df`.

[63] G. Hinton. How neural networks learn from experience. *Scientific American*, pages 144–151, September 1992.

[64] G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.

[65] Y. Hu. Better treatment learning, 2003. Masters Thesis, Department of Electrical Engineering, University of British Columbia, in preperation.

[66] Y. Ishida. Using global properties for qualitative reasoning: A qualitative system theory. In *Proceedings of IJCAI '89*, pages 1174–1179., 1989.

[67] I. Jacobson and M. Christerson. A Growing Consensus on Use Cases. *JOOP*, pages 15–19, 1995.

[68] J. Josephson, B. Chandrasekaran, M. Carroll, N. Iyer, B. Wasacz, and G. Rizzoni. Exploration of large design spaces: an architecture and preliminary results. In *AAAI '98*, 1998. Available from `http://www.cis.ohio-state.edu/~jj/Explore.ps`.

[69] A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In C. H. D.M. Gabbay and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming 5*, pages 235–324. Oxford University Press, 1998.

[70] H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, Aug. 4–8 1996. AAAI Press / MIT Press. Available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.

[71] G. Kendall. Tutorial notes on the history of ai, 2001. Available from `http://www.cs.nott.ac.uk/~gxk/courses/g5aiai/002history/history.htm`.

[72] T. Khoshgoftaar and E. Allen. Model software quality with classification trees. In H. Pham, editor, *Recent Advances in Reliability and Quality Engineering*. World Scientific, 1999.

[73] P. S. Laird, R. J. E., and A. Newell. Chunking in SOAR: The anatomy of a general learning mechanism. *Machine Learning*, 1(1):11–46, 1986.

[74] J. Laurent. Proposals for a valid terminology in kbs validation. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI-92, Vienna, Austria*, pages 829–834, 1992.

[75] N. Leveson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.

[76] R. Lutz and R. Woodhouse. Bi-directional analysis for certification of safety-critical software. In *1st International Software Assurance Certification Conference (ISACC'99)*, 1999. Available from `http://www.cs.iastate.edu/~rlutz/publications/isacc99.ps`.

[77] R. Madachy. Heuristic risk assessment using cost factors. *IEEE Software*, 14(3):51–59, May 1997.

[78] D. Marques, G. Dallemagne, G. Kliner, J. McDermott, and D. Tung. Easy Programming: Empowering People to Build Their own Applications. *IEEE Expert*, pages 16–29, June 1992.

[79] S. McConnell. The best influences on software engineering. *IEEE Software*, January/February 2000. Available from `www.computer.org/software/so2000/pdf/s1010.pdf`.

[80] J. McDermott. R1's formative years. *AI Magazine*, 2(2):21–29, 1981.

[81] J. McDermott. R1 ("xcon") at age 12: lessons from an elementary school achiever. *Artificial Intelligence*, 59:241–247, 1993.

[82] M. Mendonca and N. Sunderhaft. Mining software engineering data: A survey, September 1999. A DACS State-of-the-Art Report. Available from `http://www.dacs.dtic.mil/techs/datamining/`.

[83] T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In *Annals of Software Engineering*, 2002. Available from `http://tim.menzies.com/pdf/02itar2.pdf`.

[84] T. Menzies, R. Cohen, S. Waugh, and S. Goss. Applications of abduction: Testing very long qualitative simulations. *IEEE Transactions of Data and Knowledge Engineering (accepted for publication, 2000)*, 2003. Available from `http://tim.menzies.com/pdf/97iedge.pdf`.

[85] T. Menzies and P. Compton. Applications of abduction: Hypothesis testing of neuroendocrinological qualitative compartmental models. *Artificial Intelligence in Medicine*, 10:145–175, 1997. Available from `http://tim.menzies.com/pdf/96aim.pdf`.

[86] T. Menzies, B. Cukic, H. Singh, and J. Powell. Testing nondeterminate systems. In *ISSRE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00issre.pdf`.

[87] T. Menzies and B. Cukic. On the sufficiency of limited testing for knowledge based systems. In *The Eleventh IEEE International Conference on Tools with Artificial Intelligence. November 9-11, 1999. Chicago IL USA.*, 1999.

[88] T. Menzies and B. Cukic. Adequacy of limited testing for knowledge based systems. *International Journal on Artificial Intelligence Tools (IJAIT)*, June 2000. Available from `http://tim.menzies.com/pdf/00ijait.pdf`.

[89] T. Menzies and B. Cukic. When to test less. *IEEE Software*, 17(5):107–112, 2000. Available from `http://tim.menzies.com/pdf/00iesoft.pdf`.

[90] T. Menzies and J. S. DiStefeno. Software metrics: A case study on mccabes. In *27th NASA SEL workshop on Software Engineering (submitted)*, 2002.

[91] T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *RE '99*, 1999. Available from `http://tim.menzies.com/pdf/99re.pdf`.

[92] T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01lesstalk.pdf`.

[93] T. Menzies and Y. Hu. Reusing models for requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01reusere.pdf`.

[94] T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, *Formal Approaches to Agent-Based Systems, book chapter*, 2002. Available from `http://tim.menzies.com/pdf/01agents.pdf`.

[95] T. Menzies and Y. Hu. Just enough learning (of association rules). In *WVU CSEE tech report*, 2002. Available from `http://tim.menzies.com/pdf/02tar2.pdf`.

[96] T. Menzies and C. Michael. Fewer slices of pie: Optimising mutation testing via abduction. In *SEKE '99, June 17-19, Kaiserslautern, Germany. Available from `http://tim.menzies.com/pdf/99seke.pdf`*, 1999.

[97] T. Menzies, D. Owen, and B. Cukic. Saturation effects in testing of formal models. In *ISSRE 2002*, 2002. Available from `http://tim.menzies.com/pdf/02sat.pdf`.

[98] T. Menzies, J. Powell, and M. E. Houle. Fast formal analysis of requirements via 'topoi diagrams'. In *ICSE 2001*, 2001. Available from `http://tim.menzies.com/pdf/00fastre.pdf`.

[99] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002.

[100] T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In *2nd International Workshop on Soft Computing applied to Software Engineering (Netherlands), February*, 2001. Available from `http://tim.menzies.com/pdf/00maybe.pdf`.

[101] T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In *Proceedings ASE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00ase.pdf`.

[102] T. Menzies. *Principles for Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales, 1995. Ph.D. thesis. Available from `http://tim.menzies.com/pdf/95thesis.pdf`.

[103] T. Menzies. Knowledge maintenance: The state of the art. *The Knowledge Engineering Review*, 14(1):1–46, 1999. Available from `http://tim.menzies.com/pdf/97kmall.pdf`.

[104] T. Menzies. Practical machine learning for software engineering and knowledge engineering. In *Handbook of Software Engineering and Knowledge Engineering*. World-Scientific, December 2001. Available from `http://tim.menzies.com/pdf/00ml.pdf`.

[105] P. Meseguer. Verification of multi-level rule-based expert systems. In *Proceedings of the Ninth National Conference on Artificial Intelligence*, pages 323–328, 1991.

[106] P. Meseguer. Incremental verification of rule-based expert systems. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI-92*, pages 840–844, 1992.

[107] P. Meseguer. Towards a conceptual framework for expert system validation. *Artificial Intelligence Communications*, 5(3):119–135, 1992.

[108] T. Mitchell, R. Keller, and S. T. Kedar-Cabelli. Explanation-based generalization: A unifying view. *Machine Learning*, 1:47–80, 1986.

[109] T. Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[110] E. Motta and Z. Zdrahal. Parametric design problem solving. In *Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop*, 1996.

[111] NASA. CLIPS Reference Manual. Software Technology Branch, lyndon B. Johnson Space Center, 1991.

[112] A. Newell. The Knowledge Level. *Artificial Intelligence*, 18:87–127, 1982.

[113] A. Newell. Reflections on the Knowledge Level. *Artificial Intelligence*, 59:31–38, Feburary 1993.

[114] T. Nguyen, W. Perkins, T. Laffey, and D. Pecora. Knowledge base verification. *AI Magazine*, 8(2):69–75, 1987.

[115] R. O'Keefe and D. O'Leary. Expert system verification and validation: a survey and tutorial. *Artificial Intelligence Review*, 7:3–42, 1993.

[116] A. Parkes. Lifted search engines for satisfiability, 1999.

[117] D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, Dec. 1972.

[118] C. Pecheur and R. Simmons. From livingstone to smv: Formal verification for autonomous spacecrafts. In *Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems*, pages 5–7, April 2000. Available from `http://ase.arc.nasa.gov/pecheur/publi/Livingstone2smv-faabs.ps`.

[119] G. Prakash, E. Subramanian, and H. Mahabala. A methodology for systematic verification of ops5-based ai applications. In *Proceedings of the 12th International Joint Conference on Artificial Intelligence (IJCAI'91)*, pages 3–8, 1991.

[120] A. Preece, R. Shinghal, and A. Batarekh. Verifying expert systems: a logical framework and a practical tool. *Expert Systems with Applications*, 5(2):421–436, 1992.

[121] A. Preece and R. Shinghal. Verifying knowledge bases by anomaly detection: An experience report. In *ECAI '92*, 1992.

[122] A. Preece. Principles and practice in verifying rule-based systems. *The Knowledge Engineering Review*, 7:115–141, 2 1992.

[123] R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.

[124] R. Quinlan. *C4.5: Programs for Machine Learning*. Morgan Kaufman, 1992. ISBN: 1558602380.

[125] E. S. Raymond and B. Young. *The Cathedral and the Bazaar: Musings on Linux and Open Source by an Accidental Revolutionary*. O'Reilly & Associates, 2001. A short form is available from `http://www.tuxedo.org/~esr/writings/cathedral-bazaar/cathedral-bazaar/`.

[126] R. O. R.M., O. Balci, and E. Smith. Validating expert system performance. *IEEE Expert*, 87:81–89, Winter 1987.

[127] M. Rousset. On the consistency of knowledge bases: the covadis system. In *Proceedings of the 8th European Conference on Artificial Intelligence (ECAI'88)*, pages 79–84, 1988.

[128] J. Rushby. Quality measures and assurance for ai software, 1988. SRI-CSL-88-7R, SRI Project 4616.

[129] R. Rymon. An SE-tree based characterization of the induction problem. In *International Conference on Machine Learning*, pages 268–275, 1993.

[130] R. Rymon. An se-tree-based prime implicant generation algorithm. In *Annals of Math. and A.I., special issue on Model-Based Diagnosis*, volume 11, 1994. Available from `http://citeseer.nj.nec.com/193704.html`.

[131] F. Schneider, S. Easterbrook, J. Callahan, G. Holzmann, W. Reinholtz, A. Ko, and M. Shahabuddin. Validating requirements for fault tolerant systems using model checking. In *3rd IEEE International Conference On Requirements Engineering*, 1998.

[132] A. T. Schreiber, B. Wielinga, J. M. Akkermans, W. V. D. Velde, and R. de Hoog. Commonkads. a comprehensive methodology for kbs development. *IEEE Expert*, 9(6):28–37, 1994.

[133] G. Schreiber, editor. *Knowledge Engineering and Management : The CommonKADS Methodology*. MIT Press, 1999.

[134] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *AAAI '92*, pages 440–446, 1992.

[135] E. Y. Shapiro. *Algorithmic program debugging*. MIT Press, Cambridge, Massachusetts, 1983.

[136] M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.

[137] J. Singer, I. P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.

[138] B. Smith and M. Dyer. Locating the phase transition in binary constraint satisfaction problems. *Artificial Intelligence*, 81(1-2):155–181, 1996.

[139] E. Soloway, J. Bachant, and K. Jensen. Assessing the maintainability of xcon-in-rime: Coping with the problems of a very large rule-base. In *AAAI '87*, pages 824–829, 1987.

[140] K. Srinivasan and D. Fisher. Machine learning approaches to estimating software development effort. *IEEE Trans. Soft. Eng.*, pages 126–137, February 1995.

[141] L. Steels. Components of Expertise. *AI Magazine*, 11:29–49, 2 1990.

[142] M. Stefik, J. Aikins, R. Balzer, J. Benoit, L. Birnhaum, F. Hayes-Roth, and E. Sacerdoti. The Organisation of Expert Systems, A Tutorial. *Artificial Intelligence*, 18:135–127, 1982.

[143] M. Suwa, A. Scott, and E. Shortliffe. Completeness and consistency in rule-based expert systems. *AI Magazine*, 3(4):16–21, 1982.

[144] B. Swartout and Y. Gill. Flexible knowledge acquisition through explicit representation of knowledge roles. In *1996 AAAI Spring Symposium on Acquisition, Learning, and Demonstration: Automating Tasks for Users*, 1996.

[145] D. Tansley and C. Hayball. *Knowledge-Based Systems Analysis and Design*. Prentice-Hall, 1993.

[146] J. Tian and M. Zelkowitz. Complexity measure evaluation and selection. *IEEE Transaction on Software Engineering*, 21(8):641–649, Aug. 1995.

[147] P. M. T. Hoppe. Vvt terminology: A proposal. *IEEE Expert*, 8(3):48–55, 1993.

[148] F. van Harmelen and M. Aben. Structure-preserving specification languages for knowledge-based systems. *International Journal of Human-Computer Studies*, 44:187–212, 1996.

[149] F. van Harmelen and A. Bundy. Explanation-based generalisation = partial evaluation. *Artificial Intelligence*, pages 401–412, 1988.

[150] B. W.Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.

[151] B. Wielinga, A. Schreiber, and J. Breuker. KADS: a Modeling Approach to Knowledge Engineering. *Knowledge Acquisition*, 4:1–162, 1 1992.

[152] G. Williams and Z. Huang. A case study in knowledge acquisition for insurance risk assessment using a kdd methodology. In *Proceedings PKAW '96: Pacific Knowledge Acquisition Workshop*, 1996.

[153] I. H. Witten and E. Frank. *Data Mining: Practical Machine Learning Tools and Techniques with Java Implementations*. Morgan Kaufmann, 1999.

[154] G. Yost and A. Newell. A Problem Space Approach to Expert System Specification. In *IJCAI '89*, pages 621–627, 1989.

[155] G. Yost. Acquiring knowledge in soar. *IEEE Expert*, pages 26–34, June 1993.

[156] N. Zlatereva and A. Preece. State of the art in automated validation of knowledge-based systems. *Expert Systems with Applications*, 7:151–167, 2 1994.