Verification and Validation and Artificial Intelligence

Tim Menzies

Computer Science, Portland State University, Oregon, USA

Charles Pecheur

RIACS at NASA Ames Research Center, Moffett Field, CA, USA

Contents

Abstract

Artificial Intelligence (AI) is useful. AI can deliver more functionality for reduced cost. AI should be used more widely but won't be unless developers can trust adaptive, nondeterministic, or complex AI systems.

Verification and validation is one method used by software analysts to gain that trust. AI systems have features that make them hard to check using conventional V&V methods. Nevertheless, as we show in this chapter, there are enough alternative readily-available methods that enable the V&V of AI software.

1 Introduction

Artificial Intelligence (AI) is no longer some bleeding technology that is hyped by its proponents and mistrusted by the mainstream. In the 21st century, AI is not necessarily amazing. Rather, it is often *routine*. Evidence for the routine and dependable nature of AI technology is everywhere (see the list of applications in [1]).

The AI approach 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. AI offers improved capabilities at a reduced operational cost. For example, Figure 1 describes the AI used in NASA's Remote Agent Experiment (RAX) [2]. For over a day, this system ran a deep-space probe without any help from mission control. Such AI-based autonomy is essential to future deep space missions. NASA needs such autonomous software so that deep space probes can handle unexpected or important events billions of miles away from earth when they are hours away from assistance by mission control.

AI software can be complex and the *benefits* of complexity are clear. Some applications such as the RAX described in Figure 1 are inherently complex

---

Email addresses: `tim@menzies.us` (Tim Menzies), `pecheur@email.arc.nasa.gov` (Charles Pecheur).
URLs: `http://menzies.us` (Tim Menzies), `http://ase.arc.nasa.gov/pecheur` (Charles Pecheur).

Autonomous Control — is exemplified by → Remote Agent — performed → Successful Flight Demonstration

Remote Agent — was component of → Deep Space 1

Deep Space 1 — is → JPL-managed Technology Mission

Successful Flight Demonstration — dealt with → 3 Simulated Failures

Successful Flight Demonstration — in → May 1999

Remote Agent — joins together in constant communication → Planner & Scheduler, Smart Executive, Mode Identification and Recovery

Attitude Control System — reports to → Planner & Scheduler

Navigation System

Planner & Scheduler — creates → Plan for Given Period

Planner & Scheduler — is cognizant of → General Mission Goals, Current Spacecraft Status

Smart Executive — requests → Plan for Given Period

Smart Executive — fills in details of → Plan for Given Period

Smart Executive — sends → Commands

Smart Executive — monitors / executes → Recovery Actions

Mode Identification and Recovery — identifies → Failures

Mode Identification and Recovery — suggests → Recovery Actions

Mode Identification and Recovery — receives information from → Monitors

Monitors — are located throughout → Spacecraft

Commands — to → Other Elements of Flight Software

Other Elements of Flight Software — control → Flight Hardware

NASA's deep space missions require *autonomous spacecrafts* that don't rely on ground control. The Remote Agent Experiment (RAX) was an experiment in autonomy technology: for two days, RAX provided on-board AI control for the Deep Space One probe, while it was 60,000,000 miles from Earth.

RAX had three main components. The *Planner & Scheduler (PS)* took general goals and determined detailed activities needed to achieve the goals. If a hardware problem developed that prevents execution of the plan, the planner made a new plan, taking into account degraded capabilities. The *Smart Executive (EXEC)* interpreted the plans and added more detail to them, then issued commands to rest of the satellite. Lastly, the *Mode Identification and Recovery (MIR)* component (based on the Livingstone diagnosis system) acted like a doctor, monitoring the spacecraft's health.

If something went wrong, MIR would detect it and report it to EXEC. Exec could then consult the "doctor" for simple procedures that may quickly remedy the problem. If those simple procedures could not resolve the problem, EXEC asked PS for a new plan that still achieved the mission goals while accounting for the degraded capabilities.
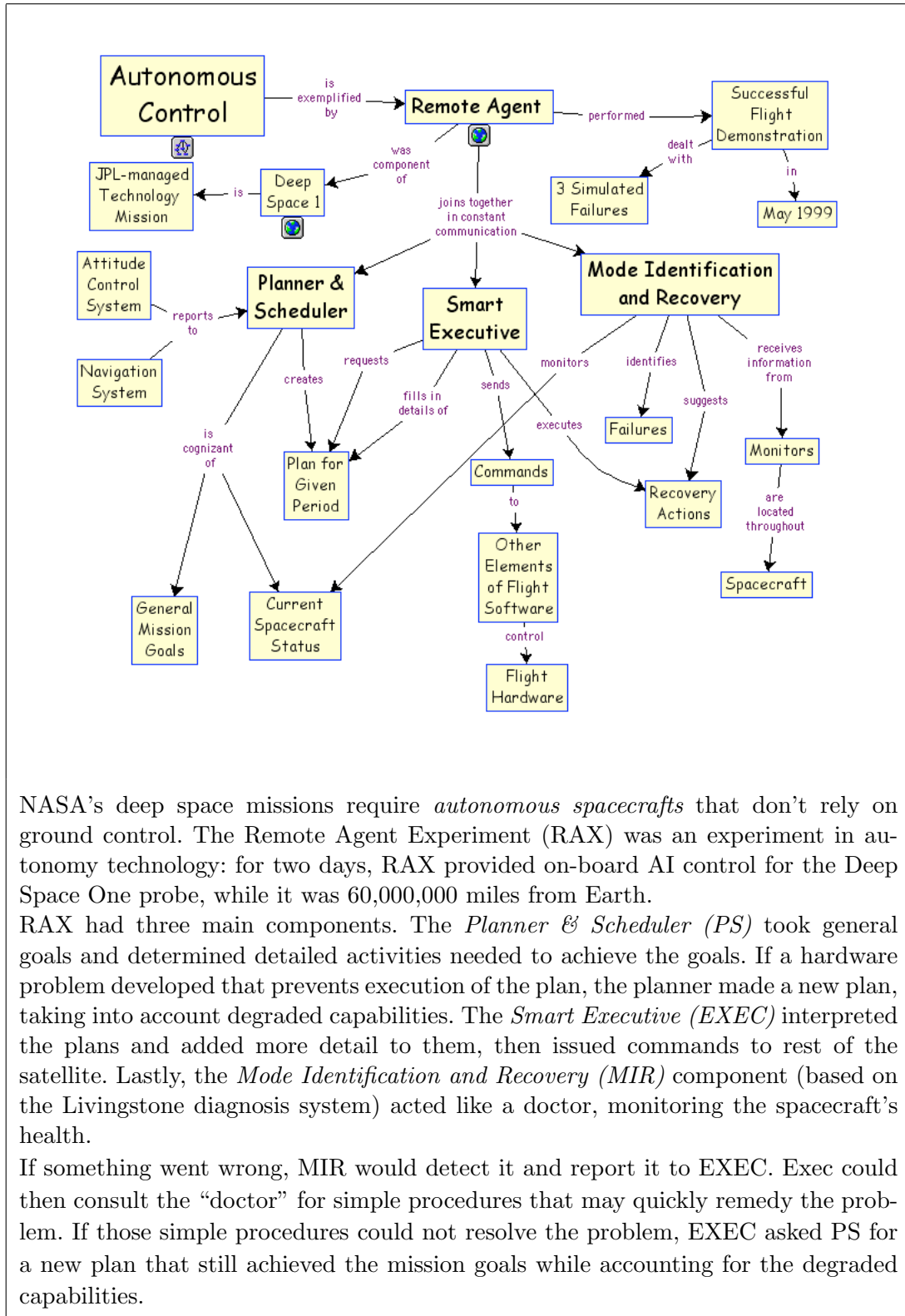
Fig. 1. The Remote Agent Experiment, from
http://cmex-www.arc.nasa.gov/CMEX/RemoteAgent.html

and require an extension to existing technology.

However, the *cost* of complexity is that complex systems are harder to understand and hence harder to test. Complex systems like can hide intricate interactions which, if they happen during flight, could compromise the mission. For example, despite a year of extensive testing, when Remote Agent was first put in control of NASA's Deep Space One mission, it froze because of a software deadlock problem (RAX was re-activated two days later and successfully completed all its mission objectives). After analysis, it turns out that the deadlock was caused by a highly unlikely race condition between two concurrent threads inside Remote Agent's executive. The scheduling conditions that caused the problem to manifest never happened during testing but indeed showed up in flight [2].

Hardware engineers solve such problems with hardware redundancy (when one component fails, its back-up wakes up and takes over). However, redundancy may not solve software reliability problems. Hardware components fail statistically because of wear or external damage. Software components programs fail almost exclusively due to latent design errors. Failure of an active system is thus highly correlated with failure of a duplicate back-up system (unless the systems use different software designs, as in the Space Shuttle's on-board computers). The classic example of how redundancy does not solve reliability is the infamous Ariane 5 rocket explosion where the backup Interial Reference system failed on the same software bug several milliseconds before the primary system failed [1].

If redundancy may not increase the reliability of AI software, what else should we do to check our AI software? How should standard verification and validation (V&V) be modified to handle AI systems? 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 chapter offers an overview of the six features of AI systems that a V&V analysts must understand. An AI system may be:

(1) *complex* software;
(2) *declarative* and *model-based*, and sometimes *knowledge-level* software;
(3) *nondeterministic* or even *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 chapter is structured around this list of features of an AI system. Each feature will be defined and their associated cost and benefits will be discussed. To help the reader who is a not an AI specialist, many of

---

[1] See http://sunnyday.mit.edu/accidents/Ariane5accidentreport.html

our sections start with short tutorials.

Note that the approach of this chapter is different to the traditional reviews of AI verification (e.g. [3–15]) or AI (e.g. [16–24]). To some extent, this difference is due to the domain within which we work. The first generation of expert system validation tools worked mostly on the rule-based or frame-based representations that were common in the 1980s and 1990s. Since then, much has changed. We work with complex model-based automonous vehciles of the kind flown by NASA. Hence, while other articles offer success stories with rule-based or frame-based representations (e.g. [25–28] and Chapt. 8,30,31,34 in [29]), this chapter focuses more on the features of modern AI that distinguishes it from conventional procedural software; e.g. nondeterministic 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, 24, 28, 29]) or one of the many excellent on-line bibliographies on V&V of AI systems [2].

2 AI Software can be Complex

The rest of this chapter stresses what is *different* about AI systems and how those differences effect V&V for AI. Before moving on to that material, this section observes that AI software is still software, albeit sometimes quite complex software. Hence, methods developed for assessing normal software systems still apply to AI systems. V&V analysts should view this chapter as techniques that *augment*, not *replace* their standard V&V methods such as peer reviews, automated test suites, etc [30–32].

In this section, we review the state-of-the-art in verifying complex software. Many of these techniques are routinely applied at NASA when verifying complex AI systems such as RAX.

We start with conventional *testing* as a baseline, then introduce more advanced formal methods: *run-time monitoring*, *static analysis*, *model checking* and *theorem proving*. Those methods vary in the strength of the verdicts they provide, as well as the level of expertise they require—generally speaking, more thorough approaches require more expertise. This can be laid out as the "verification methods spectrum" shown in Figure 2 [3].

---

[2] e.g. `http://www.csd.abdn.ac.uk/~apreece/Research/vvbiblio.html`
[3] Adapted from John Rushby, in slides of [33]. This spectrum is notional and expresses trends rather than absolute truths. In particular, a comparative experimental evaluation conducted at NASA [34] concluded that static analysis may require a deep understanding of its underlying algorithms to be used effectively.
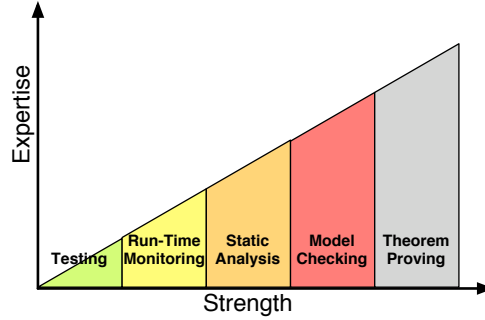
Fig. 2. The verification methods spectrum.

2.1 Testing

Traditionally, software verification is done using scenario-based testing. The system to be verified is embedded into a test harness that connects to the inputs and outputs of that component, and drives it through a suite of test runs. Each test run is an alternated sequence of provided inputs and expected outputs, corresponding to one scenario of execution of the tested component. An error is signaled when the received output does not meet the expected one.

Even for simple systems, the design and maintenance of test suites is a difficult and expensive process. It requires a good understanding of the system to be tested, to ensure that a maximum number of different situations are covered using a minimum number of test cases. Running the tests is also a time-consuming task, because the whole program code has to be executed and everything must be re-initialized before each test run. In the development of complex systems, it is quite common that testing the software actually takes more resources than developing it.

While traditional testing may be enough for more conventional software, it falls short for complex AI software, mainly because the range of situations to be tested is incomparably larger. A typical intelligent program implicitly incorporates responses to a very large space of possible inputs (for example, consider the possible interaction sequences that an autonomous agent like RAX may face). The internal state of the program is typically huge, dynamically allocated (heap memory, garbage collection) and may involve complex data structures (knowledge, rules, theories). It depends on the past history of the system in intricate ways. AI systems may involve several concurrent components, or other sources of non-determinism.

Because of all these factors, a test suite can only exercise a very limited portion of the possible configurations, and it is very hard to assess how much has been covered and therefore measure the residual risk of errors.

6

A temporal logic is a classical logic augmented with operators to reason about the evolution of the model over time. Temporal logic allows to express conditions over time, such as "any request is always eventually fulfilled". For example, (propositional) *linear temporal logic*, or LTL uses the following operators:

Property	Reads	Means p holds ...
$\circ p$	"next p"	...in the next state
$\square p$	"henceforth p"	...in all future states
$\diamond p$	"eventually p"	...in some future state
$p \cup q$	"p until q"	...in all states until q holds

Fig. 3. About temporal logic.

2.2 Run-Time Monitoring

Run-time monitoring, or run-time verification, refers to advanced techniques for scrutinizing artifacts from an executing program (variables, events, typically made available through program instrumentation), in order to detect effective or potential misbehaviors or other useful information.

Simple runtime monitoring methods have been used for decades. For example, it is standard practice for programmers to add conditionals to their code that print warning messages if some error condition is satisfied, or to generate and scan additional logging messages for debugging purposes. In essence, runtime verification automates the otherwise strenuous and error-prone task of reviewing these logs manually.

This analysis can be conducted after-the-fact on stored execution traces, but also on-the-fly while the program is executing. In this latter case, the need to store the execution trace is alleviated, and the monitor can also trigger recovery mechanisms as part of a software fault protection scheme.

Recently, the sophistication of runtime monitoring methods has dramatically increased. Commercial tools such as Temporal Rover [35] can instrument a program to execute inserted code fragments based on complex conditions expressed as *temporal logic formulae* (see Figure 3). New algorithms can detect suspicious concurrent programming patterns (data races [36], deadlocks [37]) that are likely to cause an error, even if no error occurs on the observed trace.

Runtime monitoring typically requires little computing resources and therefore scales up well to very large systems. On the other hand, it will only observe a limited number of executions and thus gives only uncertain results. In the case of error predictions, it can also give false negatives, i.e. flag potential errors

that cannot actually occur.

One example of a runtime monitoring system is Java Path Explorer (JPaX) [37]. Given a specification of properties to be monitored, JPaX instruments a Java program to generate a trace of relevant events. JPaX also produces an observer program that reads that trace and verifies the properties. The trace can be streamed through a socket, to allow local or remote on-the-fly monitoring. Both user-provided temporal logic conditions and generic deadlock and data race conditions can be monitored.

In [38], run-time monitoring is applied (in conjunction with automated test-case generation) to verify the controller of the K9 planetary rover. The controller is a large multi-threaded program (35,000 lines of C++) that controls the rover according to a flexible plan generated by a planning program. Each test case amounts to a plan, for which a set of temporal properties are derived, according to the semantics of the plan. The EAGLE system [39] is then used to monitor this properties. This system is fully automated and unveiled a flaw in plan interpretation, that indeed occurred during field tests before it was fixed in the controller. A potential deadlock and a data race were also uncovered.

2.3 Static Analysis

Static Analysis consists in exploring the structure of the source code of a program without executing it. It is an important aspect of compiler technology, supporting type analysis and compiler optimizations. A good reference textbook on the topic can be found in [40]. Static analysis includes such aspects as *control- and data-flow analysis* (tracking the flows of execution and the propagation of data in the code), *abstract interpretation* (computing abstract approximations of the allowed ranges of program variables) and *program slicing* (capturing the code portions that are relevant to a particular set of variables or a function).

In principle, static analysis can be applied to source code early in the development and is totally automatic. There is, however, a trade-off between the cost and the precision of the analysis, as the most precise algorithms have a prohibitive complexity. More efficient algorithms make approximations that can result in a large number of false positives, i.e. spurious error or warning messages.

Two commercial static analysis tools are Grammatech's CodeSurfer and the PolySpace Verifier [41]. Figure 4 shows CodeSurfer using a static control flow analysis to find the code *not* reachable from the `main` function. Such code is dead code and represents either over-specification (i.e. analysts exploring too
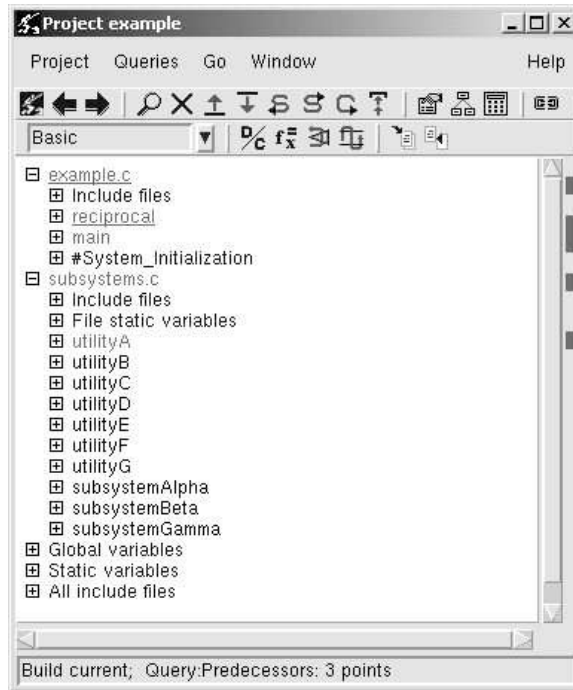
Fig. 4. Slicing in Grammatech's CodeSurfer tool (see
`http://www.grammatech.com/products/codesurfer/example.html`).

many special cases) or code defects (e.g. the wrong items are present in a
conditional). In Figure 4 the code sections reached from `main` are shown with
dark 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.

PolySpace uses abstract interpretation to find potential run-time errors in (C
or Ada) programs such as:

- access to non-initialized variables,
- unprotected shared variables in concurrent programs,
- invalid pointer references,
- array bound errors,
- illegal type conversions,
- arithmetic errors (overflow, underflow, division by zero, . . .),
- unreachable code.

The output of the tool consists of a color-coded version of the program source
code, as shown on Figure 5. Green code is guaranteed free from errors, red code
is sure to cause errors, orange code may cause errors (depending on execution
paths or because the analysis was inconclusive), and grey code is unreachable.

Once the code to be analyzed has been identified, static analysis tools such
as CodeSurfer and PolySpace are fully automatic. However, the experiment
reported in [34] concludes that their use can be a labor-intensive, highly it-

9

```
135    float beta;
136
137        static void Square_Root_conv (float alpha, float *beta_pt)
138        /* Perform arithmetic conversion of alpha to beta */
139        {   *beta_pt = (float)(1.5 + cos(alpha))/5.0;
140        }
141
142        static void Square_Root ()
143        {   float alpha = random_float();
144            float gamma;
145
146            Square_Root_conv (alpha, &beta);
147
148            gamma = (float)sqrt(beta - 0.75);    // always sqrt(negative number)
149        }
150
151
152
153
154        static void Unreachable_Code()
155        /* Here we demonstrate PolySpace Verifier's ability to
156            identify unreachable sections of code due to the
157            value constraints placed on the variables.
158        */
159        {   int x = random_int();
160            int y = random_int();
161
162            if (x > y)
163            {   x = x - y;
164                if (x < 0)
165                    x = x + 1;
166            }
```

green (ok)

red (error)

orange (warning)

grey (unreachable)

Fig. 5. An example of color-coded source code produced by the PolySpace Verifier. From `http://www.polyspace.com/datasheets/c_psde.htm` (callouts added).

erative process, in order to: (i) isolate a proper self-standing code set to be analyzed, (ii) understand and fix the reported red errors, (iii) adjust analysis parameters to reduce the number of false alarms. The last point can be very detrimental, as the analysis tends to return a very large number of mostly spurious "orange" warnings, making it very hard to identify real errors. The C Global Surveyor tool (CGS), currently under development at NASA, drastically reduces this problem by specializing the analysis algorithms to the coding practices of a specific class of applications [42].

2.4 Model Checking

Model Checking consists in verifying that a system (or a model thereof) satisfies a property by exhaustively exploring all its reachable states. Model checking was invented in the 1980s to analyze communication protocols [43,44], and is now routinely employed in verification of digital hardware. Several mature and powerful model checkers are available and widely used in the research com-

munity; SPIN [45,46] and SMV [47,48] are probably the best known. See [49] for a comprehensive theoretical presentation, and [50] for a more practical introduction.

A model checker searches all pathways of the system looking for ways to violate the property. This requires that this state space be finite and tractable: model checking is limited by the state space explosion problem, where the number of states can grow exponentially with the size of the system.[4] Tractability is generally achieved by abstracting away from irrelevant details of the system. If a violation is found, the model checker returns the counter examples showing exactly how the property is violated. Such counter examples are useful in localizing and repairing the source of the violation.

Model checking requires the construction of two models:

- The *systems model* is an abstract description of the dynamic, generally concurrent behavior of a program or system. For tractability reasons, it is usually not the complete code from the implementation but rather some abstract verification model of the application, capturing only the essential features that are relevant to the properties to be checked.
- The *properties model* is a specification of the requirements that should hold across the systems model. The properties model is often expressed as a temporal logic constraint.[5]

Model checkers often impose their own modeling language for both the systems and the properties, though more and more tools now apply directly to common design and programming languages (UML, Java), either natively or through translation.

Although model checkers are automatic tools, the benefits of model checking come at a cost that is often very significant. That cost can be divided into three components:

- *writing cost*: the initial cost of developing the systems model and the properties model, in a form accepted by the model checker;
- *running cost*: the cost of actually performing the model checking, as many times as necessary; and

---

[4] When the state space is too large or even infinite, model checking can still be applied: it will not be able to prove that a property is satisfied, but can still be a very powerful error-finding tool.

[5] Depending on the model checker being used, properties are sometimes expressed in other forms, such as invariants that must hold in every state, regular expressions that execution traces must match, or other dynamic system models whose executions must agree (in some precisely defined sense) with those of the verified system (see e.g. [51]).

- *re-writing cost*: the cost of iteratively modifying the models until model checking can complete successfully and provide acceptable results.

With traditional model checking tools, both the systems model and the properties model have to be written in their own tool-specific language, which often results in high *writing cost*. In particular, scarce and expensive PhD-level mathematical expertise may be required to properly encode properties in formal temporal logic.

Once models have been completed, the model checker 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 generally forces analysts to simplify the formal models, for example by removing parts or functions, abstracting away some details, or restricting the scope of the analysis. Such a rewrite incurs the *rewrite cost*. On the other hand, simplification may take away elements that are relevant to the properties being verified, so a balance must be found between tractability and accuracy of the analysis. This typically requires several iterations and a significant amount of expertise in both the tools and the application being verified.

Much research has tried to reduce these costs. The writing of systems models can be avoided by applying model checkers directly to readily available representations, such as design models or program code. For example, the BANDERA system extracts systems models from JAVA source code and feeds them into SMV or SPIN machine [52]. Java PathFinder applies model checking directly to Java bytecode, using its own custom Java virtual machine [53]. Other approaches such as SCR [54–56] or simple influence diagrams [57] propose simplified modeling environments where users can express their models in a simple and intuitive framework, and then map them into model checkers such as SPIN.

Interestingly enough, AI software can offer good opportunities for model checking, as a consequence of using model-based or knowledge-based approaches. This aspect is discussed in detail in Section 3; for now let us point out that the models that are interpreted in AI systems tend to be abstract enough to be readily amenable to model checking, with only minor adaptation writing costs. Pecheur and Simmons' verification of models used for autonomous diagnosis using the SMV synbolic model checker is a good example [58].

On the properties side, Dwyer, Avrunin & Corbett [59, 60] have developed a taxonomy of *temporal logic patterns* that covers most of the properties observed in real-world applications. For each pattern, they have defined an expansion from the intuitive pseudo-English form of the pattern to a formal temporal

logic formula.[6] These patterns are simpler and more intuitive than their logical counterpart, and shield the analysts away from the complexity of formal logics. For example, consider the following property 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 formula for this property is:

$$\Box((Q \wedge \Diamond R) \rightarrow ((\neg P \wedge \neg R) \cup (R \vee ((P \wedge \neg R) \cup (R \vee ((\neg P \wedge \neg R)$$
$$\cup (R \vee ((P \wedge \neg R) \cup (R \vee (\neg P \vee R)))))))))$$

In contrast, this property can be expressed using the "bounded existence" temporal logic pattern as "*transitions to P-states occur at most 2 times between Q and R*", and then be automatically translated to the form above. The translator in [58] offers a similar facility, though focused on more specific property classes.

These tools reduce the *writing cost* but don't necessary reduce the *running cost* or the *rewriting cost*. The *rewriting 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 modeling 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 [57], 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.

Despite decades of work and ever-increasing computing power, the high *running cost* of such general model checking remains a major challenge. Different techniques have proven to be useful in reducing the running cost of model checking:

- *Symbolic Methods* represent and process parts of the state symbolically to avoid enumerating each individual value that they can take. This can be done at the level of individual model variables [61] or over the model as

_____

[6] Actually, to several logic variants, including Linear Temporal Logic (LTL), Computation Tree Logic (CTL), Graphical Interval Logic (GIL), Quantified Regular Expressions (QRE) and INCA Queries.

a whole, using boolean encodings such as binary decision diagrams [47] or propositional satisfiability solvers [62].

- *Abstraction* can be applied in a principled way to map large or infinite concrete spaces into small abstract domains. *data abstraction* applies to individual variables (e.g. abstracting an integer to its sign) [63], while *predicate abstraction* uses the values of predicates obtained from the model [64]. In either case, the abstraction is usually not exact and produces spurious traces (i.e. for which no concrete trace exists in the original model).

- *Partial Order Reduction* analyzes dependency between concurrent operations to avoid exploring multiple equivalent permutations of independent operations. This is further discussed in Section 5.3.

- *Symmetry Reduction* uses symmetry in the system (e.g. identical agents) to avoid exploring multiple symmetrical states or paths [65, 66].

- *Compositional Reasoning* divides the systems model into separate components, which can be reasoned about separately [67–69]. This generally involves some form of assume/guarantee reasoning, to capture the interdependency between the different components.

- *Model Reduction* replaces the systems model by a reduced, simpler model that is equivalent with respect to the property being verified. For example, the BANDERA system [52] can automatically extract (slice) the minimum portions of a JAVA program's byte codes which are relevant to particular properties models.

While the above techniques have all been useful in their test domains, not all of them are universally applicable. Certain optimizations require expensive pre-processing, or rely on certain features of the system being studied. Not all systems exhibit reducible symmetry or concurrency that is amenable to partial order reduction. Compositional analysis is hard for tightly connected models. In the general case, model checking techniques are still limited to models of modest size, and obtaining these models from real applications requires significant work and expertise.

Despite these limitations, model checking is a widely used tool for AI systems at institutions like NASA. For example, in 1997, a team at NASA Ames used the Spin model checker to verify parts of the RAX Executive and found five concurrency bugs [70]. Although it took less than a week to carry out the verification activities, it took about 1.5 work-months to manually construct a model that could be run by Spin in a reasonable length of time, starting from the Lisp code of the executive. The errors found were acknowledged and fixed by the developers of the executive. As it turns out, the deadlock that occurred during the in-flight experiment in 1999 was caused by an improper synchronization in another part of the Executive, but was exactly of the same nature as one of the five bugs detected with Spin two years before. This demonstrates how this kind of concurrency bug can indeed pass through heavy test screens and compromise a mission, but can be found using more advanced techniques

such as model checking. In the same vein, compositional verification has been applied to the K9 Rover executive [71]. Further examples can be found in the Proceedings of the recent AAAI 2001 Spring Symposium on Model-Based Validation of Intelligence [7].

2.5   Theorem Proving

Theorem provers build a computer-supported proof of the requirement by logical induction over the structure of the program. In principle, theorem provers can use the full power of mathematical logic to analyze and prove properties of any design in its full generality. For example, the PVS system [72] has been applied to many NASA applications (e.g. [73]). However, these provers require a lot of efforts and skills from their users to drive the proof, making them suitable for analysis of small-scale designs by verification experts only. In contrast, the other methods discussed in the previous sections are largely automated, and thus more convenient for verification as part of a software development process. For this reason, theorem proving is still mostly limited to a few academic studies and regarded as inapplicable in an industrial setting.

This view may change, however, as proof systems feature increasingly powerful proof strategies, that can automatically reduce most of the simpler proof obligations. Also note that recent developments in formal verification are blurring this distinction between proof-based and state-based approaches: on one side, proof systems are extended with model checkers that can be used as decision procedures inside larger proofs [74]; on the other side, novel symbolic model checking approaches use proof-based solvers to prune out impossible paths in the symbolic state space [61, 75].

3   Model-based AI Systems

Having discussed how V&V AI can leverage techniques developed for software engineering problems, we now turn to the special features of AI that change the V&V task. This section discusses *model-based*. Subsequent sections will discuss *knowledge-level* systems, nondeterminism, adapation, and their implications for V&V.

Every V&V analyst 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

---

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

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 add a call between blocks and forget to update the high-level block diagram.

This disconnect between the specification and the implementation raises significant issues for formal verification. Often, formal verification techniques based on model checking (as opposed to those based on proof systems[8]) are able to efficiently check all possible execution traces of a software system in a fully automatic way. However, the system typically has to be manually converted beforehand into the syntax accepted by the model checker. This is a tedious and complex process, that requires a good knowledge of the model checker, and is therefore usually carried externally by a formal methods expert, rather than by the system designer themselves.

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 modeling 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[9].

These models are often *declarative* and V&V analysts can exploit such declarative knowledge for their analysis.

3.1 About Declarative Knowledge

Declarative representations can best be understood by comparing them to *procedural representations* used in standard procedural languages such as "C". Procedural representations list the ordering of activities require to complete some task. Procedural knowledge often manifests itself in the *doing of something* and may be hard to share with others except in the specific context where the procedural knowledge was developed.

V&V analysts know how hard it can be to un-tangle procedural knowledge such as a "C" program. Declarative knowledge describes facts and relation-

---

[8] Which can provide even more general results but require an even more involved and skilled guidance.

[9] This chapter defines the term "model" in its most common usage; i.e. the thing that is generated by analysts when they record information about their domain. Logic programming theorists prefer the term "theory". In logic, a "model" is some *instance* of a "theory" and is generated automatically at runtime.

ships within a domain. Declarative knowledge can be easier to understand than procedural knowledge; it can be easier to modify, easier to communicate to others, and easier to reuse for different purposes.

While procedural knowledge is about *how*, declarative knowledge is often statements about *what* is true in a domain. For example, consider the following piece of procedural knowledge. This implementation reports that you have "X" if it finds any evidence for any of the sub-types of "X". Note that this is knowledge about *how* to navigate a hierarchy of `disease`s, and their `symptom`s.

```
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!", disease, X);

              return 1
```

Suppose we wanted to report the `disease` that we have the *most* evidence for; i.e. the `disease` that has the most `symptom`s amongst the available `observation`s. In this procedural representation, this change would imply extensive modification to the code. An alternate approach would be to use declarative representations that queried the following facts:

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

Declarative representations free the analyst from specifying tedious procedural details. For example, the core logic of the above procedural code is that we have evidence for a `disease` is we have any `observation`s consistent with `subtyes` of that `disease`. This can be expressed directly and declaratively as follows:

```
%comments start with a percent sign
evidence(Disease,SubType,Evidence):-  % we have evidence if..
        subtype(Disease,SubType),           % we can find a subtype AND
        symptom(SubType,Evidence),          % AND that subtype has symptoms
        observation(Evidence).              % AND we can observe those symptoms
```

This declarative representation is useless without some interpreter that can exercise it. Our example here uses the syntax of the Prolog logic programming language [76]. In that language, upper case words are variables and lower case words are constants.

The major disadvantage of procedural knowledge is that it can be opaque and inflexible. Declarative knowledge, on the other hand, 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 fool Prolog into believing that all observations are possible. Since we are using declarative representations, this is simple to do: we just make an "anything goes" assertion:

```
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 "anything goes" 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 above `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
```

While the declarative version of knowledge might be faster to write and change, it may be slower to run. Procedural knowledge can be highly optimized to take advantage of low-level details of a system. In the 1970s, this was taken to be a major disadvantage of declarative knowledge. In the 21st century, this is much less of an issue. Not only are computers much faster now, but so are the interpreters for our declarative systems (e.g. [77, 78]), particularly when the representation can be restricted to some subset of full logic (e.g. [79, 80]).
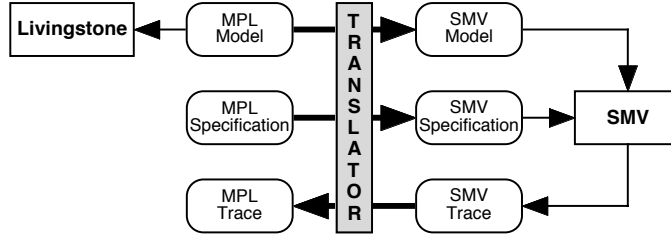
Fig. 6. Translator from Livingstone to SMV (MPL is Livingstone's modeling language).

3.2 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 [81] 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.

Declarative modeling tend to only use a small number of modeling constructs. This simplifies the construction of translators from one modeling language to another, and in particular from AI modeling languages to verification modeling languages such as those used by model checkers (as discussed in §2.4). Moreover, the reasoning algorithms used in AI typically suffer from similarly high complexity metrics as model checking, and therefore the size and complexity of AI models is already limited by the scalability of their intended usage. Indeed, AI fields such as planning and scheduling have much in common with model checking, in the way they both explore a state space described by a model. This similarity has lead to some cross-fertilization between the two fields in the recent years, with verification adopting search heuristics from planning on one hand [82] and planners based on verification technology such as BDDs on the other hand [83].

For example, Pecheur and Simmons have developed a translator to convert Livingstone models to SMV [58]. Livingstone is a model-based health monitoring system developed at NASA [84]. It tracks the commands issued to the device and monitors device sensors to detect and diagnose failures. To achieve this, Livingstone relies on a model of the device that describes, for each component, the nominal and abnormal functioning modes, how these modes are affected by commands and how they affect sensors. The translator enables exhaustive analysis of those models using the powerful SMV model checker. The essence of the translation is fairly straightforward, thanks to the similar semantics framework used in both Livingstone and SMV. The translator also supports user-oriented specification patterns and variables to express common

19

classes of properties such as consistency or existence of a broken component. These declarations are captured and converted into the core temporal logic syntax accepted by SMV. Finally, the execution traces returned by SMV can be converted back into Livingstone syntax. Together, these three translation capabilities (model, properties, trace) isolate the Livingstone application developer from the peculiarities of SMV and provide the functional equivalent of a symbolic model checker for Livingstone, as depicted if Figure 6.

The translator has been used at NASA Kennedy Space Center by the developers of a Livingstone model for the In-Situ Propellant Production (ISPP), a system that will produce spacecraft propellant using the atmosphere of Mars [85]. The latest version of the ISPP model, with 10^{50} states, could still be processed in less than a minute using SMV optimizations (re-ordering of variables). The Livingstone model of ISPP features a huge state space but little depth (all states can be reached within at most three transitions), for which the symbolic processing of SMV is very appropriate.

This tool can be used to check general sanity properties (e.g. consistency, absence of ambiguity, no conflicting transitions) or specific expected properties of the modeled device (e.g. flow conservation, functional dependency between variables). More recently, the technique has been extended to verify *diagnosability properties*, i.e. the possibility for an ideal diagnosis system to infer accurate and sufficient information on the state of the device from its observed behavior [86]. Diagnosability amounts to the absence of certain pair of traces with identical observations, which can be turned into a simple model checking problem over a duplicated version of the original model.

Model checking is powerful, but can be complicated. Feather and Smith report that a much simpler model-based technique can still be very insightful [87]. When asked to check the planner module of NASA's Remote Agent Experiment (RAX; see Figure 1 and [88]), they developed the architecture of Figure 7. RAX'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 fed 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. Note that, formally, the Feather and Smith method can be considered as an example of (after-the-fact) run-time monitoring (which was discussed above in §2.2).

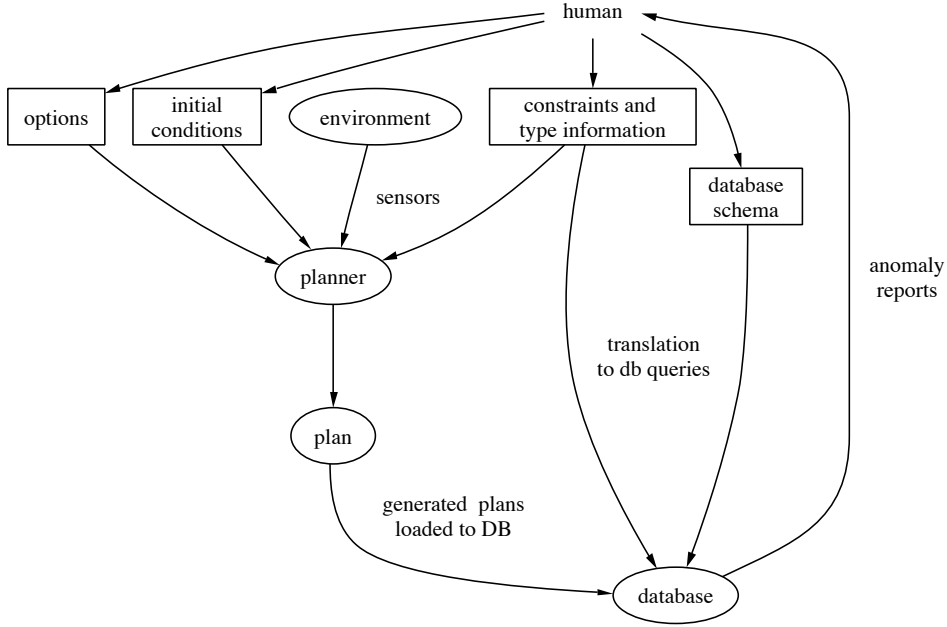The Feather and Smith method can be very cost-effective and applied quite widely:

Fig. 7. A framework for model-based V&V.

- The rectangles in Figure 7 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 7 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 [4]. 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 8) and argued that a variety of AI model-based verification tools target different subsets of these issues (perhaps using different terminology).

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.

21

$$
\text{Anomaly} \leftarrow
\begin{cases}
\text{Redundant} \leftarrow
\begin{cases}
\text{Unusable} \\
\text{Redundant} \leftarrow
\begin{cases}
\text{Duplicate} \\
\text{Subsumed}
\end{cases}
\end{cases} \\
\text{Ambivalence} \leftarrow \text{Conflict} \\
\text{Circularity} \\
\text{Deficiency} \leftarrow
\begin{cases}
\text{Missing rules} \\
\text{Missing values}
\end{cases}
\end{cases}
$$
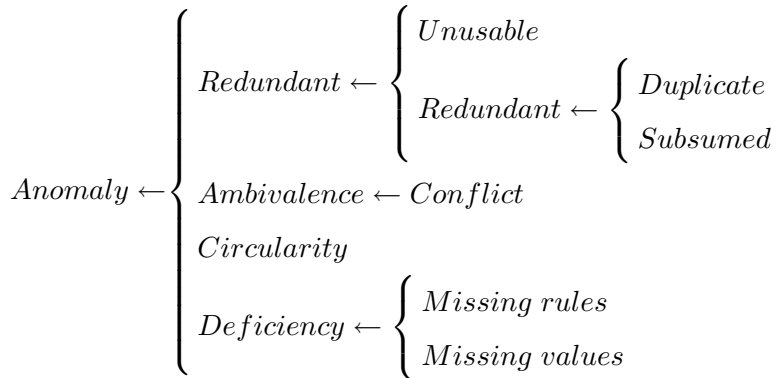
Fig. 8. The Preece hierarchy of verification errors.

	Application	
	mmu	tapes	neuron	displan	dms1	$^{\text{errors}}\!/_{\text{anomalies}}$
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	$^{20}\!/_{24} = 83\%$

Fig. 9. Ratio of $^{\text{errors}}\!/_{\text{anomalies}}$ seen in real-world expert systems. From [7]. "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.

- 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 9 shows the $^{\text{errors}}\!/_{\text{anomalies}}$ ratios seen in
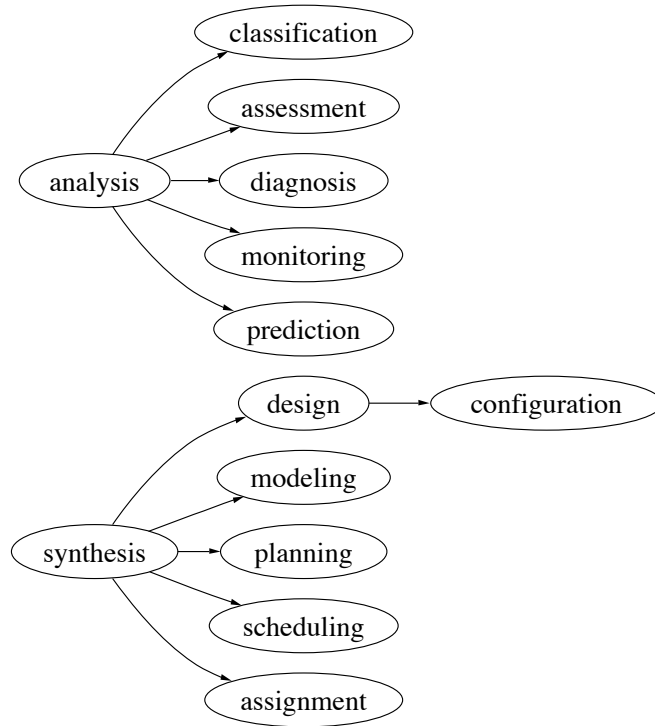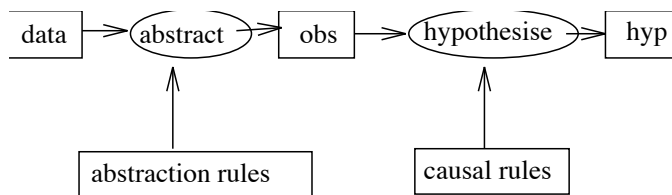
Fig. 10. A hierarchy of problem solving methods.

Fig. 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 [89].

five KBSs. Note that not all anomalies are errors.

4 The Knowledge Level

After decades of model-based programming, certain common model-based tasks were identified. That is, model-based programming became viewed like assembler code on top of which a range of *knowledge-level problem solving methods* (PSMs) are implemented.

A standard hierarchy of PSMs is shown in Figure 10. In this view, RAX's MIR (diagnosis) component (described in Figure 1) is one kind of diagnosis PSM. Another diagnosis PSM is shown in Figure 11. In that figure ovals denote functions and rectangles denote data structures. Formally, that figure

23

represents a mapping from data d to an hypothesis h via intermediaries Z and other data R_i:

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

Various design and code libraries were built around these knowledge-level PSMs e.g. cognitive patterns [90]; CommonKADS [91–93]; configurable role-limiting methods [94, 95]; MIKE [96]; the Method-To-Task approach [97]; generic tasks [98]; SPARK/BURN/FIREFIGHTER [99]; model construction operators [100]; components of expertise [101]; and the systems described in [90, 93, 98, 102–105]. The advantage of these PSMs is that they offer an organizational layer on top of model-based methods. V&V analysts can use this layer as an indexing method for their evaluation techniques.

For example, van Harmelen & Aben [89] discuss formal methods for repairing the diagnosis PSMs of Figure 11. 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 more than one sub-goal 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 [25, 106] acquired parts of the KB minority-type meta-knowledge for the XCON computer configuration system [28]. 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.

	Simple task #1				Harder task #2			
	no ETM	with ETM		no ETM		with ETM	
	S4	S1	S2	S3	S2	S 3	S1	S2
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

Fig. 12. Change times for ETM with four subjects: S1...S4. From [107]

- Users of the SPARK/ BURN/ FIREFIGHTER (SBF) [99] 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 differentiates 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 [107] 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).

5 AI Software can be Nondeterministic

The main challenge in verifying AI software (or, for that matter, any kind of complex system) comes from the number of different possible executions that have to be taken into account. We refer broadly to this uncertainty on a system's future behavior as *non-determinism*. Non-deterministic choices come from incoming external events, scheduling of concurrent tasks, or even intentional random choices in the program, to name a few. Every choice point in the execution results in as many possible subsequent execution branches. Typically, those choices compound into exponentially many possible executions. This is known as the *state space explosion* phenomenon.

Non-determinism can be either *external* or *internal*:

External non-determinism results from input or events coming from the **environment**. Examples include system configuration and initialization, invocation parameters, messages, discrete events, continuous data streams.

25

In the case of model-based systems, the models themselves constitute a huge choice space, as far as the interpreter is concerned.

Internal non-determinism result from the system itself. A common source is **concurrency**, where scheduling choices are made between concurrent executions (for example, a knowledge system that processes knowledge updates concurrently). Another case is **stochastic**, where the system itself is deliberately making random (or pseudo-random) choices.

The important distinction between these two is that while external non-determinism is *controllable* (and therefore can, in principle, be tested), internal non-determinism is not, which poses an additional problem to the verifier: the same test case may produce different results when run several times, and it is hard to measure or control the coverage of internal non-deterministic choices.

Conventional sequential programs are usually (internally) deterministic, in that they contain hard-wired decision paths that result in a functional mapping from inputs to outputs. The same is true of many applicative AI algorithms taken in isolation. However, as these algorithms get implemented and assembled together to build complex intelligent software, additional nondeterminism is introduced in the form of concurrent components executing asynchronously. This is particularly the case for reactive systems such as robotic controllers, that have to react in a timely manner to external stimuli occurring at unredictable times. These systems can resort to *anytime* algorithms, whose output depends on how long they are allowed to run. In typical concrete cases, nondeterminism is thus an issue that V&V analysts of AI systems have to face.

The rest of this section discusses V&V and these different types of nondeterminism.

5.1 Environmental Nondeterminism

The standard method of managing environmental nondeterminism is to build a *operational profile* modeling the probability that a certain variable setting will appear at runtime [108]. The range of possible inputs can now be sampled via the operational profile. The operational profile can be used to generate representative nominal inputs. Also, by inverting the profile, unlikely off-nominal test cases can also be generated.

In the case of AI agents performing tasks in dynamic environments for deep space missions, building an accurate operational profile is a very difficult task. In turns our that operational profile errors can be compensated by increasing the number of tests. Figure 13 shows Pasquini's study where the original operational profile OPe was compared to three profiles containing an increasing number of errors. The mutants were called (in order of increasing errors)
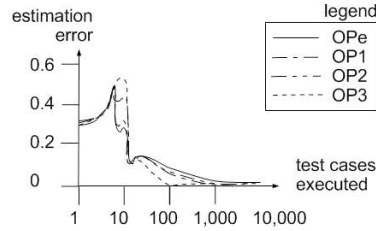
Fig. 13. Sensitivity of reliability-growth models to operational profile errors vs testing accuracy. Hand-translated from [109].

$OP1, OP2, OP3$. The inaccuracies in the operational profiles were very apparent after a small number of tests. However, above 100 tests randomly selected from each profile, the errors of the different profiles converged and after a 1000 tests, the effects of those errors were negligible.

The system studied by Pasquini was not a complex AI system. It is reasonable to assume that more complex systems would require exponentially more tests to compensate for errors in the operational profile. In many real-world situations, it is not practical to run a very large number of tests. Figure 14 shows the pre-launch test regime for the remote agent. The testing team had to share certain test rigs with numerous other development teams. As those test rigs got more elaborate (e.g. increasing fidelity to the actual in-flight software/hardware combination), they were slower to run and all the development teams, including the testers, got less and less access. Figure 14 shows that Remote Agent launched after 610 tests. Therefore, mere operational profile sampling may be inadequate for AI systems.

Various researchers have explored intelligent methods for sampling a program's input space. For example, Smith et.al. [111] characterized the inputs to the RAX planner as an n-dimensional parameter space and used orthogonal arrays to select a manageable number of cases that exercises all pair-wise combinations of parameter values. In this rig, an intelligent sample of RAX's input

| phase | test platform | | speed | |
	hardware	flight software	test:real	# tests
1	nil	some	35:1	269
2	nil	some	7:1	≈ 300
3	nil	some	1:1	≈ 20
4	nil	all	1:1	10
5	flight spare	all	1:1	10
6	flight spare	all	1:1	1
7	actual satellite	all	1:1	1
			total:	≈ 611

Fig. 14. Number of documented pre-launch tests for that RAX NASA satellite as it moved from software simulations (in phase 1) to some hardware test benches (phase 5 and 6), the finally to the actual mission (phase 7). From [110].

27

space required a number of tests that is logarithmic on the number of RAX's input parameters.

5.2 Stochastic Nondeterminism

Often, AI algorithms use some internal random choice to break out of dead-ends or to explore their models. Such stochastic methods can be remarkably effective. For example, stochastic choice has been observed to generate plans in AI systems one to two orders of magnitude bigger than ever done before with complete search [79, 112]. Yet another source of stochastic nondeterminism is **adaptation** when a system's behavior changes as a result of new experience. Adaptation is discussed later (see §6).

A concern with stochastic 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. Before a V&V analyst can certify such a wildly varying system, they might first need to *constrain* it to the point where definite predictions can be made about the system's behavior. A typical way to find these controlling constraints is to instrument the system so that internal choices become visible and controllable, effectively turning them into external choices.

Some empirical results suggest that if instrumentation is successful, then the subsequent learning of controllers may be quite simple. The *funnel* assumption is that within a nondeterministic program there exists a small number of assumptions that control which nondeterministic option the program will take [113]. This assumption appears in multiple domains, albeit using different terminology: *master-variables* in scheduling [114]; *prime-implicants* in model-based diagnosis [115] or machine learning [116], or fault-tree analysis [117]; *backbones* or back doors in satisfiability [118–120]; *dominance filtering* used in Pareto optimization of designs [121]; and the *minimal environments* in the ATMS [122]. Whatever the name, the intuition is the same: whatever happens in the larger space of the program is controlled by a few variables.

There is much empirical and analytical evidence that narrow funnels are found in many models. Menzies and Singh argue that the whole field of soft computing utilizes that assumption [113]. Later in this chapter, in §6.5, we will discuss the TAR3 learner. TAR3 was built as a test of the narrow funnel assumption. If such funnels exist, they would appear as variable bindings with a much higher frequency in preferred situations. TAR3 collects such frequency counts, normalizes and accumulates them. When applied to models with stochastic nondeterminism, TAR3 can find a very small set of treatments that constrain the nondeterminism, while selecting for better overall system behavior (see

the examples in [123, 124]).

The caveat here is that the nondeterministic system must be exercised sufficiently to give TAR3 enough data to learn the funnels. For more on that issue, see §6.2.

5.3   Concurrency Nondeterminism

At the conceptual level, AI algorithms are typically sequential (though possibly non-deterministic). In real applications, though, concurrency is often present. For example:

- A knowledge base receives and processes queries and updates concurrently.
- An intelligent robot controller responds to physical stimuli occurring at unpredictable times.
- A chess playing program is distributed over many processors for improving performance.
- Intelligent web services communicate with each other to negotiate some business contract.

Concurrency errors have been known to cause trouble in AI software. As we mentioned in the introduction, the error that caused a deadlock during the RAX experiment on-board Deep Space One was a typical concurrency error.

Concurrency is a particular form of non-determinism, and thus a major source of concern for V&V. Concurrency-related flaws typically result in *race conditions*, which manifest themselves only under specific timings of events and are very hard to detect and reproduce. This leads to critical systems being designed to minimize concurrency risks, using strictly sequenced chains or cycles of execution, or with tightly isolated concurrent components.

Model checking is well suited to verifying concurrent systems, and has been originally developed in that domain. As opposed to a conventional test-bed, the model checker has full control over the scheduling of concurrent executions and can therefore explore all non-deterministic alternatives.

Concurrency-related state-space explosion can be addressed using a particular class of optimization called *partial-order reduction (POR)*. This is based on the "diamond" principle ($\diamondsuit$) : if two concurrent operations are independent, then it does not matter in which order they are executed. This can be used to dramatically cut down the search space, provided that (sufficient conditions for) independence between operations can be asserted [125, 126]. Implementations exploiting this technique can constrain how the space is traversed [127], or constructed in the first place [128].

6 Adaptive AI Systems

In the language of the previous section, adaption is another source of nondetermism. Depending on the learning method, adapation can either be:

- *Stochastic nondeterminism* when some random choice is used in the learning, e.g. in genetic algorithms;
- *Environmental nondeterminism* when the learning changes according to the data passed into the system from the outside.

Regardless of the source of nondeterminism, the net result is that adaptive systems can adjust their own internal logic at runtime. Adaptive systems have the benefit that the software can fix itself. For example, a planning system might find a new method to generate better plans in less time. The problem with adaptive systems is that the adaption might render obsolete any pre-adaption certification.

There are many different adaptive systems such as decision explanation-based generalization [129, 130], chunking [129, 130], genetic algorithms [131], simulated annealing [132], tree learners [133], just to name a few. An example of decision tree adaptation is shown in Figure 15. In that figure, the decision tree on the right was generated from the data on the left. We see that we do not play golf on high-wind days when it might rain.

Despite there being many different learning methods, there exist several adaptive V&V criteria that can be applied to all learners: external validity, learning rates, data anomaly detectors, stability, and readability. These are discussed below.

Data   Learnt theory

```
#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
```
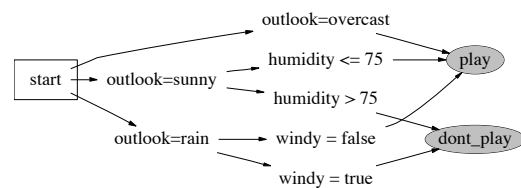

Fig. 15. Decision-tree learning. Classified examples (left) generate the decision tree (right).

6.1 External Validity

While the *method* of adaptation can vary the *goal* of the different methods is similar. Adaptation builds or tune some theory according to a set of new examples. Therefore, to validate any adaptive system, it is useful to start with validating that enough examples are available to support adequate adaptation.

When checking that enough data was available for the adaptation, good experimental technique is important. If the goal of adaptation is to generate models that have some useful future validity, then the learnt theory should be tested on data not used to build it. 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 [134]. 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 classification accuracy 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.

When assessing *different* adaptation mechanisms, the N-ways are repeated M times. In a 10-by-10 cross-validation study, the ordering of examples in a data set is randomized 10 times and a separate 10-way study is conducted for each of the ten random orderings. Such 10-by-10 study generates 100 training and 100 test sets and each of these should be passed to the different learners being studied. This will generate a mean and standard deviation on the classification accuracy for the learners being studied and these should be compared with a t-test with 10 degrees of freedom (and not 99, see [135]).

6.2 Learning Rates

Another important criteria for any learning is that of *learning rates*; i.e. how does the learnt theory change over time as more data is processed.
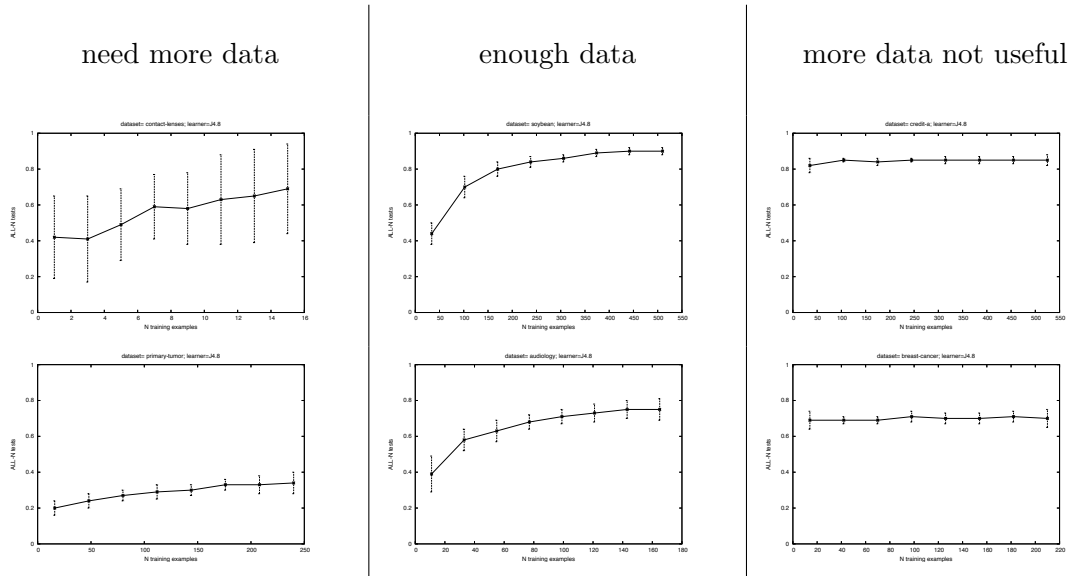
Fig. 16. *M-by-N* sequences studies on six data sets.

One way to study learning rates is via a *sequence study*. This is a variant on a 10-way study but this time training occurs on an increasing percentage of the available data. More precisely:

- The data is divided into N buckets.
- $\frac{X}{N}$-th of the data for $X \in \{1, 2, \ldots N - 1\}$ is used for training;
- The remaining $\frac{N-X}{N}$ of the data is used for testing.

Note that the sequence stops at $N - 1$ since training on $\frac{N}{N}$ of the data would leave nothing for the test suite $(1 - \frac{N}{N} = 0)$. In a *M-by-N sequence* study, the above process is repeated M times with the ordering of the example data randomized before each N-sequence study. The mean and standard deviation of the accuracy at each N value is reported. M-by-N sequence studies let us check how early learning stabilizes as more data is used in the training.

Figure 16 show results of a M-by-N sequence study for six data sets using $<M=N=20>$. In this study, the same decision tree learner was used as seen in Figure 15. The vertical-axis ranges from zero to 100% accuracy. The horizontal-axis shows the training set growing in size. The whiskers in Figure 16 show ±1 standard deviation of the 20 experiments conducted at a particular N value.

Figure 16 is divided into three groups. On the left-hand column are data sets where the adaptation needs more examples than what is currently available:

- The standard deviation in the classification accuracies of the top-left are very large. Clearly, in that data set, training is insufficient for stable future predictions.
- The mean classification of the bottom-left plot is very low (less than 40%),

32

even though all the available training data has been passed to the learner. The accuracy improves as the training set grows but much more data would be required before a V&V analyst could check if adaptation is performing adequately in that domain.

The middle column of Figure 16 shows examples were the learning improves dramatically during the sequence, rises to medium or high level accuracies, then plateaus *before* we exhaust all the data in this domain. The conclusion from these two plots would be that we are collecting adequate amounts of data in this domain and that the benefits of further data collection might be quite low.

The right-hand column of Figure 16 shows examples where, very early in the sequence, the adaption reaches medium to high levels and does not improve as more data is supplied. For these domains, a V&V analyst would conclude that *too much* data was collected in this domain.

Note that the the learning never achieves 100% accuracy. Making some errors is fundamental to the learning task. If something adapts perfectly to past example, then it can *over-fit* to the data. Such over-fitted adaption can obsess on minor details in the training data and can perform poorly on future examples if those future examples contain trivial differences to the training example.

Studying the shape of these sequence learning curves is an important V&V technique for adaptive real-time systems. Consider a real-time controller that must adapt to sudden changes to an aircraft; e.g. the flaps on the left wing have suddenly frozen. The V&V task here would be to predict the shape of these curves in the space of possible future input examples. Such predictions would inform questions such as "would the controller adapt fast enough to save the plane?".

There are many ways to explore the space of possible future input examples. One way is the sequence study shown above: the ordering of existing data is randomized many times and, each time, the learner learns for that sequence of data. Another way would be to generate artificial examples from distributions seen in current data or from known distributions in the environment. Yet another way is to define an *anomaly detector* which triggers if newly arriving data is different to data which the learner has previously managed [136, 137].

6.3 Data Anomaly Detectors

Figure 17 shows a general framework for wrapping a learner in anomaly detectors. When new data arrives a *pre-filter* could reject the new input if it is too anomalous. Any accepted data is then passed to the adaptive module.
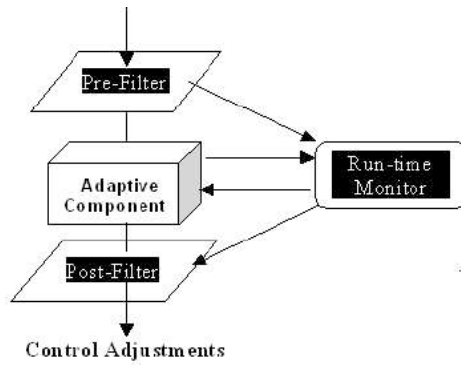
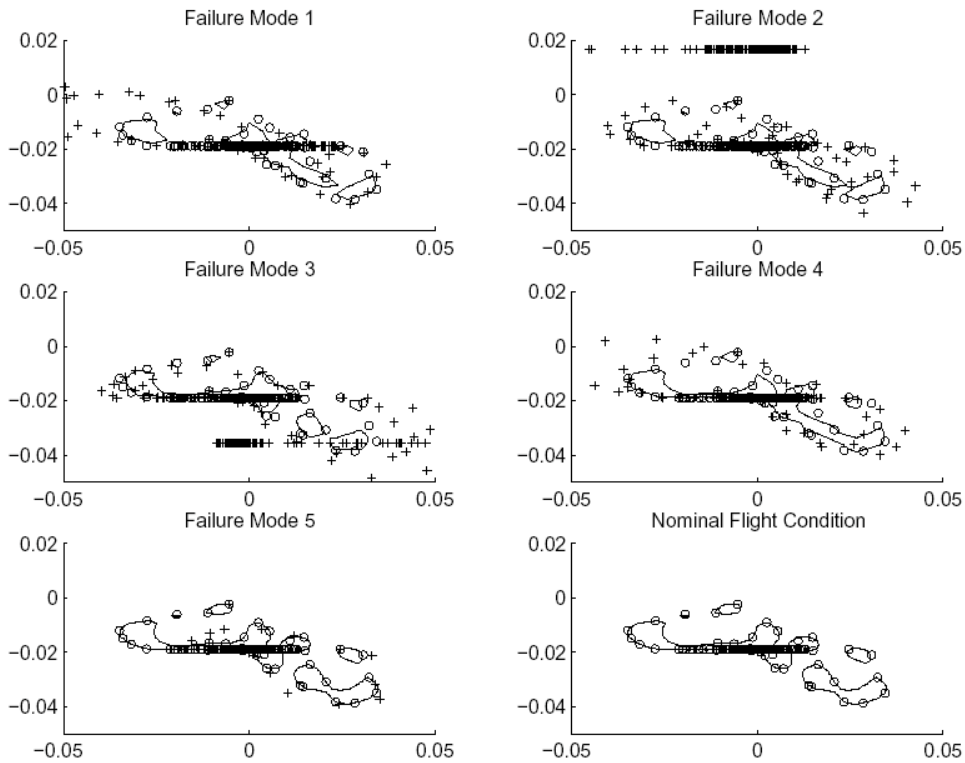Fig. 17. Applications of anomaly detection. From [138].

Fig. 18. Identifying anomalous data. From [138].

This module offers some conclusion; e.g. adds a classification tag. The output of the adaption module therefore contains more structure than the input example so a second *post-filter* anomaly detector might be able to recognize unusual output from the learner. If so, then some repair action might be taken to (e.g.) stop the output from the learner effecting the rest of the system. In effect, Figure 17 is like an automated V&V analyst on permanent assignment, watching over the adaptive device.

Figure 18 shows an example of a pre-filter anomaly detector. That figure is a representation of high dimensional data collected from nominal and five off-nominal modes from a flight simulator passed through a *support vector*

machine. Support vector machines recognizing the borderline examples that distinguish between different classes of data. Such machines run very quickly and scale very well to data sets with many attributes. The crosses in Figure 18 show training examples and the closed lines around the circles represent the border between "familiar" and "anomalous" data. Our learner should be able to handle failure mode 5 since data from that mode falls mostly in the "familiar" zone. However, Failure Mode 2 worries us the most since much of the data from that mode falls well outside the "familiar" zone.

Figure 18 only shows anomaly *detection*. After detection, some repair action is required. The precise nature of the repair action is domain-specific. For example, in the case of automatic flight guidance systems, the repair action might be either "pass control to the human pilot" or, in critical flight situations, "hit the eject button".

6.4 Stability

Apart from studying the above, a V&V analyst for an adaptive system might also care to review the *results* of the learning. When a V&V analyst is reading the output of a learner, one important property of the learning is *stability*; i.e. the output theory is the same after different runs of the learner.

Not all learners are stable. For example, decision tree learners like the one used in Figure 15 are *brittle*; i.e. minor changes to the input examples can result in very different trees being generated [139]. Also, learners that use random search can leap around within the learning process. For example, genetic programming methods randomly *mutate* a small portion of each generation of their models. Such random mutations may generate different theories on different runs. Therefore, if an output theory is to be manually inspected, it is wise to select learners that generate *stable conclusions*. For example, Burgess and Lefley report [140] that in ten runs of a neural net and genetic programming system trying to learn software cost estimation models, the former usually converged to the same conclusion while the latter could generate different answers with each run.

6.5 Readability

Lastly, another interesting V&V criteria for a learner is *readability*; i.e. assessing if the the output from the learner is clear and succinct.

Not all learners generate readable output. For example, neural net and Bayes classifiers store their knowledge numerical as a set of internal weights or table
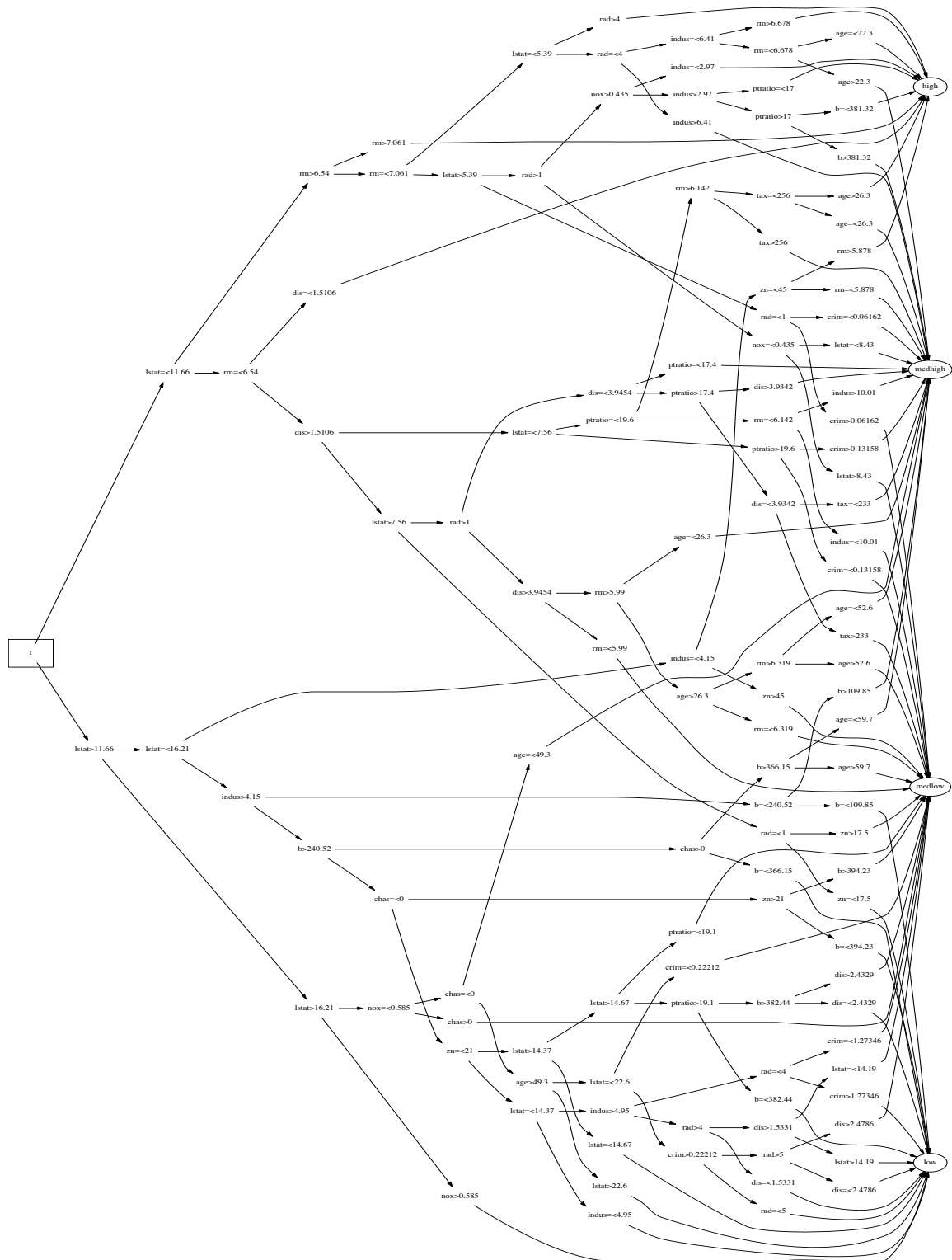
Fig. 19. A learnt decision tree. Classes (right-hand-side), top-to-bottom, are "high", "medhigh", "medlow", and "low" This indicates median value of owner-occupied homes in $1000's. Decision tree learnt from the 506 cases in HOUSING example set from the UC Irvine repository (http://www.ics.uci.edu/~mlearn/).
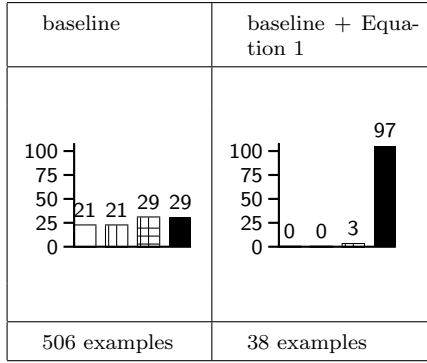
36

baseline	baseline + Equation 1
506 examples	38 examples

Fig. 20. Treatments learnt by TAR3 from the data used in Figure 19. That dataset had the class distribution shown left-hand-side. Actions that most increase housing values are shown in the right column.

values that are opaque to the human reader. Research in neural net validation often translate the neural net into some other form to enable visualization and inspection (e.g. [141]). Even learners that use symbolic, not numeric, representations can generate unreadable output. For example, Figure 19 was learnt from 506 examples of low, medium low, medium high, and high quality houses in Boston. While a computer program could apply the learnt knowledge, it is virtually unreadable by a human.

Some learners are specifically designed to generate succinct, simple, readable output. The TAR3 *treatment learner* [123,124,142–147] seek the *smallest* number of attribute ranges that *most* select for preferred classes and *least* select for undesired classes. Treatments are like constraints which, if applied to the test 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. From the same data as used in Figure 19, TAR3 learns Equation 1:

$$best = (6.7 \leq RM < 9.8) \wedge (12.6 \leq PTRATION < 15.9) \tag{1}$$

That is, good houses can be found by favoring houses with 7 to 9 rooms in suburbs with a parent-teacher ratio in the local schools of 12.6 to 15.9; The effects on the distribution of selected houses by Equation 1 are shown in Figure 20. In the raw data, the quality of houses was evenly distributed. In the treated data, most of the selected houses (97%) are now high quality.

Which is a better representation of the data? The details of Figure 19 or the high-level summary of Equation 1? That choice is up to the reader but if they are a busy V&V analyst struggling to understand an adaptive system, they might be attracted to the succinctness of Equation 1.

7 Conclusion

This chapter offers a diverse set of technologies to support the V&V of adaptive systems. If there is a single conclusion from such a diverse survey is that developers should not be afraid of AI. AI is useful. AI can deliver more functionality for reduced cost. AI should and will be used more widely.

On the other hand, AI systems have features that make them hard to check using conventional methods. Nevertheless, there are enough alternative readily-available methods that enable the V&V of AI software:

- AI software can be *complex*. Powerful methods like model checkers and static analysis tools has evolved in the software engineering area to simplify the task of checking such complex systems. Many of those methods can be applied to AI systems.
- The *model-based* nature of AI systems makes it easier for V&V analysts to extract features from a system and this can be exploited in several ways.
- Sometimes, the inference associated with those models falls into one of a small set of commonly-used *knowledge-level* problem solving methods and specialized V&V techniques are appropriate for different problem solving methods.
- AI systems can be *nondeterministic*. Different methods apply for the V&V of nondeterministic systems depending on the nature of the nondeterminism (environmental, concurrent, stochastic).
- Adaptive systems are an extreme for of stochastic nondeterministic systems. The V&V of adaptive systems can apply such criteria like external validity, learning date, stability, etc.

Contributors

Prof. Tim Menzies has recently been appointed associated research professor at Portland State University in the United States, and has been working with NASA on software quality issues since 1998. He has a CS degree and a PhD from the University of New South Wales. His recent research concerns modelling and learning with a particular focus on light weight modeling methods. His doctoral research aimed at improving the validation of, possibly inconsistent, knowledge-based systems in the QMOD specification language. He also has worked as a object-oriented consultant in industry and has authored over 150 publications and served on numerous conference and workshop programs and well as guest editor of journal special issues. He can be reached at tim@menzies.us

Prof. Charles Pecheur has recently been appointed assistant professor at Université Catholique de Louvain in Belgium, and has been a Staff Scientist for the Research Institute for Advanced Computer Science (RIACS) at NASA Ames for the last six years. He has an EE degree and a PhD from the University of Liège, Belgium. His recent research at NASA concerns the formal verification and validation of NASA's critical software systems, with a particular focus on model-based autonomous controllers and diagnosis. His doctoral research aimed at improving data type definitions in the LOTOS specification language. He also performed a couple applied formal verification projects at INRIA Rhône-Alpes in France. He has authored more than twenty publications and served on several conference and workshop programs. He can be reached at pecheur@info.ucl.ac.be.

References

[1] T. Menzies, 21st century AI: proud, not smug, Editorial, IEEE Intelligent Systems, Special Issue on AI PrideAvailable from `http://menzies.us/pdf/03aipride.pdf`.

[2] D. Bernard, et al., Spacecraft autonomy flight experience: The DS1 Remote Agent experiment, in: Proceedings of the AIAA 1999, Albuquerque, NM, 1999.

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

[4] A. Preece, Principles and practice in verifying rule-based systems, The Knowledge Engineering Review 7 (1992) 115–141.

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

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

[7] A. Preece, R. Shinghal, Verifying knowledge bases by anomaly detection: An experience report, in: ECAI '92, 1992.

[8] G. Prakash, E. Subramanian, 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), 1991, pp. 3–8.

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

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

[11] P. Meseguer, Verification of multi-level rule-based expert systems, in: Proceedings of the Ninth National Conference on Artificial Intelligence, 1991, pp. 323–328.

[12] A. Ginsberg, S. Weiss, P. Politakis, Automatic knowledge base refinement for classification systems, Artificial Intelligence 35 (1988) 197–226.

[13] 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), 1991, pp. 22–27.

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

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

[16] P. Grogono, A. Batarekh, A. Preece, R. Shinghal, C. Suen, Expert system evaluation techniques: A selected bibliography., Expert Systems (1992) 227–239.

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

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

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

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

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

[22] R. O. R.M., O. Balci, E. Smith, Validating expert system performance, IEEE Expert 87 (1987) 81–89.

[23] J. Rushby, Quality measures and assurance for AI software, sRI-CSL-88-7R, SRI Project 4616 (1988).

[24] N. Zlatereva, A. Preece, State of the art in automated validation of knowledge-based systems, Expert Systems with Applications 7 (1994) 151–167.

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

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

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

[28] J. Bachant, J. McDermott, R1 Revisited: Four Years in the Trenches, AI Magazine (1984) 21–32.

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

[30] S. McConnell, The best influences on software engineering, IEEE SoftwareAvailable from `www.computer.org/software/so2000/pdf/s1010.pdf`.

[31] B. Boehm, Safe and simple software cost analysis, IEEE Software (2000) 14–17Available from `http://www.computer.org/certification/beta/Boehm_Safe.pdf`.

[32] F. Shull, V. B. ad B. Boehm, A. Brown, P. Costa, M. Lindvall, D. Port, I. Rus, R. Tesoriero, M. Zelkowitz, What we have learned about fighting defects, in: Proceedings of 8th International Software Metrics Symposium, Ottawa, Canada, 2002, pp. 249–258, available from `http://fc-md.umd.edu/fcmd/Papers/shull_defects.ps`.

[33] J. Rushby, Disappearing formal methods, in: High-Assurance Systems Engineering Symposium, Association for Computing Machinery, Albuquerque, NM, 2000, pp. 95–96.
URL `http://www.csl.sri.com/~rushby/hase00.html`

[34] G. Brat, D. Giannakopoulou, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, A. Venet, W. Visser, Experimental evaluation of verification and validation tools on martian rover software, in: CMU/SEI Software Model Checking Workshop, Pittsburg, USA, 2003, extended version to appear in Formal Methods in System Design.

[35] D. Drusinsky, The Temporal Rover and the ATG Rover, in: SPIN Model Checking and Software Verification, Vol. 1885 of LNCS, Springer, 2000, pp. 323–330.

[36] S. Savage, M. Burrows, G. Nelson, P. Sobalvarro, T. Anderson, Eraser: A dynamic data race detector for multithreaded programs, ACM Transactions on Computer Systems 15 (4) (1997) 391–411.

[37] K. Havelund, G. Rosu, Monitoring Java programs with Java PathExplorer, in: First Workshop on Runtime Verification (RV'01), Vol. 55 of Electronic Notes in Theoretical Computer Science, Paris, France, 2001.

[38] C. Artho, D. Drusinsky, A. Goldberg, K. Havelund, M. Lowry, C. Pasareanu, G. Roşu, W. Visser, Experiments with Test Case Generation and Runtime Analysis, in: E. Börger, A. Gargantini, E. Riccobene (Eds.), Abstract State Machines (ASM'03), Vol. 2589 of LNCS, Springer, 2003, pp. 87–107, version to appear in the journal of Theoretical Computer Science.

[39] H. Barringer, A. Goldberg, K. Havelund, K. Sen, Rule-based runtime verification, in: Fifth International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI'04), Venice, Italy, 2004.

[40] F. Nielson, H. R. Nielson, C. Hankin, Principles of Program Analysis, Springer, 1999.

[41] A. Deutsch, Next generation testing tools for embedded applications, white paper, PolySpace Technologies.
URL http://www.polyspace.com/docs/Static_Verification_paper%.pdf

[42] A. Venet, G. Brat, Precise and efficient static array bound checking for large embedded c programs, in: Proceedings of the International Conference on Programming Language Design and Implementation (PLDI04), ACM Press, Washington DC, USA, 2004, pp. 231–242.

[43] J.-P. Queille, J. Sifakis, Specification and verification of concurrent systems is CESAR, in: International Symposium on Programming, Vol. 137 of LNCS, 1982, pp. 337–351.

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

[45] G. Holzmann, Software model checking with spin (in this volume) (2005).

[46] G. J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (5).

[47] J. R. Burch, E. M. Clarke, K. L. McMillan, D. L. Dill, J. Hwang, Symbolic model checking: 10^{20} states and beyond, Information and Computation 98 (2) (1992) 142–170.

[48] A. Cimatti, E. Clarke, F. Giunchiglia, M. Roveri, NuSMV: a new symbolic model verifier, in: Proc. of International Conference on Computer-Aided Verification, 1999.

[49] E. M. Clarke, O. Grumberg, D. Peled, Model Checking, Mit Press, 1999.

[50] B. Bérard, M. Bidoit, A. Finkel, F. Laroussinie, A. Petit, L. Petrucci, P. Schnoebelen, Systems and Software Verification: Model-Checking Techniques and Tools, Springer, 2001.

[51] H. Garavel, M. Jorgensen, R. Mateescu, C. Pecheur, M. Sighireanu, B. Vivien, Cadp'97 – status, applications and perspectives, in: I. Lovrek (Ed.), Proceedings of the 2nd COST 247 International Workshop on Applied Formal Methods in System Design (Zagreb, Croatia), 1997.

[52] J. Corbett, M. Dwyer, J. Hatcliff, S. Laubach, C. Pasareanu, Robby, H. Zheng, Bandera: Extracting finite-state models from java source code, in: Proceedings ICSE2000, Limerick, Ireland, 2000, pp. 439–448.

[53] W. Visser, K. Havelund, G. Brat, S. Park, Model checking programs, in: Proceedings of the IEEE International Conference on Automated Software Engineering, 2000, pp. 3–12.

[54] C. Heitmeyer, Software cost reduction, in: J. J. Marciniak (Ed.), Encyclopedia of Software Engineering, 2002, available from `http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.p%df`.

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

[56] C. Heitmeyer, B. Labaw, D. Kiskis, Consistency checking of SCR-style requirements specifications, in: International Symposium on Requirements Engineering, York, England , March 26-27, 1995.

[57] T. Menzies, J. Powell, M. E. Houle, Fast formal analysis of requirements via 'topoi diagrams', in: ICSE 2001, 2001, available from `http://menzies.us/pdf/00fastre.pdf`.

[58] C. Pecheur, R. Simmons, From Livingstone to SMV: Formal verification for autonomous spacecrafts, in: S. Verlag (Ed.), Proceedings of First Goddard Workshop on Formal Approaches to Agent-Based Systems, Vol. 1871 of Lecture Notes in Computer Science, NASA Goddard, 2000.

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

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

[61] C. Pasareanu, W. Visser, Verification of java programs using symbolic execution and invariant generation, in: Proceedings of SPIN 2004, Vol. 2989 of LNCS, Barcelona, Spain, 2004.

[62] A. Biere, A. Cimatti, E. M. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: R. Cleaveland (Ed.), Proceedings of the Fifth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS '99), Vol. 1579 of LNCS, Springer-Verlag, 1999, pp. 193–207.

[63] E. M. Clarke, O. Grumberg, D. E. Long, Model checking and abstraction, ACM Trans. Program. Lang. Syst. 16 (5) (1994) 1512–1542.

[64] S. Graf, H. Saidi, Verifying invariants using theorem proving, in: Conference on Computer Aided Verification CAV'96, Vol. 1102 of LNCS, Springer-Verlag, 1996.

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

[66] C. Ip, D. Dill, Better verification through symmetry, Formal Methods in System Design 9 (1/2) (1996) 41–75.

[67] E. Clark, D. E. Long, Compositional model checking, in: Fourth Annual Symposium on Logic in Computer Science, 1989.

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

[69] D. Giannakopoulou, C. Pasareanu, H. Barringer, Assumption generation for software component verification, in: Proc. of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, UK, 2002.

[70] K. Havelund, M. Lowry, S. Park, C. Pecheur, J. Penix, W. Visser, J. L. White, Formal analysis of the remote agent before and after flight, in: Proceedings of 5th NASA Langley Formal Methods Workshop, Williamsburg, Virginia, 2000.

[71] D. Giannakopoulou, C. Pasareanu, H. Barringer, Assumption generation for software component verification, in: Proc. of the 17th IEEE International Conference on Automated Software Engineering (ASE 2002), Edinburgh, UK, 2002.

[72] S. Owre, S. Rajan, J. M. Rushby, N. Shankar, M. K. Srivas, Pvs: Combining specification, proof checking, and model checking, in: Proceedings of the 8th International Conference on Computer Aided Verification, Springer-Verlag, 1996, pp. 411–414.

[73] J. Crow, B. Di Vito, Formalizing space shuttle software requirements: four case studies, ACM Trans. Softw. Eng. Methodol. 7 (3) (1998) 296–332.

[74] S. Owre, J. Rushby, N. Shankar, Integration in PVS: tables, types, and model checking, in: E. Brinksma (Ed.), Tools and Algorithms for the Construction and Analysis of Systems TACAS '97, no. 1217 in Lecture Notes in Computer Science, Springer-Verlag, Enschede, The Netherlands, 1997, pp. 366–383.
URL http://www.csl.sri.com/papers/tacas97/

[75] L. de Moura, S. Owre, H. Ruess, J. Rushby, N. Shankar, The ICS decision procedures for embedded deduction (Jul. 2004).

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

[77] J. A. Kalman, Automated Reasoning with OTTER, Rinton Press, 2002.

[78] M. Stickel, A prolog technology theorem prover: a new exposition and implementation in prolog, Theoretical Computer Science 104 (1992) 109–128.

[79] H. Kautz, 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, AAAI Press / MIT Press, Menlo Park, 1996, pp. 1194–1201, available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.

[80] P. P. Nayak, B. C. Williams, Fast context switching in real-time propositional reasoning, in: Proceedings of AAAI-97, 1997, available from `http://ack.arc.nasa.gov:80/ic/projects/mba/papers/aaai97.ps`.

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

[82] A. Groce, W. Visser, Heuristic model checking for java programs, in: Proceedings of SPIN 2002, Grenoble, France, 2002.

[83] A. Cimatti, M. Roveri, Conformant Planning via Symbolic Model Checking, Journal of Artificial Intelligence Research (JAIR) 13 (2000) 305–338.

[84] B. C. Williams, P. P. Nayak, A model-based approach to reactive self-configuring systems, in: Proceedings of AAAI-96, 1996.

[85] D. Clancy, W. Larson, C. Pecheur, P. Engrand, C. Goodrich, Autonomous control of an in-situ propellant production plant, in: Proceedings of Technology 2009 Conference, Miami, 1999.

[86] A. Cimatti, C. Pecheur, R. Cavada, Formal verification of diagnosability via symbolic model checking, in: Proceedings of IJCAI'03, Acapulco, Mexico, 2003.

[87] M. Feather, 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, 1999, pp. 63–72, available from `http://www-aig.jpl.nasa.gov/public/planning/papers/oracles-ase.pdf`.

[88] N. Muscettola, P. P. Nayak, B. Pell, B. Williams, Remote agent: To boldly go where no AI system has gone before, Artificial Intelligence 103 (1-2) (1998) 5–48.

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

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

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

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

[93] G. Schreiber (Ed.), Knowledge Engineering and Management : The CommonKADS Methodology, MIT Press, 1999.

[94] B. Swartout, 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.

[95] Y. Gil, E. Melz, Explicit representations of problem-soving strategies to support knowledge acquisition, in: Proceedings AAAI' 96, 1996.

[96] J. Angele, D. Fensel, R. Studer, Domain and task modelling in mike, in: A. S. et.al. (Ed.), Domain Knowledge for Interactive System Design, Chapman & Hall, 1996.

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

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

[99] D. Marques, G. Dallemagne, G. Kliner, J. McDermott, D. Tung, Easy Programming: Empowering People to Build Their own Applications, IEEE Expert (1992) 16–29.

[100] W. Clancey, Model Construction Operators, Artificial Intelligence 53 (1992) 1–115.

[101] L. Steels, Components of Expertise, AI Magazine 11 (1990) 29–49.

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

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

[104] E. Motta, Z. Zdrahal, Parametric design problem solving, in: Proceedings of the 10th Banff Knowledge Acquisition for Knowledge-Based System Workshop, 1996.

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

[106] A. V. de Brug, J. Bachant, J. McDermott, The Taming of R1, IEEE Expert (1986) 33–39.

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

[108] J. Musa, A. Iannino, K. Okumoto, Software Reliability: Measurement, Prediction, Application, McGraw Hill, 1987.

[109] A. Pasquini, N. C. A, P. Matrella, Sensitivity of reliability-growth models to pperational profile errors vs testing accuracy, IEEE Transactions on Reliability 45 (4) (1996) 531–540.

[110] N. A. R. C. N. Muscettola, Personal communication (2000).

[111] B. Smtih, M. Feather, N. Muscettola, Challenges and methods in validating the remote agent planner, in: Proceedings of the Fifth International Conference on Artificial Intelligence Planning Systems (AIPS-2000), 2000, available from `http://www-aig.jpl.nasa.gov/public/home/smith/publications.html`.

[112] B. Selman, H. Levesque, D. Mitchell, A new method for solving hard satisfiability problems, in: AAAI '92, 1992, pp. 440–446.

[113] T. Menzies, H. Singh, Many maybes mean (mostly) the same thing, in: M. Madravio (Ed.), Soft Computing in Software Engineering, Springer-Verlag, 2003, available from `http://menzies.us/pdf/03maybe.pdf`.

[114] J. Crawford, A. Baker, Experimental results on the application of satisfiability algorithms to scheduling problems, in: AAAI '94, 1994.

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

[116] R. Rymon, An SE-tree based characterization of the induction problem, in: International Conference on Machine Learning, 1993, pp. 268–275.

[117] R. Lutz, 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`.

[118] A. Parkes, Lifted search engines for satisfiability (1999). URL `citeseer.nj.nec.com/parkes99lifted.html`

[119] J. Singer, I. P. Gent, A. Smaill, Backbone fragility and the local search cost peak, Journal of Artificial Intelligence Research 12 (2000) 235–270. URL `citeseer.nj.nec.com/singer00backbone.html`

[120] R. Williams, C. Gomes, B. Selman, Backdoors to typical case complexity, in: Proceedings of IJCAI 2003, 2003, `http://www.cs.cornell.edu/gomes/FILES/backdoors.pdf`.

[121] J. Josephson, B. Chandrasekaran, M. Carroll, N. Iyer, B. Wasacz, 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`.

[122] J. DeKleer, An Assumption-Based TMS, Artificial Intelligence 28 (1986) 163–196.

[123] M. Feather, 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://menzies.us/pdf/02re02.pdf`.

[124] T. Menzies, E. Chiang, M. Feather, Y. Hu, J. Kiper, Condensing uncertainty via incremental treatment learning, in: T. M. Khoshgoftaar (Ed.), Software Engineering with Computational Intelligence, Kluwer, 2003, available from `http://menzies.us/pdf/02itar2.pdf`.

[125] A. Valmari, A stubborn attack on state explosion, in: R. P. Kurshan, E. M. Clarke (Eds.), Proceedings of the 2nd Workshop on Computer-Aided Verification (Rutgers, New Jersey, USA), Vol. 3 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science, AMS-ACM, 1990, pp. 25–42.

[126] P. Godefroid, Partial-Order Methods for the Verification of Concurrent Systems – An Approach to the State-Explosion Problem, Vol. 1032 of Lecture Notes in Computer Science, Springer-Verlag, 1996.

[127] 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 Verification, July 24-26, 1996, 1997, pp. 289–303.

[128] F. Schneider, S. Easterbrook, J. Callahan, G. Holzmann, Validating requirements for fault tolerant systems using model checking, in: 3rd IEEE International Conference On Requirements Engineering, 1998.

[129] F. van Harmelen, A. Bundy, Explanation-based generalisation = partial evaluation, Artificial Intelligence (1988) 401–412.

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

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

[132] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by simulated annealing, Science, Number 4598, 13 May 1983 220, 4598 (1983) 671–680. URL `citeseer.nj.nec.com/kirkpatrick83optimization.html`

[133] R. Quinlan, C4.5: Programs for Machine Learning, Morgan Kaufman, 1992, iSBN: 1558602380.

[134] K. Srinivasan, D. Fisher, Machine learning approaches to estimating software development effort, IEEE Trans. Soft. Eng. (1995) 126–137.

[135] R. Bouckaert, Choosing between two learning algorithms based on calibrated tests, 2003, available from `http://www.cs.pdx.edu/~timm/dm/10x10way`.

[136] Y. Liu, T. Menzies, B. Cukic, Data sniffing - monitoring of machine learning for online adaptive systems, in: IEEE Tools with AI, 2002, available from `http://menzies.us/pdf/03datasniffing.pdf`.

[137] Y. Liu, T. Menzies, B. Cukic, Detecting novelties by mining association rules, 2003, available from `http://menzies.us/pdf/03novelty.pdf`.

[138] Y. Liu, S. Gururajan, B. Cukic, T. Menzies, M. Napolitano, Validating an online adaptive system using svdd, in: IEEE Tools with AI, 2003, available from `http://menzies.us/pdf/03svdd.pdf`.

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

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

[141] B. Taylor, Development of methodologies for iv&v neural networks: Literature survey of current v&v technology, available from `http://sarpresults.ivv.nasa.gov/ViewResearch/914/11.jsp` (2004).

[142] Y. Hu, Treatment learning: Implementation and application, masters Thesis, Department of Electrical Engineering, University of British Columbia (2003).

[143] T. Menzies, Y. Hu, Reusing models for requirements engineering, in: First International Workshop on Model-based Requirements Engineering, 2001, available from `http://menzies.us/pdf/01reusere.pdf`.

[144] T. Menzies, Y. Hu, Constraining discussions in requirements engineering, in: First International Workshop on Model-based Requirements Engineering, 2001, available from `http://menzies.us/pdf/01lesstalk.pdf`.

[145] T. Menzies, Y. Hu, Agents in a wild world, in: C. Rouff (Ed.), Formal Approaches to Agent-Based Systems, book chapter, 2002, available from `http://menzies.us/pdf/01agents.pdf`.

[146] T. Menzies, Y. Hu, Just enough learning (of association rules): The TAR2 treatment learner, in: Artificial Intelligence Review (to appear), 2004, available from `http://menzies.us/pdf/02tar2.pdf`.

[147] T. Menzies, Y. Hu, Data mining for very busy people, in: IEEE Computer, 2003, available from `http://menzies.us/pdf/03tar2.pdf`.

[148] T. Menzies, E. Sinsel, T. Kurtz, Learning to reduce risks with cocomo-ii, in: Workshop on Intelligent Software Engineering, an ICSE 2000 workshop, and

NASA/WVU Software Research Lab, Fairmont, WV, Tech report # NASA-IVV-99-027, 1999, 2000, available from `http://menzies.us/pdf/00wise.pdf`.