Annals of Software Engineering, special issue on Computational Intelligence (submitted)

Condensing Uncertainty via Incremental Treatment Learning

Tim Menzies[1], **Eliza Chiang**[2], **Martin Feather**[3], **Ying Hu**[2], **James D. Kiper**[4]

[1] Lane Department of Computer Science, University of West Virginia, PO Box 6109, Morgantown, WV, 26506-6109, USA; `http://tim.menzies.com`; e-mail: `tim@menzies.com`
[2] Electrical & Computer Engineering University of British Columbia; e-mail: `yingh@ece.ubc.ca;echiang@interchange.ubc.ca`
[3] Jet Propulsion Laboratory, California Institute of Technology; e-mail: `Martin.S.Feather@Jpl.Nasa.Gov`
[4] Computer Science & Systems Analysis, Miami University; e-mail: `kiperjd@muohio.edu`

WP ref: 02/annals. April 10, 2002

Abstract Models constrain the range of possible behaviors defined for a domain. When parts of a model are uncertain, the possible behaviors may be a *data cloud*: i.e. an overwhelming range of possibilities that bewilder an analyst. Faced with large data clouds, it is hard to demonstrate that any particular decision leads to a particular outcome.

Even if we can't make definite decisions from such models, it is possible to find decisions that reduce the variance of values within a data cloud. Also, it is possible to change the range of these future behaviors such that the cloud condenses to some improved mode.

Our approach uses two tools. Firstly, a model simulator is constructed that knows the range of possible values for uncertain parameters. Secondly, the TAR2 treatment learner uses the output from the simulator to incrementally learn better constraints. In our *incremental treatment learning* cycle, users review newly discovered treatments before they are added to a growing pool of constraints used by the model simulator.

1 Introduction

Often, during early lifecycle decision making in software engineering, analysts know the *space* of possibilities, but not the *constraints* on that space. For example:

- They might know qualitatively that the *more* shared data, the *less* modifiable is a software system. However, they may not know the exact quantitative values for this relationship.
- Their experience might tell them that their source lines of code estimates are inaccurate by 50%.

What are our analysts to do? One possibility is to demand more budget and time to perform further analysis which removes these uncertainties. For example, metrics collection programs might be commenced to collect values for uncertain parameters. Elsewhere, we have documented the impressive results that can come from such a methodology/process [35].

When elaborate metrics collection is too expensive however, computational intelligence methods may be useful. If domain experts can offer a rough description of how (e.g.) variable A effects variable B, then fuzzy logic methods [14, 49] can be used to perform inference over the model, perhaps using the methods of Jahnke et.al. [22]. If the model represents a situation for which we have historical data, then genetic algorithms can be used to mutate the current model towards a model that best covers the historical data [2]. Alternatively, we could throw away the current model and use the historical data to auto-generate a new neural net model [44].

The premise of this paper is *metrics starvation*; i.e. situations in which we can access neither the relevant domain expertise required for fuzzy logic, nor the historical data required for genetic algorithms or neural nets. Our experience is that metrics starvation is common. For example, the majority of software development organizations do not conduct systematic data collection. As evidence for this, consider the Software Engineering Institute's capability maturity model (CMM [39]), which categorizes software organizations into one of five levels based on the maturity of their software development process. Below CMM level 4, there is no systematic data collection. Below CMM level 3, there is not even a written definition of the software process. Many organizations exist below CMM level 3[1]. Hence reliable data on SE projects is scarce, or hard to interpret.

However, a lack of systematic data does not mean that no inferences can be made about some software development process. If we can't constrain the range of model behavior with domain metrics, we can still make decisions by surveying the range of possible behaviors. Suppose we have a model expressing what is known within a domain. If we are uncertain over parts of that model, then we might supplied *ranges* for those uncertain parameters. When we run this model, if we ever require some uncertain parameter, we might *select* and *cache* a value for that parameter, based on the supplied ranges. To survey the range of possible behaviors, we just re-run the model many times, taking care to clear the cache

[1] Personal communication with SEI researchers.

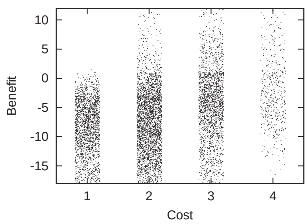


Figure 1.i: Cloud1 (from §4.1)

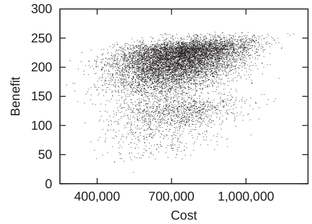


Figure 1.ii: Cloud2 (from §4.3)

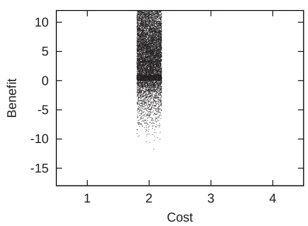


Figure 1.iii: Cloud1, condensed

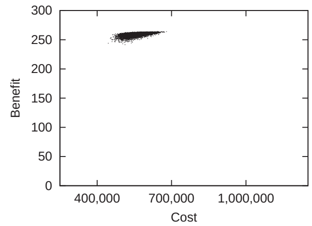


Figure 1.iv:Cloud2, condensed

Fig. 1 Examples of condensing clouds. The right-hand model's cost values are continuous while the left-hand model has discrete costs.

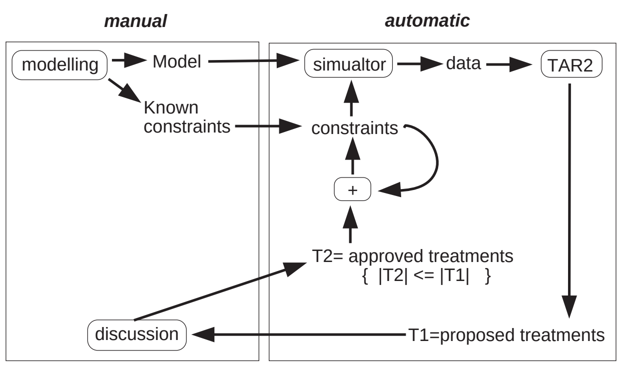


Fig. 2 Incremental treatment learning

between each run. Many variants on this scheme have been discussed in the literature. For example:

- This scheme is the same as Monte Carlo simulations when uncertain parameters are just system inputs.
- This scheme is the same as abductive inference [24] where the uncertain parameters are truth assignments to assumptions within the model, and some global invariant checking executes before a new value is assigned.

The advantage of this "select and cache" method is that the range of possible behaviors can be explored *without* expensive further analysis. The disadvantage of this approach is *data clouds*: an overwhelming amount of data that clouds and confuses the issues. For example, Figure 1.i and Figure 1.ii show data clouds generated from case studies described later in this paper. In these figures, each mark represents the *cost* and *benefits* associated with a set of decisions about the structure of a software project. Note the large variance in the possible cost and benefits from the different possible decisions.

Faced with such large data clouds, it is hard to demonstrate that any particular decision leads to a particular outcome. What is required is some method for condensing these clouds of uncertainty *without* expensive data collection for all the uncertain parameters. Ideally, condensation methods should be *minimal*; i.e. they require a commitment to only a small portion of the uncertain variables within a model.

This papers experiments with minimal condensation using the TAR2 *treatment learner* [16, 21, 28–31]. A treatment learner seeks the *least number* of attribute ranges that *most differentiate* between desired and undesired behavior. Figure 2 shows how TAR2 can be applied incrementally to explore data clouds. A simulator executes a model generated by some manual modelling process. TAR2 reduces the data generated by the simulator to a set of *proposed treatments*. After some discussion, users add the *approved treatments* to a growing set of constraints for the simulation. The cycle repeats until users see no further improvement in the behavior of the simulator. Experiments with this approach have shown that TAR2 can:

- *Reduce the variance* of values within a data cloud
- *Improve the mean* of values within data clouds

For example, Figure 1.iii and Figure 1.iv show the results of applying incremental treatment learning to Figure 1.i and Figure 1.ii. Note that in both studies, the mean of the benefits increased, the mean of the costs decreased, and the variance in both measures was greatly decreased.

The notion that extra constraints can reduce the space of uncertainties is hardly surprisingly. However, what is surprising is *how few* extra constraints TAR2 needs to condense (e.g.) Figure 1.ii to Figure 1.iv; and how *easily* TAR2 can automatically find those constraints. The claim of this paper is that:

> *In the average case, a simple algorithm (TAR2) can quickly find a very small number of key constraints that result in massive condensations of data clouds towards some desired goal.*

There are three implications of this claim. Firstly, even when we don't know exactly what is going on within a model, it is possible to define minimal strategies to grossly decrease the uncertainty in that model's behavior.

Secondly, even if we aren't sure about the impact of certain decisions, we can be sure that certain other decisions will be ineffective. Decisions about treatment variables will override decisions about variables not found within a treatment. Hence, decisions about variables outside the treatments are redundant.

Thirdly, incremental treatment learning can reduce the cost of software modelling. Before applying elaborate modelling techniques or tools, it is wise to try cheaper and simpler techniques. Our results here shown that even hastily built incomplete models can be used for effective decision making. Since much can be learnt, even from sketchy data, it may be possible to avoid elaborate and extensive and expensive metrics collection. Further, once the treatments are known, then a minimal metrics collection program can be defined, just for the few variables in the treatments.

The rest of this paper describes the details of our condensation technique. TAR2 was motivated by *funnel theory* which is a claim that most decisions are redundant or irrelevant. In models containing funnels, a small number of key variables are enough to control a model, despite the large range of possibilities outside the funnel. Funnel theory is discussed in §2. Our algorithm for finding the key decisions within the funnels is discussed in §3. Case studies are then explored in §4 where TAR2 can reduce the variance and improve the mean of three case studies. After that, §5 discusses when this approach may not be appropriate and §6 discusses related work.

2 Funnel Theory

The premise of this paper is that within the space of possible decisions, there exist a small number of key decisions that determine all others. After Menzies, Easterbrook, Nuseibeh, and Waugh, we call this premise *funnel theory*- the metaphor being that all processing runs down the same narrow funnel [27].

To introduce funnels, we first say that a decision space supports *reasons*; i.e. chains of reasoning that link inputs in a certain context to desired goals. Chains have links of at least two types. Firstly, there are links that clash with other links. Secondly, there are the links that depend on other links. One method of optimizing the decision making process would be to first decide about the non-dependent clashing links. These are the *key decisions* since they determine most of the other non-key decisions.

For example, suppose the following decision space is explored using the invariant $nogood(X, \neg X)$ and everything that is not a $context$ or a $goal$ is open to debate:

$$\begin{array}{rl} & a \longrightarrow b \longrightarrow c \longrightarrow d \longrightarrow e \\ context1 \longrightarrow & f \longrightarrow g \longrightarrow h \longrightarrow i \longrightarrow j \ \ \longrightarrow goal \\ context2 \longrightarrow & k \rightarrow \neg g \longrightarrow l \longrightarrow m \rightarrow \neg j \ \longrightarrow goal \\ & n \longrightarrow o \longrightarrow p \longrightarrow q \longrightarrow \neg e \end{array}$$

Like any model, any of $\{a, b, ..q\}$ is subject to discussion. However, in the context of reaching some specified goals from *context1* and *context2*, the only important discussions are the clashes $\{g, \neg g, j, \neg j\}$ (the $\{e, \neg e\}$ clash is not exercised in the context of $context1, context2 \ \vdash \ goal$, since no reason uses e or $\neg e$). Further, since $\{j, \neg j\}$ are fully dependent on $\{g, \neg g\}$, then the core decision must be about variable ({g}) with two disputed values: true and false.

The *funnel* of an decision space contains the non-dependent clashing links; e.g. $\{g\}$. The decisions with *greatest information content* are the decisions about the funnel variables, since these variables set the others. If the space contains *narrow funnels* (i.e. funnels with small cardinality) then the total decision space can be greatly reduced to a small number of highly informative disputes about funnel variables. Analysts are still free to debate whatever they want (and they will, seemingly endlessly), but with this approach, a funnel-aware analyst can steer the discussion towards the issues that tell us most about a domain. The net effect can be less argument. Suppose our analysts agree that *g* is true, then in the context of arguing about how $context1, context2 \ \vdash \ goal$, the decision space reduces to:

$$context1 \longrightarrow f \longrightarrow g \longrightarrow h \longrightarrow i \longrightarrow j \longrightarrow goal$$

The reasoning starting with k has been culled since, by endorsing *g*, we must rejects all lines of reasoning that use $\neg g$. In addition, the reasoning starting with a, n are ignored since they are irrelevant in this context; i.e. they do not participate in reaching a desired goal. Further, in this context, there is little point arguing about $\{f, h, i, j\}$ since if any of these are false, then no goal can be reached.

This small example suggests how funnels can condense data clouds. Data clouds are the result of a wide variation in model behavior. Such variation come from choices within a model relating to uncertain ranges. The more commitments we make about funnel variables, the more we collapse the space of possibilities outside the funnel. Hence, decisions about funnel variables condense data clouds, since they restrict the behavior of a system. Decision making in spaces containing funnels can be simple and short. Once values for the funnel variables are decided, all other decisions become redundant. In the above example, we have a decision space containing potentially $2^{16} = 65536$ debates about 16 boolean variables $\{a..q\}$. A decision about one variable (i.e. "is g true or false?") has reduced this space to one option.

Relying on narrow funnels may seem an overly optimistic approach. Yet a literature review suggests that such optimism is well-founded. There are many examples of funnel-like behavior in the literature. For example Horgan & Mathur [20] report that testing often exhibits a *saturation* effect; i.e. most program paths get exercised early with little further improvement as testing continues. Saturation is consistent with funnels controlling the reachable parts of a program. If these funnels were narrow, there would be few options in a program's execution and test inputs would quickly sample them all. Further testing over systems with narrow funnels would yield little further information since anything not connected to the funnels would be, by definition, unreachable.

An analogous effect to saturation is *homogenous propagation*. Despite numerous perturbations on data values using a program mutator[2], Michael found that in 80 to 90% of cases, there were no changes in the behavior of a range of programs [36]. Another study compared results using X% of a library of mutators, randomly selected (X $\in$ {10,15,...40,100}). Most of what could be learnt from the program could be learnt using only X=10% of the mutators; i.e. after a very small number of mutators, new mutators acted in the same manner as previously used mutators [48]. The same observation has been made elsewhere in the mutation literature [1,6].

[2] A *mutant* of a program is a syntactically valid but randomly selected variation to a program; e.g. swapping all plus signs to a minus sign.

$$\left.\begin{array}{l} \xrightarrow{a_1} M_1 \\ \xrightarrow{a_2} M_2 \\ \quad\dots \\ \xrightarrow{a_m} M_m \end{array}\right\} \xrightarrow{c} goal_i \xleftarrow{d} \left\{\begin{array}{l} N_1 \xleftarrow{b_1} \\ N_2 \xleftarrow{b_2} \\ N_3 \xleftarrow{b_2} \\ N_4 \xleftarrow{b_2} \\ \quad\dots \\ N_n \xleftarrow{b_n} \end{array}\right.$$

Fig. 3 Alternate funnels that lead to some goal.

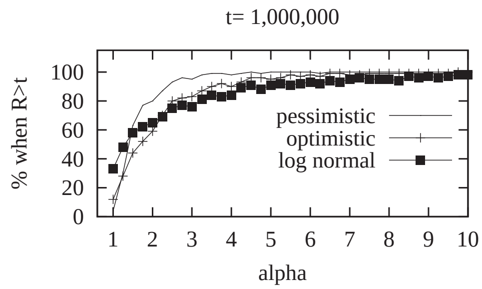


Fig. 4 10,000 runs of the funnel simulator. Y-axis shows what percentage of the runs satisfies $\left(\frac{narrow}{wide} = R\right) > t$. The $pessimistic$, $lognormal$, and $optimistic$ distributions assume a worst-case, average-case, and best-case (respectively) distribution for $\{a_i, b_i, c_i, d_i\}$. For more details, see [32].

Like saturation, homogenous propagation is consistent with funnel theory. If the overall behavior of a system is determined by a small number of key variables, then random mutation is unlikely to find those variables and the net effect of those mutations would be very small.

Homogenous propagation is observed in procedural programs. An analogous effect has been seen in declarative systems; i.e. most choices within a declarative set of constraints have little effect on the average behavior. Menzies & Waugh studied choices in millions of mutations of a nondeterministic system. In their *abductive framework* [24,26], a consistent set of choices generated a *world* of belief. Given N binary choices, there are theoretically 2^N possible worlds. However, after studying millions of generated worlds they found the maximum number of goals found in any world was often close to the number of goals found in a world selected at random [34] (on average, the difference was less 6%). This observation is inexplicable without narrow funnels. If choices had a large impact on what was reached within a declarative system, then there should be much variability in what is found in each world. Since the observed variability was so small, the number of critical choices (a.k.a. funnel variables) must also be small.

In fact, the concept of a funnel has been reported in many domains under a variety of names including:

– *Master-variables* in scheduling [12];
– *Prime-implicants* in model-based diagnosis [42] or machine learning [41], or fault-tree analysis [25].
– *Backbones* in satisfiability [37,45];
– *The dominance filtering* used in Pareto optimization of designs [23];
– *Minimal environments* in the ATMS [13];
– The *base controversial assumptions* of HT4 [26].

Whatever the name, the core intuition in all these terms is the same: what happens in the total space of a system can be controlled by a small critical region. The frequency of the funnel effect have made Menzies & Singh suspect that funnels are some average case phenomenon that is emergent in decision spaces [32]. To test this, they consider a device that can choose between a narrower and a wider funnel. Let some goal in a system be reachable by a narrow funnel M or a wide funnel N shown in Figure 3. Under what circumstances will the narrow funnel be favored over the wide funnel? The following definitions let us answer this question:

– Let the cardinality of the narrow funnel and wide funnels be m and n respectively.
– Each m members of the narrow funnel are reached via a path with probability a_i while each n members of the wider funnel are reached via a path with probability b_i.
– Two paths exist from the funnels to this goal: one from the narrow neck with probability c and one from the wide neck with probability d. Therefore, the probability of reaching the goal via the narrow pathway is $narrow = c\prod_{i=1}^{m} a_i$ while the probability of reaching the goal via the wide pathway is $wide = d\prod_{i=1}^{n} b_i$.

With these definitions, the Menzies & Singh study can be redefined as the search for conditions under which

$$\left(\frac{narrow}{wide} = R\right) > t \qquad (1)$$

where t is some threshold value.

To explore Equation 1, Menzies & Singh built a small simulator of Figure 3, and performed 150,000 runs using different distributions for a_i, b_i, c, d and a wide range of values for m, n. The results are shown in Figure 4. For comparison purposes, the size of the two funnels is expresses as a ratio $alpha$ where $n = alpha * m$. As might be expected, at $alpha = 1$ the funnels are the same size and the odds of using one of them is 50%. As $alpha$ increases, then increasingly $R > t$ is satisfied and the narrower funnel is preferred to the wider funnel. The effect is quite pronounced. For example, for all the studied distributions, after the wider funnel is 2.25 times bigger than the narrow funnel, then in 75% or more of the random searches, accessing the narrow funnel is at least 1,000,000 times more likely as accessing the wider funnel (see the lower graph of Figure 4). Interestingly, as the probability of using any of a_i, b_i, c_i, d_i decreases, the odds of using the narrow funnel increase (see the *pessimistic curves* in Figure 4). That is, narrow funnels are likely, especially in spaces that are difficult to search.

The average case analytical result of Menzies & Singh is suggestive evidence, but not conclusive evidence, that narrow funnels are common. Perhaps a more satisfying test for narrow funnels would be to check if, in a range of applications, a small number of variables are enough to control the other

outlook	*temp(^oF)*	*humidity*	*windy?*	*class*
sunny	*85*	*86*	*false*	*none*
sunny	*80*	*90*	*true*	*none*
sunny	*72*	*95*	*false*	*none*
rain	*65*	*70*	*true*	*none*
rain	*71*	*96*	*true*	*none*
rain	*70*	*96*	*false*	*some*
rain	*68*	*80*	*false*	*some*
rain	*75*	*80*	*false*	*some*
sunny	*69*	*70*	*false*	*lots*
sunny	*75*	*70*	*true*	*lots*
overcast	*83*	*88*	*false*	*lots*
overcast	*64*	*65*	*true*	*lots*
overcast	*72*	*90*	*true*	*lots*
overcast	*81*	*75*	*false*	*lots*

Fig. 5 A log of some golf-playing behavior.

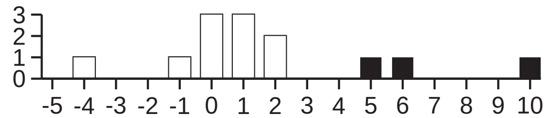


Fig. 6 Δ distribution seen in golf data sets. The X-axis shows the range of Δ values seen in the gold data set. The Y-axis shows the number of attribute ranges that have a particular Δ. Outstandingly high Δ values shown in black.

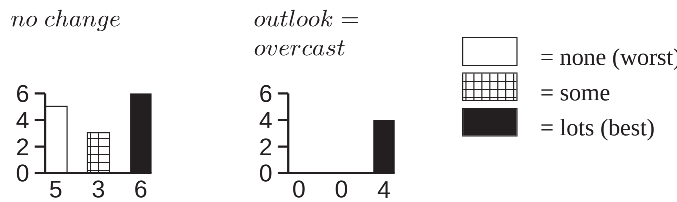


Fig. 7 Finding treatments that can improve golf playing behavior. With no treatments, we only play golf lots of times in $\frac{6}{5+3+6} = 57\%$ of cases. With the restriction that *outlook=overcast*, then we play golf lots of times in 100% of cases.

variables in a model. The rest of this paper implements that check.

3 Finding the Funnel

A traditional approach to funnel-based reasoning is to find the funnels using some dependency-directed backtracking tool such as the ATMS [13] or HT4 [26]. Dependency-directed backtracking is very slow, both theoretically and in practice [26]. Further, in the presence of narrow funnels, it may be unnecessary. There is no need to *search* for the funnel in order to *exploit* it. Any reasoning pathway to goals must pass through the funnels (by definition). Hence, all that is required is to find attribute ranges that are associated with desired behavior.

TAR2 is a machine learning method for finding attribute ranges associated with desired behavior. Traditional machine learners generate classifiers that assign a class symbol to an example [40]. TAR2 finds the difference between classes. Formally, the algorithm is a *contrast set learner* [3] that uses *weighted classes* [7] to steer the inference towards the preferred behavior. The algorithm differs from other learners in that it seeks contrast sets of *minimal* size.

TAR2 can best be introduced via example. Consider the log of golf playing behavior shown in Figure 5. This log contains four attributes and 3 classes. Recall that TAR2 accesses a *score* for each class. For a golfer, the classes in Figure 5 could be scored as *none=2* (i.e. worst), *some=4*, *lots=8* (i.e. best).

TAR2 seeks attribute ranges that occur more frequently in the highly scored classes than in the lower scored classes. Let $a.r$ be some attribute range e.g. *outlook.overcast*. $\Delta_{a.r}$ is a heuristic measure of the worth of $a.r$ to improve the frequency of the $best$ class. $\Delta_{a.r}$ uses the following definitions:

$X(a.r)$: the number of occurrences of that attribute range in class X; e.g. *lots(outlook.overcast)=4*.
$all(a.r)$: total number of occurrences of that attribute range in all classes; e.g. *all(outlook.overcast)=4*.
$best$: the highest scoring class; e.g. $best = lots$;
$rest$: the non-best class; e.g. $rest = \{none, some\}$;
$\$Class$: score of a class $Class$ is $\$Class$.

$\Delta_{a.r}$ is calculated as follows:

$$\Delta_{a.r} = \frac{\sum_{X \in rest} (\$best - \$X) * (best(a.r) - X(a.r))}{all(a.r)}$$

When *a.r* is *outlook.overcast*, then $\Delta_{outlook.overcast}$ is calculated as follows:

$$\frac{\overbrace{((8-2)*(4-0))}^{lots \to none} + \overbrace{((8-4)*(4-0))}^{lots \to some}}{4+0+0} = \frac{40}{4} = 10$$

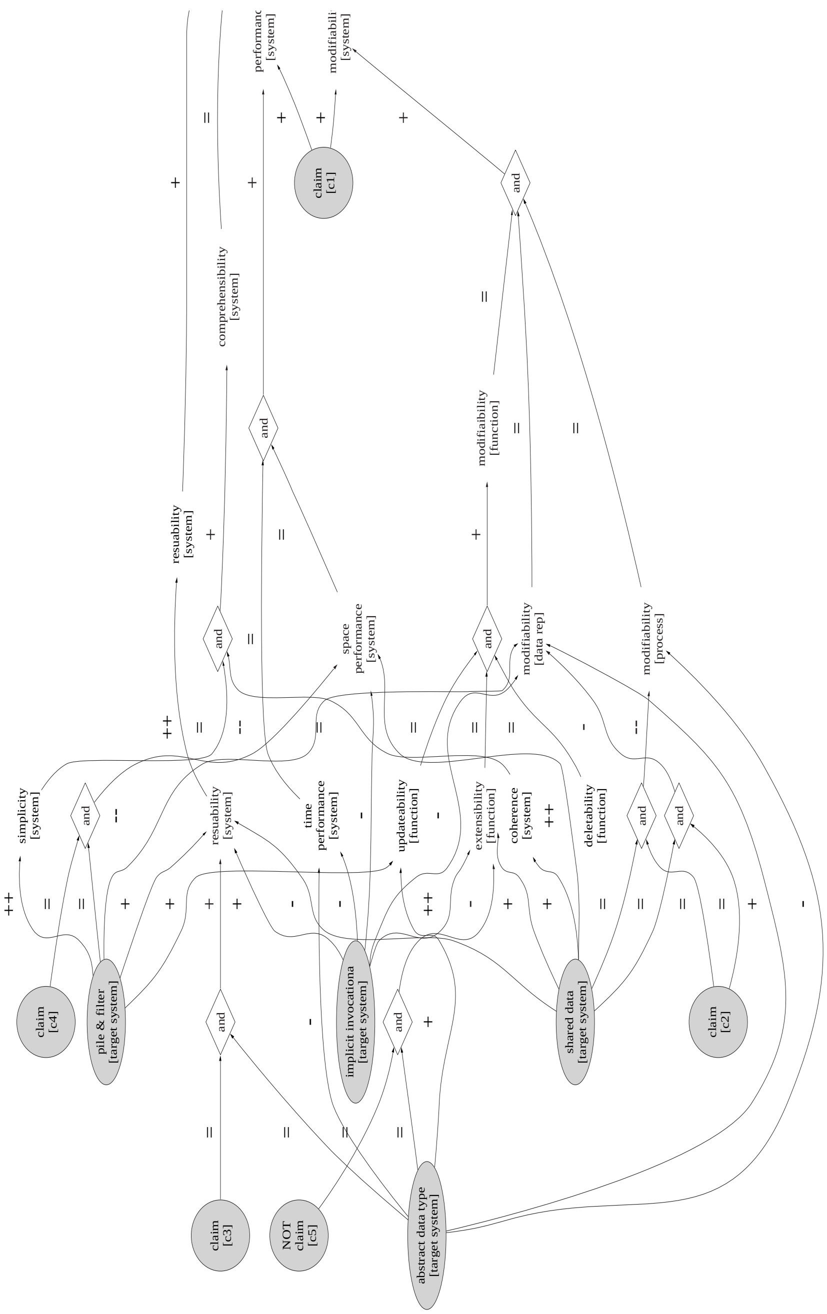
The attribute ranges in our golf example generate the Δ histogram shown in Figure 6. Note that *outlook=overcast*'s Δ is the highest, potentially most effective, attribute range.

A *treatment* is a subset of the attribute ranges with an *outstanding* $\Delta_{a=r}$ value. For our golf example, such attributes can be seen in Figure 6: they are the outliers with outstandingly large Δs on the right-hand-side. (These outliers include *outlook=overcast*).

To *apply* a treatment, TAR2 rejects all example entries that contradict the conjunction of attribute ranges in the treatment. The ratio of classes in the remaining examples is compared to the ratio of classes in the original example set. The *best treatment* is the one that most increases the relative percentage of preferred classes. In the case where N treatments increase the relative score by the same amount, then N *best treatments* are generated and TAR2 picks one at random. In our golf example, a single best treatment was generated containing *outlook=overcast*; Figure 7 shows the class distribution before and after that treatment, i.e. if we choose a vacation location that is generally overcast, then in 100% of cases we should be playing lots of golf, all the time.

4 Case Studies

This section presents three examples of incremental treatment learning. The examples are sorted by model size: smallest to



Claim : Notes
c1 : “among the few vital goals”
c2 : “a claim by David Parnas” [38]
c3 : “few assumptions among interacting modules”
c4 : “expected size of data is huge”
c5 : “many implementors familiar with ADTs” (from domain experts)

Inference rules:

The $benefit$ of the this network is the $benefit$ computed for the top-level node $good$. This $benefit$ is defined recursively as follows:

– The benefit of a leaf node is 1 if it is selected, or 0 otherwise. Leaf nodes represent choices in the network. Leaf nodes are shown in gray.
– The benefit of a non-leaf node is computed from its input influences.
– An influence of an edge on an upstream node is the product of the edge weight and the $benefit$ of the downstream node.
– Edge weights are set by tables that offer numeric values for $\{$++,+,=,-,–$\}$.
– Nodes are either disjunctions or conjunctions. Conjunctions are shown as diamonds. The benefit of a conjunction is minimum of the input influences. The benefit of a disjunction is the average of the input influences.

For a rationale on why these rules were selected, see [8]. In summary: these rules were not unreasonable and the users wanted it that way. Future experiments in this domain will explore variants to these rules.

Fig. 8 An model that assesses architectural choices within software. Options within the model are the leaf nodes shown in gray. These options can be architectural decisions such as the use of *abstract data types*, *implicit invocation*, *pipe & filter* methods, or *shared data*. Some links in the model are dependant of various claims $c1..c5$ shown top, right of the diagram. For example, *claim[c2]* is Parnas’s [38] argument that having a single share data model across an entire application has a negative impact on the modifiability of that process. The inference rules of this diagram are shown middle, right.

largest. The largest and final model is too detailed to explain here but the second largest model is explained in sufficient detail for the reader to reproduce the entire experiment. In all examples, the results of incremental treatment learning is to find a subset of all possible decisions that *reduces the variance* and *improves the mean* of the important variables within a data cloud.

4.1 Case Study A: Software Architectures

Figure 8 shows some architectural assessment knowledge taken from Shaw & Garlan's Software Architectures book [43]. The knowledge is expressed in our variant of the *softgoal* notation of Chung et.al. [10]. In the softgoal approach, a *softgoal* is distinguished from a normal goal as follows:

- While a $goal$ has well-defined non-optional feature of a system that *must* be available, a $softgoal$ is a goal that has no clear cut criteria for success.=;
- While goals can be conclusively demonstrated to be *satisfied* or *not satisfied*, softgoals can only be *satisficed* to some degree.

Much is under-constrained in Figure 8. In fact, there are $4^{21} * 2^9 \approx 10^{15}$ possibilities within this model:

- The nine boolean choices in the model are leaf nodes representing software architecture options or claims about the application. Hence, there are 2^9 combinations of these choices.
- Edges between nodes in Figure 8are annotated with a symbol denoting how strongly the downstream node impacts the upstream node. These annotations are $\{++, +, =, -, --\}$ denoting $makes, helps, equals, hurts, breaks$ (respectively). For the sake of exposition, we say that the values for four of these annotations come from a range of 21 possible values

$$1 \geq X_{makes} > X_{helps} > X_{hurts} > X_{breaks} \geq 1$$
$$X_I \in \{-1, -0.9, -0.8, ..0, ...0.9, 1\}$$

(The exact value of $equals$ is not varied since this annotation is used to propagate influences unchanged over an edge; i.e. $weight(equals) = 1$.)
Hence, in the worst case, there are $4^{21} \approx 10^{12}$ possible edge weights.

These possibilities generate a wide range of behavior. Our softgoal interpreter [8] computes a $cost$ and $benefit$ figure resulting from a selection of edge weights and choices in diagrams like Figure 8 (the details of this computation are discussed in the *inference rules* table of Figure 8). Figure 1.i shows the range of $benefit$s and $cost$s seen after 10,000 random selection of choices and edge weights. Note the large variance in these figures.

To apply incremental treatment learning for this case study, we first require a scoring scheme for the different classes. In 10,000 runs of Figure 8, with no constraints on any selections, the observed $costs$ ranged from 1 to 4 and the $benefit$

scoring function:

	Cost			
Benefit	1	2	3	4
12	1	2	3	4
6	5	6	7	8
0	9	10	11	12
-6	13	14	15	16
-12	17	18	19	20
-18	21	22	23	24

Fig. 9 Class scoring function.

round 0:

	Cost				
Benefit	1	2	3	4	Totals
12					
6		1	2	1	4
0	13	19	15	4	51
-6	10	12	4	1	27
-12	4	6	2		12
-18	3	2	1	1	7
Totals	30	40	24	6	100

Fig. 10 Percentage distributions of $benefit$s and $cost$s seen in 10,000 runs of Figure 8, assuming Equation 2 and a random selection of architectural options and claims.

ranged from -18 to 12 (see Figure 1.i). Since high $benefit$ and low $cost$ is preferable to high $cost$ and low $benefit$, these ranges were scored as shown in Figure 9. In that figure, the best range is $benefit \geq 12$ and $cost = 1$ and the worst range is $benefit \leq -18$ and $cost = 4$.

TAR2 was applied to Figure 8 four times. Each round comprised 10,000 runs where:

- Edge weights were selected at random at the start of each run from Equation 2.
- From the space of remaining choices, architectural options and claims were selected at random.

Initially, no restrictions were imposed on the architectural options and claims. This generated the ranges of $cost$ and $benefit$ shown in Figure 1.i. Such a data cloud is hard to read. A more informative representation is the *percentile matrix* of Figure 10. Each cell of this matrix shows the percent of runs that falls into a certain range. Each cell is colored on a scale ranging from white (0%) to black (100%).

Figure 11 shows the results of applying incremental treatment learning to Figure 8. Each round took the key decisions learnt by TAR2 from 10,000 examples generated in the previous round. 10,000 more runs were then performed, with the selection of architectural options and claims restricted according to the current set of key decisions. Note that as the key decisions accumulate, the variance in the behavior decreases and the means improve (decreased $cost$ and increased $benefit$).

This experiment stopped after four rounds since there was little observed improvement between round 3 and round 4. Figure 1.iii shows the results of the round 3, not round 4; i.e. this experiment returned the results from round 3, and not

$$KEY_1 = (claim[c1] = yes)$$
$$\wedge(pipe\&filter[target\,system] = yes)$$

round 1:

	Cost				
Benefit	1	2	3	4	Totals
12		1	2	1	4
6		5	9	3	17
0	11	30	26	7	74
-6	1	2	1	1	5
-12					
-18					
Totals	12	38	38	12	100

$$KEY_2 = KEY_1 \wedge (shared\,data[target\,systems] = yes)$$
$$\wedge(implicit\,invocation[target\,system] = no)$$

round 2:

	Cost				
Benefit	1	2	3	4	Totals
12		3	6		9
6	4	17	14		35
0	28	28			56
-6					
-12					
-18					
Totals	32	48	20		100

$$KEY_3 = KEY_2 \wedge (abstract\,data\,type[target\,systems] = no)$$
$$\wedge(claim[c3] = no)$$

round 3:

	Cost				
Benefit	1	2	3	4	Totals
12		12			12
6		39			39
0		48			48
-6		1			1
-12					
-18					
Totals		100			100

$$KEY_4 = KEY_3 \wedge (claim[c2] = yes)$$
$$\wedge(claim[c4] = yes)$$

round 4:

	Cost				
Benefit	1	2	3	4	Totals
12		20			20
6		38			38
0		42			42
-6					
-12					
-18					
Totals		100			100

Fig. 11 Percentile matrices showing four rounds of incremental treatment learning for Figure 8

round 4. By stopping at round 3, analysts can avoid excessive decision making since they need never discuss $c2, c4, c5$ with their users. Alternatively, if in some dispute situation, an analyst could use $c2, c4, c5$ as bargaining chips. Since these claims have little overall impact, our analyst could offer them in any configuration as part of some compromise deal in exchange for the other key decisions being endorsed.

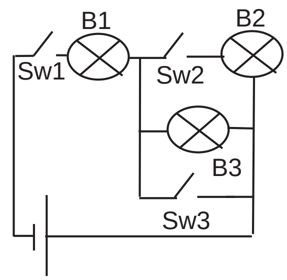


Fig. 12 A qualitative circuit. From [5].

```
%sum(X,Y,Z).
 sum(+,+,+).   sum(+,0,+). sum(+,-,Any).
 sum(0,+,+).   sum(0,0,0). sum(0,-,-).
 sum(-,+,Any). sum(-,0,-). sum(-,-,-).
```

Fig. 13 Qualitative mathematics using a Prolog syntax [5].

4.2 Case Study B: Circuit Design

Our next example contains a model somewhat more complex than §4.1. This example is based on models first developed by Bratko to demonstrate principles of qualitative reasoning [5].

While our last example generated cost and benefit figures for a software project, this example is a qualitative model of a circuit design shown in Figure 12. Such qualitative descriptions of a planned piece of software might appear early in the software design process. We will assume that the goal of this circuit is to illuminate some area; i.e. *the more bulbs that glow, the better*.

For exposition purposes, we assume that much is unknown about our circuit. All we will assume is that the topology of the circuit is known, plus some general knowledge about electrical devices (e.g. the voltage across components in series is the sum of the voltage drop across each component). What we don't know about this circuit are the precise quantitative values describing each component.

When quantitative knowledge is unavailable, we can use qualitative models. A qualitative model is a quantitative mode whose numeric values x are replaced by a qualitative value x' having one of three qualitative states: +, -, 0; i.e.

$$\begin{aligned} x' &= +\ if\ x > 0 \\ x' &= 0\ if\ x = 0 \\ x' &= -\ if\ x < 0 \end{aligned}$$

The `sum` relation of Figure 13 describes our qualitative knowledge of addition using a Prolog notation. In Prolog,

```
%blub(Mode,Light,Volts,Amps)
 bulb(blown,dark,  Any, 0).
 bulb(ok,   light, +,   +).
 bulb(ok,   light, -,   -).
 bulb(ok,   dark,  0,   0).

%num(Light, Glow).   %switch(State,Volts,Amps)
 num( dark,  0).      switch(on,    0,    Any).
 num( light, 1).      switch(off,   Any,  0).
```

Fig. 14 Definitions of qualitative bulbs and switches. Adapted from [5].

```
 1  circuit(switch(Sw1,VSw1,C1),
 2          bulb(B1,L1,VB1,C1),
 3          switch(Sw2,VSw2,C2),
 4          bulb(B2,L2,VB2,C2),
 5          switch(Sw3,VSw3,CSw3),
 6          bulb(B3,L3,VB3,CB3),
 7          Glow) :-
 8      VSw3 = VB3,
 9      sum(VSw1, VB1, V1),   %  9 options
10      sum(V1,VB3,+),        %  1 option
11      sum(VSw2,VB2,VB3),    %  9 options
12      switch(Sw1,VSw1,C1),  %  2 options
13      bulb(B1,L1,VB1,C1),   %  4 options
14      switch(Sw2,VSw2,C2),  %  2 options
15      bulb(B2,L2,VB2,C2),   %  4 options
16      switch(Sw3,VSw3,CSw3),%  2 options
17      bulb(B3,L3,VB3,CB3),  %  4 options
18      sum(CSw3,CB3,C3),     %  9 options
19      sum(C2,C3,C1),        %  9 options
20      num(L1,N1),
21      num(L2,N2),
22      num(L3,N3),
23      Glow is N1+N2+N3.
```

Fig. 15 Figure 12, modelled in Prolog. Adapted from [5].

variables start with upper case letters and constants start with lower-case letters or symbols. For example,

$$\mathtt{sum}(+,+,+)$$

says that the addition of two positive values is a positive value. There is much uncertainty within qualitative arithmetic. For example

$$\mathtt{sum}(+,-,\mathtt{Any})$$

says that we cannot be sure what happens when we add a positive and a negative number.

The `bulb` relation of Figure 14 describes our qualitative knowledge of bulb behavior. For example,

$$\mathtt{bulb}(\mathtt{blown},\mathtt{dark},\mathtt{Any},0)$$

says that a blown bulb is dark, has zero current across it, and can have any voltage at all. Also shown in Figure 14 are the `num` and `switch` relations. `Num` defines how bright a dark or light bulb glows while `switch` describes our qualitative knowledge of electrical switches. For example

$$\mathtt{switch}(\mathtt{on},0,\mathtt{Any})$$

says that if a switch is on, there is zero voltage drop across it while any current can flow through it.

The `circuit` relation of Figure 15 describes qualitative knowledge of a circuit using `bulb, num, sum` and `switch`. This relation just records what we know of circuits wired together in series and in parallel. For example:

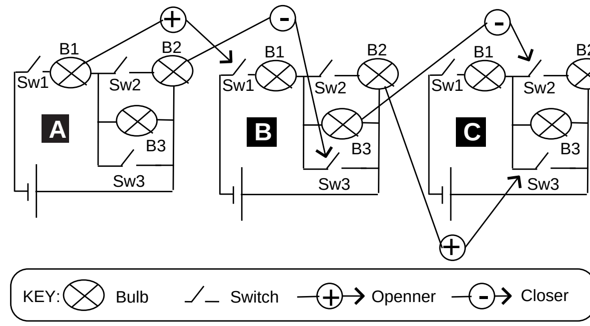


Fig. 16 A device modelled using the Prolog of Figure 16.

```
%inf(Sign,Bulb,Switch)
 inf(Inf,bulb(_,Shine,_,_),switch(Pos,_,_)) :-
    inf1(Inf,Shine,Pos).

%inf1(Sign,Glow,SwitchPos)
 inf1(+,dark,  off). inf1(+,light, on).
  inf1(-,dark,  on). inf1(-,light, off).
```

Fig. 17 The `inf1/3` predicate used to connect bulb brightness to switches.

- `Switch3` and `Bulb3` are wired in parallel. Hence, the voltage drop across these components must be the same (see line 8).
- `Switch2` and `Bulb2` are wired in series so the voltage drop across these two devices is the sum of the voltage drop across each device. Further, this summed voltage drop must be the same as the voltage drop across the parallel component `Bulb3` (see line 11).
- `Switch1` and `Bulb1` are in series so the same current `C1` must flow through both (see line 12 and line 13)

In order to stress test our method, our case study will wire up three copies of Figure 15 in such a way that solutions to one copy won't necessarily work in the other copies. Figure 16 shows our circuit connected by a set of *openers* and *closers* that open/close switches based on how much certain bulbs are glowing. For example, the *closer* between bulb *B2A* and switch *Sw1B* means that if *B2A* glows then *Sw1B* will be closed. These openers and closers are defined in Figure 17. The full model is shown in Figure 18.

The less that is known about a model, the greater the number of possible behaviors. This effect can easily be seen in our qualitative model. Each line of Figure 15 is labelled with the number of possibilities it condones: i.e.

$$9*1*9*2*4*2*4*2*4*9*9 = 3,359,232$$

Copied three times, this implies a space of up to $3,359,232^3 = 10^{19}$ options. Even when many of these possibilities are ruled out by inter-component constraints, the `circuits` relation of Figure 18 can still succeed 35,228 times (some sample output is shown in Figure 19).

Given the goal that the *more* lights that shine, the better the circuit, we assume 10 classes: $0, 1, 2, 3, ..9$, one for every possible number of glowing bulbs. As shown in Figure 20,

```
1 circuits :-
2    % some initial conditions
3    value(light,bulb,B1a),
4    % Uncomment to constrain Sw2c
5    % value(off,switch,Sw2c),
6    % Uncomment to constrain Sw1c
7    % value(on,switch,Sw1c),
8    % Uncomment to constrain Sw3c
9    % value(on,switch,Sw3c),
10   % explore circuit A
11   circuit(Sw1a,B1a,Sw2a,B2a,Sw3a,B3a,GlowA),
12   % let circuit A influence circuit B
13   inf(+,B1a,Sw1b),
14   inf(-,B2a,Sw3b),
15   % let circuit B influence circuit C
16   circuit(Sw1b,B1b,Sw2b,B2b,Sw3b,B3b,GlowB),
17   % propagate circuit B to circuit C
18   inf(-,B3b,Sw2c),
19   inf(+,B2b,Sw3c),
20   % explore circuit C
21   circuit(Sw1c,B1c,Sw2c,B2c,Sw3c,B3c,GlowC),
22   % compute total shine
23   Shine is GlowA+GlowB+GlowC,
24   % make one line of the examples
25   format('~p,~p,~p,~p,~p,~p,~p,~p,~p',
26   [Sw1a,Sw2a,Sw3a,Sw1b,Sw2b
27   ,Sw3b,Sw1c,Sw2c,Sw3c]),
28   format('~%,~%,~%,~%,~%,~%,~%,~%,~%,~p',
29   [B1a,B2a,B3a,B1b,B2b,B3b
30   ,B1c,B2c,B3c,Shine]),nl.
31
32 data :- tell('circ.data'),
33   forall(circuits,true), told.
34
35 % some support code
36 value(Sw,    switch, switch(Sw,_,_)).
37 value(Light, bulb,bulb(_,Light,_,_)).
38
39 :- format_predicate('%',bulbIs(_,_)).
40
41 bulbIs(_,bulb(X,_,_,_)) :-
42     var(X) -> write('?') |write(X).
43
44 portray(X) :- value(Y,_,X), write(Y).
```

Fig. 18 Figure 16 expressed in Prolog.

Sw1a	*Sw2a*	*Sw3a*	*Sw1b*	*Sw2b*	*Sw3b*	*Sw1c*	*Sw2c*	*Sw3c*	*B1a*	*B2a*	*B3a*	*B1b*	*B2b*	*B3b*	*B1c*	*B2c*	*B3c*	*Shine*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*blown*	*blown*	*blown*	*blown*	*blown*	*blown*	*2*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*blown*	*blown*	*blown*	*blown*	*blown*	*blown*	*2*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*blown*	*blown*	*blown*	*blown*	*blown*	*blown*	*2*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*blown*	*blown*	*blown*	*ok*	*blown*	*blown*	*2*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*blown*	*blown*	*ok*	*blown*	*blown*	*blown*	*2*
on	*off*	*off*	*on*	*on*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*blown*	*blown*	*blown*	*blown*	*blown*	*ok*	*2*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*ok*	*blown*	*ok*	*blown*	*ok*	*blown*	*3*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*ok*	*blown*	*ok*	*blown*	*ok*	*ok*	*3*
on	*off*	*off*	*on*	*off*	*on*	*off*	*on*	*off*	*ok*	*blown*	*ok*	*ok*	*blown*	*ok*	*blown*	*blown*	*blown*	*3*
on	*off*	*off*	*on*	*off*	*on*	*on*	*on*	*off*	*ok*	*blown*	*ok*	*ok*	*blown*	*blown*	*ok*	*ok*	*blown*	*5*

Fig. 19 Some output seen in `circ.data` generated using `data` (line 32 of Figure 18). Columns denote values from Figure 16. For example, Sw1a and Sw1b denotes switch 1 in ciruit A and ciruit A respectively.

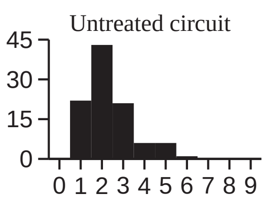


Fig. 20 Frequency count of number of bulbs glowing in the 35,228 solutions of `circuits` of Figure 18.

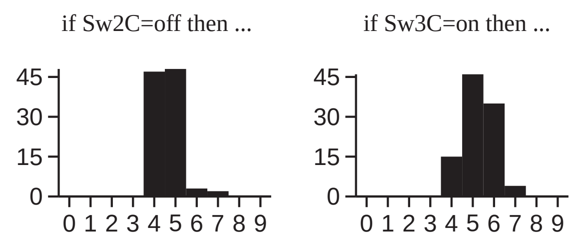


Fig. 21 Run#1 of TAR2 over the data seen in Figure 20.

within the 35,228 runs, there are very few lights shining. In fact, on average within those runs, only two lights are shining. TAR2's mission is to explore the space, trying to find key decisions which, when applied to the circuit, can most improve this low level of lighting.

4.2.1 Round 1 After learning treatments from the all 35,228 initial runs, and applying them to the data, TAR2 generated Figure 21.

In summary, Figure 21 is saying that making a single decision will change the average illumination of the circuit from 2 to 5 (if Sw2C=off) or 6 (if Sw3C=on).

For exposition purposes, this example assumed that something prevents our users from making this key decision; i.e. Sw3C=on. Our experience with incremental treatment learning is that this is often the case. When users are presented with the next key decision, they often recall some key knowledge that they neglected to mention previously. In this case, we assumed that it is preferable if switch 3 in circuit C is not closed- since that would violate (say) the warranty on circuit C. Our analysts therefore agreed to the next best treatment, i.e. Sw2C=off; shown in Figure 21, left hand side (LHS).

4.2.2 Round 2 After constraining the model to Sw2C=off (i.e. by uncommenting line 5 in Figure 18), fewer behaviors were generated: 3,264 as compared to the 35,228 solutions seen previously. The frequency distribution of the shining lights in this new situation is shown in Figure 22.

Happily, Figure 22 has the same distribution as Figure 21.LHS; i.e. in this case, when the constraints proposed by TAR2's best treatment were applied to the model, the resulting new behavior of the model matched the new behavior predicted by the treatment.

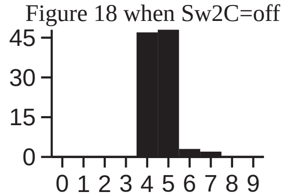


Fig. 22 Frequency count of number of bulbs glowing in the 3,264 solutions of `circuits` of Figure 18 when Sw2C=off.

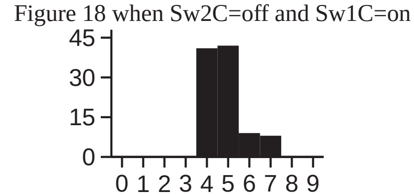


Fig. 24 Frequency count of number of bulbs glowing in the 648 solutions of `circuits` of Figure 18 when Sw2C=off and Sw1C=on.

Executing TAR2 again found the next most informative decision, as shown in Figure 23. Here, TAR2 said that our best treatment would be to guarantee that bulb 3 in circuit C is never blown. Perhaps this is possible if we were to use better light bulbs with extra long life filaments. However, for the sake of exposition, we assumed that there is no budget for such expensive hardware. Hence, to avoid this expense, our analysts agreed that always closing switch 1 in circuit C (as proposed by Figure 23.LHS) is an acceptable action.

4.2.3 Round 3 After further constraining the model to Sw1C=on (i.e. by uncommenting line 7 in Figure 18), fewer behaviors were generated: 648 as compared to the 3,264 solutions seen previously. The frequency distribution of the shining lights in this new situation is shown in Figure 24.

Figure 24 has the same distribution as Figure 23.LHS. That is, once again, TAR2's predictions proved accurate. Executing TAR2 again generated Figure 25 and finds the next most informative decision.

The cycle could stop here since the next best treatments are not acceptable. Figure 25.LHS wants to use overly-expensive hardware to ensure that bulb 3 in circuit C is always not blown. Figure 25.RHS wants to use an undesirable action and close switch 3 in circuit C. However, our engineers have enough information to propose some options to their manager: if they increase their hardware budget, they could make the improvements shown in Figure 25.LHS. Alternatively, if there was some way to renegotiate the warranty, then the improvements shown in Figure 25.RHS could be achieved.

To verify this, our engineers continue constraining Figure 18 to the case of Sw3c=on by uncommenting line 9 in Figure 18. The resulting distributions looked exactly like Figure 25.RHS. Further, only 64 solutions were found. Note that this observation is consistent with funnel theory: resolving three of the top treatments proposed by TAR2 constrained our system to one fifth of one percent of its original 35,228 behaviors.

4.3 Case Study C: Satellite Design

Our third example is much larger that then two previous. For reasons of confidentiality, the full details of this third model cannot be presented here. Further, this model uses so much domain-specific knowledge of satellite design that the general reader might learn little from its full exposition.

Analysts at the Jet Propulsion Laboratory sometimes debate design issues by building a semantic network connecting design decisions to requirements [46]. This network links *faults* and risk mitigation *actions* that effect a tree of *requirements* written by the stakeholders (see Figure 26). Potential faults within a project are modelled as influences on the edges between requirements. Potential fixes are modelled as influences on the edges between faults and requirements edges.

This kind of requirements analysis seeks to maximize benefits (i.e., our coverage of the requirements) while minimizing the costs of the risk mitigation actions. Optimizing in this manner is complicated by the interactions inside the model - a requirement may be impacted by multiple faults, a fault may impact multiple requirements, an action may mitigate multiple faults, and a fault may be mitigated by multiple actions. For example, in Figure 26, *fault2* and *require4* are interconnected: if we cover *require4* then that makes *fault2* more likely which, in turn, makes *fault1* more likely which reduces the contribution of *require5* to *require3*.

The net can be executed by selecting actions and seeing what benefits results. One such network included 99 possible actions; i.e. $2^{99} \approx 10^{30}$ combinations of actions. The data cloud of Figure 1.ii was generated after 10,000 runs where each run selected at random from the 99 options. Note the wide variance in the possible behaviors.

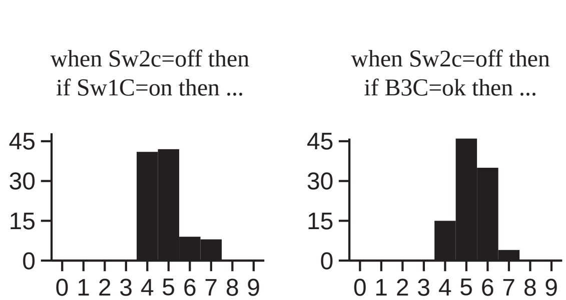


Fig. 23 Run #2 of TAR2 over the data seen in Figure 22.

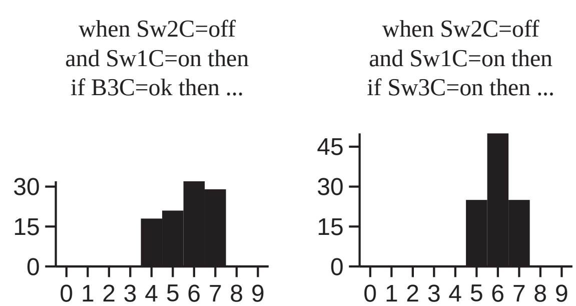
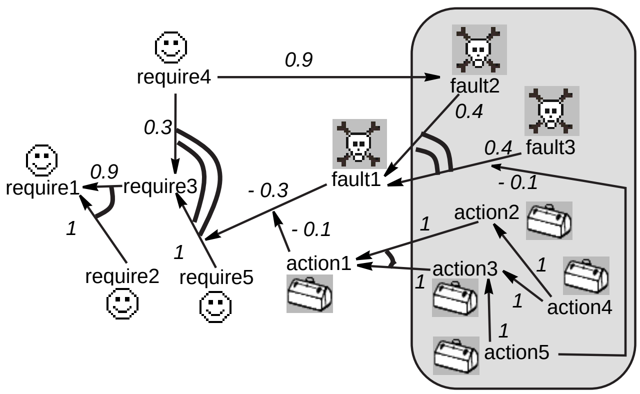


Fig. 25 Run#3 of TAR2 over the data seen in Figure 24.



Faces denote requirements;
Toolboxes denote actions;
Skulls denote faults;
Conjunctions are marked with one arc; e.g. *require1* if *require2* and *require2*.
Disjunctions are marked with two arcs; e.g. *fault1* if *fault2* or *fault3*.
Numbers denote impacts; e.g. *action5* reduces the contribution of *fault3* to *fault1*, *fault1* reduces the impact of *require5*, and *action1* reduces the negative impact of *fault1*.
Oval denotes structures that are expressible in the latest version of the JPL semantic net editor (under construction).

Fig. 26 Above: a semantic net of the type used at JPL [15] Below: explanation of symbols.

The results of incremental treatment learning is shown in Figure 27. The first percentile matrix (called **round 0**) summarizes Figure 1.ii. As with all our other examples, as incremental treatment learning is applied, the variance is reduced and the mean values improve (compare **round 0** with **round 4** in Figure 27). In a result consistent with funnel theory, TAR2 could search a space of 10^{30} decisions to find 30 (out of 99) that crucially effected the cost/benefit of the satellite; i.e. TAR2 found 99-30=69 decisions that can be ignored [16].

For comparison purposes, a genetic algorithm (GA) was also applied to the same problem of optimized satellite design [47]. The GA also found decisions that generated high benefit, low cost projects. However, each such GA solution commented on every possible decisions and there was no apparent way to ascertain which of these are the most critical decisions. The TAR2 solution was deemed superior to the GA solution by the domain experts, since the TAR2 solution required just 30 actions.

round 0:

	Cost				
Benefit	400K	600K	800K	1,000K	Totals
250		6	15	5	26
200	1	22	27	4	54
150	1	6	5	1	13
100		3	3		6
50		1%			1
Totals	2	38	50	10	100

round 1:

	Cost				
Benefit	400K	600K	800K	1,000K	Totals
250	7	45	13		65
200	12	22	1		35
150					
100					
50					
Totals	19	67	14		100

round 2:

	Cost				
Benefit	400K	600K	800K	1,000K	Totals
250	9	8	7		24
200	18	58			76
150					
100					
50					
Totals	27	66	7		101

round 3:

	Cost				
Benefit	400K	600K	800K	1,000K	Totals
250	9	70	11		90
200	3	7			10
150					
100					
50					
Totals	12	77	11		100

round 4:

	Cost				
Benefit	400K	600K	800K	1,000K	Totals
250	1	81	17		99
200		1			1
150					
100					
50					
Totals	1	82	17		100

Fig. 27 Percentile matrices showing four rounds of incremental treatment learning for JPL satellite design. The data clouds for **round 0** and **round 4** appear as Figure 1.ii and Figure 1.iv (respectively).

5 When Not to Use Incremental Treatment Learning

Our approach is an inexpensive method of generating coarse-grained controllers from rapidly written models containing uncertainties. This kind of solution is inappropriate for certain classes of software such as mission critical or safety critical systems. For those systems, analysts should move beyond TAR2 and apply heavyweight modelling methods and extensive data collection to ensure exact and optimal solutions.

There are several other situations where incremental treatment learning should not be used. When trusted and powerful heuristics are available for a model, then a heuristic search for model properties may yield insight than random trashing within a model. Such heuristics might be modelled via (e.g.)

fuzzy membership functions or bayesian priors reflecting expert intuitions on how variables effect each other. Of course, such heuristics must be collected, assessed, and implemented. When the cost of such collection and assessment and implementation is too great, then our approach could be a viable alternative.

Also, our approach requires running models many thousands of times and therefore can't be applied to models that are too expensive or too slow to execute many times. For example:

– It may be too expensive or dangerous to conduct Monte Carlo simulations of in-situ process control systems for large chemical plants or nuclear power stations.
– Suppose some embedded piece of software must be run on a specialized hardware platform. In the case where several teams must access this platform (e.g. the test team, the development team, the government certification team, and the deployment team), then it may be impossible to generate sufficient runs for incremental treatment learning.
– Many applications connect user actions on some graphical user interface to database queries and updates. Monte Carlo simulations of such applications may be very slow since each variable reference might require a slow disk access or a user clicking on some "OK" button. An ideal application suitable for incremental treatment learning comprises a *separate model* for the business logic which can be executed without requiring (e.g.) screen updates or database accesses.

6 Related Work

Other publications on treatment learning have assumed a "one-shot" use of TAR2 [21, 29–31]. This paper assumes an iterative approach. Our experience with users is that this iterative approach encourages their participation in the process and increases their sense of ownership in the conclusions.

Our development on funnel theory owes much to the deKleer's ATMS (assumption-based truth maintenance system) [13]. As new inferences are made, the ATMS updates its network of dependencies and sorts out the current conclusions into maximally consistent subsets (which we would call worlds). *Narrow funnels* are analogous to *minimal environments of small cardinality* from the ATMS research. However, funnels differ from the ATMS. Our view of funnels assume a set-covering semantics and not the consistency-based semantics of the ATMS (the difference between these two views is detailed in [11]). The worlds explored by funnels only contain the variables see in the subset of a model exercised by the supplied inputs. An ATMS world contains a truth assignment to every variable in the system. Consequently, the user of an ATMS may be overwhelmed with an exponential number of possible worlds. In contrast, our heuristic exploration of possible worlds, which assumes narrow funnels, generates a more succinct output. Further, the ATMS is only defined for models that generate logical justifications for each conclusions. Iterative treatment learning is silent on the form of the model: all it is concerned with is that a model, in whatever form, generates outputs that can be classified into desired and undesired behavior.

We are not the first to note variability in knowledge extracted from users. Leveson [19] reports very large variances in the calculation of root node likelihood in software fault tree analysis:

– In one case study of 10 teams from 17 companies and 9 countries, the values computed for root node likelihood in trees from different teams differed by a factor of up to 36.
– When a unified fault tree was produced from all the teams, disagreements in the internal probabilities of the tree varied less, but still by a factor of 10.

The work presented here suggests a novel method to resolve Leveson's problem with widely varying root node liklihoods. If funnel theory is correct, then within the space of all disagreements in the unified fault tree, there exist a very small number of key values that crucially impact the root node likelihood. Using TAR2 the feuding teams could restrict their debates to just those key decisions.

We do not use Bayesian reasoning for uncertain models for the same reason we don't use computational intelligence methods. Recall from our introduction that this work assumes *metrics starvation*: i.e. the absence of relevant domain expertise or specific numeric values in the domain being studied. Bayesian methods have been used to sketch out subjective knowledge (e.g. our software management oracle), then assess and tune that knowledge based on available data. Success with this method includes the COCOMO-II effort estimation tool [9] and defect prediction modelling [17]. In the domains where statistical data on cause-and-effect are lacking (e.g. our metrics starved domains), we have to approximate (i.e guess/make-up) some values to describe the model. Since there are too many uncertainties within the model, Bayesian reasoning may not yield stable result.

Other research has explored simulation for making design decisions. Bricconi et.al. [18] built a simulator of a server on a network, then assessed different network designs based on their impact on the server. Menzies and Sinsel assessed software project risk by running randomly selected combinations of inputs through a software risk assessment model [33]. Josephson et.al. [23] executed all known options in a car design to find designs that were best for city driving conditions. Bratko et.al. [4] built qualitative models of electrical circuits and human cardiac function. Where uncertainty existed in the model, or in the propagation rules across the model, a Bratko-style system would backtrack over all possibilities.

Simulation is usually paired with some summarization technique. Our research was prompted by certain short-comings with the summarization techniques of others. Josephson et.al. used *dominance filtering* (a Pareto decision technique) to reduce their millions of designs down to a few thousand options. However, their techniques are silent on automatic methods for determining the difference between dominated and undominated designs. Bratko et.al. used standard machine learners to summarize their simulations. Menzies and Sinsel attempted the same technique, but found the learnt theories to be too large to manually browse. Hence, they evolved a tree-query language (TAR1) to find attribute ranges that were of very different frequencies on decision tree branches that lead to different classifications. TAR2 grew out of the realization that all the TAR1 search operations could be applied to the example set directly, without needing a decision tree learner as an intermediary. TAR2 is hence much faster than TAR1 (seconds, not hours).

7 Conclusion

When not all values within a model are known with certainty, analysts have at least three choices. Firstly, they can take the time to nail down those uncertain ranges. This is the preferred option. However, our experience strongly suggests that funding restrictions and pressing deadlines often force analysts to make decisions when many details are uncertain.

Secondly, analysts might use some sophisticated uncertainty reasoning scheme like bayesian inference or the computational intelligence methods such as neural nets, genetic algorithms or fuzzy logic. These techniques require some minimal knowledge of expert opinion, plus perhaps some historical data to tune that knowledge. In situations of metrics starvation, that knowledge is unavailable.

This paper has explored a third option: try to understand a model by surveying the space of possible model behaviors. Such a survey can generate a data cloud: a dense mass of possibilities with a such a wide variance of output values that they can confuse, not clarify, the thinking of our analysts. However, in the case of data clouds generated from models containing narrow funnels, there exist key decisions which can condense that cloud.

Incremental treatment learning is a method for controlling the condensation of data clouds. At each iteration, users are presented with list of treatments that have most impact on a system. They select some of these and the results are added to a growing set of constraints for a model simulator. This human-in-the-loop approach increases user "buy-in" and allows for some human control of where a data cloud condenses. In the case studies shown above, data clouds where condensed in such a way as to decrease variance and improve the means of the behavior of the model being studied.

As stated in the introduction, there are several implications of this work. Even when we don't know exactly what is going on with a model, it is often possible to:

- Define minimal strategies that grossly decrease the uncertainty in that model's behavior.
- Identify which decisions are redundant; i.e. those not found within any funnel.

Also, when modelling is used to assist decision making, it is possible to reduce the cost of that modelling:

- Even hastily built models containing much uncertainty can be used for effective decision making.
- Further, for models with narrow funnels, elaborate and extensive and expensive data collection may not be required prior to decision making.

TAR2 exploits narrow funnels and is a very simple method for finding treatments at each step of iterative treatment learning. Iterative treatment learning is general to all models with narrow funnels. Empirically and theoretically, there is much evidence that many real-world models have narrow funnels. To test if a model has narrow funnels, it may suffice just to try TAR2 on model output. If a small number of key decisions can't be found, then iterative treatment should be rejected in favor of more elaborate techniques.

Acknowledgements

The research described in this paper was carried out at the University of British Columbia, Vancouver, Canada and at the Jet Propulsion Laboratory, California Institute of Technology, under a contract with the National Aeronautics and Space Administration. Reference herein to any specific commercial product, process, or service by trade name, trademark, manufacturer, or otherwise, does not constitute or imply its endorsement by the United States Government or the Jet Propulsion Laboratory, California Institute of Technology.

References

1. A. Acree. *On Mutations*. PhD thesis, School of Information and Computer Science, Georgia Institute of Technology, 1980.
2. T. Bäck, F. Hoffmeister, and H.-P. Schwefel. A survey of evolution strategies. In R. Belew and L. Booker, editors, *Proceedings of the Fourth International Conference on Genetic Algorithms*, pages 2–9, San Mateo, CA, 1991. Morgan Kaufman.
3. S. Bay and M. Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, 1999. Available from `http://www.ics.uci.edu/~pazzani/Publications/stucco.pdf`.
4. I. Bratko, I. Mozetic, and N. Lavrac. *KARDIO: a Study in Deep and Qualitative Knowledge for Expert Systems*. MIT Press, 1989.
5. I. Bratko. *Prolog Programming for Artificial Intelligence. (third edition)*. Addison-Wesley, 2001.
6. T. Budd. *Mutation analysis of programs test data*. PhD thesis, Yale University, 1980.
7. C. Cai, A. Fu, C. Cheng, and W. Kwong. Mining association rules with weighted items. In *Proceedings of International Database Engineering and Applications Symposium (IDEAS 98)*, August 1998. Available from `http://www.cse.cuhk.edu.hk/~kdd/assoc_rule/paper.pdf`.
8. E. Chiang. Learning controllers for nonfunctional requirements, 2003. Masters thesis, University of British Columbia, Department of Electrical and Computer Engineering. In preperation.
9. S. Chulani, B. Boehm, and B. Steece. Bayesian analysis of empirical software engineering cost models. *IEEE Transaction on Software Engineerining*, 25(4), July/August 1999.

10. L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.
11. L. Console and P. Torasso. A Spectrum of Definitions of Model-Based Diagnosis. *Computational Intelligence*, 7:133–141, 3 1991.
12. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.
13. J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.
14. C. Elkan. The paradoxical success of fuzzy logic. In R. Fikes and W. Lehnert, editors, *Proceedings of the Eleventh National Conference on Artificial Intelligence*, pages 698–703, Menlo Park, California, 1993. AAAI Press.
15. M. Feather, H. In, J. Kiper, J. Kurtz, and T. Menzies. First contract: Better, earlier decisions for software projects. In *Submitted to the ACM CIKM 2001: the Tenth International Conference on Information and Knowledge Management*, 2001. Available from `http://tim.menzies.com/pdf/01first.pdf`.
16. M. Feather and T. Menzies. Converging on the optimal attainment of requirements. In *RE'03 (submitted)*, 2002.
17. N. E. Fenton and M. Neil. A critique of software defect prediction models. *Software Engineering*, 25(5):675–689, 1999. Available from `http://citeseer.nj.nec.com/fenton99critique.html`.
18. E. T. G. Bricconi, E. Di Nitto. Issues in analyzing the behavior of event dispatching systems. In *Proceedings of the 10th International Workshop on Software Specification and Design (IWSSD-10), San Diego, USA*, November 2000.
19. M. Heimdahl and N. Leveson. Completeness and consistency analysis of state-based requirements. *IEEE Transactions on Software Engineering*, May 1996.
20. J. Horgan and A. Mathur. Software testing and reliability. In M. R. Lyu, editor, *The Handbook of Software Reliability Engineering*, pages 531–565, McGraw-Hill, 1996.
21. Y. Hu. Better treatment learning, 2002. Masters Thesis, Department of Electrical Engineering, University of British Columbia, in preperation.
22. J. Jahnke and A. Zundorf. Rewriting poor design patterns by good design patterns. In *Proc. of ESEC:FSE '97 Workshop on Object-Oriented Reengineering*, 1997.
23. J. Josephson, B. Chandrasekaran, M. Carroll, N. Iyer, B. Wasacz, and G. Rizzoni. Exploration of large design spaces: an architecture and preliminary results. In *AAAI '98*, 1998. Available from `http://www.cis.ohio-state.edu/~jj/Explore.ps`.
24. A. Kakas, R. Kowalski, and F. Toni. The role of abduction in logic programming. In C. H. D.M. Gabbay and J. Robinson, editors, *Handbook of Logic in Artificial Intelligence and Logic Programming 5*, pages 235–324. Oxford University Press, 1998.
25. R. Lutz and R. Woodhouse. Bi-directional analysis for certification of safety-critical software. In *1st International Software Assurance Certification Conference (ISACC'99)*, 1999. Available from `http://www.cs.iastate.edu/~rlutz/publications/isacc99.ps`.
26. T. Menzies and P. Compton. Applications of abduction: Hypothesis testing of neuroendocrinological qualitative compartmental models. *Artificial Intelligence in Medicine*, 10:145–175, 1997. Available from `http://tim.menzies.com/pdf/96aim.pdf`.
27. T. Menzies, S. Easterbrook, B. Nuseibeh, and S. Waugh. An empirical investigation of multiple viewpoint reasoning in requirements engineering. In *RE '99*, 1999. Available from `http://tim.menzies.com/pdf/99re.pdf`.
28. T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01lesstalk.pdf`.
29. T. Menzies and Y. Hu. Reusing models for requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://tim.menzies.com/pdf/01reusere.pdf`.
30. T. Menzies and Y. Hu. Agents in a wild world. In C. Rouff, editor, *Formal Approaches to Agent-Based Systems, book chapter*, 2002. Available from `http://tim.menzies.com/pdf/01agents.pdf`.
31. T. Menzies and Y. Hu. Just enough learning (of association rules). In *KDD'02 (submitted)*, 2002. Available from `http://tim.menzies.com/pdf/02tar2.pdf`.
32. T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In *2nd International Workshop on Soft Computing applied to Software Engineering (Netherlands), February*, 2001. Available from `http://tim.menzies.com/pdf/00maybe.pdf`.
33. T. Menzies and E. Sinsel. Practical large scale what-if queries: Case studies with software risk assessment. In *Proceedings ASE 2000*, 2000. Available from `http://tim.menzies.com/pdf/00ase.pdf`.
34. T. Menzies and S. Waugh. On the practicality of viewpoint-based requirements engineering. In *Proceedings, Pacific Rim Conference on Artificial Intelligence, Singapore*. Springer-Verlag, 1998. Available from `http://tim.menzies.com/pdf/98pracai.pdf`.
35. T. Menzies. Practical machine learning for software engineering and knowledge engineering. In *Handbook of Software Engineering and Knowledge Engineering*. World-Scientific, December 2001. Available from `http://tim.menzies.com/pdf/00ml.pdf`.
36. C. Michael. On the uniformity of error propagation in software. In *Proceedings of the 12th Annual Confererence on Computer Assurance (COMPASS '97) Gaithersburg, MD*, 1997.
37. A. Parkes. Lifted search engines for satisfiability, 1999.
38. D. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 5(12):1053–1058, Dec. 1972.
39. M. Paulk, B. Curtis, M. Chrissis, and C. Weber. Capability maturity model, version 1.1. *IEEE Software*, 10(4):18–27, July 1993.
40. R. Quinlan. Induction of decision trees. *Machine Learning*, 1:81–106, 1986.
41. R. Rymon. An SE-tree based characterization of the induction problem. In *International Conference on Machine Learning*, pages 268–275, 1993.
42. R. Rymon. An se-tree-based prime implicant generation algorithm. In *Annals of Math. and A.I., special issue on Model-Based Diagnosis*, volume 11, 1994. Available from `http://citeseer.nj.nec.com/193704.html`.
43. M. Shaw and D. Garlan. *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
44. H. T. Siegelmann and E. D. Sontag. On the computational power of neural nets. *Journal of Computer and System Sciences*, 50(1):132–150, 1995.

45. J. Singer, I. P. Gent, and A. Smaill. Backbone fragility and the local search cost peak. *Journal of Artificial Intelligence Research*, 12:235–270, 2000.
46. M. F. . K. H. S.L. Cornford. Ddp a tool for life-cycle risk management. In *IEEE Aerospace Conference, Big Sky, Montana*, pages 441–451, March 2001.
47. J. D. . M. F. S. Cornford. Optimizing the design of end-to-end spacecraft systems using risk as a currency. In *IEEE Aerospace Conference, Big Sky Montana*, pages 9–16, March 2002.
48. W. Wong and A. Mathur. Reducing the cost of mutation testing: An empirical study. *The Journal of Systems and Software*, 31(3):185–196, December 1995.
49. L. A. Zadeh. Outline of a new approach to the analysis of complex systems and decision processes. *IEEE Transactions on Systems, Man and Cybernetics*, SMC-3(1):28–44, Jan. 1973.