To appear,15th IEEE International Conference on Automated Software Engineering September 11-15, 2000, IMAG Grenoble, France.
http://sigart.acm.org/Conferences/ase/

Practical Large Scale What-if Queries: Case Studies with Software Risk Assessment

Tim Menzies
NASA/WVU IV&V Facility
100 University Drive
Fairmont WV 26554, USA
1-304-367-8447
tim@menzies.com
http://www.tim.menzies.com

Erik Sinsel
NASA/WVU IV&V Facility
100 University Drive
Fairmont WV 26554, USA
1-304-367-8226
esinsel@csee.wvu.edu

ABSTRACT

When a lack of data inhibits decision making, large scale what-if queries can be conducted over the uncertain parameter ranges. Such what-if queries can generate an overwhelming amount of data. We describe here a general method for understanding that data. Large scale what-if queries can guide Monte Carlo simulations of a model. Machine learning can then be used to summarize the output. The summarization is an ensemble of decision trees. The TARZAN system can poll the ensemble looking for majority conclusions regarding what factors change the classifications of the data. TARZAN can succinctly present the results from very large what-if queries. For example, in one of the studies presented here, we can view on $\frac{1}{2}$ a page the significant features from 10^9 what-ifs.

KEYWORDS: Machine learning, ensemble learning, Monte-Carlo simulations, risk assessment, COCOMO-II, decision support systems.

1. Introduction

Incomplete information can cripple decision making. For example, suppose a software manager wants to reduce the odds of time over-runs on her project. To perform this task, our manager could use a software effort estimation model like COCOMO-II (see Figure 1). Our manager may be uncertain about all the details of the current state of the project, or is debating multiple changes to the project. If COCOMO-II is run for each combination of current and possible new values, then our manager would be buried under a mountain of reports. For example, Figure 2 shows the

The COCOMO project aims at developing an open-source, public-domain software effort estimation model. The project has collected information on 161 projects from commercial, aerospace, government, and non-profit organizations[3]. As of 1998, the projects represented in the database were of size 20 to 2000 KSLOC (thousands of lines of code) and took between 100 to 10000 person months to build.
COCOMO measures effort in calendar months where one month is 152 hours (and includes development and management hours). The core intuition behind COCOMO-based estimation is that as systems grow in size, the effort required to create them grows exponentially, i.e. $effort \propto KSLOC^x$. More precisely:

$$months = a * \left(KSLOC^{\left(1.01+\sum_{i=1}^{5} SF_i\right)}\right) * \left(\prod_{j=1}^{17} EM_j\right)$$

where a is a domain-specific parameter, and KSLOC is estimated directly or computed from a function point analysis. SF_i are the scale factors (e.g. factors such as "have we built this kind of system before?") and EM_j are the cost drivers (e.g. required level of reliability).
Software effort-estimation models like COCOMO-II should be tuned to their local domain. Off-the-shelf "untuned" models have been up to 600% inaccurate in their estimates, e.g. [16, p165] and [6]. However, tuned models can be far more accurate. For example, [3] reports a study with a bayesian tuning algorithm using the COCOMO project database. After bayesian tuning, a cross-validation study showed that COCOMO-II model produced estimates that are within 30% of the actuals, 69% of the time.

Figure 1. Some background notes on COCOMO-II. For more details, see [1]

.

			NASA software projects					
			KC-1 (very new project)		FB-3 (moderately new project)			BJ-1 (very mature project)
	ranges		now1	changes1	now2	changes2$_a$	changes2$_b$	now3
Scale drives	prec = 0..5	precedentness	0, 1		2,3			4,5
	flex = 0..5	development flexibility	1, 2, 3, 4	1	?	3,4	0-5	0,1
	resl = 0..5	architectural analysis or risk resolution	0, 1, 2	2	?		0-5	4,5
	team = 0..5	team cohesion	1, 2	2	4			3,4
	pmat = 0..5	process maturity	0, 1, 2, 3	3	?		0-5	4,5
Product attributes	rely = 0..4	required reliability	4		4			4
	data = 1..4	database size	2		?		1-4	1,2
	cplx = 0..5	product complexity	4, 5		3,4,5			3,4,5
	ruse = 1..5	level of reuse	1, 2, 3	3	?		1-5	4,5
	docu = 0..4	documentation requirements	1, 2, 3	3	1			3,4
Platform attributes	time = 2..5	execution time constraints	?		5	4		2,3
	stor = 2..5	main memory storage	2, 3, 4	2	?		2-5	3,4
	pvol = 1..4	platform volatility	1		?		2-4	1,2
Personnel attributes	acap = 0..4	analyst capability	1, 2	2	2			2,3,4
	pcap = 0..4	programmer capability	2		2			2,3
	pcon = 0..4	programmer continuity	1, 2	2	?		0-4	2,3
	aexp = 0..4	analyst experience	1, 2		?		0-4	3,4
	pexp = 0..4	platform experience	2		?		0-4	3,4
Project attributes	ltex = 0..4	experience with language and tools	1, 2, 3	3	2			3,4
	tool = 0..4	use of software tools	1, 2		1	2,3		3,4
	site = 0..5	multi-site development	2		?		0-5	?
	sced = 0..4	time before delivery	0, 1, 2	2	?		0-4	2
		# of what-ifs (combinations of $nowX \cup changesX$) =	$6 * 10^6$		$3 * 10^9$		10^9	10^7

Figure 2. Three software NASA projects: "now"= current situation; "changes"= some proposed changes. Attributes come from the COCOMO-II software cost estimation model described in Figure 1. Attribute values of "2" are nominal; lower values usually denote some undesirable situation; higher values usually denote some desired situation. Each "don't know " (denoted "?") requires what-if queries for the entire range of that parameter. No changes are shown for BJ-1 since, in the view of the managers, this project has already had years of useful process improvement.

current state and proposed changes to three NASA projects. In the bottom row, we see these projects contain 10^6 to 10^9 combinations of current and proposed parameters.

This paper proposes a new method for practical large scale what-if queries. We characterize exploring a space of what-ifs as the search for the *significant ranges*; i.e. a small set of parameter ranges that have most impact on achieving some desired results. Our approach has three stages. Firstly, we build the space of potentially significant what-ifs by asking "what are the different models postulated for this domain?". The significant ranges lies somewhere within the input parameter ranges of these models (e.g. column 2 of Figure 2). Secondly, we generate and cache behavior from our range of models using random Monte Carlo simulations. This behavior is summarized by a decision tree learner. This study used C4.5 [17]. Thirdly, we pass the learnt trees to our TARZAN package. TARZAN uses the trees and domain information such as Figure 2 to savagely prune the ranges. TARZAN's pruning methods (described below) can be very effective. For example, in this study (see below), TARZAN generated a $\frac{1}{2}$ page report that shows the significant features within a space of 10^9 what-ifs.

This rest of this paper is structured as follows. First, we discuss the novel contributions and generality of this research. Second, we offer some briefing notes on technologies that are core to our technique: random Monte Carlo simulations, decision tree learning, and the TARZAN pruning methods. Third, we describe the Madachy risk model which will assess the software risk of the projects in Figure 2 (the Madachy model was first reported at KBSE'94 [9]). Finally, we show how these technologies work on the NASA case studies of Figure 2.

1.1. Originality

This is the first report to detail the use of automatic software engineering techniques for the independent assessment of software projects. We believe this research offers the possibility for better conflict reduction in requirements engineering, faster reasoning in the presence of uncertainty, and easier explanation of automatically generated theories.

Independent assessment: Recent satellite losses have highlighted NASA's need for quality software. An independent assessment is an early life-cycle assessment of the risks within a software project. The assessment is performed by a consultant from outside the development team. NASA can save high risk projects if we can find them early enough; e.g. by allocating additional resources. For such assessments to be useful, however, they must be cheap to conduct and be performed early in the life cycle. Our method supports such early life cycle risk assessments, even before key parameters are not known with certainty. Further, we not only assess risk, but also explore how to alter it. After building decision trees that summarize our risk knowledge, we then query those trees looking for forks that can alter the classifications. In particular, TARZAN returns those parameter ranges that, if applied to some project, significantly reduce that project's risk.

Requirements engineering: Requirements engineers note that decision making can be complicated by the conflicting plans of multiple stakeholders [5]. One subtle feature of TARZAN is that it is a conflict reduction tool. Given a set of proposed changes (e.g. the *changesX* columns in Figure 2), TARZAN returns a smaller set containing the changes that are most influential. For example, only two of the 11 members of *changes1* are effective in reducing the risk of the KC-1 project (see below). Without TARZAN, our stakeholders might have ($2^{11} = 2048$) arguments as they discuss which portions of *changes1* to apply. With TARZAN, our stakeholders need only have ($2^2 = 4$) arguments.

Reasoning in the presence of uncertainty: Uncertainty is fundamental to many under-measured domains, e.g. human internal medicine, economics [13], and software development. For example, software development data may be scarce if the development team was not funded to maintain a metrics repository, or the collected data does not relate to the business case, or if contractors prefer to retain control of their own information. After two years of watching our NASA colleagues chase data, we suspect that data famines cannot be solved by management directives to collect more data. Our view is that we should take data famines as a premise, and then research how to reason in their presence. Our prior research into reasoning about uncertain domains used abductive logics to implement HT4: a set-covering truth maintenance system [13]. Such logical approaches to uncertain reasoning can be too slow: HT4's inference is provably NP-hard and experimentally exponential on model size. TARZAN, on the other hand, is a much simpler, faster method. TARZAN learns emergent stable properties from a wide range of randomly selected behavior. Assuming master-variable systems, such emergent stable properties should be few in number and fast to find.

Explanation of automatically generated theories: We show below that the explanation of a large learnt theory can require special explanation facilities. Such explanation facilities are not usually researched by the machine learning community. Most machine learning research merely generates theories but does not describe the post-processing of that theory. Two exceptions are the researchers who explored alternate representations to decision trees: Quinlan's rule generator [17] and Muggleton's work on learning horn clauses [15]. In part, the work of Muggleton and Quinlan was motivated by the problem of explaining large decision trees. Unlike these studies, our solution to the explanation problem only requires standard decision tree learners, plus (< 300) lines of simple Prolog to explore the learnt trees. We assume that when a human operator "understands" a tree, they can answer the following question:

> Here are some things I am planning to change; tell me the smallest set that will change the classifications of this system.

Note that this question can be answered by hiding the learnt trees and merely showing the operator the significant ranges. TARZAN swings through the decision trees looking for the significant ranges that pick interesting branches.

1.2. Generality

The Madachy model is not core to our work; rather it is just an example we use to demonstrate a new technique for automatically understanding models. In theory, there is nothing in our system preventing us from swapping the Madachy model for another. That is, while the case study in this paper relates to software risk management, our technique potentially applies more broadly. For example, in our conclusion, we discuss applications of this technique to the treatment of diabetics.

On the other hand, this approach is only practical when a model's behavior can be quickly sampled, then quickly pruned. Thus:

- To quickly sample a model's behavior, it has be run many times (e.g. this study ran a model 900,000 times). Hence, we cannot use this approach for systems that are slow to execute, or which generate extensive side-effects when executed; e.g. adding megabytes to a database with each execution.

- To quickly prune a system's behavior, that system must contain a small number of *master variables* that control the larger number of slave variables in the rest of the system. In such master-variable systems, pruning terminates on a very small set or master variable ranges. Elsewhere, we have offered evidence from theory [12], experimentation [14], and an extensive literature review [11] suggesting master-variable systems are common, even for indeterminate systems.

2. The Madachy Risk Model

For our experiments, we used the Madachy COCOMO-based effort-risk model [10]. Dr. Madachy is one of the authors of the COCOMO-II model description [1]. The Madachy model was an experiment in explicating the heuristic nature of effort estimation. The model contains 94 tables of the form of Figure 3. Each such table implements a context-dependent modification to internal COCOMO parameters. Two important features of the Madachy model are its classifications and its validation. In the first case, the model generates a numeric effort-risk index which is then mapped into the classifications *low, medium, high*. In the second, the model has survived at least one validation study. Most risk models come with no validation information. The Madachy model is the rare exception. Studies with the COCOMO-I project database have shown that the Madachy index correlates well with $\frac{months}{KDSI}$ (where KDSI is thousands of delivered source lines of code) [10].

The Madachy model was exercised using Monte Carlo simulations, then summarized with a machine learner (described below). Hence, our summaries reflect the biases of the Madachy model. Recall that Madachy validated his model by finding a good correlation between the risk assessments of his model and the development times of the systems recorded in the COCOMO database. That is, Madachy defines risk as a *development* issue. This is different to the standard software risk assessment view, which defines risk as some measure of the runtime performance of the system; in other words, in the standard view risk is defined as an *operational* issue (e.g. [7, 8]). Since we are using the Madachy model, we must adopt his definitions. Hence, when we say "software risk"we mean "development risk"; i.e. the risk that the project while take longer than planned to build. Note that our methods could also be used for operational risk assessment, if we could access such a development risk model.

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

Figure 3. A Madachy factors table. From [10]. This table reads as follows. In the exceptional case of high reliability systems and very tight schedule pressure (i.e. *sced=low or very low* **and** *rely= high or very high***), add some increments to the built-in parameters (increments shown top-right). Otherwise, in the non-exceptional case, add nothing to the built-in parameters.**

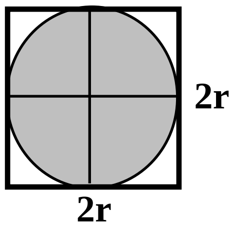

Figure 4. A circle of radius r within a square of side $2r$. From [18]

Madachy's model uses off-the-shelf COCOMO, which can be up to 600% inaccurate in its estimates (see the discussion in Figure 1). In practice, users tune COCOMO's parameters using historical data in order to generate accurate estimates. For the projects in Figure 2, we lacked the data to calibrate the model. Therefore we ran our simulations using three different COCOMO tunings and found that we generated the same significant ranges from all three tunings (see below). Hence, we were not motivated to explore other tunings.

3. Monte Carlo Simulations

The Madachy model was exercised using Monte Carlo simulations. That is, instead of running (e.g.) 10^9 simulations, we ran a small number picked at random to see what we could learn. We then ran a larger number of randomly picked simulations. A conclusion was deemed *stable* if it did not change when we used a larger sample size. The rest of this section offers some introductory notes on Monte Carlo simulations and their application in our domain.

Traditionally, MC methods were used for mathematical integration. For example, to compute the value of π using MC, we might ask a very bad darts player to throw darts at Figure 4. In this approach, our darts player is a stochastic generator of the data. Assuming all the darts land within the square of Figure 4, then the ratio of darts hitting the circle will be:

$$\frac{\#\ darts\ hitting\ circle}{\#\ throws} = \frac{\pi r^2}{(2r)^2}$$

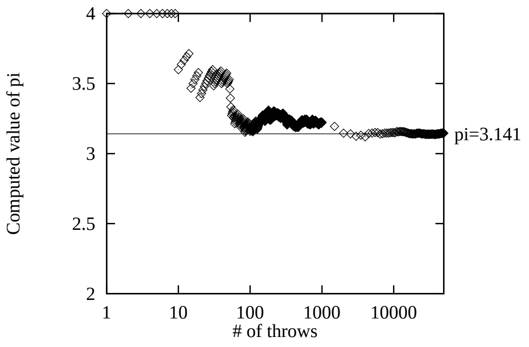

Figure 5. Finding π using MC methods; i.e. throwing darts at Figure 4 and applying Equation 1. Adapted from [18]

which we can rearrange to

$$\pi = \frac{4 * \#\ darts\ hitting\ circle}{\#\ throws} \tag{1}$$

Note that the results of an MC study can be skewed by inadequate sampling. If our darts player only makes a few throws, our value for π will be inaccurate. Figure 5 shows how the value of Equation 1 varied as a computer program simulated the darts player. The value for π seen after 10 "throws" was very different to the value seen after 1000 "throws". However, after 5000 "throws", the value *stabilized*; i.e. more throws did not significantly change the value. The general lesson from this example is that the sample size of an MC study must be extended until the conclusions *stabilize*.

In order to apply MC to software risk management, we ran the Madachy model using randomly selected inputs as follows:

- Since the outputs of a COCOMO model are dependent on its tunings, we picked our tunings from several published sources. Those sources showed tunings generated via genetic algorithms [4], tunings generated via bayesian learning [3], and the standard tunings found within the Madachy model.
- COCOMO estimations are based on source lines of code (SLOC). SLOC is notoriously hard to estimate. Hence, we ran our MC simulations on different ranges of SLOCs. From Boehm's text *Software Engineering Economics*, we saw that using $SLOC = 10K, SLOC = 100K, SLOC = 2000K$ would cover an interesting range of software systems [2].
- Next, for the three different SLOCs and three tunings, we generated 100,000 random examples by picking one value at random for each of the parameters from column 2 of Figure 2.

This generated 900,000 examples classified *low*, *medium*, *high*, divided into different SLOCs and tunings. We made our conclusions via machine learning (see below) using N randomly selected examples from each division. We increased the sample size (to $2N, 3N, ...$) until we found conclusions that were stable across all divisions. In our experiments, $N = 10,000$ samples and our conclusions usually stabilized at $3N = 30,000$ samples. These 45 training sets were used to build an ensemble of 45 trees:

$$3\ SLOCs\ * 3\ tunings\ *\ 5\ samples\ =\ 45\ trees$$

4. Decision Trees

Our conclusions were found using the TARZAN system. TARZAN swings through the 45 trees generated from the above 45 experiments looking for parameter changes that altered the classifications. Before discussing TARZAN, we digress for a brief tutorial on decision tree learners.

Decision tree learners input classified examples and output decision trees (see Figure 6). This study used C4.5 [17].

INPUT:

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

OUTPUT (estimated error=38.5%):

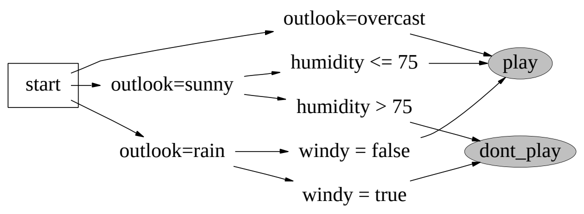

Figure 6. Decision-tree learning. Classified examples (above) generate the decision tree (below).

C4.5 is an international standard in machine learning; most new machine learners are benchmarked against this program. The algorithm uses a heuristic *entropy* measure of information content to build its trees. The parameter range with the most information content (highest entropy) is selected as the root of a decision tree. The example set is then divided up according to which examples do/do not satisfy the test in the root. For each divided example set, the process is then repeated recursively. A statistical measure is then used to estimate the classification error on unseen cases.

For example, consider the decision tree learnt by C4.5 in Figure 6. In that tree, C4.5 has decided that the weather *outlook* has the most information content. Hence, it has placed *outlook* near the root of the learnt tree. If *outlook=rain*, a sub-tree is entered whose next-most critical parameter is *wind*. We see that we should not play golf on high-wind days when it might rain. Note that C4.5 estimates that this tree will lead to incorrect classifications 38.5 times out of 100 on future cases. We should expect such a large classification errors when learning from only 15 examples. In general, C4.5 needs hundreds to thousands of examples before it can produce trees with low classification errors. Hence, when we build trees, we should increase the sample size till the estimated error drops to an acceptable level. Figure 7 shows the error levels in our 45 trees built from 10-50K samples: our errors fall to low levels after tens of thousands of samples. Observe that after $4 * 10^4$, the error curve is flat; i.e. we learn nothing further about the system. That is, we need not explore all the (e.g.) 10^9 what-ifs from *changes2*$_b$ in Figure 2 (since most of these what-ifs have the same overall result). Secondly, we see that by 50K samples, our trees are over 6000 nodes in size. Humans have much difficulty reading these large trees. Such large trees need an automatic explanation facility such as TARZAN.

5. TARZAN's Pruning Methods

This section describes the pruning methods used by TARZAN. Note that:

- Pruning method P_i culls some output Out_j as follows: $Out_i = P_i\,(Out_{i-1})$.
- Out_0 is initialized to include all the known ranges of all the known variables.
- TARZAN returns the ranges Out_8 after applying the prunings $P_1, P2, ..., P_8$.

5.1. Pruning Methods: The Golf Example

The first four pruning methods $P_1..P_4$ will be demonstrated using the golf example of Figure 6.

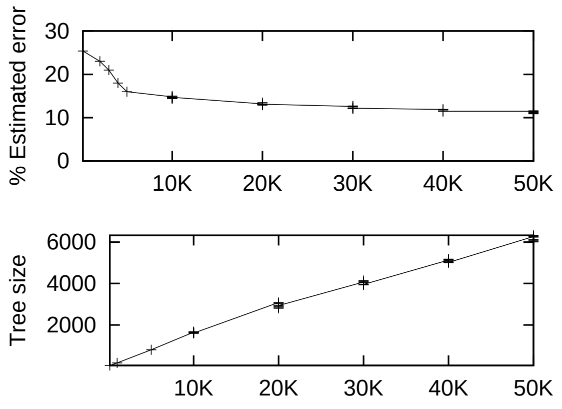

Figure 7. Error (top) and size (bottom) in the 45 trees learnt at five different sampling sizes.

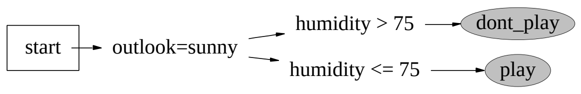

Figure 8. P_2 pruning: example #1. The golf tree, after pruning branches that contradict *outlook=sunny***.**

$Out_1 = P_1\,(Out_0)$: The entropy measure of C4.5 performs the first pruning. Recall that C4.5 selects the parameter ranges for the tree using entropy. Parameter ranges with high entropy appear high in the tree (e.g. *outlook*), while low entropy attributes may disappear from the tree all together. For example, note that *temperature* has been pruned from the learnt trees of Figure 6.

$Out_2 = P_2\,(Out_1)$: P_2 prunes ranges that contradict the domain constraints; e.g. $nowX \cup changesX$ from Figure 2. Also, when applying P_2, we prune tree branches that use ranges from the discarded set. For example, suppose some golf course had weather control technology and so was considering changes for $constraints = \{outlook = sunny\}$. Figure 8 shows how this second prune using shrink the tree learnt in Figure 6.

$Out_3 = P_3\,(Out_2)$: P_3 prunes ranges that do not change classifications in the trees. For example, in Figure 8, the range *outlook=sunny* does not appear in some change that alters our golf-playing behavior.

$Out_4 = P_4\,(Out_3)$: P_4 prunes ranges that are not interesting to the user, i.e. those that are not mentioned in the $changesX$ set ($Out_4 = Out_3 \wedge changesX$).

5.2. More Pruning Methods: The KC-1 Example

$P1, ...P4$ apply to single trees. The remaining pruning methods take Out_4s generated from all the single trees, then applied that across the ensemble. These remaining pruning methods $P_5..P_8$ will be demonstrated using one of our software project management examples; i.e. KC-1 from Figure 2.

$Out_5 = P_5\,(Out_4)$: P_5 prunes ranges that do not change classifications in the majority of the members of the ensemble (our threshold is 66%). For example, in the case of the KC-1 domain from Figure 2, we applied $P_1..P_5$ to generate a set of ranges that change risk classifications. Of the 11 changes found in KC-1's $changes1$ set, only 4 ranges change the classifications in more than 66% of the decision trees. Hence, Out_5 only contained 4 ranges.

$Out_6 = P_6\,(Out_5)$: P_6 prunes ranges that are not stable, i.e. those ranges that are not always found at the larger samples of the random Monte Carlo simulations.

$Out_7 = P_7\,(Out_6)$: P_7 explores all subsets of Out_6 to reject the combinations that have low impact on the classifications. The impact of each subset is assessed via how that subset changes the number of branches to the different classifications. For example, Figure 9 shows how some of the subsets of the ranges found in Out_6 affect KC-1. In that figure, bar chart A1 shows the average number of branches to *low*, *medium*, and *high* risk seen in the 45 trees. For each subset $X \subseteq Out_6$, we make a copy of the trees pruned by P_2, then delete all branches that contradict X. This generates 45 new trees that are consistent with X. The average number of branches to each classification in these new branches is then calculated. P_7 would reject the subsets shown in B1, C1, and B2 since these barely alter the current situation in A1.

$Out_8 = P_8\,(Out_7)$: P_8 compares members of Out_7 and rejects a combination if some smaller combination has a similar effect. For example, in Figure 9, we see in A2 that having moderately talented analysts and no schedule pressure (*acap=[2], sced=[2]*) reduces our risk nearly as much as any larger subset. Exception: C2 applies all four actions from KC-1's Out_6 set to remove all branches to medium and high risk projects. Nevertheless, we still recommend A2, not C2, since A2 seems to achieve most of what C2 can do, with much less effort.

OUT_8 is reported back to the user. Note that, in the KC-1 study, we have found 2 significant ranges out of a range of $6 * 10^6$ what-ifs and 11 proposed new changes.

TARZAN took 8 minutes to process KC-1 on a 350MHz machine. That time was divided equally between Prolog code that implements the tree processing and some inefficient shell scripts that filter the Prolog output to generate Figure 9. We are currently porting TARZAN to "C" and will use bit manipulations for set processing. We are hopeful that this new language and smarter processing will make TARZAN much faster.

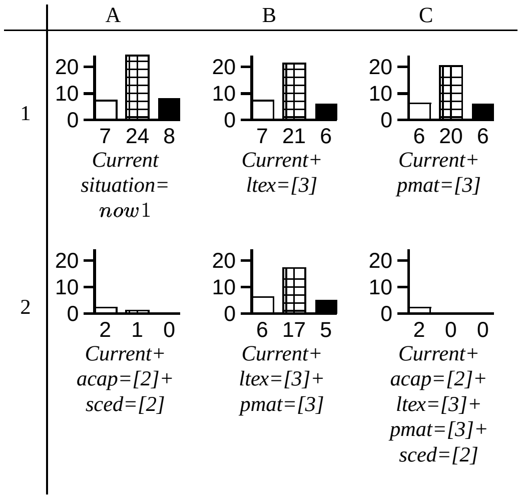

Figure 9. Average number of branches to different classifications in the 45 trees in KC-1, assuming different subsets of the ranges seen in KC-1's Out_6 set. Legend: ☐ =low risk ▦ =medium risk ■ =high risk.

6. Other Case Studies

The above technique was applied to the BJ-1 project, i.e. *now3* from Figure 2. BJ-1 is the most mature software program at NASA (and KC-1 is one of the newest programs). Over the years, significant resources have been allocated to BJ-1 to improve its quality. The pre-experimental expectation was that BJ-1 would be assessed as a much lower risk project than KC-1. This was indeed the case. Nearly all the branches remaining after P_2 went to low risk. Hence, P_3 returned very few non-empty sets and BJ-1's processing was terminated after P_3.

Our next case study used the FB-3 project; i.e. $now2 \cup changes2_a$ from Figure 2. FB-3's current situation is the A1 bar chart in Figure 10: there are numerous ways to high risk projects in FB-3. Unfortunately, TARZAN was unable to offer risk mitigation strategies. $Changes2_a$ lists so few possible changes that P_4 always returned. Hence, we decided to explore the changes not mentioned in $changes2_a$, just to see if there existed useful changes that we had missed. In this next study, we used $now2 \cup changes2_b$. This generated the largest what-if space yet processed by TARZAN (10^9). Out_4 contained hundreds of entries so we increased the P_5 threshold to 100%; i.e. P_5 culls any ranges that do not change classifications in all trees. This

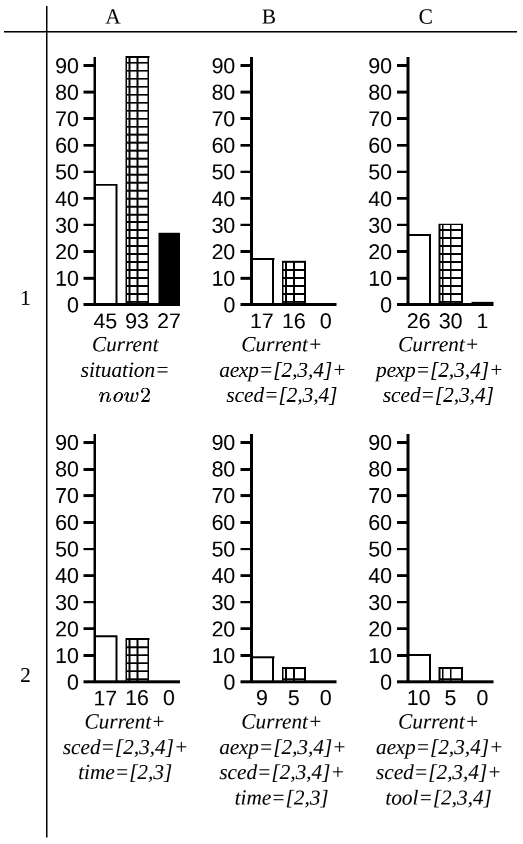

Figure 10. Average number of branches to different classifications in the 45 trees in FB-3, assuming some subsets of $changes2_b$ from Figure 2. Legend: ☐ =low risk ▦ =medium risk ■ =high risk.

very strict pruning resulted in Out_5 containing 50 items. Before P_7 explored all subsets of 50 items, we manually culled Out_5 back to six items that we believed would be inexpensive to change (e.g. we culled changes to process maturity). The $now2 \cup changes2_b$ study took 2 hours to complete. Figure 10 shows some results from the P_8 pruning. Note that some change are clearly inferior (e.g. C1). If our users could tolerate more than two changes, we would recommend either B2 or C2. Otherwise, if our users are seeking the least change for the most benefit, we would recommend either B1 or A2.

7. Discussion

We have shown that the results from large scale what-if queries can be automatically surveyed and succinctly summarized via:

Part 1: Identifying the ranges implied by the what-if queries;

Part 2: Conducting Monte Carlo simulations of some model across those ranges to generate examples;

Part 3: Performing machine learning to convert the examples into an ensemble of decision trees;

Part 4: Using TARZAN to find mitigation strategies that changes classifications in a majority of the trees.

This technique is useful for more than just the domain of software risk mitigation. By swapping the model used in **Part 2**, we should be able to apply this method to other domains. For example, we are currently exploring applying this technique to diet and exercise planning for diabetic patients. We have accessed from the world-wide web a commonly-used 15-parameter model of blood sugar levels in humans. This model is being installed into **Part 2** of our rig. Using the model, we intend to learn the simplest action patients should take to maintain their blood sugar at the correct level.

This technique may also be useful for more than just the task of automatically generating plans to improve the behavior of a system. TARZAN can support a range of other tasks such as classification, prediction, diagnosis, monitoring, validation, maintenance, and multiple-stakeholder requirements engineering in uncertain domains:

- Decision tree learners can auto-generate classifiers and predictors.
- Decision-tree learners can generate diagnosis knowledge for detecting faults if they learn from examples containing faulty behavior.
- TARZAN can also rationalize and reduce the cost of monitoring a system. For example, once TARZAN has identified the crucial change parameters, metrics collection on software projects could be restricted to just those variables. This could significantly reduce the cost of metrics collection.
- To validate a TARZAN-based system, test engineers can inspect Out_8 to find the effects of changing key parameters in the system. This validation scheme could fault the model used in **Part 2** if the test engineers find that system behavior changes inappropriately when the key control parameters change. We envision that this style of validation will become very important to organizations like NASA in the near future. NASA already has hundreds of simulators of flight systems. Such simulators are used to explore alternatives in system design and flight profiles. Tools like TARZAN can be used to check if those simulators are generating sensible output.

- TARZAN-based systems are simple to maintain. When domain knowledge changes, we must manually change the model used in **Part 2**. However, once that change has been made, we can then automatically generate classifiers, predictors, diagnosis engines, planners, validation tools, and monitors.
- As noted above, TARZAN could also assist a community of stakeholders as they debate different method of implementing their classifiers, predictors, diagnosis engines, planners, monitoring, validation, and maintenance regimes. When conflicts arise between stakeholders, TARZAN can find which decisions are crucial and which arguments do not impact the system. Debates could then be shortened to just the crucial issues.
- Lastly, the examples presented here suggest that TARZAN can perform the above tasks in domains with less-than-certain information.

What we cannot show at this time is evidence that the software risk mitigation strategies found above actually decrease software development risk. Given the data famine in the software engineering industry in general, and at NASA in particular, we are unsure when such an assessment could be applied. While we await an end to the data famine, software must be built and project managers must make decisions based on the models and data at hand. In the FB-3 study, we saw that TARZAN could (1) handle large scale what-if queries to (2) find risk mitigation strategies overlooked by humans. From the BJ-1 and KC-1 studies, we saw that (3) TARZAN was able to distinguish low risk from higher risk projects, even when many parameters were unknown precisely. All three observations suggestion that TARZAN could be a useful tool for assessing decisions when faced with a poverty of data and uncertainty within the models.

Acknowledgements

Tim Kurtz was patient enough to try and correct some of our early misunderstandings of COCOMO. Barry Boehm and Ray Madachy were kind enough to grant us access to the source code of the Madachy COCOMO-based risk assessment model. John R. Josephson first proposed that we run a small number of randomly selected inputs to the system. Naresh Iyesr enthusiastically supported our earlier experiments with summarizing hundreds of thousands of calls to COCOMO-II. This work was partially supported by NASA through cooperative agreement #NCC 2-979.

References

[1] C. Abts, B. Clark, S. Devnani-Chulani, E. Horowitz, R. Madachy, D. Reifer, R. Selby, and B. Steece. COCOMO II Model Definition Manual. Technical report, Center for Software Engineering, USC,, 1998. `http://sunset.usc.edu/COCOMOII/cocomox.html#downloads`.

[2] B. Boehm. *Software Engineering Economics*. Prentice Hall, 1981.

[3] 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.

[4] R. Cordero, M. Costamagna, and E. Paschetta. A Genetic Algorithm Approach for the Calibration of COCOMO-like Models. In *12th COCOMO Forum*, 1997.

[5] S. Easterbrook. Handling conflicts between domain descriptions with computer-supported negotiation. *Knowledge Acquisition*, 3:255–289, 1991.

[6] C.F. Kemerer. An Empirical Validation of Software Cost Estimation Models. *Communications of the ACM*, 30(5):416–429, May 1987.

[7] N. Levenson. *Safeware System Safety And Computers*. Addison-Wesley, 1995.

[8] M.R. Lyu. *The Handbook of Software Reliability Engineering*. McGraw-Hill, 1996.

[9] R. Madachy. Knowledge-based risk assessment and cost estimation. In *Proceedings Ninth Knowledge-Based Software Engineering Conference*, pages 172 –178, 1994.

[10] R. Madachy. Heuristic Risk Assessment Using Cost Factors. *IEEE Software*, 14(3):51–59, May 1997.

[11] T. Menzies and B. Cukic. On the Sufficiency of Limited Testing for Knowledge Based Systems. In *The Eleventh IEEE International Conference on Tools with Artificial Intelligence. November 9-11, 1999. Chicago IL USA.*, 1999. Available from `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-011.pdf`.

[12] Tim Menzies, Bojan Cukic, Harhsinder Singh, and John Powell. Testing Indeterminate Systems, 2000. ISSRE 2000 (to appear).

[13] T.J. 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://www.cse.unsw.edu.au/~timm/pub/docs/96aim.ps.gz`.

[14] T.J. Menzies, S. Easterbrook, Bashar Nuseibeh, and Sam Waugh. An Empirical Investigation of Multiple Viewpoint Reasoning in Requirements Engineering. In *RE '99*, 1999. Available from `http://research.ivv.nasa.gov/docs/techreports/1999/NASA-IVV-99-009.pdf`.

[15] S. Muggleton. Inductive Logic Programming. *New Generation Computing*, 8:295–318, 1991.

[16] T. Mukhopadhyay, S.S. Vicinanza, and M.J. Prietula. Examining the Feasibility of a Case-based Reasoning Tool for Software Effort Estimation. *MIS Quarterly*, pages 155–171, June 1992.

[17] J.R. Quinlan. Induction of Decision Trees. *Machine Learning*, 1:81–106, 1986.

[18] J. Woller. The Basics of Monte Carlo Simulations, 1996. University of Nebraska-Lincoln Physical Chemistry Lab. Available at `http://wwitch.unl.edu/zeng/joy/mclab/mcintro.html`.