# Qualitative Modeling for Requirements Engineering

Tim Menzies
Computer Science
Portland State University
tim@menzies.us

Julian Richardson
RIACS/USRA, NASA Ames Research Center
Moffett Field, CA 94035, USA
julianr@riacs.edu

## Abstract

*Acquisition of "quantitative" models of sufficient accuracy to enable effective analysis of requirements tradeoffs is hampered by the slowness and difficulty of obtaining sufficient data. "Qualitative" models, based on expert opinion, can be built quickly and therefore used earlier. Such qualitative models are nondeterminate which makes them hard to use for making categorical policy decisions over the model. The nondeterminacy of qualitative models can be tamed using "stochastic sampling" and "treatment learning". These tools can quickly find and set the "master variables" that restrain qualitative simulations. Once tamed, qualitative modeling can be used in requirements engineering to assess more options, earlier in the life cycle.*

## 1 Introduction

In *model-based requirements engineering* (RE) [14], the constraints in a model are like an *umprire* that can resolve stakeholder feuds by e.g.:

- Showing how some options are clearly undesirable;
- Finding a novel solution that keeps most of the stakeholders mostly satisfied.

There is a tension between how *fast* a model is written and how *soon* we can use it to make definitive decisions. For example, early in the software life cycle, it is simple and fast to generate qualitative models of a domain. However such nondeterministic qualitative models can generate a wide range of output. Hence, they may be useless for model-based RE since they neither constrain nor refine an argument. Clancy and Kuipers observe that…

> Intractable branching due to (nondeterminacy) is one of the major factors hindering the application of qualitative reasoning techniques to large real-world problems [2].

This paper is far more optimistic about using qualitative models for RE. In some models, the space of options in the whole model reduces to just the choices in a small number of *master variables*. Such qualitative models can be used for model-based requirements engineering, despite their nondeterminacy as follows: *learn* from *stochastic samples* of qualitative models to find *settings to the master variables* — *treatments* — that *improve the performance* of the qualitative models.

Previously, we have argued the *theoretical potential* of our method. At RE'99, with Easterbrook, Nuseibeh and Waugh, we reported results from millions of simulations of thousands of randomly generated models showing that most of what was seen in any option could be found in a small number of randomly selected models [17]. A subsequent mathematical analyses suggested that (a) these empirical results are the expected results from models with master variables and (b) in the usual case, most models have only a few master variables [21]. These results inspired tools such TAR3, an efficient search engine for *treatments*: i.e. settings to the *fewest* master variables that *most* improve the behavior of a model [18].

Despite all that theoretical work, this paper is this the first time that our method has been *field-tested* on a real-world requirements document[1]. In this study, we read a document, quickly sketched out a qualitative model, then used treatment learning to find definitive decisions. Our document was a discussion of software process options for NASA Near Earth Orbit Rendezvous (NEAR) missions [11].

Figure 1 shows the distribution of utilities for ($E_0$) a baseline of random process options, ($E_1$) process options which include the "best" single process action found by the TAR3 learner, but are otherwise random, and ($E_2$) process options which are based on the treatments (which can be conjunctions of several process actions) produced by the TAR3 learner. Figure 1 shows that $E_1$ performs better than the $E_0$ baseline, but is inferior to $E_2$. Note that the $E_2$ treatments nearly doubled that baseline performance from 20 to

[1]See [6,20] for studies with treatment learning on more precise models

35. Despite qualitative nondeterminacy, we can make definite decisions about *ranking* and *rejecting* software process options.

The rest of this paper describes how Figure 1 was generated and discusses the generality of our technique.

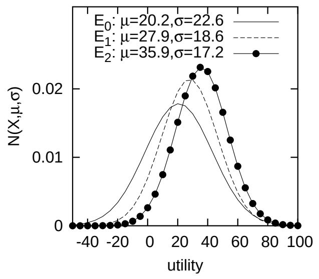


**Figure 1. Results.**

The framework we describe in this paper has several ingredients:

1. Section §2 describes a simple representation of the effects of different process options. The simplicity of the representation is advantageous for early requirements engineering: it does not require large or complex axiomatizations of a possibly poorly-understood domain, and represents effects qualitatively. This representation is easily translated into executable form (which is thus amenable to analysis). The translation is simple (each row of the influence table is translated into a single Prolog clause), adaptable, but has good foundations in previous work on qualitative simulation.

2. Section §3 describes how we can extract definite recommendations from the model above using a simulator which quickly generates very large numbers of random or semi-random process options. The process options are semi-randomly ranked and fed to a learner which produces recommendations of which actions should or should not be performed.

In §4 we describe the use of the framework and present results. In §5 we conclude.

## 2 Qualitative Modeling Early Requirements

A degree of imprecision in modeling can help early life cycle requirements engineering. Goel studied designers using *well-structured* diagramming tools (MacDraw) and an *ill-structured* diagramming tool (freehand sketches using pencil and paper). Structured tools were found to inhibit creativity while ill-structured tools generated more design variants (i.e. more drawings, more ideas, more use of old ideas) [7].

Because such imprecise models are useful, it is common to see RE frameworks where the semantics of link information is under-constrained. For example, Chung et.al's soft-goals use qualitative influences like "makes", "breaks", "helps" and "hurts" [1]. Chung's framework is silent on computational mechanisms for handling combinations of competing influences such as "helps" plus "hurts".

Similar issues appear in other work. Shaw and Garlan use qualitatively influences such as "strong", "weak", "medium" to model trade-offs between alternative software architectures using QFD ("Quality Functional Deployment") tables [28, p119]. In QFD it is under-defined what happens when "weak positives" combine with "weak negatives". Also, MacLean et.al. use "encourage" and "discourage" links in their a Questions Options Criteria (QOC) graphs to rapidly record trade-offs between issues during a decision discussion [13]. The QOC work does not specify what happens when the same option is influenced by combinations "encourages" and "discourages".

To be fair, the above researchers never claimed that their representations are executable. For the most part, softgoals, QFDs, and QOCs are manual browsing tools to be used interactively in design meetings. Commonly, these representations are precursors to other, more elaborate and more time-consuming modeling.

Our research takes a different approach: our imprecise models *must* execute. NASA often conducts week-long intensive *tiger team* design discussions that develop *mission concept* documents[2]. These documents can be used to make decisions about projects that will cost hundreds of millions of dollars. Often those decisions are based on qualitative information such as the operational parameters of a satellite that does not yet exist nor has never been built before.

We seek to add value to those tiger team meetings with automatic agents that point out the best decisions that could be made from the available qualitative information. These agents have two requirements. First, they must run fast enough to keep with the dialogue of the tiger team. Second, these agents must not slow down the tiger team's dialogue by, e.g. asking for precise details about satellites that don't exist yet. Hence, our agents need a fast execution mechanism for qualitative models. Our representation is considerably more lightweight than for example *i** [31, 32], which allows rich representations of processes involving different actors, logical and soft/qualitative task decompositions, but where we lose out on the ability to model complex dependencies in requirements problems, we gain in the speed with which models can be constructed and, more significantly, from the fact that we can *execute* our models — enabling us to explore different scenarios (cf. [30]) — and *learn from* our models.

Figure 2 is an example of the kind of information we can extract from those design meetings. To be precise, we

[2]At the NASA Jet Propulsion Laboratory in California, they are called *Team X* meetings. At the NASA Goddard Space Flight Center, these are the meetings conducted at the *Integrated Mission Design Center* `http://imdc.nasa.gov/`.

$$Utility = \sum_{x \in Goals} U_x \sum_{y}^{options} Impact_{xy}$$

$$\begin{array}{c} 0 \leq U_{safety} \leq 10 \\ 0 \leq U_{devTime} \leq 10 \\ 0 \leq U_{devCost} \leq 10 \\ 0 \leq U_{lifeCycleCost} \leq 10 \\ 0 \leq U_{capability} \leq 10 \end{array} \qquad (1)$$

| id | software process option | goals | | | | |
|---|---|---|---|---|---|---|
| | | safety | development | | life cycle operational cost | capability |
| | | | time | cost | | |
| 1 | target critical mission phases | + | + | + | - | - |
| 2 | target critical commands | + | + | + | - | - |
| 3 | target critical events | + | + | + | - | - |
| 4 | onboard checking | + | - | - | + | 0 |
| 5 | reduce flight complexity | + | + | + | ? | - |
| 6 | test fly prototypes | + | + | + | ? | ? |
| 7 | enhance safing | + | - | - | + | ? |
| 8 | certification | + | ? | ? | ? | ? |
| 9 | increase vv | + | - | - | + | ? |
| 10 | reduce onboard autonomy | ? | + | + | - | - |
| 11 | reuse across missions | ? | + | + | ? | ? |
| 12 | increase developer capabilities | + | + | + | ? | ? |
| 13 | increase developer tool use | + | + | + | ? | ? |
| 14 | implement optional functions after launch | ? | + | ? | ? | ? |
| 15 | reduce vv cost | 0 | 0 | + | + | 0 |
| 16 | increase vv speed | 0 | + | 0 | 0 | 0 |
| 17 | increase vv capabilities | + | + | + | 0 | + |

**Figure 2. Part of the NEAR model: "+" denotes "increases"; "-" denotes "decreases"; "?" denotes "an influence of unknown sign exists"; and "0" denotes "no influence".**

built that figure from a text document. Nevertheless, it is representative of the kind of information we have observed in the tiger team meetings. The model was quick to build (just a few hours for one of us to generate Figure 2 based on [11]).

The impact matrix containing $y = 17$ software process options is shown on the bottom of Figure 2. Note that each process option impacts multiple goals, sometimes in contradictory ways. For example, *targeting critical mission phases improves* mission safety but *decreases* mission capabilities (since less resources are devoted to peripheral functionality).

In general, a process option will combine a number of single process options[3] from Figure 2. We make the simplifying assumption here that the order in which process actions is applied does not matter.

Equation 1 of Figure 2 allows us to compute the utility of single process actions (when *options* is a singleton set), and of combinations of more complex process options (when *options* contains several process actions). The utility can then be used to judge between points in the trade space of process options. Different stakeholders assign different values to the $U_i$ weights to reflect the relative importance to them of the project subgoals of safety, development cost, capability, etc.. Given suitable definitions of summation for qualitative values (defined by the `sum(.,.,.)` and `value(.,.)` predicates below), Equation 1 allows us to incorporate these into an overall utility score.

The economy of the representation of Figure 2 is advantageous for early requirements analysis: it can be used when the structure of the domain is not well understood, and when stakeholders disagree about the relative importance of project subgoals. It is also very easily transformed into an executable representation. We represent Figure 2 as a set of Prolog tuples (one per row of the table), e.g.:

```
table1(target_critical_mission_phases,+,+,+,-,-).
table1(target_critical_commands      ,+,+,+,-,-).
```

In Figure 2, +,-,0 denote "positive", "negative", or "zero" numeric values (respectively). If "?"is the non-determinant case and "0" indicates "no influence", then the following fairly standard definition of a qualitative `sum(.,.,.)` predicate is used to define the inner summation of Equation 1.

```
sum(+,+,+). sum(-,-,-). % same + same is same
sum(+,-,?). sum(-,+,?). % this + that is unknown
sum(_,?,?). sum(?,_,?). % anything + unknown= unknown
sum(0,X,X). sum(X,0,X). % nothing + something= something
```

The results of qualitative summation are converted by `value(.,.)` below to integers which can be multiplied by the $U_i$ weights in the outer summation of Equation 1:

```
value(+,1).  value(-,-1). value(0,0).
value(?,-1). value(?,0). value(?,1).
```

Figure 2 *seems* quite small but its behavior is surprisingly complex. For example, Figure 3 shows 1000 outputs from Figure 2 using a small qualitative simulator described later in the paper. Due to qualitative nondeterminacy, the utilities generated in 1000 runs vary quite widely. The resulting pattern is very noisy and, hence, it is difficult (impossible?) to make a decision about the effects of those four process options on the overall utility of the project.

Another problem with qualitative modeling is that the simulators themselves can be quite complex. Formally, a qualitative model can reach a contradiction; i.e. a belief that the model can be in two or more mutually exclusive states. Most theorem provers are not contradiction tolerant since they are based in classical logic. In classical propositional logic, if a theory can reach any contradiction then everything else in the theory can becomes trivially true. Intuitively, the argument is as follows: if an argument permits conclusions on all sides of an issue then we may as well stop arguing now and agree that anything at all could be true.

[3]Single process options are henceforth called *process actions*. The term *process option* is used to denote a combination of one or more process actions.

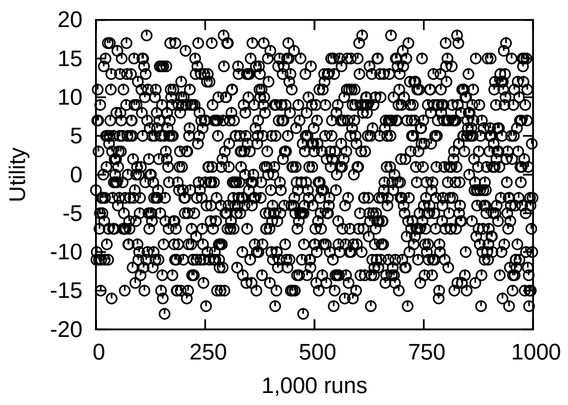


**Figure 3. Utilities seen after selecting four options, then 1000 times, computing Equation 1.**

Hence, qualitative simulators use complex non-classical semantics. Qualitative simulators build multiple *worlds of beliefs* (a.k.a. envisionments [3], extensions of a default theory [25], scenarios [23]). Each world holds a consistent set of beliefs which conflict with at least one other world. The proper implementation of such a multi-worlds reasoner is a non-trivial task and most researchers reuse a small number of well-studies tools such as the ATMS [4], THEORIST [23], or QSIM [10]. All these tools scale badly since larger theories with more contradictions lead to an exponential forking in the number of worlds.

Previously (with Cohen, Houle, Waugh, Goss, and Powell) we have tried using restrictive modeling languages to simplify qualitative simulation and restrain the exponential forking of behavior. A restrictive language was developed with the property that *any model* written in that language would generate tractable qualitative simulation [15, 19, 24]. Those restrictions are quite severe. In the following sections we show how stochastic simulation and treatment learning can make general qualitative modeling useful for requirements engineering.

## 3 Simpler Qualitative Simulation

Our tools use *treatment learning* from *stochastic sampling* to find and set the *master variables* that restrain qualitative simulations. These terms are explain below.

### 3.1 Master Variables

Nondeterminacy can prevent requirements engineers from making decisions from qualitative models. For example, 20 binary unknowns implies $2^{20} > 1,000,000$ different possibilities. We have never found a domain expert willing to resolve disputes by tediously exploring such a large range of options.

Fortunately, the space of possible behaviors in a qualitative model can be greatly reduced by setting a small number of key *master variables*. According to Menzies, Easterbrook, Nuseibeh, and Waugh [17], within most models there are a small number of master variables which set the remaining *slave variables*. For such *master-slave models*, the space of options is just the space of options within the master variables.

Crawford and Baker used the term *master-variables* in their study of scheduling problems. Similar concepts have been reported elsewhere:

- *Prime-implicants* in model-based diagnosis [27] or machine learning [26], or fault-tree analysis [12].
- *Backbones* in satisfiability [22, 29];
- *The dominance filtering* used in Pareto optimization of designs [8];
- *Minimal environments* in the ATMS [4];
- The *base controversial assumptions* of HT4 [16].

Whatever the name, the core intuition in all these terms is the same: what happens in the total space of a system is controlled by a small critical region.

An interesting property of master variable systems is that, by definition, inference pathways from inputs to outputs pass through the master variables. Hence, a stochastic sampling of those pathways will repeatedly stumble over the master variables. The tools described below use such a stochastic sampling policy, thereby avoiding the exponential runtime times of other qualitative reasoning tools.

### 3.2 The TAR3 Treatment Learner

The TAR3 *treatment learner* performs stochastic *forward select* search for a *treatment*; i.e. a small conjunction of master variables which, when compared to some baseline scenario, most improves the score of the system under study [18].

A *forward select* search adds candidate attribute values to a growing set of selected attribute values until the larger set does no better than a smaller set. For example, Kohavi and John's WRAPPER assesses attributes by their classification accuracy when they are used by a target learner [9]. Forward select search can be slow; e.g. WRAPPER does not scale to large data sets.

A faster method is to select new attributes for the growing set using some heuristic preference criteria. Hence, treatment learning starts by computing the *lift* of each individual attribute value. *Lift* is the increase in average oracle score within a defined sub population, compared to the full population:

- Let $D_0$ be the training examples with attributes $A$ with ranges $R$;
- Let $X$ be an attribute range within $A$; i.e. $X \in A \times R$.
- Let $D_1$ be the subset with attribute range $X$, $D_1 \subseteq D_0$;
- If $D_0$ and $D_1$ have mean oracle score $\mu_0$ and $\mu_1$, then $\mathit{lift}_X = log\left(\frac{\mu_1}{\mu_0}\right)$.
- By definition, setting $X$ to a master variable dramatically improves the score; i.e. $\mathit{lift}_X \gg 0$.
- By the same reasoning, to find bad settings to the master variables, look for the range $Y$ with $\mathit{lift}_Y \ll 0$.

Often there is added value in defining treatments as conjunctions of ranges. Treatment conjunctions $X_1 \wedge X_2 \ldots X_N$ are grown by selecting a treatment size $N$ at random from 1 to $maxSize$ (e.g. 10). $N$ attribute ranges are then combined into a conjunction by selecting attribute ranges $X_i$ at random, preferring those with higher lifts. This is repeated, say, 100 times and the best $B$ treatments are collected. This process is repeated until no new best treatments are found.

## 3.3 Stochastic Sampling

The complexities of multiple worlds reasoning were discussed in §2 above. Not only can they be difficult to implement, but they can scale poorly to larger problems. To reduce the implementation complexity, we use a result from Druzdel [5]. If software has $n$ variables, each with its own assignment probability distribution of $p_i$, then the probability that a system will fall into a particular state is

$$p = p_1 p_2 p_3 ... p_n = \prod_{i=1}^{n} p_i.$$

By taking logs of both sides, this equation becomes

$$\ln p = \ln \prod_{i=1}^{n} p_i = \sum_{i=1}^{n} \ln p_i \qquad (2)$$

The asymptotic behavior of such a sum of random variables is addressed by the central limit theorem. As long as $p_i$ is not uniform, then the expected case is that $p$ will exhibit a log-normal distribution; i.e. a small fraction of states can be expected to cover a large portion of the total probability space; and the remaining states have practically negligible probability. A stochastic sample of a small number of the reachable states will therefore sample a large percentage of the likely states.

This result can reduce the cost of searching through the worlds of belief generated by qualitative simulations. If we stochastically sample a subset of the reachable worlds, then Equation 2 promises that sample will include a large percentage of the states that are not negligibly unlikely.

Implementing such a stochastic sampling method is trivial. For example, the following `any` predicate converts standard SLD resolution used in logic programming into a random sampling method. Standard PROLOG processes clauses lists in a top-down manner. `Any` returns solutions in a random order:

```
any(X) :- setof(R/X,(X,R is random(2**30)),L),
```

The `any` predicate is the core of our qualitative simulator for the NEAR knowledge. It is used, for example, in the `row` predicate to select a random process option from Table2

```
row(table1,X,[A,B,C,D,E]) :- any(table1(X,A,B,C,D,E)).
```

Apart from `any`, there are two main predicates in our simulator: `across` and `down`.

```
sim(Used,Score) :-
    Table = table1,         % what table to look at
    Inits = [  0,0,0,0,0 ], % initial values for all colums
    Used  = [ _1,_2,_3,_4], % how many options to select?
    Utils = [ 10,3,3,2,2 ], % Equations 2 to 6
    down(Used,[],Table,Inits,Impacts),
    across(Utils,Impacts,Score).
```

`Across` sums over the goals to implement the *utility* calculation of Equation 1.

```
across(U,V,S) :- maplist(score,U,V,S0), add(S0,S).
score(Util,Val0,Util*Val) :- any(value(Val0,Val)).

add([],0).  %adds  a list of numbers.
add([H|T],X) :- add(T,X0), X is X0+H.
```

`Across` is called *after* `down` sums downs the columns of Figure 2 to find the net qualitative influences on each goal. `Down` tries to fill in a list of software process options (in its first argument) with rows from the impact table.

```
down([],_,_,Out,Out).
down([One|Rest],Used,Table, Old,Out) :-
  row(Table,One,Next),  % "One" is a row ..
  \+ member(One,Used),  % .. which is not used before
  sums(Old,Next,New),   % add this to "Old"
  down(Rest,            % and recurse
       [One|Used],Table,New,Out).

sums([],[],[]).
sums([H0|T0],[H1|T1],[H|T]):-
   any(sum(H0,H1,H)),sums(T0,T1,T2).
```

Note the nondeterministic choice within `sums` and `score` for the qualitative mathematics in `any(sum,H0,H1.H))`.

The data from Figure 3 can now be generated by *first* finding any solution, then *next* repeatedly scoring that solution:

```
generateFigureThree :-
   sim(Used,_),
   tell('figure2.data'),
   forall(between(1,1000,_),
         (sim(Used,Score)
         ,format(''~p\n'',Score)
         )),
   told.
```

The above code demonstrates the implementation simplicity of stochastic qualitative simulation. The above code is small (fits into half a page) and can be easily adapted to another model syntax or other user requests. For example, we can introduce a new qualitative value "++" which is to be stronger than "+". The change to above code was trivial: we just defined the scores for the qualitative symbols to be normal distributions with different means and changed `score` to sum the normals.

```
value(++,N) :- normal( 2, 0.5, N). % "++" has a mean of  2
value( +,N) :- normal( 1, 0.5, N). %  "+" has a mean of  1
value( -,N) :- normal(-1,   0, N). %  "-" has a mean of -1
value( ?,N) :- normal( 0,   1, N). %  "?" has a mean of  0
```

This scoring scheme produces very similar results to the scheme defined in §2, but is more flexible. We used this scheme for the experiments described in the following sections.

## 4 Results

A standard run with the above tools was to, one hundred times, generate 5000 process options, each combining from one to four process actions from Figure 2. $E_0$ in Figure 1 shows our *baseline* results. These were the distribution of utilities seen when the $U_x$ values of Equation 1 where allowed to vary randomly from 0 to 10. Subsequently, we explored the effects of restricting the range of possible $U_x$ values.

The outputs were based on *two* runs of TAR3. In run one, TAR3 was used to find the master variable settings that most *improved* the utility score. In run two, TAR3 was used to find the settings that most *decreased* the score. This process was not slow: in 20 minutes on a standard Linux box we could stochastically sample our qualitative model 500,000 and execute TAR3 200 times.

Figure 4 shows the start of one of the 200 runs of TAR3 on $E_0$. Note that 23 attributes were passed to TAR3:

- 17 "do"s and "dont"s that show which process actions were selected.
- 5 numeric $U_x$ weights from Equation 1.
- The final $Utility$ value from Equation 1. In our experiments, we wrote a small pre-processor to divide these into three classes of equal size.

Figure 4 also shows the `Options` used in this experiment. The following settings are the default settings for TAR3:

`granularity=4`: Discretize all numeric attributes (e.g. the 5 $U_x$ values) into four unique values.

`maxNumber=20`: Only report the top 20 treatments.

`randomTrials=100` Build 100 conjunctive treatments of size `minSize=1` to `maxSize=10`

```
Read 5000 cases (23 attributes) from m1.data

Options: granularity  4
         maxNumber    20
         minSize      1
         maxSize      10
         randomTrials 100
         futileTrials 5
         bestClass    10.00%

Baseline:

   CLASS                                 SUPPORT
   =====                                 ==============
  -20:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~     [  1666 - 33%]
    0:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    [  1667 - 33%]
   20:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    [  1667 - 33%]

Lift1s:
  -3:                                  [     1 -  1%]
  -1:                                  [     1 -  1%]
   0:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~    [    59 - 63%]
   1:~~~~~~~~~~                        [    20 - 22%]
   2:~~                                [     5 -  5%]
   3:~~~                               [     6 -  6%]
   4:                                  [     1 -  1%]
```

**Figure 4. TAR3 pre-processing $E_0$.**

`futileTrials=5`: If, after building 100 treatments, there are no new "top 20" treatments, then that is a "futile trial". In our experiments, TAR3 kept generating sets of 100 treatments until it found 5 consecutive futile trials.

`bestClass=10%`: This option stops TAR3 generating treatments with too little support. The `Baseline` in Figure 4 shows that our 5000 input examples contain 1,667 "best" examples. In our experiments, TAR3 rejected all treatments that selects for less than 10%*1667=167 of those best. examples.

The `Lift1s` distribution of Figure 4 shows the *lift* values for single ranges of the $U_x$ values and the software process options `dos` and `donts`. Most of the ranges have equal frequencies in all the classes (see the 59 ranges with *lift* $=$ 0). However, in a result consistent with the master-variable hypothesis, there exist a small number of ranges that have a large impact on the overall score (see the handful of ranges with *lift* scores approaching -3 and 4).

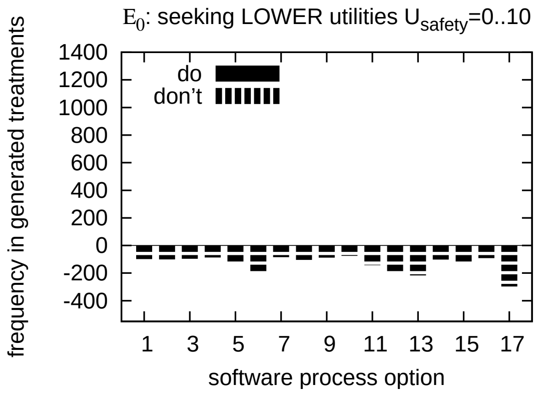
Figure 5 and Figure 6 show some TAR3 results. Note that the ordering of the classes in the two runs is reversed In Figure 5 TAR3 is seeking attribute ranges that select for higher utilities (the "20" class) but in Figure 6 TAR3 is seeking attribute ranges that select for lower utilities (the "-20" class). The `SUPPORT` column shows the effect of applying the treatment to the training data. These support results should be compared the *Baseline* of Figure 4. Note that these treatments make large changes to the utility distributions.

```
20 Treatments learned after 58 random trials

IF a16=dont AND a12=do THEN...

 CLASS                               SUPPORT
 =====                               ==============
 -20:~~~~~~                          [    46 - 12%]
   0:~~~~~~~~~~~~                    [    94 - 25%]
  20:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  [   230 - 62%]

IF a4=dont AND a17=do THEN

 CLASS                                SUPPORT
 =====                                ==============
 -20: ~~~~                            [    63 -  9%]
   0: ~~~~~~~~~~~~~~~                 [   205 - 31%]
  20: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  [   399 - 60%]

etc
```

**Figure 5.** $E_0$**: seeking HIGHER utilities.**

```
20 Treatments learned after 55 random trials

IF a13=dont AND a17=dont THEN

 CLASS                                SUPPORT
 =====                                ==============
  20: ~~                              [    39 -  4%]
   0: ~~~~~~~~~~~~~~~~                [   298 - 34%]
 -20: ~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  [   551 - 62%]

IF a17=dont THEN
 CLASS                               SUPPORT
 =====                               ==============
  20:~~~~                            [    33 -  9%]
   0:~~~~~~~~~~~~~                   [   103 - 28%]
 -20:~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~  [   228 - 63%]

etc
```

**Figure 6.** $E_0$ **seeking LOWER utilities.**



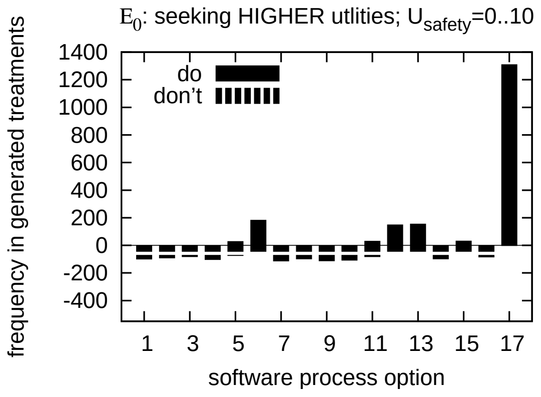


**Figure 7. Frequency of "dos" and "donts".**

## 4.1 Evaluating Process Recommendations

The treatments if Figure 5 and Figure 6 show "dos" and "donts"; i.e. they comment on *both* what process options to apply and what process options to avoid. Each run of the system (generating and scoring 5000 random process options, then applying TAR3) produces a ranking of 20 best and 20 worst process options. These top/bottom 20 treatments differ somewhat on successive runs of the system, i.e. the recommendations produced by a single run still have some randomness in them. This noise can be substantially reduced by collating the results from a large number of runs of the system.

Figure 7 shows the frequencies of what "dos" and "donts" were seen in one experiment of 100 repeats of 5000 stochastic simulations run twice through TAR3. In that figure:

- The x-axis numbers correspond to the software process actions listed in Figure 2.
- The y-axis show the frequency of ranges: positive numbers (solid bars) count the occurrences of "dos" and negative numbers (striped bars) count the occurrences of "donts".

In the two plots of Figure 7 TAR3 was looking for higher and lower utilities. When seeking lower utilities, TAR3's treatments were full of "donts"; i.e. actions which lead to a poor score if we do not do them. On the other hand, when seeking higher utilities, TAR3's treatments were full of "dos" and "donts". Experiment $E_0$ reveals that four process actions lead to positive treatments if chosen, and negative treatments if not chosen:

6. Test fly prototypes;
12. Increase developer capabilities;
13. Increase developer tool use;
17. Increase V&V capabilities.

These results assume that all the $U_x$ values in Figure 2 range from 0 to 10. The influence of changes to the $U_x$ distribution is examined in §4.3.

Based on Figure 7, we might limit our conclusions to the recommendation that process action 17 should be included in any process option. Is this the best we can do?

We can evaluate the utility of a (possibly singleton or empty) set of treatments by repeating our experiments, re-

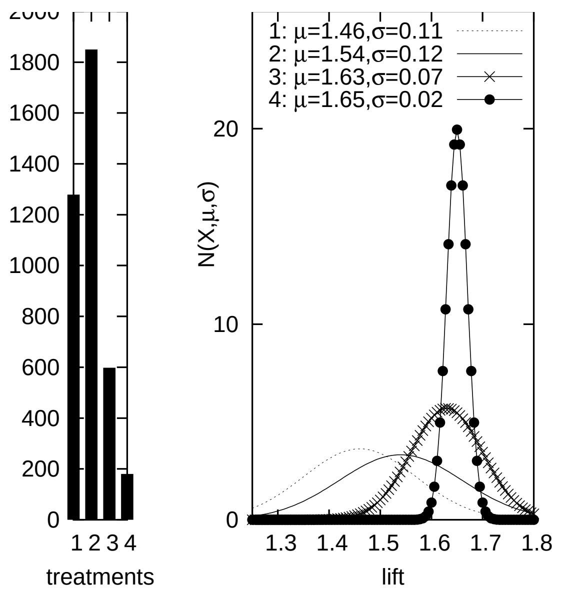


**Figure 8. LEFT: TAR3 generated thousands of treatments of size 1,2,3 or 4. RIGHT: larger treatments had higher mean lifts with less variance.**

stricting process option sets to those which are compatible with at least one of the treatments (i.e. which, for some treatment, perform all of the positive actions (plus maybe some others) and none of the negative actions).

The $E_1$ results shown in Figure 1 came from imposing software process 17 (increase V&V capabilities) onto the NEAR model and running another 5000 simulations. As shown in that figure, $E_1$ out-performed the baseline. However, as shown below, certain combinations of *multiple* software process options out-perform $E_1$.

## 4.2 Exploring Large Treatments

At the end of the previous section, we evaluated the effect of using the best single process option (17) as the basis for process options. Here, we study the effects of selecting up to ten of the process actions listed in Figure 2. The results indicate that simply collating treatments from many experiments and choosing the single best process action is simplistic: basing process options on more complex learned treatments can produce better results.

Recall from TAR3's options shown in Figure 4 that TAR3 hunted for treatments of size 1 to 10. The left-hand-side of Figure 8 shows the size of best treatments found by TAR3 in any of its 200 runs over $E_0$ data. In these results, TAR3 is advising us to do more than *just* apply singleton software process options selected from Figure 7. The right-hand-side of Figure 8 shows that as treatments grow in size, their mean lift increases somewhat and their variance decreases sharply. Hence, in the case of Figure 2, it is better to set *multiple process options* rather than just reading off one action from Figure 7.

The effects of the best treatment found by TAR3 were shown as $E_2$ in Figure 1. Those $E_2$ results were generated as follows: TAR3's `maxNumber` configuration parameter was set to one; i.e. TAR3 only reported the *best* treatment it ever found in any run. This process was repeated 100 times, producing 100 best treatments, for example, the four most frequently chosen were 1. do(17), 2. do(17)& dont(9), 3. do(17) & dont(10), 4. do(17) & dont(4). Intuitively, these treatments are combinations which advise doing 17 while avoiding 9, 10, 4 (which negate or introduce uncertainty into goals which 17 would otherwise influence positively).

In experiment $E_2$, 500000 random process options were generated which were compatible with the set of 100 selected treatments. The utility score nearly doubled the baseline distribution, and significantly increased over that of $E_1$ (extensions of 17).

## 4.3 The Effects of "Magic Weights"

Models of user preferences often contain "magic weights" representing the relative strengths of various factors on the final outcome. In the case of Figure 2, those "magic weights" are the $U_x$ weights which range from 0 to 10.

In the general case, changing these weights can change the conclusions of the model. However, in the specific case of Figure 2, we can use our rig to show that the implications of the NEAR model are quite insensitive to some of the possible changes in the magic weights.

Recall that Figure 7 assumed that all $U_x$ values varied at random from 0 to 10. Figure 9 shows the effect of changing the $U_{safety}$ distribution. The left-hand column of that figure shows results from a run where $U_{safety}$ was constrained to middle-range values; i.e. $3 \leq U_{safety} \leq 6$. The right-hard column shows results from a run when $U_{safety}$ was given a maximal weight; i.e. $U_{safety} = 10$. Figure 7 and Figure 9 show similar distributions; i.e. the above conclusions are relatively *insensitive* to different perceptions on the relative merits of safety versus other goals. This is a good thing to know: stakeholders with different opinions on the importance of safety can agree in this case on the conclusions of our analysis.

## 5 Conclusion

Early life cycle models *should* be less precise than models generated later on. Such imprecise models are faster to generate and let users explore more options, faster.

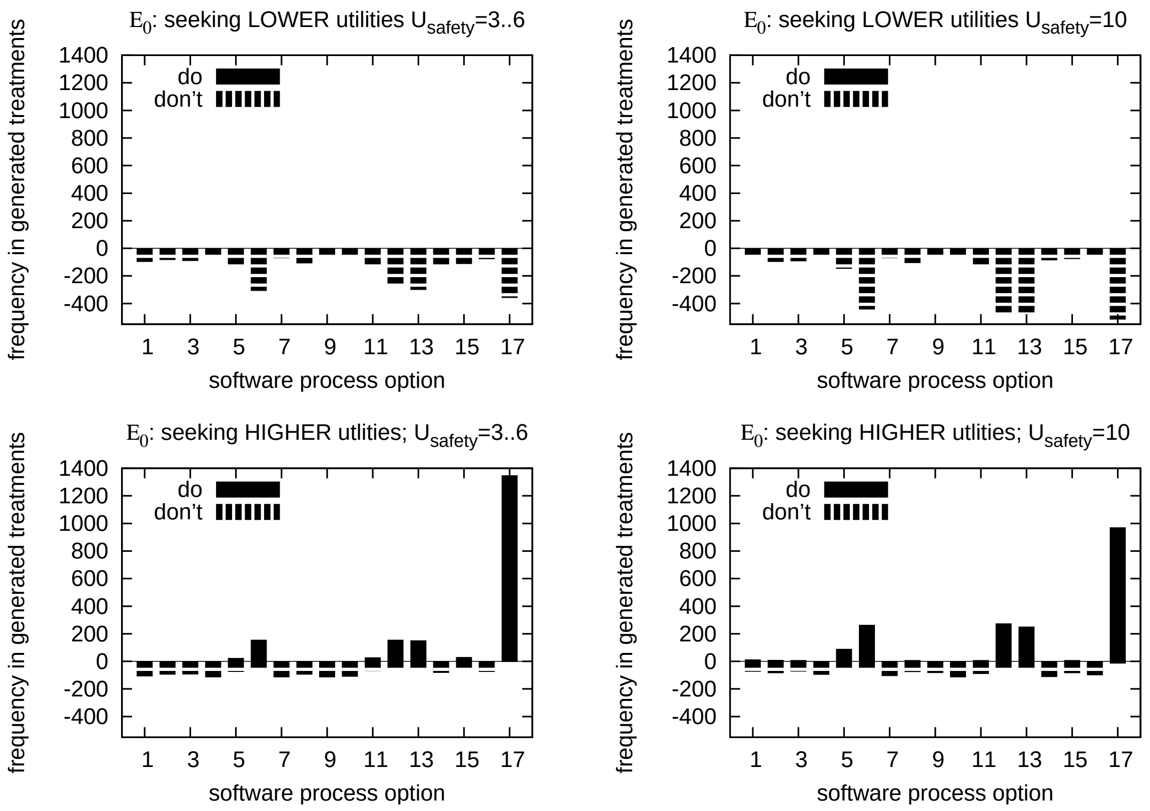


**Figure 9.** $E_0$**: effects of changing utilities**

Until now, it seemed that speed meant recklessness, that models written quickly were not suitable for model-based RE. Such quickly written models are often so under-constrained that they produce a wide range of outputs. Such wide-ranging output may neither restrain or nor refine an argument. We saw such a large range of options here: Figure 2 generated the noise of Figure 3.

However, we show here that master variables can be exploited to explore the space of options as follows:

- Quickly sketch a qualitative model.
- *Learn treatments* from *stochastic samples* of those qualitative models
- Use the treatments as *settings to the master variables* that *improve the performance* of the qualitative models.

Our choice of a simple representation had a number of benefits: models can be quickly formulated, are immediately executable, can be executed very fast, and yield data which is very amenable to treatment learning. This is very suitable for quick early RE.

Our goal is not of course the construction of Platonic truths, but explicating tacit influences within hastily written early life cycle models. After our treatment learners find the master variable settings that most improve or degrade a qualitative model, our intent is that users will focus their dialogue on those settings. Our expectation is that, as a result of that extra attention, users will often elaborate parts of the qualitative models (i.e. those including the master variables). That is, our tools let users find and focus on those parts of the model which most influence the decisions. An alternative approach, which would be much more time-consuming is that users elaborate *all* parts of a model. For our target domain (real-time support of NASA's tiger teams), this is *not* the preferred options since it would be too slow.

## Acknowledgements

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. This research was conducted at the NASA Ames Research Center, funded by NASA, and at Portland State University, partially sponsored by the NASA Office of Safety and Mission Assurance under the Software Assurance Research Program led by the

NASA IV&V Facility.

## References

[1] L. Chung, B. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering*. Kluwer Academic Publishers, 2000.

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

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

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

[5] M. Druzdzel. Some properties of joint probability distributions. In *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 187–194, 1994.

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

[7] V. Goel. "ill-structured diagrams" for ill-structured problems. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 66–71, 1992.

[8] 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`.

[9] R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

[10] B. Kuipers. Qualitative simulation. *Artificial Intelligence*, 29:229–338, 1986.

[11] S. Lee and A. G. Santo. Tradeoffs in functional allocation between spacecraft autonomy and ground operations: the NEAR (Near Earth Asteroid Rendezvous) Experience, John Hopkins APL, August 9-12, Utah State University , Eccles Conference Center, Logan, Utah, 2004.

[12] 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`.

[13] A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options and criteria: Elements of design space analysis. In T. Moran and J. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 53–106. Lawerence Erlbaum Associates, 1996.

[14] T. Menzies. Model-based requirements engineering. *Requirements Engineering*, 2003. Available from `http://menzies.us/pdf/03mbre.pdf`.

[15] T. Menzies, R. Cohen, S. Waugh, and S. Goss. Applications of abduction: Testing very long qualitative simulations. *IEEE Transactions of Data and Knowledge Engineering*, pages 1362–1375, November/December 2003. Available from `http://menzies.us/pdf/97iedge.pdf`.

[16] 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://menzies.us/pdf/96aim.pdf`.

[17] 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://menzies.us/pdf/99re.pdf`.

[18] T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.

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

[20] T. Menzies, D. Raffo, S. on Setamanit, Y. Hu, and S. Tootoonian. Model-based tests of truisms. In *Proceedings of IEEE ASE 2002*, 2002. Available from `http://menzies.us/pdf/02truisms.pdf`.

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

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

[23] D. Poole. Normality and Faults in Logic-Based Diagnosis. In *IJCAI '89*, pages 1304–1310, 1989.

[24] J. Powell. A graph theoretic approach to assessing tradeoffs on memory usage for model checking,, 1999. Masters thesis, Computer Science and Electrical Engineering, West Virginia University.

[25] R. Reiter. A Logic for Default Reasoning. *Artificial Intelligence*, 13:81–132, 1980.

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

[27] 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`.

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

[29] 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.

[30] A. Sutcliff and A. Gregoriades. Validating functional system requirements with scenarios. In *IEEE Joint Conference On Requirements Engineering ICRE'02 and RE'02, 9-13th September, University of Essen, Germany*, 2002.

[31] E. S. K. Yu. Towards modeling and reasoning support for early-phase requirements engineering. In *RE '97: Proceedings of the 3rd IEEE International Symposium on Requirements Engineering (RE'97)*, page 226. IEEE Computer Society, 1997.

[32] E. S. K. Yu and J. Mylopoulos. Understanding "why" in software process modelling, analysis, and design. In *ICSE '94: Proceedings of the 16th international conference on Software engineering*, pages 159–168. IEEE Computer Society Press, 1994.