Constraining Discussions in Requirements Engineering via Models[1]

Tim Menzies[2], Ying Hu[3]
University of British Columbia[4]
tim@menzies.com, yingh@ece.ubc.ca

Abstract

Models are used in requirement engineering to inform the process of goal elaboration and refinement. Models written in early life cycle are often based on incomplete information. Hence, such models easily generate an unmanageably large space of possibilities. We show here that this very large space can be reduced by identifying the *most informative issues* in the search space. By iteratively resolving the next most informative issue, we are able to quickly constrain the space of possibilities to just the issues most important for the requirements. In this paper, we build a qualitative model to sample requirement options and use the TAR2 *treatment learner* to find out the most informative question as further constrains to the model. By iteratively exploring the most informative issues, new constraints can be discovered which, when applied to the model, dramatically reduce the space to a degree where the number of options becomes manageably small.

1 Introduction

Requirements engineering (RE) is (1) the elicitation of high-level goals of some envisioned system followed by (2) the refinement of these goals into services and constraints, and (3) the assignment of responsibilities for the resulting requirements to agents such as humans, devices, and software [12]. RE is typically performed within a community of stakeholders, who may have different goals and priorities.

Model-based requirements engineering (MBRE) uses some formal, possibly executable, model to inform the requirements process. Models offer additional constraints to a discussion. Options can be quickly rejected if the model informs the analyst that those options are impossible. Various what-if queries can be answered by exploring the space of options offered by the model.

The ability of models to inform and constrain a requirements discussion is quite important. After Parnas [10] we say that requirements elaboration is essentially the exploration of paths down a decision tree, with each node corresponding to a design decision. Each decision injects new constraints into the tree of options. Decisions towards the top of the tree are the hardest to change (because they require more back-tracking), whereas decisions near the leaves of the tree are much easier to change.

When exploring a tree of requirements options, some issues should be explored first. We call the issues that most divide the tree of options the *most informative issues*. These issues should be explored first because:

- Answers to the big issues can rule out much of the space of options. That is, the answers to some questions will render irrelevant many of more trivial issues.
- Resolving some issues may require gathering domain details. However, if details are requested in same order as the most informative issues, then a *minimum* of domain questions will be asked.

This paper proposes an iterative cycle for exploring options in model-based requirements engineering:

[1]Submitted to the first International Workshop on Model-based Requirements Engineering, November 30, 2001, San Diego, http://www.bfsng.com/mbre01/.

[2]http://tim.menzies.com, phone (604) 822-3381

[3]http://www.ece.ubc.ca/twiki/bin/view/Softeng/YingHu.

[4]Department of Electrical & Computer Engineering; 2356 Main Mall; Vancouver, B.C. Canada V6T1Z4.

- Given a model and some known constraints...
- Execute the model within those constraints to generate an experience base.
- Query the experience base to learn the most informative issues. This step is crucial. If this query mechanism is effective, then this cycle will yield the *least* number of questions that constrains the options space the *most*.
- Explore the most informative issues to find an extension to the constraints.
- Repeat until the number of options becomes manageably small.

A similar iterative cycle has been proposed for model-based diagnosis using entropy measures [3]. Given that each *probe* into a device is expensive, the best probe can be selected via an heuristic entropy measure (entropy is the technical name for information content). Here, we explore a similar cycle but use a *treatment learner* (defined below) to identify the most informative issues.

The rest of the paper is structured as follows. The next section is a description of our sample model which can generate many alternatives (in fact, 35,228 alternative behaviors). This is followed by an introduction to the TAR2 treatment learner. Finally, we will show that treatment learners can dramatically reduce options space. Specifically, in our case study, three questions found by a treatment learner were enough to reduce the space of options to one fifth of one percent of its former size.

2 Representing an Options Generator

The technology discussed in this paper is being developed for a large financial institution that wants to develop requirements for a particular software process (handling customer sales and queries over the phone). Their model of software process is corporate confidential.

Without a real model to show, we will demonstrate the technique using a qualitative model of an electrical circuit. The qualitative model has many features in common with the real business model:

- The general topology of the model is known but precise numeric values for any of the inter-relationships are unknown.

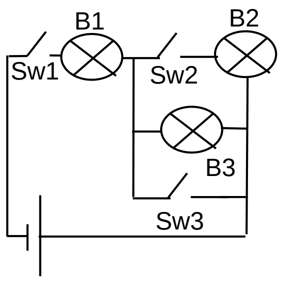

Figure 1: A qualitative circuit.

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

Figure 2: Qualitative mathematics using a Prolog syntax [1].

- Certain business sub-processes are reused in multiple places across the model. Various qualitative constraints are known for these "business sub-routines".
- The model is not determinate. Due to the various unknowns within the model, a set of inputs will generate a range of outputs, not a unique value.
- Model output can be assessed using a coarse-grained evaluation function that declares some outputs better than others.

Our model will be on the qualitative electrical circuit of Figure 1. 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 [1]; i.e.

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

The `sum` relation of Figure 2 describes our qualitative knowledge of addition using a Prolog notation. In Prolog, variables start with upper case letters and constants start with lower-case letters or symbols. For example,

```
%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).
```

Figure 3: Definitions of qualitative bulbs and switches. Adapted from [1].

`sum(+,+,+)` says that the addition of two positive values is a positive value. Some qualitative additions are undefined. For example `sum(+,-,Any)` says that we cannot be sure what happens when we add a positive and a negative number.

The `bulb` relation of Figure 3 describes our qualitative knowledge of bulb behavior. For example, `bulb(blown,dark,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 3 is 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 `switch(on,0,Any)` says that if a switch is on, there is zero voltage drop across it while any current can flow throw it.

The `circuit` relation of Figure 4 describes our qualitative knowledge of the 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:

- `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)

```
 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.
```

Figure 4: Figure 1, modelled in Prolog. Adapted from [1].

In order to stress test our method, our case study will wire up three copies of Figure 4 in such a way that solutions to one copy won't necessarily work in the other copies. Figure 5 shows our circuit connected by a set of *openners* 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 openners and closers are defined in Figure 6. The full model is shown in Figure 7.

3 The TAR2 Treatment Learner

The model described in the previous section is capable of generating tens of thousands of examples. Some summarization technology is required to understand all these examples. Treatment learners are one such summarization method since they report a small number of actions that can most change the behavior of a system.

Treatment learners are different to standard machine learners; e.g. the C4.5 decision tree learners [11]. Standard machine learners output classifiers that mapping at-

```
circuits :-
   % some initial conditions
   value(light,bulb,B1a),
   % Uncomment to constrain Sw2c
   % value(off,switch,Sw2c),
   % Uncomment to constrain Sw1c
   % value(on,switch,Sw1c),
   % Uncomment to constrain Sw3c
   % value(on,switch,Sw3c),
   % explore circuit A
   circuit(Sw1a,B1a,Sw2a,B2a,Sw3a,B3a,GlowA),
   % let circuit A influence circuit B
   inf(+,B1a,Sw1b),
   inf(-,B2a,Sw3b),
   % let circuit B influence circuit C
   circuit(Sw1b,B1b,Sw2b,B2b,Sw3b,B3b,GlowB),
   % propagate circuit B to circuit C
   inf(-,B3b,Sw2c),
   inf(+,B2b,Sw3c),
   % explore circuit C
   circuit(Sw1c,B1c,Sw2c,B2c,Sw3c,B3c,GlowC),
   % compute total shine
   Shine is GlowA+GlowB+GlowC,
   % make one line of the examples
   format(’~p,~p,~p,~p,~p,~p,~p,~p,~p’,
   [Sw1a,Sw2a,Sw3a,Sw1b,Sw2b
   ,Sw3b,Sw1c,Sw2c,Sw3c]),
   format(’~%,~%,~%,~%,~%,~%,~%,~%,~%,~p’,
   [B1a,B2a,B3a,B1b,B2b,B3b
   ,B1c,B2c,B3c,Shine]),nl.

data :- tell(’circ.data’),
   forall(circuits,true), told.

% some support code
value(Sw,    switch, switch(Sw,_,_)).
value(Light, bulb,bulb(_,Light,_,_)).

:- format_predicate(’%’,bulbIs(_,_)).

bulbIs(_,bulb(X,_,_,_)) :-
     var(X) -> write(’?’) |write(X).

portray(X) :- value(Y,_,X), write(Y).
```

Figure 7: Figure 5 expressed in Prolog.

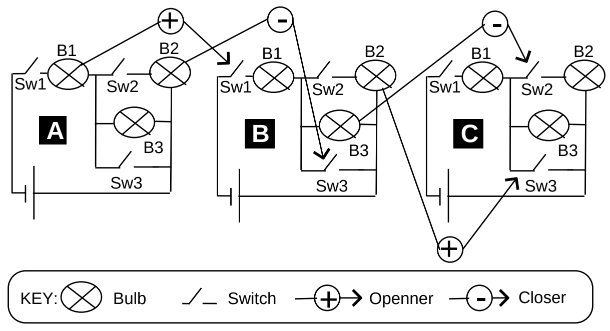

Figure 5: A device modelled using the Prolog of Figure 5.

```
%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).
```

Figure 6: The `inf1/3` predicate used to connect bulb brightness to switches.

tributes to classes. These standard learners treated their classes the same way. The TAR2 *treatment learner* assumes that some partial ordering has been defined for the classes; i.e. some classes are less desirable than others and one class is *best*.

TAR2 outputs a *treatment*, which is a constraint on future controllable inputs of a system. The intent of a constraint is to increase the ratio of preferred classes. Unlike normal machine learners, TAR2 does not output a classifier. Rather, it outputs a strategy that “nudge” the system towards the better classes.

TAR2 can best be introduced via example. Consider the log golf playing behavior shown in Figure 8. This log contains four attributes and 3 classes. Recall that TAR2 accesses a *score* for each class. For a golfer, the classes in Figure 8 could be scored as *none=2* (i.e. worst), *some=4*, *lots=8* (i.e. best). Note that the preferred classes score exponentially higher than the non-preferred classes. As we shall see, this disproportionate weighting scheme strongly encourages TAR2 to chase the better classes.

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

outlook	*temp(^{o}F)*	*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*

Figure 8: A log of some golf-playing behavior.

of $a.r$ to improve the frequency of the $best$ class. $\Delta_{a.r}$ uses the following definitions:

$X(a.r)$: is the number of occurrences of that attribute range in class X; e.g. *lots(outlook.overcast)=4.*

$all(a.r)$: is the 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. $rest = \{none, some\}$;

$score$: The score of a class X is $\$X$;.

$\Delta_{a.r}$ 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:

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

The attribute ranges in our golf example generate the Δ histogram shown in Figure 9. Note that *outlook=overcast*'s Δ is the highest, potentially most effective, attribute range.

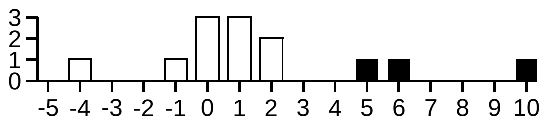

Figure 9: Δ distribution seen in golf data sets. Outstandingly high Δ values shown in black. Y-axis is the number of attribute ranges that have a particular Δ.

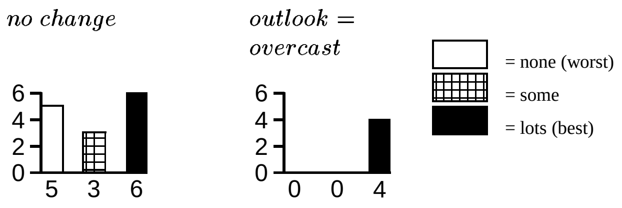

Figure 10: Finding treatments that can improve golf playing behavior. With no treatments, we only play golf lots of times in $\frac{6}{5+3+6} = 57\%$ of cases. With the restriction that *outlook=overcast*, then we play golf lots of times in 100% of cases.

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 9: 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 the attribute rages 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 our golf example, the best treatment is *outlook=overcast*; Figure 10 shows the class distribution before and after that treatment. i.e. if we bribe disc jockeys to always forecast overcast weather, then in 100% of cases, we should be playing lots of golf, all the time.

4 Experiments

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 4 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

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

Figure 11: Some output seen in `circ.data` generated using `data` (line 32 of Figure 7). Columns denote values from Figure 5. For example, Sw1a and Sw1b denotes switch 1 in ciruit A and ciruit A respectively.

times ,these implies a space of up to $3,359,232^3 = 10^{19}$ options. Even many of these possibilities are ruled out by inter-component constraints, the `circuits` relation of Figure 7 can still succeed 35,228 times (some sample output is shown in Figure 11).

Given the goal that the *more* lights that shine, the better the circuit, we assess 10 classes: $0, 1, 2, 3, ..9$, one for every possible number of glowing bulbs. As shown in Figure 12, 35,228 runs there are very few lights shining. TAR2's mission is to explores the space, trying to find the key constrain which, when apply to the circuit, can most improve this low level of lighting.

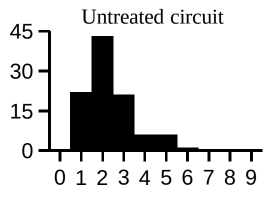

Figure 12: Frequency count of number of bulbs glowing in the 35,228 solutions of `circuits` of Figure 7.

4.1 Exploration 1

After learning treatments, and applying some of them to the data, TAR2 generated Figure 13.

In summary, Figure 13 is saying that answering a single question will change the average illumination of the circuit from 2 to 5 (if Sw2C=off) or 6 (if Sw3C=on). it

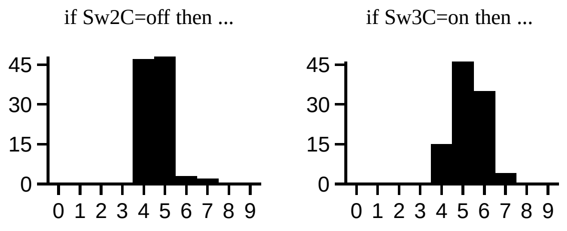

Figure 13: Run#1 of TAR2 over the data seen in Figure 12.

is preferrable if switch 3 in circuit C is not be closed- since that would violate (say) the warranty on circuit C. Our analysts therefore agree to the next best treatment, i.e. Sw2C=off; shown in Figure 13, left hand side (LHS).

4.2 Exploration 2

After constraining the model to Sw2C=off (i.e. by uncommenting line 5 in Figure 7), 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 14.

Happily, Figure 14 has the same distribution as Figure 13.LHS; i.e. in this case, TAR2's predictions proved accurate.

Executing TAR2 again finds the next most informative question, as shown in Figure 15. Here, TAR2 is saying that our best treatment would be to guarantee that bulb 3 in circuit C is never blown. Perhaps this is possible if

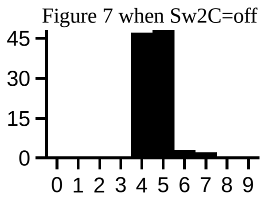

Figure 14: Frequency count of number of bulbs glowing in the 3,264 solutions of `circuits` of Figure 7 when Sw2C=off.

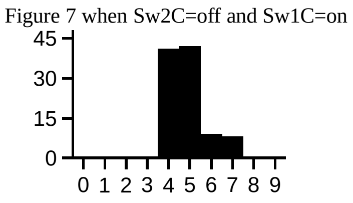

Figure 16: Frequency count of number of bulbs glowing in the 648 solutions of `circuits` of Figure 7 when Sw2C=off and Sw1C=on.

we were to use better light bulbs with extra long life filaments. However, for the sake of argument, we will assume that there is no budget for such expensive hardware. Hence, to avoid this expense, our analysts agree that always closing switch 1 in circuit C (as proposed by Figure 15.LHS) is an acceptable action.

4.3 Exploration 3

After further constraining the model to Sw1C=on (i.e. by uncommenting line 7 in Figure 7), 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 16.

Figure 16 has the same distribution as Figure 15.LHS. That is, once again, TAR2's predictions proved accurate. Executing TAR2 again generated Figure 17 and finds the next most informative question.

The cycle could stop here since the next best treatments are not acceptable. Figure 17.LHS wants to use overly-expensive hardware to ensure that bulb 3 in circuit C is always not blown. Figure 17.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 17.LHS. Alternatively, if there was some way to renegotiate the warranty, then the improvements shown in Figure 17.RHS could be achieved.

To verify this, our engineers continue constraining Figure 7 to the case of Sw3c=on by uncommenting line 9 in Figure 7. The resulting distributions looked exactly like Figure 17.RHS. Further, only 64 solutions were found; i.e. resolving three of the top treatments proposed by TAR2 constrained our system to one fifth of one percent of its original 35,228 behaviors.

5 Discussion

The above example assumed an exhaustive enumeration of the behaviors of our model. As models grow, the number of such exhaustive behaviors can grow exponentially-especially for qualitative models. Hence, some sub-sampling of the total space will be required. Methods for

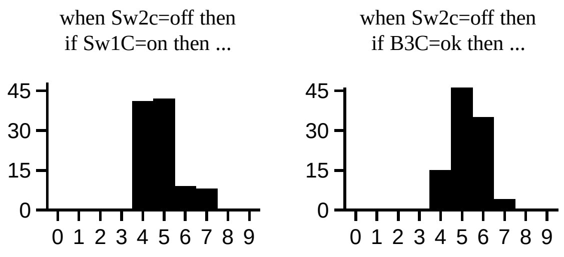

Figure 15: Run #2 of TAR2 over the data seen in Figure 14.

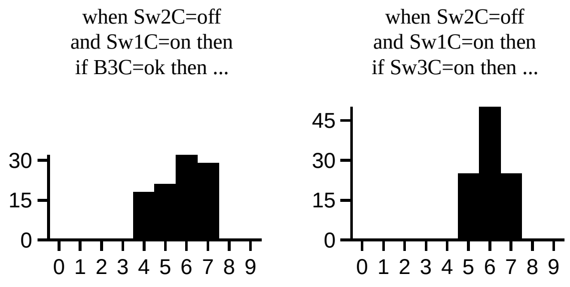

Figure 17: Run#3 of TAR2 over the data seen in Figure 16.

sub-sampling for treatment learners are discussed in [8,9]. In summary, sub-sampling can still yield adequate treatments.

In basing our requirements engineering around an nondeterministic model, we are somewhat at odds with the conventional wisdom in the requirements engineering field. For example, the software safety guru Nancy Leveson argues that “nondeterminism is the enemy of reliability” and proposes that a valid requirements model is deterministic [6]. In the case of safety critical systems where sufficient resources are available for data collection, Leveson’s view is clearly correct. Incremental treatment learning is more suitable to resource bounded exercises where data collection can be prohibitively expensive.

6 Conclusion

We have presented one example where answering a very small number of key questions greatly constrained and improved the behavior of a qualitative theory. The key questions were identified by the TAR2 treatment learner.

Our claim, as yet untested,is that this technology will apply to general business models, provided that they are executable and some oracle can assess their behavior. We have several reasons for this optimistism:

- Treatment learning is simple and fast. Further, the algorithm scales well. Given 200MB of RAM,TAR2 has processed 500,000 examples (with 12 attributes) in 80 seconds.
- While precise deterministic models are slow and expensive to build, nondeterministic models can be easily generated by sketching out local intuitions as qualitative models. Hence, approximate business models can be built very quickly. Examples of such approximate business models can be found in [2, 8]. Related work by Feather et.al. [4] and Kaplan & Norton [5] are also very interesting to us.
- The core assumption of treatment learning is that there exists a small number of attribute ranges that can have a major impact on the overall behavior of the system. Elsewhere, we have explored this assumption and have found theoretical and empirical grounds for believing that the assumptions is widely applicable [7].

References

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

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

[3] J. DeKleer and B. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32:97–130, 1 1987.

[4] M. Feather, S. Cornford, and T. Larson. Combining the best attributes of qualitative and quantitative risk management tool support. In *15th IEEE International Conference on Automated Software Engineering, Grenoble, France*, pages 309–312, September 2000.

[5] R. Kaplan and D. Norton. *The Balanced Scorecard: Translating Strategy into Action*. Harvard Business School Press. Boston, 1996.

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

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

[8] T. Menzies and J. Kiper. Better reasoning about software engineering activities. In *ASE-2001*, 2001. Available from `http://tim.menzies.com/pdf/01ml4re.pdf`.

[9] 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`.

[10] D. Parnas. On the design and development of program families. *IEEE Transactions on Software Engineering*, March 1976.

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

[12] A. van Lamsweerde. Requirements engineering in the year 00: A research perspective. In *Proceedings ICSE2000, Limmerick, Ireland*, pages 5–19, 2000.