July 29, 2006
WP ref: our/src/sh/xomoglue/doc/out1.tex
Download this paper at http://timmenzies.net/pdf/05xomo101.pdf.

XOMO: Understanding Development Options for Autonomy

Tim Menzies

Computer Science, Portland State University, USA
tim@timmenzies.net
URL: http://timmenzies.net

Abstract We seek an AI agent that is fast enough to keep up with debates between humans and and can offer suggestions regarding what is the next most important issue to explore.

To assess the merits of *treatment learning* for such an agent, this kind of learning was applied to three models of a mythical (but plausible) development project building software for autonomous systems. The models were the COCOMO effort estimation model; the COQUALMO defect introduction and removal model; and Madachy's Heuristic Schedule Risk model.

The experiment was successful; i.e. the learner found ways to improve reduce the residual defects per thousand lines of code by 85% while halving the risk that the schedule over run. Also, the development effort was nearly halved. Hence, we conclude that treatment learners can understand how to improve the process of building autonomous software.

Contents

List of Figures

This research was conducted at Portland State University under the Universities Space Research Association (USRA) sub-contract 8023-004. All software discussed here is available from the author under the GNU Public License (version 2: see www.gnu.org/copyleft/gpl.html). The USRA sub-contract was funded, in turn, from a NASA Human and Robotic Technologies grant. 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.

1 Introduction

Suppose a group of talented Ph.D.-level developers were bold enough to try something new- develop an autonomous software system for part of the new NASA Crew Exploration Vehicle (CEV). What software process decisions would reduce the risks associated with such a brave undertaking?

To make the problem interesting, we'll assume that this team has previously built many successful products, none of which used autonomy. Also, we'll assume that the project is being analyzed very early in its life cycle when many issues are still open and many decisions have yet to be made.

To solve this problem, we'll model the project including all its "maybes" and "what-ifs" A Monte Carlo simulator called XOMO (pronounced "x-o-mow") will then sample that space of possibilities. XOMO combines three models:

- The COCOMO effort estimation model [2, p29-57];
- The COQUALMO defect model [2, p254-268];
- The Madachy's schedule risk model [2, 284-291].

XOMO's output will then be based to a data miner that seeks the decisions that most:

1. Decrease the mean development times, while...
2. Decreasing the mean chance of schedule over-run, while...
3. Leaving the fewest mean number of defects.
4. Also, the learner tries to reduce the variance in the model behavior so predictions can be made with more certainty.

The particular data miner used here is the TAR3 *treatment learner* [13]. The premise of treatment learning is that we are all busy people and busy people don't need (or can't use) complex models. Rather, busy people need to know the *least* they need to do to achieve the *most* benefits. For example, when dealing with complex situations with many unknowns (e.g. developing autonomous system), it can be a wise tactic to focus your efforts on a small number of key factors rather than expending great effort trying to control all possibilities.

It is shown below that, at least for this case study, XOMO and treatment learning can reach all the above four goals:

1. The mean development effort will be nearly halved;
2. The mean risk of schedule over run will be halved;
3. The mean defects densities will be reduced by 85%.
4. The variance on the above measures will also be significantly reduced.

The case study was fast to run: all the results shown below took ten minutes to run on a standard computer (a Mac OS X box running at 1.5GHz). Those ten minutes included 5000 runs of the model and five runs of the data miners. Hence, this study gives us confidence that AI-based decision support agents can run fast enough to keep up with humans debating software process options for autonomous systems.

The rest of this paper describes how the XOMO models and treatment learning were applied to the case study. Before that, we first pause for a quick digression.

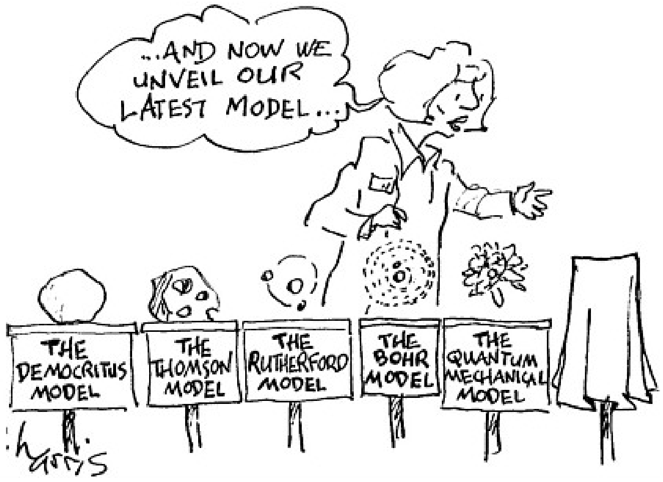
2 Digressions: Are our Models "Correct"?

One drawback with our results is that they come from simulation and not from empirical observations of real world development teams applying the policy decisions made by the learners. If our models are wrong (e.g. poorly calibrated) then our results are suspect.

Our methodology partially addresses this concern. For example, the COCOMO effort model contains two calibration parameters and the above results hold for simulations across the space of possible calibrations. That is, Monte Carlo plus data mining can find stable conclusions within the space of possibilities (this is a conclusion we have made elsewhere, many times [5, 11–13, 15–17]).

Nevertheless, simulations across the space of options will never give the right answers if that model is fundamentally flawed; e.g. important domain factors are missing from the model. This is a problem with all model-based reasoning: if the model is wrong then the reasoning is wrong as well.

However, as George Box says, "all models are wrong but some are useful". Certainly this has been the recent experience in physics. Over the last 100 models, numerous revisions to the atomic theory of matter have been proposed:



Each new model was wrong since it was superseded by a newer model. But each new model was useful in the sense that it explained more effects than the previous model.

The lesson here is that committing to a model of the current best understanding of a phenomenon is good practice, even if that model is not "correct" in some absolute sense. And once that new model is generated, it is right and proper that it be exercised, criticized, and improved. In the next few months, the XOMO models will be used in panel sessions were experts will convene to debate cost and risk models for autonomous NASA software. At those sessions, it is expected that the XOMO models will be critiqued and extensively revised.

During those panels, it is important that the experts' time is put to best use. Autonomy experts are scarce and it will take a significant administrative effort to collect them all together at the same place and at the same time. It is therefore vital that

```
# thousands of lines of codes
_ANY(ksloc, 2,   10000)

# scale factors: exponential effect on effort
ANYi(prec,  1,   6)
ANYi(flex,  1,   6)
ANYi(resl,  1,   6)
ANYi(team,  1,   6)
ANYi(pmat,  1,   6)

# effort multipliers: linear effect on effort
ANYi(rely,  1,   5)
ANYi(data,  2,   5)
ANYi(cplx,  1,   6)
ANYi(ruse,  2,   6)
ANYi(docu,  1,   5)
ANYi(time,  3,   6)
ANYi(stor,  3,   6)
ANYi(pvol,  2,   5)
ANYi(acap,  1,   5)
ANYi(pcap,  1,   5)
ANYi(pcon,  1,   5)
ANYi(aexp,  1,   5)
ANYi(plex,  1,   5)
ANYi(ltex,  1,   5)
ANYi(tool,  1,   5)
ANYi(site,  1,   6)
ANYi(sced,  1,   5)

# defect removal methods
_ANYi(automated_analysis,           1,  6)
_ANYi(peer_reviews,                 1,  6)
_ANYi(execution_testing_and_tools, 1,  6)

# calibration parameters
_ANY(a,      2.25,3.25)
_ANY(b,      0.9, 1.1)
```

Fig. 1 XOMO: specifying legal ranges.

```
function ksloc0() {
  # Low-level primitive that returns a ksloc
  return from(2,10000,ratio( min("ksloc"),max("ksloc")))}

function ksloc()  {
  # On the first call, there is nothing in the cache.
  # So call the primitive function and cache the result.
  # On subsequent calls, return the value in the cache
  if   ("ksloc" in Cache) { return Cache["ksloc"] }
  else { return Cache["ksloc"] = ksloc0()} }

function Ksloc()  {
  # Return an unfiltered ksloc
  return ksloc()}
```

Fig. 2 XOMO: expanding `_ANY(ksloc,2,10000)`

no time be wasted in discussing irrelevancies. The XOMO toolkit can be used to quickly prune debates about relatively unimportant issues. The panel moderator could (gently) guide the discussion onto other matters if the matters under debate have little effect on the model behaviors. Similar, the moderator could ask XOMO for the next most important issue to discuss (that issue would be the one that most changes the current model's behavior).

3 XOMO

XOMO is a general framework for Monte Carlo simulations that has been customized for processing COCOMO-like models. An XOMO user begins by defining a set of *ranges* for model variables. For example, in Figure 1, `_ANY(x,n1,n2)` defines some variable x that can take any value fro $n1$ to $n2$.

```
function prec0() {
  return from(1,6,ratioInt(min("prec"),max("prec")))}

function prec()  {
  if   ("prec" in Cache) { return Cache["prec"] }
  else { return Cache["prec"] = prec0()} }

function Prec() {
  return scaleFactor("prec",
                        pred())}
```

Fig. 3 XOMO: expanding `ANYi(prec,1,6)`. Very similar to Figure 2, but `Prec()` takes the value returned by `prec()` and maps it into the COCOMO regression parameters.

Many of the COCOMO parameters map some integer index into a table of regression which can be defined in XOMO using `ANYi(y,n1,n1)`. So the first two entries of Figure 1 define `ksloc` using `_ANY` and `prec` (which is a COCOMO parameter) using `ANYi`.

Internally, XOMO represents its variables as memoed, possible filtered, functions. A variable `foobar` gets three functions: `foobar0`, `foobar()` and `Foobar()`:

- `foobar0()` computes a new value for "foobar"; e.g. see `ksloc0` in Figure 2.
- `foobar()` calls and traps the results from `foobar0` in a memo table called `Cache`. This `Cache`d value is then return by all subsequent calls to `foobar()` so multiple calls to `foobar()` all return the same value.
- `Foobar()` returns the results of `foobar()` and filters then through some other function. For example, in Figure 3, `Prec()` converts the results of `prec()` to a COCOMO regression parameter. On the other hand, `Ksloc()` in Figure 2, returns `ksloc)` without any filtering.

Much of this detail is invisible to an XOMO application programmer. They just define ranges (e.g. Figure 1) and XOMO generates code like Figure 2 and Figure 3 automatically. The programmer can then access variables via a call to the `Foobar()` functions. More experienced programmers can modify the auto-generation process by editing XOMO's macro expansion files (which are written in the M4 language).

4 Case Study

XOMO picks model inputs using Figure 1, plus any additional restraints supplied on the command line. The command line can set exact values (using the "`-=`" flag) or can define a range from some lower to upper value (using the "`-l`" and "`-u`" flags).

Using that syntax, we define the inputs to our case study as follows:

`-p ksloc -l 75 -u 125` : We assume that some developer from a prior autonomy project has guess-timate that this new project will require 75,000 to 125,000 lines of code.

`-p rely -=5` : At NASA, everything must be have highest reliability. In COCOMO, `rely`'s maximum value is very high; i.e. 5.

```
runxomo() {
  Scenario="-p ksloc -l 75 -u 125
            -p rely -= 5
            -p prec -= 1
            -p acap -= 5
            -p aexp -= 1
            -p cplx -= 6
            -p ltex -= 1
            -p ruse -= 6"
  xomo $Scenario }
```

Fig. 4 XOMO: specifying restraints.

`-p prec -= 1` : Since this team has never done this sort of thing before, the precedence (or `prec`) is set to the lowest value.

`-p acap -= 5` : This team is skillful; i.e. has highest analyst capability.

`-p aexp -= 1` : Their experience in this kind of software is non-existence.

`-p cplx -= 6` : The software is very complex.

`-p ltex -= 1` : The team has no experience with the languages and tools used for autonomous systems.

`-p ruse -= 6` : This team, in their enthusiasm, believe that the tools they are building here will be reused by many developers in the future.

Figure 4 summarizes the `XOMO` command line used in this study. Each run of Figure 4 generates one line of Figure 5. Between each run, the $Cache$ is cleared so that the next run is free to select another set of inputs. Observe in Figure 5 how the selected values satisfy *both* the ranges of Figure 1 and the restraints of Figure 4. For example, the `rely` values are all 5 (since the command line included `-p rely -= 5`) while the other values can range more widely.

5 Multi-Dimensional Optimization using "BORE"

Our goal is reducing development effort *and* the risk of schedule risk *and* the defect density in our code. Optimizing for all these three goals can be difficult. The last 3 columns of Figure 5 show scores from COCOMO, the risk model, and COQUALMO. The rows are sorted by the COQUALMO scores; i.e. by the estimated number of defects per 1000 lines of code. Interestingly, high number of remaining defects are not correlated with high schedule risk or development effort:

- The second and last rows have *similar* efforts but very *different* defect densities.
- Row two has the *highest* schedule risk but one of the *lowest* defect densities.

The reason for these non-correlations is simple: even though the three models within XOMO using the *same* variables, they predict for *different* goals. This complicates optimization since any gain achieved in one dimension may have detrimental effects on other dimensions.

26 inputs							*3 outputs*		
rely	*plex*	*ksloc*	…	*pcap*	*time*	*aa*	*effort*	*schedule risk*	*defects*
5	*1*	*118.80*	…	*5*	*3*	*5*	*2083*	*69*	*0.50*
5	*1*	*105.51*	…	*1*	*3*	*5*	*4441*	*326*	*0.86*
5	*4*	*89.26*	…	*3*	*5*	*3*	*1242*	*63*	*0.96*
5	*2*	*89.66*	…	*1*	*4*	*5*	*2118*	*133*	*2.30*
5	*1*	*105.45*	…	*2*	*4*	*5*	*6362*	*170*	*2.66*
5	*3*	*118.43*	…	*2*	*6*	*2*	*7813*	*112*	*4.85*
5	*4*	*110.84*	…	*4*	*4*	*4*	*4449*	*112*	*6.81*
…									

Fig. 5 XOMO: output from Figure 4.

rely	*plex*	*ksloc*	…	*pcap*	*time*	*aa*	*effort*	*secdRisk*	*defects*
best:									
5	*4*	*89.26*	…	*3*	*5*	*3*	*1242*	*63*	*0.96*
5	*1*	*118.80*	…	*5*	*3*	*5*	*2083*	*69*	*0.50*
5	*2*	*89.66*	…	*1*	*4*	*5*	*2118*	*133*	*2.30*
rest:									
5	*1*	*105.51*	…	*1*	*3*	*5*	*4441*	*326*	*0.86*
5	*4*	*110.84*	…	*4*	*4*	*4*	*4449*	*112*	*6.81*
5	*3*	*118.43*	…	*2*	*6*	*2*	*7813*	*112*	*4.85*

Fig. 6 BORE: classification of Figure 5.

To model this multi-dimensional optimization problem, XOMO uses a multi-dimensional classification scheme called BORE (short for "best or rest"). BORE maps simulator outputs into a hypercube which has one dimension for each utility; in our case, one dimension for effort, remaining defects, and schedule risk, These utilities are normalized to "zero" for "worst", and "one" for "best". The corner of the hypercube at 1,1,... is the *apex* of the cube and represents the desired goal for the system. All the examples are scored by their Euclidean distance to the apex. The N best examples closest to the apex are then labeled $best$. A random sample of N of the remaining examples are then labeled $rest$. Figure 6 shows a BORE report of the three $best$ and three $rest$ examples from XOMO output. Note how the average efforts, schedule risk, and defects are lower in $best$ than $rest$.

BORE's classifications are passed to a data miner to find what settings select for *best* and avoid the *rest*. Before describing that data mining process, we first describe the COCOMO, COQUALMO and schedule risk models that generated the output columns of Figure 5.

6 Models

This section describes the three models within XOMO:

- Boehm et.al.'s COCOMO-II (2000) model that computes development effort;
- Madachy's heuristic risk model that computes the risk that schedules will over run;
- Boehm et.al.'s COQUALMO model that estimates the number of defects remaining in delivered code;

6.1 The COCOMO Effort Model

COCOMO measures effort in calendar months where one month is 152 hours (and includes development and management hours). COCOMO assumes that as systems grow in

	Definition	Low-end	Medium	High-end
Scale factors:				
flex	development flexibility	development process rigorously defined	some guidelines, which can be relaxed	only general goals defined
pmat	process maturity	CMM level 1	CMM level 3	CMM level 5
prec	precedentedness	we have never built this kind of software before	somewhat new	thoroughly familiar
resl	architecture or risk resolution	few interfaces defined or few risk eliminated	most interfaces defined or most risks eliminated	all interfaces defined or all risks eliminated
team	team cohesion	very difficult interactions	basically co-operative	seamless interactions
Effort multipliers				
acap	analyst capability	worst 15%	55%	best 10%
aexp	applications experience	2 months	1 year	6 years
cplx	product complexity	e.g. simple read/write statements	e.g. use of simple interface widgets	e.g. performance-critical embedded systems
data	database size (DB bytes/SLOC)	10	100	1000
docu	documentation	many life-cycle phases not documented		extensive reporting for each life-cycle phase
ltex	language and tool-set experience	2 months	1 year	6 years
pcap	programmer capability	worst 15%	55%	best 10%
pcon	personnel continuity (% turnover per year)	48%	12%	3%
plex	platform experience	2 months	1 year	6 years
pvol	platform volatility ($\frac{frequency\ of\ major\ changes}{frequency\ of\ minor\ changes}$)	$\frac{12\ months}{1\ month}$	$\frac{6\ months}{2\ weeks}$	$\frac{2\ weeks}{2\ days}$
rely	required reliability	errors mean slight inconvenience	errors are easily recoverable	errors can risk human life
ruse	required reuse	none	multiple program	multiple product lines
sced	dictated development schedule	deadlines moved closer to 75% of the original estimate	no change	deadlines moved back to 160% of original estimate
site	multi-site development	some contact: phone, mail	some email	interactive multi-media
stor	main storage constraints (% of available RAM)	N/A	50%	95%
time	execution time constraints (% of available CPU)	N/A	50%	95%
tool	use of software tools	edit,code,debug		integrated with life cycle

Fig. 7 Parameters of the COCOMO-II effort risk model; adapted from `http://sunset.usc.edu/COCOMOII/expert_cocomo/drivers.html`. “`Stor`” and “`time`” score “N/A”” for low-end values since they have no low-end defined in COCOMO-II.

size, the effort required to create them grows exponentially, i.e. $effort \propto KSLOC^x$. More precisely, COCOMO-II uses the variables of Figure 7 as follows:

$$months = a * \left(KSLOC^{\left(b + 0.01 * \sum_{i=1}^{5} SF_i\right)} \right) * \left(\prod_{j=1}^{17} EM_j \right) \quad (1)$$

where a and b are 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). Scale factors have an exponential impact on software cost while effort multipliers have a linear impact.

```
function Effort() {
  return A() * Ksloc() ^ E()   * Rely()* Data()* Cplx()*
         Ruse()* Docu()* Time()* Stor()* Pvol()* Acap()*
         Pcap()* Pcon()* Aexp()* Plex()* Ltex()* Tool()*
         Site()* Sced()
}

function E() {
  return B() + 0.01*(Prec() + Flex()
         + Resl() + Team() + Pmat())
}
```

Fig. 8 COCOMO: computing effort.

Figure 8 shows XOMO implementation of the COCOMO effort equation. This implementation using the functions generated by Figure 1. Values such as (e.g.) `flex=1` get converted to numerics as follows. First, the integers {1, 2, 3, 4, 5, 6} are converted to the symbols {vl, l, n, h, vh, xh} (respectively) representing very low, low, nominal, high, very high,

	vl	l	n	h	vh	xh
Scale factors:						
flex	5.07	4.05	3.04	2.03	1.01	
pmat	7.80	6.24	4.68	3.12	1.56	
prec	6.20	4.96	3.72	2.48	1.24	
resl	7.07	5.65	4.24	2.83	1.41	
team	5.48	4.38	3.29	2.19	1.01	
Effort multipliers:						
acap	1.42	1.19	1.00	0.85	0.71	
aexp	1.22	1.10	1.00	0.88	0.81	
cplx	0.73	0.87	1.00	1.17	1.34	1.74
data		0.90	1.00	1.14	1.28	
docu	0.81	0.91	1.00	1.11	1.23	
ltex	1.20	1.09	1.00	0.91	0.84	
pcap	1.34	1.15	1.00	0.88	0.76	
pcon	1.29	1.12	1.00	0.90	0.81	
plex	1.19	1.09	1.00	0.91	0.85	
pvol		0.87	1.00	1.15	1.30	
rely	0.82	0.92	1.00	1.10	1.26	
ruse		0.95	1.00	1.07	1.15	1.24
sced	1.43	1.14	1.00	1.00	1.00	
site	1.22	1.09	1.00	0.93	0.86	0.80
stor			1.00	1.05	1.17	1.46
time			1.00	1.11	1.29	1.63
tool	1.17	1.09	1.00	0.90	0.78	

Fig. 9 COCOMO: co-efficients

and extremely high. Next, these are mapped into the look-up table of Figure 9.

Ideally, software effort-estimation models like COCOMO-II should be tuned to their local domain. Off-the-shelf "un-tuned" models have been up to 600% inaccurate in their estimates, e.g. [18, p165] and [8]. However, tuned models can be far more accurate. For example, [6] 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.

Elsewhere, with Boehm, Chen, Port, Hihn, and Stukes [3, 4,10,14] we have explored calibration methods for COCOMO. Here, we take a new approach and ask "what conclusions hold across the space of possible tunings"?. Hence we treat the tuning parameters "`a`" and "`b`" as random variables (see Figure 1, last two lines).

6.2 SCED-RISK: a Heuristic Risk Model

The Madachy Heuristic Risk model (hereafter SCED-RISK) was an experiment in explicating the heuristic nature of effort estimation. It returns a heuristic estimate of the chances of a schedule over run in the project. Values of 0-5 are considered to be "low risk"; 5-15 "medium risk"; 15-50 "high risk"; and 50-100 "very high risk". Studies with the COCOMO-I project database have shown that the Madachy SCED-RISK index correlates well with $\frac{months}{KDSI}$ (where KDSI is thousands of delivered source lines of code) [9].

Internally, the model contains dozens of tables of the form of Figure 10. Each such table adds some "riskiness" value to the overall project risk. These tables are read as follows. Consider the exceptional case of building high reliability systems with very tight schedule pressure (i.e. `sced=vl` or and `rely=vh` or `vh`). Recalling Figure 9, the COCOMO co-efficients for these ranges are 1.43 (for `sced=vl`) and 1.26 (for `rely=vh`). These co-efficients also have a risk factor of 2 (see Figure 10). Hence, a project with these two attribute ranges would contribute 1.43*1.26*2=3.6036 to the schedule risk.

	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

Fig. 10 SCED-RISK: an example risk table

```
Total_risk =
 (Schedule_risk  + Product_risk   + Personnel_risk +
  Process_risk   + Platform_risk  + Reuse_risk)/3.73

Schedule_risk=
  Sced_Rely_risk + Sced_Time_risk + Sced_Pvol_risk +
  Sced_Tool_risk + Sced_Acap_risk + Sced_Aexp_risk +
  Sced_Pcap_risk + Sced_Plex_risk + Sced_Ltex_risk +
  Sced_Pmat_risk

Product_risk =
  Rely_Acap_risk + Rely_Pcap_risk + Cplx_Acap_risk +
  Cplx_Pcap_risk + Cplx_Tool_risk + Rely_Pmat_risk +
  Sced_Cplx_risk + Sced_Rely_risk + Sced_Time_risk +
  Ruse_Aexp_risk + Ruse_Ltex_risk

Personnel_risk =
  Pmat_Acap_risk + Stor_Acap_risk + Time_Acap_risk +
  Tool_Acap_risk + Tool_Pcap_risk + Ruse_Aexp_risk +
  Ruse_Ltex_risk + Pmat_Pcap_risk + Stor_Pcap_risk +
  Time_Pcap_risk + Ltex_Pcap_risk + Pvol_Plex_risk +
  Sced_Acap_risk + Sced_Aexp_risk + Sced_Pcap_risk +
  Sced_Plex_risk + Sced_Ltex_risk + Rely_Acap_risk +
  Rely_Pcap_risk + Cplx_Acap_risk + Cplx_Pcap_risk +
  Team_Aexp_risk
Process_risk =
  Tool_Pmat_risk + Time_Tool_risk + Tool_Pmat_risk +
  Team_Aexp_risk + Team_Sced_risk + Team_Site_risk +
  Sced_Tool_risk + Sced_Pmat_risk + Cplx_Tool_risk +
  Pmat_Acap_risk + Tool_Acap_risk + Tool_Pcap_risk +
  Pmat_Pcap_risk

Platform_risk =
  Sced_Time_risk + Sced_Pvol_risk + Stor_Acap_risk +
  Time_Acap_risk + Stor_Pcap_risk + Pvol_Plex_risk +
  Time_Tool_risk

Reuse_risk =
  Ruse_Aexp_risk + Ruse_Ltex_risk
```

Fig. 11 SCED-RISK: the calculations.

The details of the SCED-RISK calculations are shown in Figure 11. The risk tables of the current model are shown in Figure 12.

6.3 COQUALMO: defect introduction and removal

COQUALMO models how process options *add* and *remove* defects to software during *requirements*, *design*, and *coding*. For example, poor documentation leads to more errors since developers lack the guidance required to code the right system. So, Figure 13 offers its large defect introduction values

		vl	l	n	h	vh	xh
sced	rely						
	vl				1	2	
	l					1	
sced	cplx						
	vl				1	2	4
	l					1	2
	n						1
sced	time						
	vl				1	2	4
	l					1	2
	n						1
sced	pvol						
	vl				1	2	
	l					1	
sced	tool						
	vl	2	1				
	l	1					
sced	pexp						
	vl	4	2	1			
	l	2	1				
	n	1					
sced	pcap						
	vl	4	2	1			
	l	2	1				
	n	1					
sced	aexp						
	vl	4	2	1			
	l	2	1				
	n	1					
sced	acap						
	vl	4	2	1			
	l	2	1				
	n	1					
sced	ltex						
	vl	2	1				
	l	1					
sced	pmat						
	vl	2	1				
	l	1					

		vl	l	n
rely	acap			
	n	1		
	h	2	1	
	vh	4	2	1
rely	pcap			
	n	1		
	h	2	1	
	vh	4	2	1
cplx	acap			
	h	1		
	vh	2	1	
	xh	4	2	1
cplx	pcap			
	h	1		
	vh	2	1	
	xh	4	2	1
cplx	tool			
	h	1		
	vh	2	1	
	xh	4	2	1
rely	pmat			
	n	1		
	h	2	1	
	vh	4	2	1
pmat	acap			
	vl	2	1	
	l	1		
stor	acap			
	h	1		
	vh	2	1	
	xh	4	2	1
time	acap			
	h	1		
	vh	2	1	
	xh	4	2	1
tool	acap			
	vl	2	1	
	l	1		
tool	pcap			
	vl	2	1	
	l	1		

		vl	l	n
ruse	aexp			
	h	1		
	vh	2	1	
	xh	4	2	1
ruse	ltex			
	h	1		
	vh	2	1	
	xh	4	2	1
pmat	pcap			
	vl	2	1	
	l	1		
stor	pcap			
	h	1		
	vh	2	1	
	xh	4	2	1
time	pcap			
	h	1		
	vh	2	1	
	xh	4	2	1
ltex	pcap			
	vl	4	2	1
	l	2	1	
	n	1		
pvol	pexp			
	h	1		
	vh	2	1	
tool	pmat			
	vl	2	1	
	l	1		
time	tool			
	vh	1		
	xh	2	1	
team	aexp			
	vl	2	1	
	l	1		
team	sced			
	vl	2	1	
	l	1		
team	site			
	vl	2	1	
	l	1		

Fig. 12 SCED-RISK: the details. For example, looking at the top-left matrix, the `Sced_Rely_risk` is highest when the reliability is very high but the schedule pressure is very tight.

	rely	data	ruse	docu	cplx	time	stor	pvol	acap	pcap	pcon	aexp	plex	ltex	tool	site	sced
requirements:																	
xh			1.05		1.32	1.08	1.08	1.16								0.83	
vh	0.7	1.07	1.03	0.86	1.21	1.05	1.05	1.1	0.75	1	0.82	0.81	0.9	0.93	0.92	0.89	0.85
h	0.85	1.04	1.02	0.93	1.1	1.03	1.03	1.05	0.87	1	0.91	0.91	0.95	0.97	0.96	0.95	0.92
n	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
l	1.22	0.93	0.95	1.08	0.88			0.86	1.17	1	1.11	1.12	1.05	1.04	1.05	1.1	1.09
vl	1.43			1.16	0.76				1.33	1	1.22	1.24	1.11	1.07	1.09	1.2	1.18
design:																	
xh			1.02		1.41	1.2	1.18	1.2								0.83	
vh	0.69	1.1	1.01	0.85	1.27	1.13	1.12	1.13	0.83	0.85	0.8	0.82	0.86	0.88	0.91	0.89	0.84
h	0.85	1.05	1	0.93	1.13	1.06	1.06	1.06	0.91	0.93	0.9	0.91	0.93	0.91	0.96	0.95	0.92
n	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
l	1.23	0.91	0.98	1.09	0.86			0.83	1.1	1.09	1.13	1.11	1.09	1.07	1.05	1.1	1.1
vl	1.45			1.18	0.71				1.2	1.17	1.25	1.22	1.17	1.13	1.1	1.2	1.19
coding:																	
xh			1.02		1.41	1.2	1.15	1.22								0.85	
vh	0.69	1.1	1.01	0.85	1.27	1.13	1.1	1.15	0.9	0.76	0.77	0.88	0.86	0.82	0.8	0.9	0.84
h	0.85	1.05	1	0.92	1.13	1.06	1.05	1.08	0.95	0.88	0.88	0.94	0.94	0.91	0.9	0.95	0.92
n	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
l	1.23	0.91	0.98	1.09	0.86			0.82	1.05	1.16	1.15	1.07	1.08	1.11	1.13	1.09	1.1
vl	1.45			1.18	0.71				1.11	1.32	1.3	1.13	1.16	1.22	1.25	1.18	1.19

Fig. 13 COQUALMO: effort multipliers and defect introduction

	prec	flex	resl	team	pmat
requirements:					
xh	0.7	1	0.76	0.75	0.73
vh	0.84	1	0.87	0.87	0.85
h	0.92	1	0.94	0.94	0.93
n	1	1	1	1	1
l	1.22	1	1.16	1.17	1.19
vl	1.43	1	1.32	1.34	1.38
design:					
xh	0.75	1	0.7	0.8	0.61
vh	0.87	1	0.84	0.9	0.78
h	0.94	1	0.92	0.95	0.89
n	1	1	1	1	1
l	1.17	1	1.22	1.13	1.33
vl	1.34	1	1.43	1.26	1.65
coding:					
xh	0.81	1	0.71	0.86	0.63
vh	0.9	1	0.84	0.92	0.79
h	0.95	1	0.92	0.96	0.9
n	1	1	1	1	1
l	1.12	1	1.21	1.09	1.3
vl	1.24	1	1.41	1.18	1.58

Fig. 14 COQUALMO: scale factors and defect introduction

```
function defectsIntroduced() {
 return 10*Ksloc()*defectsIntroduced1("requirements") +
        20*Ksloc()*defectsIntroduced1("design") +
        30*Ksloc()*defectsIntroduced1("coding") }

function defectsIntroduced1(table) {
  # return the product of the Figure 13 and
  # and the Figure 14 figures  }
```

Fig. 15 COQUALMO: defects introduced.

	automated	peer	execution_testing
	analysis	reviews	_and_tools
requirements:			
xh	0.4	0.7	0.6
vh	0.34	0.58	0.57
h	0.27	0.5	0.5
n	0.1	0.4	0.4
l	0	0.25	0.23
vl	0	0	0
design:			
xh	0.5	0.78	0.7
vh	0.44	0.7	0.65
h	0.28	0.54	0.54
n	0.13	0.4	0.43
l	0	0.28	0.23
vl	0	0	0
coding:			
xh	0.55	0.83	0.88
vh	0.48	0.73	0.78
h	0.3	0.6	0.69
n	0.2	0.48	0.58
l	0.1	0.3	0.38
vl	0	0	0

Fig. 16 COQUALMO: defect removal

when the effort multiplier `docu=vl` is very low. See also Figure 14 for the defects introduced by various settings to the scale factors.

As shown in Figure 15 the COQUALMO defect introduction factors are effects-per-1000 lines of code. A small weighting factor (10,20,30) is added to show an increasing number of defects as the life cycle progresses.

The defects remaining in software is the product of the defects introduced times the percentage removed (see Figure 16 and Figure 17). The removal percentage is calculated in Figure 18 which shows how various actions (`automated analysis`,`peer reviews`, and `execution testing and tools`) remove defects during *requirements*, *design* and *coding*. These values are ratios per 1000 lines of code so their complement represents the remaining defects (see Figure 18).

```
function Total_defects() {
  return defects("requirements",Coqualr) +
         defects("design",      Coquald) +
         defects("coding",      Coqualc)
}

function defects(what,table) {
  introduced     = defectsIntroduced1(what,table);
  percentRemoved = defectsRemovedRatio(what);
  return percentRemoved*introduced
}
```

Fig. 17 COQUALMO: defects added and removed

```
function defectsRemovedRatio(table, auto,review,tool) {
  return (1 - drf(table,"automated_analysis")) *
         (1 - drf(table,"peer_reviews")) *
         (1 - drf(table,"execution_testing_and_tools"))
}

function drf(table,x ) {
  # return x's value in table from Figure 16
}
```

Fig. 18 COQUALMO: ratio of defects removed

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

Fig. 19 TAR3: Playing golf.

7 Learning

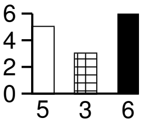
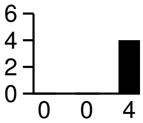
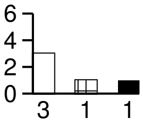
Once the above models run, and BORE classifies the output into $best$ and $rest$, a data miner is used to find input settings that select for the better outputs. This study uses treatment learning since this learning method return the *smallest* theories that *most* effect the output. In terms of software process changes, such minimal theories are useful since they require the fewest management actions to improve a project.

7.1 Treatment Learning

Treatment learning inputs a set of training examples E. Each example maps a set of attribute ranges to some class sym-

input:	`SELECT class`
	`FROM golf`
		`SELECT class`
		`FROM golf`
		`WHERE`
		`outlook = 'overcast'`
			`SELECT class`
			`FROM golf`
			`WHERE`
			`humidity >= 90`

output:	`none none none none none`
	`some some some lots lots`
	`lots lots lots lots`
		`lots lots lots lots`	`none none none some lots`

distributions:			

legend: □ none ▦ some ■ lots

Fig. 20 TAR3: Class distributions selected by different conditions in Figure 19.

bol; i.e. $\{R_i, R_j, ... \rightarrow C\}$ The class symbols $C_1, C_2..$ are stamped with some utility score that ranks the classes; i.e. $\{U_1 < U_2 < .. < U_C\}$. With E, these classes occur at frequencies $F_1\%, F_2\%, ..., F_C\%$. A treatment T of size X is a conjunction of attribute ranges $\{R_1 \wedge R_2... \wedge R_X\}$. Some subset of $e \subseteq E$ are consistent with the treatment. In that subset, the classes occur at frequencies $f_1\%, f_2\%, ...f_C\%$. A treatment learner seeks the seek smallest T which most changes the weighted sum of the utilities times frequencies of the classes. Formally, this is called the $lift$ of a treatment:

$$lift = \frac{\sum_C U_C f_C}{\sum_C U_C F_C}$$

For example, consider the log of golf playing behavior seen in Figure 19. In that log, we only play *lots* of golf in $\frac{6}{5+3+6} = 43\%$ of cases. To improve our game, we might search for conditions that increases our golfing frequency. Two such conditions are shown in the `WHERE` test of the select statements in Figure 20. In the case of `outlook= overcast`, we play *lots* of golf all the time. In the case of `humidity` $\leq$ `90`, we only play *lots* of golf in 20% of cases. So one way to play lots of golf would be to select a vacation location where it was always overcast. While on holidays, one thing to watch for is the humidity: if it rises over 90%, then our frequent golf games are threatened.

The tests in the `WHERE` clause of the select statements in Figure 20 is a treatment. Classes in treatment learning get a score U_C and the learner uses this to assess the class frequencies resulting from *applying a treatment* (i.e. using them in a `WHERE` clause). In normal operation, a treatment learner does *controller learning* that finds a treatment which selects for better classes and reject worse classes By reversing the scoring function, treatment learning can also select for the worse classes and reject the better classes. This mode is called *monitor learning* since it finds the thing we should most watch for. In the golf example, *outlook = 'overcast'* was the controller and $humidity \geq 90$ was the monitor.

Formally, treatment learning is a weighted-class minimal contrast-set association rule learner. The treatments are associations that occur with preferred classes. These treatments serve to contrast undesirable situations with desirable situation where more of the outcomes are favorable. Treatment learning is different to other contrast set learners like STUCCO [1] since those other learners don't focus on minimal theories.

Conceptually, a treatment learner explores all possible subsets of the attribute ranges looking for good treatments. Such a search is impractical in practice so the art of treatment learning is quickly pruning unpromising attribute ranges. This study uses the TAR3 treatment learner [7] that uses stochastic search to find its treatments.

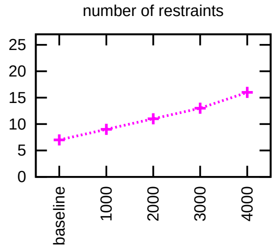
7.2 Iterative Treatment Learning

Sometimes, one round of treatment learning is not enough. *Iterative treatment learning* runs by conducting Monte Carlo simulations over the ranges of any uncertain variables. For example, there are 28 variables in the XOMO models:

- Ksloc;
- 5 scale factors;
- 17 effort multipliers;
- 2 calibration parameters ("`a,b`");
- 3 defect removal activities (automated analysis, peer reviews, execution testing and tools).

The restraints of Figure 4 only offers hard constraints on seven of the variables: `rely, prec, acap, ...`[1]. XOMO's Monte Carlos execute by picking random values

[1] The constraint on `ksloc` is softer- it can vary from 75K to 125K).



baseline	learned restraints			
	1000	2000	3000	4000
75 ≤ ksloc ≤ 125	sced=4	pmat=5	tool=4	team=5
rely = 5	peer_reviews=5	pcap=4	execution_testing-_and_tools=5	resl=5
prec = 1				automated-analysis=5
acap = 5				
aexp = 1				
cplx = 6				
ltex = 1				
ruse = 6				

Fig. 21 TAR3: learned restraints

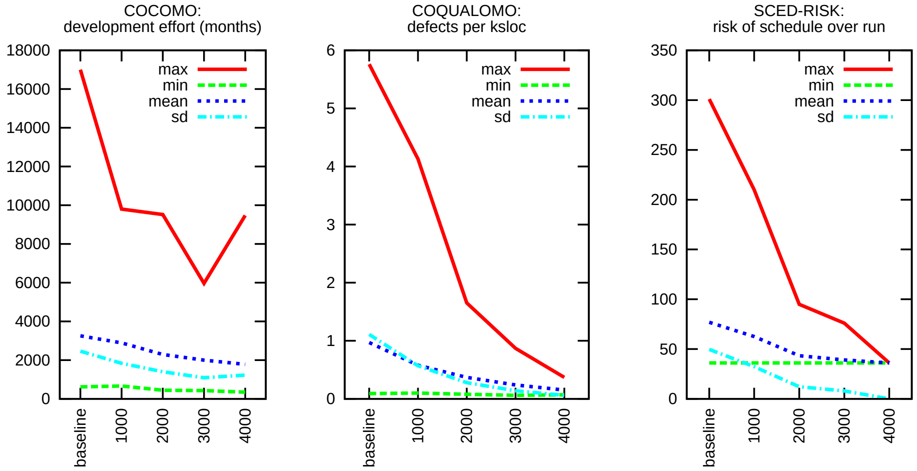


Fig. 22 TAR3: impact of Figure 21's restraints.

from valid ranges for all known inputs. After, say, 1000 Monte Carlo runs, BORE classifies the outputs as either the 100 *best* or 100 *rest*. The treatment learner studies the results and notes which input ranges select for *best*. The ranges found by the learner then become restraints for future simulations. The whole cycle looks like this:

$$restraints_i \rightarrow simulation_i \rightarrow learn \rightarrow \\ \rightarrow restraints_{i+i} \rightarrow simulation_{i+1}$$

8 Results

For this study the initial baseline restraints were set according to our autonomy settings; i.e. Figure 4. XOMO was run 1000 times each iteration and BORE returned the 100 *best* examples and a random sample of 100 of the *rest*. These *best* and *rest* examples were passed to TAR3 and the best learned treatment was imposed as restraints on subsequent iterations.

Figure 21 shows the restraints learned by four iterations of iterative treatment learning. Figure 22 shows the effects of these restraints on the output of the XOMO models:

1. The mean development effort was nearly halved: 3257 to 1780 months;
2. The mean SCED-RISK halved: 77 to 36;
3. The mean defects densities were reduced by 85% from 0.97 to 0.15.
4. The variance on the above measures was significantly reduced: the COQUALMO and SCED-RISK standard deviations nearly reached zero.

Several of the Figure 22 curves flatten out after 2000 runs of XOMO. A parsimonious management strategy could be formed from just the results of the first two rounds of learning. Interestingly, in those first two rounds, process changes were more important than the application of technology. Technology-based techniques such as `tool` support or `execution testing and tools` did not arise till iteration three. On the other hand, the first two iterations labeled "1000,2000" in Figure 21 want to decrease schedule pressure (`sced`), increase process maturity (`pmat`), and programmer capability (`pcap`) and requested the user of peer reviews.

Another interesting feature of the results is that many of the inputs were never restrained. The left-hand-side plot of Figure 21 shows that even after four rounds of learning, only 17 of the 28 inputs were restrained. That is, management commitments to 11 of the 28 inputs would have been a waste of time. Further, if management is content with the improvements gained from the first two iterations, then only 12 restraints are required and decisions about the remaining 16 inputs would have been superfluous.

9 Discussion

Software models like COCOMO, COQUALMO, and SCED-RISK contain many assumptions about their domain. The conclusions gained from this models should be scrutinized by domain experts. Early in the life cycle of a software project, such scrutiny is complicated by all the unknowns associated with a project. Exploring all those unknowns can lead to massive data overload as domain experts are buried beneath a mountain of data coming from their simulators.

Tools like XOMO, BORE, and treatment learners like TAR3 can assist in that scrutiny. These tools can find automatically find software process decisions that reduce defects and effort and risk of schedule over run. These tools sample the space of options and report sample conclusions within the space of possibilities.

To demonstrate that technique, this paper conducted a case study with software development for autonomous systems. Certain special features of autonomous systems were identified. These features included high complexity and little experience with building these kinds of systems in the past. These features were then mapped into general software cost and risk models.

It is encouraging that the analysis is so fast: the above case study took less than ten minutes to run on a standard computer. Hence, we can use these tools during early life cycle debates about options within a software project.

While the particular case study examined here is quite specific, the analysis method is quite general. Our case study related to autonomous systems, but there is nothing stopping an analyst from using XOMO to study other kinds of software development. The only requirement is that the essential features of that software can be mapped onto COCOMO-like models.

- All the models used here contain most of their knowledge in easy-to-modify tables representing the particulars of different domains.
- All the tools used here are portable and use simple command-line switches that allow an analyst to quickly run through a similar study for a different kind of project.

References

1. S. Bay and M. Pazzani. Detecting change in categorical data: Mining contrast sets. In *Proceedings of the Fifth International Conference on Knowledge Discovery and Data Mining*, 1999. Available from `http://www.ics.uci.edu/~pazzani/Publications/stucco.pdf`.
2. B. Boehm, E. Horowitz, R. Madachy, D. Reifer, B. K. Clark, B. Steece, A. W. Brown, S. Chulani, and C. Abts. *Software Cost Estimation with Cocomo II*. Prentice Hall, 2000.
3. Z. Chen, T. Menzies, and D. Port. Feature subset selection can improves software cost estimation. In *Proceedings, PROMISE workshop, ICSE 2005*, 2005. Available from `http://menzies/pdf/05/fsscocomo.pdf`.
4. Z. Chen, T. Menzies, D. Port, and B. Boehm. Finding the right data for software cost modeling. *IEEE Software*, Nov 2005.
5. E. Chiang and T. Menzies. Simulations for very early lifecycle quality evaluations. *Software Process: Improvement and Practice*, 7(3-4):141–159, 2003. Available from `http://menzies.us/pdf/03spip.pdf`.
6. 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.
7. Y. Hu. Treatment learning, 2002. Masters thesis, Unviersity of British Columbia, Department of Electrical and Computer Engineering. In preperation.
8. C. Kemerer. An empirical validation of software cost estimation models. *Communications of the ACM*, 30(5):416–429, May 1987.
9. R. Madachy. Heuristic risk assessment using cost factors. *IEEE Software*, 14(3):51–59, May 1997.
10. T. Menzies, Z. Chen, D. Port, and J. Hihn. Simple software cost estimation: Safe or unsafe? In *Proceedings, PROMISE workshop, ICSE 2005*, 2005. Available from `http://menzies.us/pdf/05safewhen.pdf`.
11. T. Menzies, E. Chiang, M. Feather, Y. Hu, and J. Kiper. Condensing uncertainty via incremental treatment learning. In T. M. Khoshgoftaar, editor, *Software Engineering with Computational Intelligence*. Kluwer, 2003. Available from `http://menzies.us/pdf/02itar2.pdf`.
12. T. Menzies and Y. Hu. Constraining discussions in requirements engineering. In *First International Workshop on Model-based Requirements Engineering*, 2001. Available from `http://menzies.us/pdf/01lesstalk.pdf`.
13. T. Menzies and Y. Hu. Data mining for very busy people. In *IEEE Computer*, November 2003. Available from `http://menzies.us/pdf/03tar2.pdf`.
14. T. Menzies, D. Port, Z. Chen, J. Hihn, and S. Stukes. Validation methods for calibrating software effort models. In *Proceedings, ICSE*, 2005. Available from `http://menzies.us/pdf/04coconut.pdf`.
15. 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`.
16. T. Menzies, S. Setamanit, and D. Raffo. Data mining from process models. In *PROSIM 2004*, 2004. Available from `http://menzies.us/pdf/04dmpm.pdf`.
17. 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://menzies.us/pdf/00ase.pdf`.
18. T. Mukhopadhyay, S. Vicinanza, and M. Prietula. Examining the feasibility of a case-based reasoning tool for software effort estimation. *MIS Quarterly*, pages 155–171, June 1992.