Towards Faster Model-Aided Decision Making

Andres Orrego
Global Science & Technology
Fairmont, WV, USA
andres.orrego@gst.com

Tim Menzies
West Virginia University
Morgantown, WV, USA
tim@menzies.us

Gregory Gay
West Virginia University
Morgantown, WV, USA
gregoryg@csee.wvu.edu

ABSTRACT

Previously we have achieved significant performance improvements in model optimization using our KEYS2 search algorithm on defect detection and prevention (DDP) models from JPL. In these models we seek the reduction of software development costs while maximizing requirements attainment. Unfortunately, as these models grow larger, the search space typically grows at an exponential rate, and the demand for faster optimizations methods increases. In this study we generate large risk models and attempt to speed up the KEYS2 search engine. After some experimentation and analysis we arrived at a fairly simple optimization technique that maintains the level of attainment and cost reductions of KEYS2 but also significantly reduces the run time on larger models so the engineers making decisions based on model simulations do not have to spend their valuable time waiting for the results of the tool to come back.

Keywords: Model optimization, AI search

1. INTRODUCTION

Model-based design is now a software engineering technique employed at various degrees by software engineering teams in numerous large organizations such as as Microsoft [9]; Lockheed Martin [14]; the Object Management Group [2]; and the NASA's Jet Propulsion Laboratory [7].

Such models can be queried to find combinations of options that might be otherwise missed. For example, with DDP, the goal is a non-linear optimization that seeks the *least* costly project options that *most* increases the chance of attaining more requirements.

Paradoxically, prior successes [7, 8, 12] with DDP have caused a problem. The reality is that as the DDP models grow larger, the demand for faster optimization methods also increases, particularly when those models are used by a large room of debating experts as part of rapid interactive dialogues. Hence, there is a pressing need for "real-time requirements optimization"; i.e. requirements optimizers that can offer advice before an expert's attention wanders to other issues. Our user community now expects an automatic model-based cost-benefit analysis for larger and larger JPL models containing more variants. Further, as requirements change (as is frequently the case), our users are demanding a complete re-analysis of all past decisions. Extrapolating into the near future, we expect to fall off a computational cliff where our models will be too complex for automatic analysis.Accordingly, we explore optimizations for model-based design.

Prior experiments with simulated annealing [7] or treatment learning [8] terminated in minutes to hours. Subsequent optimization attempts by Jalali et. al. resulted in a method, called "KEYS", that runs significantly faster, e.g. for one model, KEYS ran 13,000 times faster than treatment learning (40 minutes to 0.18 secs) [10]. Although this speed gain is very impressive, KEYS performance starts to degrade rapidly for growing models. Results from experiments in this study show that KEYS takes on average 99 seconds to find a solution for the largest DDP model we generated.

We recently achieved another significant speedup in DDP model optimization with KEYS2. This later algorithm is based on the original KEYS algorithm but runs four orders of magnitude faster. In our experiments, KEYS2's takes on average an impressive 8.7 seconds over our largest model.

Given that these models will keep growing in size and complexity, in this study we analyze the KEYS and KEYS2 algorithms searching for areas of improvement in their implementation. We then address address those areas in order to achieve faster performance, particularly for larger models.

The rest of this paper described JPL's DDP modeling systems; the KEYS and KEYS2 algorithms; our latest improvements; and experiments performed on large, randomly-generated DDP models.

2. DDP MODEL

The defect detection and prevention (DDP) approach was first invented in 1998 by Steven Cornford, at the Jet Propulsion Laboratory [4]. It is a risk-based requirements model that assists in early life-cycle decision making to help developers select assurance activities in a cost-effective maner. That is, maximizing requirements attainment while minimizing mitigation costs.

The process by which DDP is employed is as follows:

- 6 to 20 experts are gathered together for short, intensive knowledge acquisition sessions (typically, 3 to 4 half-day sessions). These sessions *must* be short since it is hard to gather together these experts for more than a very short period of time.
- The DDP tool supports a graphical interface for the rapid entry of the assertions. Such rapid entry is essential, lest using the tool slows up the debate.
- Assertions from the experts are expressed in using an ultra-lightweight decision ontology (e.g. see Figure 1). The ontology *must* be ultra-lightweight since:

  - Only brief assertions can be collected in short knowledge acquisition sessions.
  - If the assertions get more elaborate, then experts may be unable to understand technical arguments from outside their own field of expertise.

The result of these sessions is a network of influences connecting project requirements to risks to possible mitigations.

The ontology of Figure 1 may appear too weak for useful reasoning. However, in repeated sessions with DDP, it has been seen that the ontology is rich enough to structure and guide debates between NASA experts. For example, DDP has been applied to over a dozen applications to study advanced technologies such as

- a computer memory device;
- gyroscope design;
- software code generation;
- a low temperature experiments apparatus;
- an imaging device;
- circuit board like fabrication;
- micro electro-mechanical devices;
- a sun sensor;
- a motor controller;
- photonics; and
- interferometry.

DDP assertions are either:

- *Requirements* (free text) describing the objectives and constraints of the mission and its development process;
- *Weights* (numbers) associated with requirements, reflecting their relative importance;
- *Risks* (free text) are events that damage requirements;
- *Mitigations*: (free text) are actions that reduce risks;
- *Costs*: (numbers) effort associated with mitigations, and repair costs for correcting Risks detected by Mitigations;
- *Mappings*: directed edges between requirements, mitigations, and risks that capture quantitative relationships among them. The key ones are *impacts*, each one of which is a quantitative estimate of the proportion of a requirement that would be lost should a risk occur, and *effects*, each one of which is a quantitative estimate of the proportion by which a risk would be reduced were a mitigation to be employed (the ontology is also able to capture the phenomenon of a mitigation making some risks *worse*).
- *Part-of relations* structure the collections of requirements, risks and mitigations;

Figure 1: DDP's ontology

In those studies, DDP sessions have led to cost savings exceeding $1 million in at least two instances, and lesser amounts (exceeding $100,000) in some others. The DDP sessions have also generated numerous design improvements such as savings of power or mass, and shifting of risks from uncertain architectural ones to better understood (and hence more predictable and manageable) design ones. Further, at these meetings, some non-obvious significant risks have been identified and mitigated. Lastly, DDP can be used to document the information and decision making of these studies. Hence, DDP, although not mandated, remains in use at JPL:

- not only for its original purpose (group decision support);
- but also as a design rationale tool to document decisions.

Note that DDP is not just a software design tool. At JPL, software and hardware are designed together. DDP is best viewed as a *software systems engineering* tool where the interactions between hardware and software can be quickly explored.

2.1 DDP Model Format

In the implementation of DDP models, values for risks, requirements, and mitigations are assigned to reflect importance, likelihood, and cost respectively. Each requirement is assigned a numeric *Weight* ranging from 0 to a MAXWEIGHT, usually 100. This number denotes the priority of the requirement in terms of how important it is to attain it compared to other requirements. In terms of risks, each one of them is assigned a likelihood indicating the probability of its occurrance in case no mitigation is exercised. This a-priori likelihood or *rAPL* is measured as a floating-point number ranging from 0 to 1 . Lastly, each mitigation is assigned a *Cost* which is usually the financial cost it would take to take the steps necessary to prevent a risk (or risks) from happening. Mitigations are also assigned a boolean, *Selected*, that is set to *true* it will be performed, *false* otherwise.

In additon to these factors, DDP models also consider the relationships among them. For instance, risks and requirements are related in that if the former occurs, the attainment of the latter is negatively impacted. Given that our goal is to maximize requirement, this impact is measured as the loss of attainment imposed by the risk should it occur in floating-point values ranging from 0 to 1 (inclusive). An *Impact(risk,requirement)* value of 0.5 means that should the risk happen, the requirements attainment is reduced by one half. A second relationship is established between risks and mitigatins. The *effect* of a mitigation indicates how much it reduces each risk, and it is also measured in decimal values ranging from 0 to 1 inclusive. An *Effect(mitigation,risk)* value of 0.5 means that the given mitigation reduces the risk by one half.

Given these factors and the relationships among them we can then search for the best way to control them to achieve the maximum requirements attainment at the lowest mitigation costs. Keep in mind that maximizing attainment requires minimizing risks by implementing costly mitigations. At the same time, minimizing costs prevents projects from implementing mitigations to reduce risks.

One of the measures that we have taken to improve the runtimes of previous experiments is to use a method called *knowledge compilation*. Knowledge compilation is a technique where certain information is compiled into a target language. Previous work with knowledge compilation has focused on compilation languages utilizing CNF equations, state machines, or BDD [1,5]. None of these made sense for our experiments, we instead compiled the DDP models into a structure used by the C programming language.

These compiled models are used to rapidly answer a large number of queries while the main program is running [5, 13]. This

```
include ``model.h''

#define M_COUNT 2
#define O_COUNT 3
#define R_COUNT 2

struct ddpStruct
{
  float oWeight[O_COUNT+1];
  float oAttainment[O_COUNT+1];
  float oAtRiskProp[O_COUNT+1];
  float rAPL[R_COUNT+1];
  float rLikelihood[R_COUNT+1];
  float mCost[M_COUNT+1];
  float roImpact[R_COUNT+1][O_COUNT+1];
  float mrEffect[M_COUNT+1][R_COUNT+1];
};

ddpStruct *ddpData;

void setupModel(void)
{
  ddpData = (ddpStruct *) malloc(sizeof(ddpStruct));
  ddpData->mCost[1]=11;
  ddpData->mCost[2]=22;
  ddpData->rAPL[1]=1;
  ddpData->rAPL[2]=1;
  ddpData->oWeight[1]=1;
  ddpData->oWeight[2]=2;
  ddpData->oWeight[3]=3;
  ddpData->roImpact[1][1] = 0.1;
  ddpData->roImpact[1][2] = 0.3;
  ddpData->roImpact[2][1] = 0.2;
  ddpData->mrEffect[1][1] = 0.9;
  ddpData->mrEffect[1][2] = 0.3;
  ddpData->mrEffect[2][1] = 0.4;
}

void model(float *cost, float *att, float m[])
{
  float costTotal, attTotal;
  ddpData->rLikelihood[1] = ddpData->rAPL[1] *
             (1 - m[1] * ddpData->mrEffect[1][1]) *
             (1 - m[2] * ddpData->mrEffect[2][1]);
  ddpData->rLikelihood[2] = ddpData->rAPL[2] *
             (1 - m[1] * ddpData->mrEffect[1][2]);
  ddpData->oAtRiskProp[1] = (ddpData->rLikelihood[1] *
             ddpData->roImpact[1][1]) +
             (ddpData->rLikelihood[2] *
             ddpData->roImpact[2][1]);
  ddpData->oAtRiskProp[2] = (ddpData->rLikelihood[1] *
             ddpData->roImpact[1][2]);
  ddpData->oAtRiskProp[3] = 0;
  ddpData->oAttainment[1] = ddpData->oWeight[1] *
             (1 - minValue(1, ddpData->oAtRiskProp[1]));
  ddpData->oAttainment[2] = ddpData->oWeight[2] *
             (1 - minValue(1, ddpData->oAtRiskProp[2]));
  ddpData->oAttainment[3] = ddpData->oWeight[3] *
             (1 - minValue(1, ddpData->oAtRiskProp[3]));
  attTotal = ddpData->oAttainment[1] +
             ddpData->oAttainment[2] +
             ddpData->oAttainment[3];
  costTotal = m[1] * ddpData->mCost[1] +
              m[2] * ddpData->mCost[2];

  *cost = costTotal;
  *att = attTotal;
}
```

Figure 2: A trivial DDP model

pushes a large percentage of the computational overhead into the compilation phase. This cost is amortized over time as we increase the number of on-line queries. Some of the algorithms that we use make thousands of calls to these DDP models; therefore, the run-time increase from knowledge compilation is significant.

This knowledge compilation process stores a flattened form of the DDP requirements tree. In standard DDP:

- Requirements form a tree;
- The relative influence of each leaf requirement is computed via a depth-first search from the root down to the leaves.
- This computation is repeated each time the relative influence of a requirement is required.

In our compiled form, the computation is performed once and added as a constant to each reference of the requirement.

For example, here is a trivial DDP model where `mitigation1` costs $10,000 to apply and each requirement is of equal value (100):

$$\overbrace{mitigation1}^{\$10{,}000} \underbrace{\rightarrow}_{0.9} risk1 \rightarrow \left\langle \begin{array}{l} \overbrace{\rightarrow}^{0.1}(requirement1 = 100) \\ \underbrace{\rightarrow}_{0.99}(requirement2 = 100) \end{array} \right.$$

(The other numbers show the impact of mitigations on risks, and risks on requirements).

The knowledge compiler converts this trivial DDP model into `setupModel` and `model` functions similar to those in Figure 2. The `setupModel` function is called once and sets several constant values. The `model` function is called whenever new cost and attainment values are needed. The topology of the mitigation network is represented as terms in equations within these functions. As our models grows more complex, so do these equations. For example, our biggest model, which contains 99 mitigations, generates 1427 lines of code.

Knowledge compilation is useful for more than just runtime optimization, it has actually made some of this research possible in the first place. This method allows JPL to retain their proprietary information. In turn, this has given researchers outside of JPL access to their models. For our experiments, JPL ran the knowledge compiler and passed West Virginia University models like those shown in Figure 2. These models have been anonymized to remove proprietary information while still maintaining their computational nature. JPL could assure its clients that any project secrets would be safe while allowing outside researchers to perform valuable experiments.

3. GENERATING BIGGER MODELS

Currently, the number of DDP models generated at JPL is very limited due to the relatively short period of time these models have been around and the long time it takes to get the right project people to meet and discuss requirements, risks, and mitigations.So far we have access to the five models summarized in Figure 3.

Model	LOC	Objectives	Risks	Mitigations	Run-Time*
model1.c	43	3	2	2	0.0018
model2.c	260	1	30	31	0.0139
model3.c	58	3	2	3	0.0019
model4.c	1226	50	31	58	0.0906
model5.c	1412	32	70	99	0.1751

*average over 100 runs (in seconds)

Figure 3: Details of Five DDP Models.

	Model	LOC	Objectives	Risks	Mitigations
Random-generated Models	model2_1.c	264	1	30	31
	model2_2.c	502	2	60	62
	model2_4.c	978	4	120	124
	model2_8.c	1930	8	240	248
	model2_16.c	3834	16	480	496
	model4_1.c	1233	50	31	58
	model4_2.c	2428	100	62	116
	model4_4.c	4818	200	124	232
	model4_8.c	9598	400	248	464
	model4_16.c	19158	800	496	928
	model5_1.c	1545	32	70	99
	model5_2.c	3053	64	140	198
	model5_4.c	6069	128	280	396
	model5_8.c	12101	256	560	792
	model5_16.c	24165	512	1120	1584

Figure 4: Details of incrementally bigger randomly-generated DDP Models.

One aspect to note from these available models is the increase in size; from 43 lines of code to 1,412. In order to find the best search method to find the best solution to these increasingly larger models, more instances need to be explored and therefore an instance generator is required. Such instance generator has to follow the constraints set by the model and randomize the values of its variables so the external validity of studies done using them is not compromised.

For this study we focused on models 2, 4 and 5, and developed our own random generator of models and artificially grow the original JPL models by a factor of 2, 4, 8, and 16, obtaining the models in Figure 4. We use these models for comparing our optimization algorithms.

4. KEYS

The premise of KEYS is that within the space of possible decisions, there exist a small number of *key* decisions that determine all others [11]. If a model contains keys, then a general search through a large space of options is superfluous. A better (faster, simpler) approach would be to just explore the keys. KEYS uses support-based Bayesian sampling to quickly find these important variables.

There are two main components to KEYS - a greedy search and the BORE ranking heuristic.

4.1 Greedy Search

KEYS searches a space of M mitigations over the course of M "eras". Initially, the entire set of mitigations is set randomly. During each era, one more mitigation is set to $M_i = X_j$, $X_j \in \{true, false\}$. Each era e generates a set $< input, score >$ as follows:

1a: $Selected[1 \ldots (e-1)]$ are settings from previous eras.

1b: $Guessed$ are randomly selected values for unfixed mitigations.

1c: $Input = selected \cup guessed$.

1d: Call `model` to compute $score = ddp(input)$;

After 100 repeats of steps 1a,1b,1c,and 1d:

2: The 100 $score$s are divided into 10% $best$ and 90% $rest$.

3: The mitigation values in the $input$ sets are then scored using BORE (described below).

```
1. Procedure KEYS
2. while FIXED_MITIGATIONS != TOTAL_MITIGATIONS
3.   for I:=1 to 100
4.     SELECTED[1...(I-1)] = best decisions up to this step
5.     GUESSED = random settings to the remaining mitigations
6.     INPUT = SELECTED + GUESSED
7.     SCORES= SCORE(INPUT)
8.   end for
9.   for J:=1 to NUM_MITIGATIONS_TO_SET
10.    TOP_MITIGATION = BORE(SCORES)
11.    SELECTED[FIXED_MITIGATIONS++] = TOP_MITIGATION
12.  end for
13. end while
14. return SELECTED
```

Figure 5: Pseudocode for KEYS

4: The top ranked mitigation value is fixed and stored in $selected[e]$.

KEYS then moves to the era $e+1$ and repeats steps 1,2,3,4. KEYS stops when every mitigation has a setting. For full details, see Figure 5.

4.2 BORE = Best Or Rest

BORE [3] is bayesian ranking measure we use for find those most influential "key" variables. BORE assumes that the output scores are divided into two classes - one class for $best$ outcomes and one for the $rest$. In such two-class systems, BORE searches for mitigation values that have a high probability of belonging to the $best$ set.

BORE divides numeric scores seen over K runs and stores the top 10% in $best$ and the remaining 90% scores in the set $rest$. It then computes the probability that a value is found in $best$ using Bayes theorem. The theorem uses evidence E and a prior probability $P(H)$ for hypothesis $H \in \{best, rest\}$, to calculate a posteriori probability $P(H|E) = P(E|H)P(H)\ /\ P(E)$. Such simple Bayes classifiers are often called "naïve" since they assume independence of each variable. Domingos and Pazzani have shown that the independence assumption is a problem in a vanishingly small percent of cases [6]. This explains the repeated empirical result that seemingly naïve Bayes classifiers perform as well as other more sophisticated schemes (e.g. see Table 1 in [6]).

When applying the theorem, *likelihoods* are computed from observed frequencies. These likelihoods are then normalized to create probabilities (this normalization cancels out $P(E)$ in Bayes theorem). For example, after $K = 10,000$ runs are divided into 1,000 *best* solutions and 9,000 *rest*, the value $mitigation31 = false$ might appears 10 times in the $best$ solutions, but only 5 times in the $rest$. Hence:

$$\begin{aligned}
E &= (mitigation31 = false) \\
P(best) &= 1000/10000 = 0.1 \\
P(rest) &= 9000/10000 = 0.9 \\
freq(E|best) &= 10/1000 = 0.01 \\
freq(E|rest) &= 5/9000 = 0.00056 \\
like(best|E) &= freq(E|best) \cdot P(best) = 0.001 \\
like(rest|E) &= freq(E|rest) \cdot P(rest) = 0.000504 \\
P(best|E) &= \frac{like(best|E)}{like(best|E) + like(rest|E)} = 0.66 \qquad (1)
\end{aligned}$$

Previously [3], we have found that Bayes theorem is a poor ranking heuristic since it is easily distracted by low frequency evidence. For example, note how the probability of E belonging to the best class is moderately high even though its support is very low; i.e.

$P(best|E) = 0.66$ but $freq(E|best) = 0.01$.

To avoid the problem of unreliable low frequency evidence, we augment Equation 1 with a support term. Support should $increase$ as the frequency of a value $increases$, i.e. $like(best|E)$ is a valid support measure. Hence, step 3 of our greedy search raks values via

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \quad (2)$$

4.3 KEYS2

For each era, KEYS samples the DDP models and fixes the top $N = 1$ settings. $N = 1$ is, perhaps, an overly conservative search policy.

At least for the DDP models, the improvement in costs and attainments generally increase for each era of KEYS. This lead to the speculation that we could jump further and faster into the solution space by fixing $N \geq 1$ settings per era. Such a *jump* policy can be implemented as a small change to KEYS:

- Standard KEYS assigns the value of one to `NUM_MITIGATIONS_TO_SET` (see the pseudo-code of Figure 5);
- Other variants of KEYS assigns larger values.

KEYS2 sets i settings in each era i. Note that, in era 1, KEYS2 behaves exactly the same as KEYS while in (say) era 3, KEYS2 will fix the top 3 ranked ranges. Since it sets more variables at each era, KEYS2 terminates earlier than KEYS.

4.4 Areas of Improvement

KEYS2 achieves a significant improvement in speed by setting an increasing number of mitigations at a time which in turn reduces the number of model instances generated and scored. For further speed up, we could simply set more mitigations at each era, but prior experimentation showed that doing so reduces the algorithm's ability to reach more optimal solutions. This may be caused by the incremental decrease of support.

Note that in (say) era 5, KEYS2 will fix the top 5 ranked ranges. The first key set in this era is selected using BORE on 100 instances. Let us assume that this key is set to $true$, so $Key_{5-1} = true$. For each one of the 100 instances scored in era 5, there is a 50% probability that they have $Key_{5-1} = false$, rendering half of the instances unfit to rank the second key. Thus, Key_{5-2} is selected from about 50 instances. Similarly, Key_{5-3} is selected from around 25, and so on. On large models, this effect can have some performance consequences.

We therefore need to explore other areas of improvemnt that do not affect the cost and attainment performance of the algorithm. One effective way to achieve a more efficient algorithm is profiling its implementation to find bottlenecks. Running "gprof" on KEYS with our largest model shows that the algorithm spends 86% of its runtime scoring instances of the model. All other functions run either very fast or are not executed as many times.This finding prompted us to try find ways to reduce the number of calls to the model without compromising support.

A simple way to address both problems is to only call the model for the instances that become invalid after each era. This can be achieved calling a new function that searches for the roughly 50% of mitigations that become invalid for selection of the next mitigation, and a slight modification of the original code so that it only generates and scores replacement instances for the invalid ones. Such method effectively uses valid instances already scored and saves roughly 50% of the calls to the model while always having a full set of valid instances to select the next mitigation. Figure 6 displays the pseudo-code for the proposed changes.

```
1.  Procedure KEYS
2.  while FIXED_MITIGATIONS != TOTAL_MITIGATIONS
3.   for I:=1 to 100
A.    if INVALID then
4.     SELECTED[1...(I-1)] = best decisions up to this step
5.     GUESSED = random settings to the remaining mitigations
6.     INPUT = SELECTED + GUESSED
7.     SCORES= SCORE(INPUT)
B.    end if
8.   end for
9.   for J:=1 to NUM_MITIGATIONS_TO_SET
10.   TOP_MITIGATION = BORE(SCORES)
11.   SELECTED[FIXED_MITIGATIONS++] = TOP_MITIGATION
12.  end for
C.   INVALIDS = INPUT $\cup$ SELECTED
13. end while
14.return SELECTED
```

Figure 6: Pseudocode for KEYS improvement. Additional steps A. B. and C.

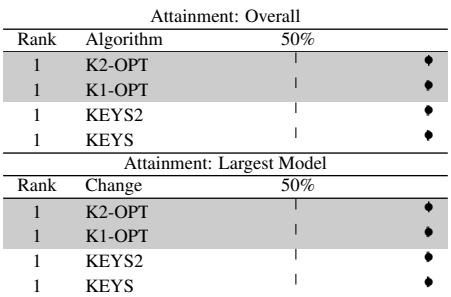

Figure 7: Attainment: Maximum attainment achieved at lowest cost (normalized 0..100%): optimizations are shaded.

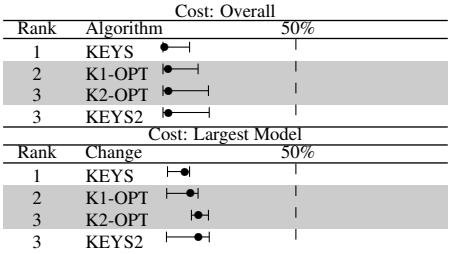

Figure 8: Cost: Minimum cost to achieve maximum attainment (normalized 0..100%): optimizations are shaded.

5. RESULTS

Our experiments with KEYS, KEYS2, and their optimizations are summarized in Figures 8, 7, and 9. In those figures:

- All results are normalized to run 0..100, 0..max, where max is calculated by running the model with all mitigations set.
- Each row shows the 25% to 75% quartile range of the normalized scores collected during the simulation.
- The median result is shown as a black dot.
- All the performance scores get *better* when the observed scores get *smaller*.

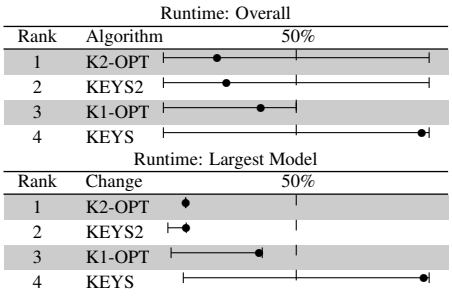

Figure 9: Runtime: Time to produce a solution (normalized 0..100%): optimizations are shaded.

- The "none" row comes from Monte Carlo simulations of the current ranges, without any changes.
- The top sumarizes the performance of the algorithms across all fifteen randomly-generated models, giving us a general indication of their performance on small and large models.
- The bottom part of each figure depicts the performance of each algorithm specific to their execution on the largest of our models, model5_16.c (see Figure 4).
- "K1-OPT" and "K2-OPT" denote the runs of the optimized version of KEYS and KEYS2 respectively.

In each figure, the rows are sorted by the number of times an algorithm scores below (*looses* to) another. In order to assess number of *losses*, we used the Mann-Whitney test at 95% confidence (this test was chosen due to (a) the random nature of Monte Carlo simulations which results in non-paired tests; and (b) ranked tests make no assumption about the normality of the results). Two rows have the same rank if there is no statistical difference in their distributions. The shaded rows in Figures 8, 7, and 9 identify our latest optimizations.

In terms of attainment, Figure Figure 7 shows that it is consistently maximized across all algorithms with no significant differences among them. They all achieve (very close to) the maximum possible attainment indicating that our new version of the algorithms do not affect the performance on this aspect. It is important to note that achieving maximum requirements attainment simplifies our analysis because it reduces the dimensionality of the resuls such that we can concentrate on the other two aspects of this study, cost and runtime reduction.

The first differences among algorithms start to appear when we look at cost reduction. Not surprisingly, Figure 8 shows the original KEYS as the best algorithm to reduce costs. What strikes us a surprising is that our K1-OPT has a slightly lower performance than KEYS given that both select their *keys* based on the same number of simulations. We expected to see a tie similar to the performances of KEYS2 and K2-OPT. They perform, on average, very close to each other, but K2-OPT is more stable (lower variance). This phenomenon may be explained by the fact that at each era K2-OPT is selecting *keys* from half of the same instances from which the previous *key* was selected. That might also explain the slim decrease in average cost minimization performance by K1-OPT, but it needs further study.

The third and main aspect of our study is the runtimes. The main goal of our study is the reduction of the optimization runtimes so that results can be obtained in near real-time. Figure 9 shows results very close to our expectations. Overall, applying our optimization improving technique on both KEYS and KEYS2 for the

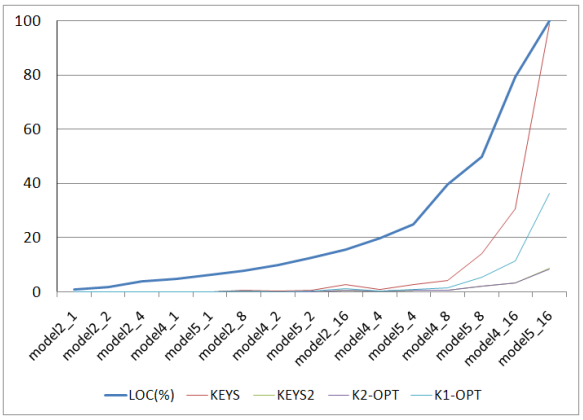

Figure 10: Runtime (secs) Vs. model size increase (LOC%)

largest model results in shorter runtimes. In the case of KEYS, K1-OPT reduces the average execution time to 36.5 seconds from 99 seconds. A 63% runtime speedup. K2-OPT also improves KEYS2 runtime on the largest model but not as much. KEYS2 takes on average 8.7 seconds while K2-OPT takes 8.5 seconds. A speedup of 2.3%.

A more clear view of the speed gains achieved by our improvement technique can bee seen in Figure 10. Note the big difference in the KEYS and K1-OPT slopes. Following the trend, the bigger the model the more KEYS beneffits from our improvement. The same applies to KEYS2 and K2-OPT, but at the model sizes we are running our experiments it is barely apparent.

Although our technique's gain for KEYS2 is not as large as it is for KEYS, it is statistically significant at 95% confidence level and we believe it increases as the models become even larger. Furthermore, applying our simple speed up technique is easy and maintains the performance of the other aspects of the model-optimization algorithms within acceptable margins.

6. CONCLUSION

In this paper we evaluate and analyze the performance of the KEYS and KEYS2 model optimization algorithms using artificially-grown models. We then identify and address their deficiencies and inefficiencies and attempt to speed up their runtimes so optimal solutions for incrementally larger models can be obtained near real-time.

Our analysis identifies the execution of the model as the main bottleneck in the efficiency of the algorithms. KEYS spends 86% of the runtime, running model simulations. Our modified versions attempt to reduce the number of times the model is called without significantly affecting the ability to optimize the search for a solution. The technique employed to achieve that reduction consists of maintaining valid model instances between eras and generating and executing new instances only to replace the invalid ones.

The results show that this fairly simple technique significantly reduces the runtimes at a slight decrease in cost minimization. The time reduction is particulaly large (more than 50%) for the KEYS algorithm.

7. FUTURE WORK

We believe that time savings benefit of our improvemnt technique increases as the models become larger. As part of the future work we would like to perform of the relevant experimentation with

even bigger models.

Another aspect worth exploring is multi-threading. In theory this technique is promising for large to huge models due to the fact that KEYS bottleneck lies in calling the model scoring function millions of times but these calls can be done in paralel at every era. Early experimentation shows a slightly better runtimes when using available cores in a multi-processor, multi-core server environment. Cloud computing is the logical next step for our future experimentation.

8. REFERENCES

[1] A. Biere, A. Cimatti, E. M. Clarke, and Y. Zhu. Symbolic model checking without bdds. In *In Proceedings of Tools and Algorithms for the Analysis and Construction of Systems*, page 193207, May 1999.

[2] A. Brown, S. Iyengar, and S. Johnston. A rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.

[3] R. Clark. Faster treatment learning. Master's thesis, Computer Science, Portland State University, 2005.

[4] S. L. Cornford. Managing risk as a resource using the defect detection and prevention process. In *Internationa Conference on Probabilistic Safety Assessment and Management*, September 1998.

[5] A. Darwiche and P. Marquis. A knowledge compulation map. *Journal of Artifical Intelligence Research*, 17:229–264, 2002. Available from `ww.jair.org/media/989/live-989-2063-jair.pdf`.

[6] P. Domingos and M. J. Pazzani. On the optimality of the simple bayesian classifier under zero-one loss. *Machine Learning*, 29(2-3):103–130, 1997.

[7] M. Feather, S. Cornford, K. Hicks, J. Kiper, and T. Menzies. Application of a broad-spectrum quantitative requirements model to early-lifecycle decision making. *IEEE Software*, 2008. Available from `http://menzies.us/pdf/08ddp.pdf`.

[8] 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`.

[9] J. Greenfield and K. Short. *Software factories : assembling applications with patterns, models, frameworks, and tools*. Wiley Publishing, Indianapolis, IN, 2004.

[10] O. Jalali, T. Menzies, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings of the PROMISE 2008 Workshop (ICSE)*, 2008. Available from `http://menzies.us/pdf/08keys.pdf`.

[11] T. Menzies, D.Owen, and J. Richardson. The strangest thing about software. *IEEE Computer*, 2007. `http://menzies.us/pdf/07strange.pdf`.

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

[13] B. Selman and H. Kautz. Knowledge compilation and theory approximation. *Journal of the ACM*, 43(2):193–224, 1996. Available from `http://citeseer.nj.nec.com/article/selman96knowledge.html`.

[14] D. Waddington and P. Lardieri. Model-centric software development. *IEEE Computer*, 39(2):28–29, February. 2006.