Finding Robust Solutions in Requirements Models

Gregory Gay[1], Tim Menzies[1], Omid Jalali[1], Gregory Mundy[2],
Beau Gilkerson[1], Martin Feather[3], and James Kiper[4]

[1] West Virginia University, Morgantown, WV, USA
[2] Alderson-Broaddus College, Philippi, WV
[3] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA
[4] Dept. of Computer Science and Systems Analysis, Miami University, Oxford, OH, USA

greg@greggay.com, tim@menzies.us, jalali.omid@gmail.com,
beau.gilkerson@gmail.com, mundyge@ab.edu,
martin.s.feather@jpl.nasa.gov, kiperjd@muohio.edu

Abstract. Solutions to non-linear requirements engineering problems may be "brittle"; i.e. small changes may dramatically alter solution effectiveness. Therefore, it is not enough to merely generate solutions to requirements problems- we must also assess the robustness of that solution. This paper introduces the KEYS2 algorithm that can generate *decision ordering diagrams*. Once generated, these diagrams can assess solution robustness in linear time. In experiments with real-world requirements engineering models, we show that KEYS2 can generate decision ordering diagrams in $O(N^2)$. When assessed in terms of terms of (a) reducing inference times, (b) increasing solution quality, and (c) decreasing the variance of the generated solution, KEYS2 out-performs other search algorithms (simulated annealing, ASTAR, MaxWalkSat).

1 Introduction

Consider a "requirements model" where stakeholders write:

- Their various goals;
- Their different methods to reach those goals;
- Their view of the possible risks that compromise those goals;
- What mitigations they believe might reduce those risks.

We say that a "solution" to such a model is the the *least* cost combination of mitigations that reduce the *most* risks, thereby enabling the *most* number of requirements. In theory, software tools can find the solution that best exploits and most satisfies the various goals of our different stakeholders. Such software might find good solutions that were missed by stakeholders in early lifecycle design discussions. Finding solutions to these requirements models is a non-linear optimization problem (*minimize* the sum of the mitigation costs while *maximizing* the number of achieved requirements).

There are many heuristic methods that can generate solutions to non-linear problems (see *Related Work*, below). However, such heuristic methods can be *brittle*; i.e. small changes may dramatically alter the effectiveness of the solution. Therefore, it important to understand the *neighborhood around the solution*. A naive approach to understanding the neighborhood might be to run a system N times then report:

– the solutions appearing in more than (say) $\frac{N}{2}$ cases;
– results with a $\pm 95\%$ confidence interval.

Note that both these approaches requires multiple runs of an analysis method. Multiple runs are undesirable since, in our experience [20], stakeholders often ask questions across a range of "scenarios"; i.e. hard-wired constraints that cannot be changed in that scenario. For example, three scenarios might be "what can be achieved assuming a maximum budget of one, two, or five million dollars?". Scenario analysis can be time consuming. Reflecting over (say) $d = 10$ possible decisions a statistically significant number of times (e.g. $N = 20$) requires up to $20 * 2^{10} > 20,000$ repeats of the analysis.

Therefore, this paper proposes a rapid method for exploring decision neighborhoods. *Decision ordering diagrams* are a visual representation of the effects of changing a solution. We show below that:

– Using these diagrams, the region around a solution can be explored in linear time.
– A greedy Bayesian-based method called KEYS2 can generate the decision ordering diagrams in $O(N^2)$ time.
– KEYS2 yields solutions of higher quality that several other methods (simulated annealing, MaxWalkSat, ASTAR).
– Also, the variance of the solutions found by KEYS2 is less (and hence, better) than those found by the other methods.

This paper is structured as follows:

– After some background notes on the solution robustness, we describe the *DDP* requirements engineering tool used at NASA's Jet Propulsion Laboratory (the case studies for this paper come from real-world early lifecycle DDP models);
– DDP inputs and outputs are then reviewed;
– Next, decision ordering diagrams are introduced;
– We then define and compare five different algorithms for generating solutions from DDP models: KEYS, KEYS2, Simulated Annealing, MaxWalkSat, ASTAR. Experimental results will show that KEYS2 out-performs the other methods (measured in terms of quickly generating high quality solutions that allow us to reflect over solution robustness).
– Finally, we offer some notes on related work and conclusions.

This paper extends a prior publication [39] in several ways:

– That paper did not address concerns of solution robustness.
– That paper did not explore a range of alternate algorithms.
– This paper introduces KEYS2, which is an improved version of KEYS,.
– This paper offers extensive notes on related work.

2 Background

According to Harman [30], understanding the neighborhood of solutions is an open and pressing issue in search-based software engineering (SBSE). He argues that many

software engineering problems are over-constrained and no precise solution over all variables is achievable; therefore partial solutions based on heuristic search methods are preferred. *Solution robustness* is a major problem for such partial heuristic searches. The results of such partial heuristic search may be "brittle"; i.e. small changes to the search results may dramatically alter the effectiveness of the solution [31].

When offering partial solutions, it is very important to also offer insight into the space of options around the proposed solution. Such neighborhood information is very useful for managers with only partial control over their projects since it can give them confidence that even if only some of their recommendations are effected, then at least the range of outcomes is well understood. Harman [30] comments that understanding the neighborhood of our solutions is an open and pressing issue in search-based software engineering (SBSE):

> "In some software engineering applications, solution robustness may be as important as solution functionality. For example, it may be better to locate an area of the search space that is rich in fit solutions, rather than identifying an even better solution that is surrounded by a set of far less fit solutions."

> "Hitherto, research on SBSE has tended to focus on the production of the fittest possible results. However, many application areas require solutions in a search space that may be subject to change. This makes robustness a natural second order property to which the research community could and should turn its attention [30]."

This paper reports a set of experiments on AI search for robust solutions. Our experiments address two important concerns. Firstly, *is demonstrating solution robustness a time consuming task*? Secondly, is it necessary, as Harman suggests that *solution quality must be traded off against solution robustness*? That is, in our search for the conclusions that were stable within their local neighborhood, would we have to reject better solutions because they are less stable across their local neighborhood?

At least for the NASA models described in the next section, both these concerns are unfounded. KEYS2 terminates in hundreths of a second (where as our prior implementations took minutes to terminate [22]). Also, the solutions found by KEYS2 where not only of highest quality, they were also exhibited the lowest variance. Further, KEYS2 generates the decision ordering diagrams that can assess solution robustness in linear time.

3 Requirements Modeling Using "DDP"

This section introduces the DDP requirements modeling tool [15, 20]. used to interactively document the "Team-X" early lifecycle meetings at NASA's Jet Propulsion Laboratory (JPL). These meetings are the source of the real-world requirements models used in this paper.

At "Team X" meetings, a large and diverse group of up to 30 experts from various fields (propulsion, engineering, communication, navigation, science, etc) meet for very short periods of time (hours to days) to produce a "mission concept" document. This

1. Requirement goals:
   - Spacecraft ground-based testing & flight problem monitoring
   - Spacecraft experiments with on-board Intelligent Systems Health Management (ISHM)
2. Risks:
   - Obstacles to spacecraft ground-based testing & flight problem monitoring
     - Customer has no, or insufficient, money available for our use
     - Difficulty of building the models / design tools
   - ISHM Experiment is a failure (without necessarily causing flight failure)
   - Usability, User/Recipient-system interfaces undefined
   - V&V (certification path) untried and scope unknown
   - Obstacles to Spacecraft experiments with on-board ISHM
     - Bug tracking / fixes / configuration management issues, Managing revisions and upgrades (multi-center tech. development issue)
     - Concern about our technology interfering with in-flight mission
3. Mitigations:
   - Mission-specific actions
     - Spacecraft ground-based testing & flight problem monitoring
     - Become a team member on the operations team
     - Use Bugzilla and CVS
   - Spacecraft experiments with on-board ISHM
     - Become a team member on the operations team
     - Utilize xyz's experience and guidance with certification of his technology

Fig. 1. Sample DDP requirements, risks, mitigations.

document may commit the project to, say, solar power rather nuclear power or a particular style of guidance software. All of the subsequent work is based on the initial decisions documented in the mission concept.

DDP allows for the representation of the goals, risks, and risk-removing mitigations that belong to a specific project. During a Team X meeting, users of DDP explore combinations of mitigations that cost the least and support the most number of requirements. DDP propagate influences over matrices. For example, here is a trivial DDP model where `mitigation1` costs $10,000 to apply and each requirement is of equal value (100). Note that the mitigation can remove 90% of the risk. Also, unless mitigated, the risk will disable 10% to 99% of requirements one and two (respectively):

$$\overbrace{mitigation1}^{\$10{,}000} \underbrace{\rightarrow}_{0.9} risk1 \rightarrow \left\langle \begin{array}{l} \overbrace{\rightarrow}^{0.1}(requirement1 = 100) \\ \underbrace{\rightarrow}_{0.99}(requirement2 = 100) \end{array} \right. \quad (1)$$

The other numbers show the impact of mitigations on risks, and the impact of risks on requirements. The core of DDP is two matrices: one for *mitigations*risks* and another for *risks*requirements*.

DDP is used as follows. A dozen experts, or more, gather 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, time is crucial and no tool should slow the debate. Therefore, DDP uses a lightweight representations for its model. Such representations are essential for early lifecycle decision making since only high-level 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). Therefore, DDP uses the following lightweight ontology:

- *Requirements* (free text) describing the objectives and constraints of the mission and its development process;
- *Weights* (numbers) of each 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.
- *Part-of relations* structure the collections of requirements, risks and mitigations;

Note that DDP models are the same as the "requirements models" we defined in the introduction. For examples of risks, requirements, and mitigations, see Figure 1. For an example of the network of connections between risks and requirements and mitigations, see Figure 2.

Sometimes, we are asked if the analysis of DDP requirements models is a real problem. The usual question is something like: "With these ultra-lightweight languages, aren't all open issues just obvious?". Such a question is usually informed by the small model fragments that appear in the ultra-lightweight modeling literature. Those sample model fragments are typically selected according to their ability to fit on a page or to succinctly illustrate some point of the authors. Real world ultra-lightweight models can be much more complex, paradoxically perhaps due to their simplicity: if a model is easy to write then it is easy to write a lot of it. Figure 2, for example, was generated in under a week by four people discussing one project. It is complex and densely-connected (a close inspection of the left and right hand sides of Figure 2 reveals the requirements and fault trees that inter-connect concepts in this model) and it is, by no means, the biggest or most complex DDP model that has ever been built.

We base our experimentation around DDP for three reasons. Firstly, one potential drawback with ultra-lightweight models is that they are excessively lightweight and contain no useful information. DDP's models are demonstrably useful (that is, we are optimizing a real-world problem of some value). Clear project improvements have been seen from DDP sessions at JPL. Cost savings in at least two sessions have exceeded $1 million, while savings of over $100,000 have resulted in others [20]. Cost savings are not the only benefits of these DDP sessions. Numerous design improvements such as savings of power or mass have come out of DDP sessions. Likewise, a shifting of risks has been seen from uncertain architectural ones to more predictable and manageable ones. At some of these meetings, non-obvious significant risks have been identified and subsequently mitigated.

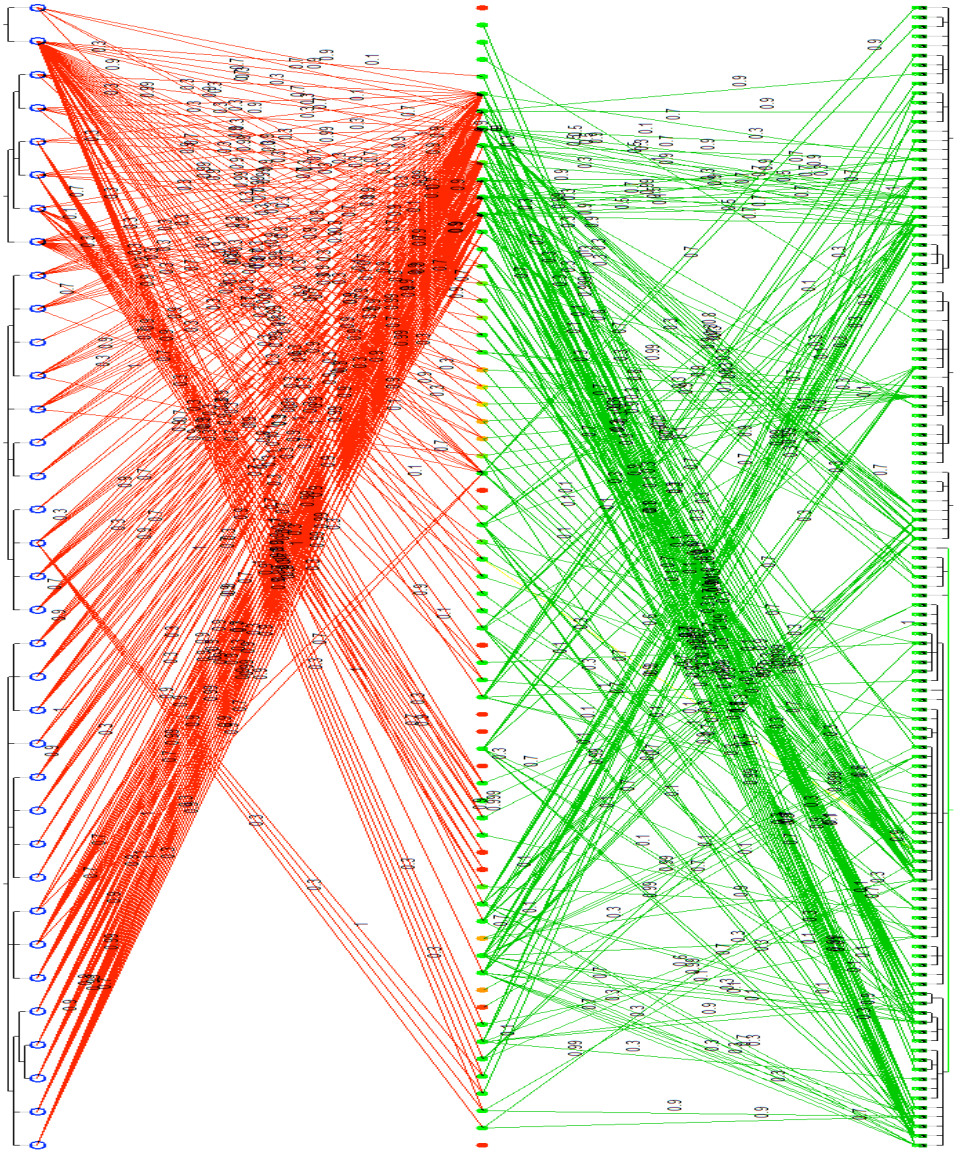

Fig. 2. An example of a model formed by the DDP tool. Red lines connect risks (middle) to requirements (left). Green lines connect mitigations (right) to the risks.

Our second reason to use DDP is that we can access numerous real-world requirements models written in this format, both now and in the future. The DDP tool can be used to document not just final decisions but also to review the rationale that led to those decisions. Hence, DDP remains in use at JPL: not only for its original purpose (group decision support), but also as a design rationale tool to document decisions. Recent DDP sessions included:

- An identification of the challenges of intelligent systems health management (ISHM) technology maturation (to determine the most cost-effective approach to achieving maturation) [23];
- A study on the selection and planning of deployment of prototype software [21].

Our third, and most important reason to use DDP in our research is that the tool is representative of other requirements modeling tools in widespread use. At its core, DDP is a set of influences expressed in a hierarchy, augmented with the occasional equation. Edges in the hierarchy have weights that strengthen or weaken influences that flow along those edges. At this level of abstraction, DDP is just another form of QOC [64] or a quantitative variant of Mylopoulos' s qualitative soft goal graphs [54].

4 Model Inputs and Outputs

Before describing experimental comparisons of different methods for generating decision ordering diagrams, we will first offer more details on the DDP models.

4.1 Pre-Processing

To enable fast runtimes, a simple compiler exports the DDP models into a form accessible by our algorithms. This compiler stores a flattened form of the DDP requirements tree. In our compiled form, all computations are performed once and added as a constant to each reference of the requirement. For example, the compiler converts the trivial model of Equation 1 into `setupModel` and `model` functions similar to those in Figure 3. The `setupModel` function is called only 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 grow more complex, so do these equations. For example, our biggest model, which contains 99 mitigations, generates 1427 lines of code. Figure 4 compares the largest model to four other real-world DDP models.

Currently it takes about two seconds to compile a model with 50 requirements, 31 risks, and 58 mitigations. This compilation only has to happen once, after which we will run our $2^{|d|}$ what-if scenarios. While this is not a significant bottleneck, the current compiler (written in unoptimized Visual Basic code) can certainly be sped up. Experts usually change a small portion of the model then run $2^{|d|}$ what-if scenarios to understand the impact of that change. Therefore, an incremental compiler (that only updates changed portions) would run much faster than a full compilation of the entire DDP model.

```
#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;
}
```

Fig. 3. A trivial DDP model after knowledge compilation

Model	LOC	Objectives	Risks	Mitigations
model1.c	55	3	2	2
model2.c	272	1	30	31
model3.c	72	3	2	3
model4.c	1241	50	31	58
model5.c	1427	32	70	99

Fig. 4. Details of Five DDP Models.

4.2 Objective Function

When the `model` function is called, a pairing of the total cost of the selected mitigations and the number of reachable requirements (attainment) is returned. All of our algorithms then use that information to obtain a "score" for the current set of mitigations. The two numbers are normalized to a single score that represents the distance to a *sweet spot* of maximum requirement attainment and minimum cost:

$$score = \sqrt{\overline{cost}^2 + (\overline{attainment} - 1)^2} \tag{2}$$

Here, $\overline{x}$ is a normalized value $0 \leq \frac{x - min(x)}{max(x) - min(x)} \leq 1$. Hence, our scores ranges $0 \leq score \leq 1$ and *higher* scores are *better*.

4.3 Decision Ordering Diagrams

The objective function described above summarizes *one* call to a DDP model. This section describes *decision ordering diagrams*, which are a tool for summarizing the results of thousands of calls to DDP models.

Consider some recommendation for changes to a project that requires decisions d of size $|d|$. In the general case, d is a subset of the space of all solutions D ($d \subseteq D$). When checking for solution robustness, or reflecting over modifications to d, a stakeholder may need to consider up to $d' \subseteq N^{|d|}$ possibilities (and $N = 2$ for binary decisions of the form "should I or should I not do this"). This can be a slow process, especially if evaluating each decision requires invoking a complex and slow simulator.

Decision ordering diagrams are a linear time method for studying the robustness and neighborhood of a set of decisions. The diagrams assume that some method could offer a *linear ordering* of the decisions $x \in d$ ranked from *most-important* to *least-important*. They also assume that some method offers information on the effects of applying the top-ranked $1 \leq x \leq |d|$ decisions (e.g. the median and variance seen in the model's objective function after applying solution $\{d_1..d_x\}$). For example, the *decision ordering diagram* of Figure 5 shows such a linear ordering (this figure presents $benefit$ and $cost$ results). In that figure:

- The x-axis denotes the number of decisions made.
- The y-axis shows performance statistics of an objective function seen after imposing the conjunction of decisions $1 \leq i \leq x$.

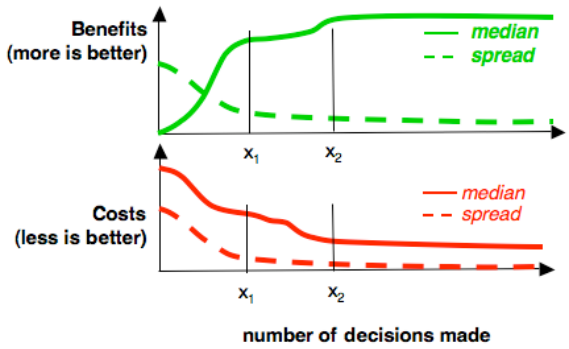

Fig. 5. A Decision Ordering Diagram. The median and spread plots show 50%-the percentile and the (75-50)%-th percentile range (respectively) values generated from some objective function.

For performance, we run some objective function and report the median (50th percentile) and *spread* (the range given by the 75th percentile - the 50th percentile). We use median and spread to avoid any parametric assumptions.

These diagrams can comment on the robustness and neighborhood of solution $\{d_1..d_x\}$ as follows:

- By considering the variance of the performance statistics after applying $\{d_1..d_x\}$.
- By comparing the results of using the first x decisions to that of using the first $x-1$ or $x+1$ actions.

The neighborhood of a solution that uses decisions $\{d_1..d_x\}$ are solutions that use the decisions $\{d_1..d_{x\pm j}\}$. Since j is bounded $0 \leq |d| - 1$, this means that *reflecting over solution neighborhoods takes time linear on the number of decisions* d.

Decision ordering diagrams are a natural representation for "trade studies," the activity of a multidisciplinary team to identify the most balanced technical solution among a set of proposed viable solutions [2]. For example, minimum costs and maximum benefits are achieved at point x_2 of Figure 5. However, after applying only half the decisions (see x_1) most of the benefits could be achieved, albeit at a somewhat higher cost.

Decision ordering diagrams are *useful* under at least three conditions:

- The scores output by the objective functions are *well-behaved*; i.e. move smoothly to a plateau.
- The decisions *tame* the variance; i.e. the spread falls to value much lower than then median (otherwise, it is hard to show that decisions have any effect on the system performance).
- The are generated in a *timely* manner. Fast runtimes are required in order to keep up with fast moving discussion.

According to these definitions, Figure 5 is a *useful* decision ordering diagram if it can be generated in a *timely* manner.

It is an open issue if real worlds requirements models generate useful decision ordering diagrams. The following experiments test if, in practice, decision ordering diagrams generated from real world requirements models are *timely* to generate while being *well-behaved* and *tame*.

5 Searching for Solutions

Our experiments compare the results of numerous algorithms. We selected these comparison algorithms with much care. Numerous researchers have stressed the difficulties associated with comparing radically different algorithms. For example, Uribe and Stickel [67] tried to make some general statement about the value of Davis-Putnam proof (DP) procedures or binary-decision diagrams (BDD) for constraint satisfaction problems. In the end, they doubted that it was fair to compare algorithms that perform constraint satisfaction and no search (like BDDs) and methods that perform search and no constraint satisfaction (like DP). For this reason, model checking researchers like Holzmann (pers. communication) eschew comparisons of tools like SPIN [35], which are search-based, with tools like NuSMV [11], which are BDD-based. Hence we take care to only select algorithms which are similar to KEYS.

In terms of the Gu et al. survey [27], our selected algorithms (simulated annealing, ASTAR and MaxWalkSat) share four properties with KEYS and KEYS2. They are each discrete, sequential, *unconstrained* algorithms (constrained algorithms work towards a pre-determined number of possible solutions while unconstrained methods are allowed to adjust to the goal space).

For full details on simulated annealing, ASTAR, and MaxFunWalk, see below. We observe that these algorithms share the property that at each step of their processing, they comment on all model inputs. KEYS2, on the other hand, explores the consequences of setting only a subset of the possible inputs.

5.1 SA: Simulated Annealing

Simulated Annealing is a classic stochastic search algorithm. It was first described in 1953 [51] and refined in 1983 [43]. The algorithm's namesake, annealing, is a technique from metallurgy, where a material is heated, then cooled. The heat causes the atoms in the material to wander randomly through different energy states and the cooling process increases the chances of finding a state with a lower energy than the starting position.

For each round, SA "picks" a neighboring set of mitigations. To calculate this neighbor, a function traverses the mitigation settings of the current state and randomly flips those mitigations (at a 5% chance). If the neighbor has a better score, SA will move to it and set it as the current state. If it isn't better, the algorithm will decide whether to move to it based on the mathematical function:

$$prob(w, x, y, temp(y, z)) = e^{((w-x)*\frac{y}{temp(y,z)})} \tag{3}$$

$$temp(y, z) = \frac{(z - y)}{z} \tag{4}$$

If the value of the `prob` function is greater than a randomly generated number, SA will move to that state anyways. This randomness is the very cornerstone of the Simulated

```
1. Procedure SA
2. MITIGATIONS:= set of mitigations
3. SCORE:= score of MITIGATIONS
4. while TIME < MAX_TIME && SCORE < MIN_SCORE //minScore is a constant score (threshold)
5.    find a NEIGHBOR close to MITIGATIONS
6.    NEIGHBOR_SCORE:= score of NEIGHBOR
7.    if NEIGHBOR_SCORE > SCORE
8.        MITIGATIONS:= NEIGHBOR
9.        SCORE:= NEIGHBOR_SCORE
10.   else if prob(SCORE, NEIGHBOR_SCORE, TIME, temp(TIME, MAX_TIME)) > RANDOM)
11.       MITIGATIONS:= NEIGHBOR
12.       SCORE:= NEIGHBOR_SCORE
13.   TIME++
14. end while
15. return MITIGATIONS
```

Fig. 6. Pseudocode for SA

Annealing algorithm. Initially, the "atoms" (current solutions) will take large random jumps, sometimes to even sub-optimal new solutions. These random jumps allow simulated annealing to sample a large part of the search space, while avoiding being trapped in local minima. Eventually, the "atoms" will cool and stabilize and the search will converge to a simple hill climber.

As shown in line 4 of Figure 6, the algorithm will continue to operate until the number of tries is exhausted or a score meets the threshold requirement.

5.2 MaxFunWalk

The design of simulated annealing dates back to the 1950s. In order to benchmark our own search engine (KEYS2) against a more state-of-the-art algorithm, we implemented the variant of MaxWalkSat described below.

WalkSat is a local search method designed to address the problem of boolean satisfiability [42]. MaxWalkSat is a variant of that algorithm that applies weights to each clause in a conjunctive normal form equation [62]. While WalkSat tries to satisfy the entire set of clauses, MaxWalkSat tries to maximize the sum of the weights of the satisfied clauses.

In one respect, both algorithms can be viewed as a variant of simulated annealing. Whereas simulated annealing always selects the next solution randomly, the WalkSat algorithms will *sometimes* perform random selection while, other times, conduct a local search to find the next best setting to one variable.

MaxFunWalk is a generalization of MaxWalkSat:

- MaxWalkSat is defined over CNF formulae. The success of a collection of variable settings is determined by how many clauses are "satisfiable" (defined using standard boolean truth tables).
- MaxWalkFun, on the other hand, assumes that there exist an arbitrary function that can assess a collection of variable settings. Here, we use the DDP model as a assessment function.

```
1. Procedure MaxFunWalk
2. for TRIES:=1 to MAX-TRIES
3.    SELECTION:=A randomly generate assignment of mitigations
4.    for CHANGED:=1 to MAX-CHANGES
5.        if SCORE satisfies THRESHOLD return
6.        CHOSEN:= a random selection of mitigations from SELECTION
7.        with probability P
8.            flip a random setting in CHOSEN
9.        with probability (P-1)
10.           flip a setting in CHOSEN that maximizes SCORE
11.    end for
12. end for
13. return BESTSCORE
```

Fig. 7. Pseudocode for MaxFunWalk

Note that $MaxWalkFun = MaxWalkSat$ if the assessment is conducted via a logical truth table.

The MaxFunWalk procedure is shown in Figure 7. When run, the user supplies an ideal cost and attainment. This setting is normalized, scored, and set as a goal threshold. If the current setting of mitigations satisfies that threshold, the algorithm terminates.

MaxFunWalk begins by randomly setting every mitigation. From there, it will attempt to make a *single* change until the threshold is met or the allowed number of changes runs out (100 by default). A random subset of mitigations is chosen and a random number P between 0 and 1 is generated. The value of P will decide the form that the change takes:

- P$\leq \alpha$: A stochastic decision is made. A setting is changed completely at random within the set CHOSEN.
- P$> \alpha$: Local search is utilized. Each mitigation in CHOSEN is tested until one is found that improves the current score.

The best setting of α is domain-specific. For this study, we used $\alpha = 0.3$.

If the threshold is not met by the time that the allowed number of changes is exhausted, the set of mitigations is completely reset and the algorithm starts over. This measure allows the algorithm to avoid becoming trapped in local maxima. For the DDP models, we found that the number of retries has little effect on solution quality.

If the threshold is never met, MaxFunWalk will reset and continue to make changes until the maximum number of allowed resets is exhausted. At that point, it will return the best settings found.

As an additional measure to improve the results found by MaxFunWalk, a heuristic was implemented to limit the number of mitigations that could be set at one time. If too many are set, the algorithm will turn off a few in an effort to bring the cost factor down while minimizing the effect on the attainment.

5.3 A* (ASTAR)

A* is a best-first path finding algorithm that uses distance from origin (G) and estimated cost to goal (H) to find the best path [33]. The algorithm is widely used [36, 57, 59, 66].

A* is a natural choice for DDP optimization since the objective function described above is actually a Euclidean distance measure to the desired goal of maximum attainment and minimum costs. Hence, for the second potion of the ASTAR heuristic, we can make direct use of Equation 2.

The ASTAR algorithm keeps a *closed* list in order to prevent backtracking. We begin by adding the starting state to the closed list. In each "round," a list of neighbors is populated from the series of possible states reached by making a change to a single mitigation. If that neighbor is not on the closed list, two calculations are made:

- G = Distance from the start to the current state plus the additional distance between the current state and that neighbor.
- H = Distance from that neighbor to the goal (an ideal spot, usually 0 cost and a high attainment). For DDP models, we use Equation 2 to compute H.

The *best* neighbor is the one with the lowest F=G+H. The algorithm "travels" to that neighbor and adds it to the closed list. Part of the optimality of the A* algorithm is that the distance to the goal is underestimated. Thus, the final goal is never actually reached by ASTAR. Our implementation terminates once it stops finding better solutions for a total of ten rounds. This number was chosen to give ample time for ASTAR to become "unstuck" if it hits a corner early on.

```
1. Procedure ASTAR
2. CURRENT_POSITION:= Starting assignment of mitigations
3. CLOSED[0]:= Add starting position to closed list
4.
5. while END:= false
6.    NEIGHBOR_LIST:=list of neighbors
7.    for each NEIGHBOR in NEIGHBOR_LIST
8.        if NEIGHBOR is not in CLOSED
9.            G:=distance from start
10.           H:=distance to goal
11.           F:=G+H
12.           if F<BEST_F
13.               BEST_NEIGHBOR:=NEIGHBOR
14.   end for
15.   CURRENT_POSITION:= BEST_NEIGHBOR
16.   CLOSED[++]:=Add new state to closed list
17.   if STUCK
18.       END:= true
19. end while
20. return CURRENT_POSITION
```

Fig. 8. Pseudocode for ASTAR

5.4 KEYS and KEYS2

The core premise of KEYS and KEYS2 is that the above algorithms perform over-elaborate searches. Suppose that the behavior of a large system is determined by a small number of *key* variables. If so, then a very rapid search for solutions can be found by (a) finding these *keys* then (b) explore the ranges of the key variables.

As documented in our *Related Work* section, this notion of $keys$ has been discovered and rediscovered many times by many researchers. Historically, finding the keys has seen to be a very hard task. For example, finding the keys is analogous to finding the *minimal environments* of DeKleer' ATMS algorithm [17]. Formally, this *logical abduction*, which is an NP-hard task [8].

Our method for finding the keys uses a Bayesian sampling method. If a model contains keys then, by definition, those variables must appear in all solutions to that model. If model outputs are scored by some oracle, then the key variables are those with ranges that occur with very different frequencies in high/low scored model outputs. Therefore, we need not search for the keys- rather, we just need to keep frequency counts on how often ranges appear in $best$ or $rest$ outputs.

KEYS contains an implementation of this Bayesian sampling method. It has two main components - a greedy search and the BORE ranking heuristic. The greedy search explores 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 $M_i = X_j$, $X_j \in \{true, false\}$. In the original version of KEYS [48], the greedy search fixes one variable per era. A newer variant, KEYS2, fixes an increasing number of variables as the search progresses (see below for details).

In KEYS (and KEYS2), each era e generates a set $< input, score >$ as follows:

1: $MaxTries$ times repeat:
   - $Selected[1 . . . (e-1)]$ are settings from previous eras.
   - $Guessed$ are randomly selected values for unfixed mitigations.
   - $Input = selected \cup guessed$.
   - Call `model` to compute $score = ddp(input)$;
2: The $MaxTries$ $score$s are divided into β% "best" and remainder become "rest".
3: The $input$ mitigation values are then scored using BORE (described below).
4: The top ranked mitigations (the default is one, but the user may fix multiple mitigations at once) are fixed and stored in $selected[e]$.

The search moves to era $e + 1$ and repeats steps 1,2,3,4. This process stops when every mitigation has a setting. The exact settings for $MaxTries$ and β must be set via engineering judgment. After some experimentation, we used $MaxTries = 100$ and $\beta = 10$. For full details, see Figure 9.

KEYS ranks mitigations using a support-based Bayesian ranking measure called BORE. BORE [12] (short for "best or rest") divides numeric scores seen over K runs and stores the top 10% in $best$ and the remaining 90% scores in the set $rest$ (the $best$ set is computed by studying the delta of each score to the best score seen in any era). 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 $P(H|E) = P(E|H)P(H)\ /\ P(E)$. When applying

```
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
```

Fig. 9. Pseudocode for KEYS

the theorem, *likelihoods* are computed from observed frequencies. These likelihoods (called "like" below) 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 appear 10 times in the $best$ solutions, but only 5 times in the $rest$. Hence:

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

Previously [12], 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 5 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 ranks values via

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \tag{6}$$

For each era, KEYS samples the DDP models and fixes the top $N = 1$ settings. KEYS2 assigns progressively larger values. 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.

Note that decision ordering diagrams could be directly generated during execution, just by collection statistics from the `SCORES` array used in line 7 of Figure 9.

6 Results

Each of the above algorithms was tested on the five models of Figure 4. Note that:

- Models one and three are trivially small. They were used them to debug our code, but not in the core experiments. We report our results using models two, four and five since they are large enough to stress test real-time optimization.
- Model 4 was discussed in [49] in detail. The largest, model 5 was optimized previously in [22]. At that time (2002), it took 300 seconds to generate solutions using our old, very slow, rule learning method.

We also studied how well KEYS and KEYS2 scale to larger models. Further, we instrumented KEYS and KEYS2 to generate decision ordering diagrams. The results from all of these experiments are shown below.

6.1 Attainment and Costs

We ran all of our algorithms 1000 times on each model. This number was chosen because it yielded enough data points to give a clear picture of the span of results. At the same time, it is a low enough number that we can generate a set of results in a fairly short time span.

The results are pictured in Figure 10. Attainment is along the x-axis and cost (in thousands) is along the y-axis. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also, better solutions exhibit less variance; that is, the results are clumped closely together.

These graphs give a clear picture of the results obtained by our various algorithms. Two methods are clearly inferior:

- Simulated annealing exhibits the worst variance, lowest attainments, and highest costs.
- MaxFunWalk is better than SA (less variance, lower costs, higher attainment) but its variance is still far too high to use in any critical situation.

As to the others:

- On larger models such as model4 and model5, KEYS and KEYS2 exhibit lower variance, lower costs, and higher attainments than ASTAR.
- On smaller models such as model2, ASTAR usually produces higher attainments and lower variance than the KEYS algorithms (this advantage disappears on the larger models). However, observe the results near the $(0, 0)$ point of model2's ASTAR results: sometimes ASTAR's heuristic search failed completely for that model

6.2 Runtime Analysis

Measured in terms of attainment and cost, there is little difference between KEYS and KEYS2. However, as shown by Figure 11, KEYS2 runs twice to three times as fast as its predecessor. Interestingly, Figure 11 ranks two of the algorithms in a similar order to Figure 10:

- Simulated annealing is clearly the slowest;
- MaxFunWalk is somewhat better but not as fast as the other algorithms.

As to ASTAR versus KEYS or KEYS2:

- ASTAR is faster than KEYS;
- and KEYS2 runs in time comparable to ASTAR.

Measured purely in terms of runtimes, there is little to recommend KEYS2 over ASTAR. However, ASTAR's heuristic guesses were sometimes observed to be sub-optimal (recall the above discussion on the $(0, 0)$ results in model2's ASTAR results). Such sub-optimality was never observed for KEYS2.

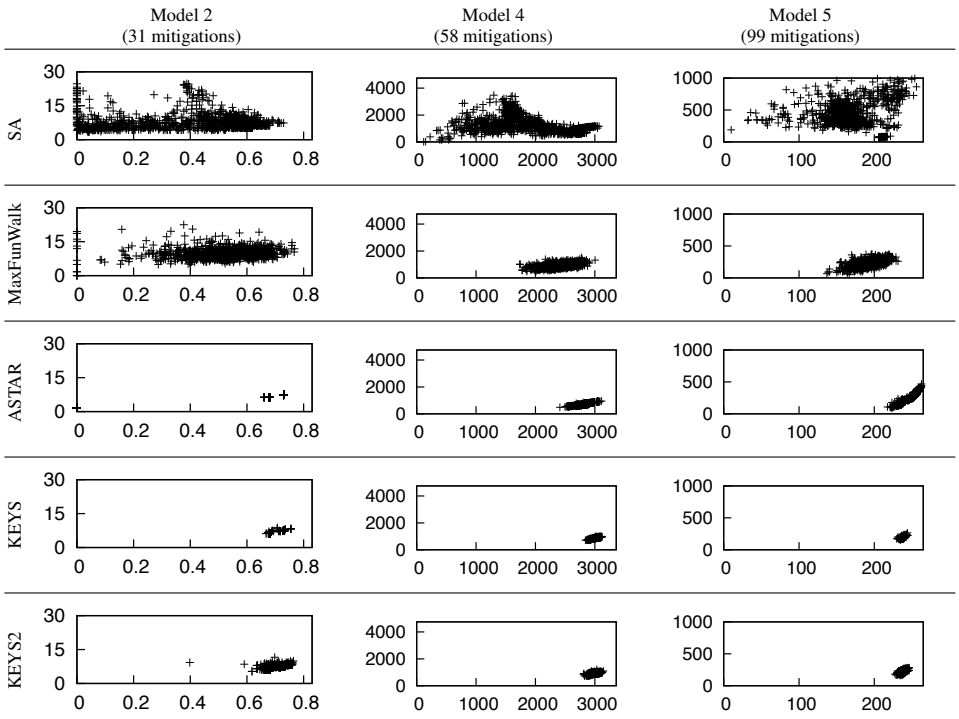

Fig. 10. 1000 results of running five algorithms on three models (15,000 runs in all). The y-axis shows cost and the x-axis shows attainment. The size of each model is measured in number of mitigations. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also better solutions exhibit less variance, i.e. are clumped tighter together.

	Model 2 (31 mitigations)	Model 4 (58 mitigations)	Model 5 (99 mitigations)
SA	0.577	1.258	0.854
MaxFunWalk	0.122	0.429	0.398
ASTAR	0.003	0.017	0.048
KEYS	0.011	0.053	0.115
KEYS2	0.006	0.018	0.038

Fig. 11. Runtimes in seconds, averaged over 100 runs, measured using the "real time" value from the Unix `times` command. The size of each model is measured in number of mitigations (and for more details on model size, see Figure 4).

			Runtime (secs)		$\frac{KEYS}{KEYS2}$
model	expansion	model size	KEYS	KEYS2	
2	1	62	0.01	0.01	1.07
2	2	124	0.03	0.02	1.23
4	1	139	0.04	0.02	2.29
5	1	201	0.13	0.04	3.18
2	4	248	0.10	0.05	2.09
4	2	278	0.17	0.05	3.48
5	2	402	0.50	0.12	4.26
2	8	496	0.44	0.14	3.21
4	4	556	0.73	0.16	4.66
5	4	804	1.98	0.38	5.21
4	8	1112	2.97	0.52	5.71
5	8	1608	8.06	1.35	5.96

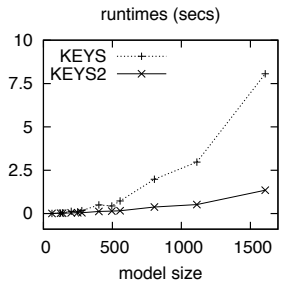

Fig. 12. Runtimes KEYS vs KEYS2 (medians over 1000 repeats) as models increase in size. The "model" number in column one corresponds to Figure 4. The "expansion factor" of column two shows how much the instance generator expanded the model. The "model sizes" of column three are the sum of mitigations, requirements, and risks seen in the expanded model.

6.3 Scale-Up Studies

Figure 12 and Figure 13 shows the effect of changing the size of the model on the number of times that the model is asked to generate a score for both KEYS and KEYS2. To generate these graph, an *instance generator* was created that:

- Examined the real-world DDP models of Figure 4;
- Extracted statistics related to the different types of nodes (mitigations or risks or requirements) and the number of edges between different types of nodes;
- Used those statistics to build random models that were 1,2,4 and 8 times larger than the original models.

Figure 14 shows the results of curve fitting to the plots of Figure 12 and Figure 13. The KEYS and KEYS2 performance curves fit a low-order polynomial (of degree two) with very high coefficients of determination ($R^2 \geq 0.98$).

Figure 14 suggests that we could scale *either* KEYS or KEYS2 to larger models. However we still recommend KEYS2. The column marked $\frac{KEYS}{KEYS2}$ in Figure 13 shows the ratio of the number of calls made by KEYS vs KEYS2. As models get larger, the number of calls to the model are an order of magnitude greater in KEYS than in KEYS2. If applied to models with slower runtimes than DDP, then this order of magnitude is highly undesirable.

model	expansion	model size	Calls to `model` KEYS	KEYS2	KEYS / KEYS2
2	1	62	3100	800	3.9
2	2	124	6200	1100	5.6
4	1	139	5800	1100	5.3
5	1	201	9900	1400	7.1
2	4	248	12400	1600	7.8
4	2	278	11600	1500	7.7
5	2	402	19800	2000	9.9
2	8	496	24800	2200	11.3
4	4	556	23200	2200	10.5
5	4	804	39600	2800	14.1
4	8	1112	46400	3000	15.5
5	8	1608	79200	4000	19.8

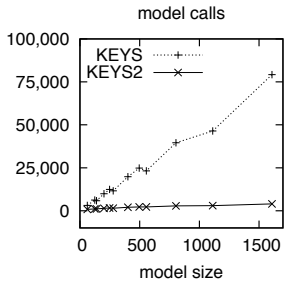

Fig. 13. Number of model calls made by KEYS vs KEYS2 (medians over 1000 results) as models increase in size. This figure uses the same column structure as Figure 12.

	KEYS		KEYS2	
	runtimes	model calls	runtimes	model calls
exponential	0.82	0.83	0.88	0.93
polynomial (of degree 2)	0.99	0.99	0.99	0.98

Fig. 14. Coefficients of determination R^2 of KEYS/KEYS2 performance figures, fitted to two different functions: exponential or polynomial of degree two. Higher values indicate a better curve fit. In all cases, the best fit is not exponential.

6.4 Decision Ordering Algorithms

The decision ordering diagrams of Figure 15 show the effects of the decisions made by KEYS and KEYS2. For both algorithms, at $x = 0$, all of the mitigations in the model are set at random. During each subsequent era, more mitigations are fixed (KEYS sets one at a time, KEYS2 an incrementally increasing number). The lines in each of these plots show the median and spread seen in the 100 calls to the `model` function during each round.

Note that the these diagrams are *tame* and *well-behaved*:

- *Tame:* The "spread" values quickly shrink to a small fraction of the median.
- *Well-behaved:* The median values move smoothly to a plateau of best performance (high attainment, low costs).

On termination (at maximum value of x), KEYS and KEYS2 arrive at nearly identical median results (caveat: for model2, KEYS2 attains slightly more requirements at a slightly higher cost than KEYS). The spread plots for both algorithms are almost indistinguishable (exception: in model2, the KEYS2 spread is less than KEYS). That is, KEYS2 achieves the same results as KEYS, but (as shown in Figure 11 and Figure 12) it does so in less time.

A core assumption of this work is the "keys" concept; i.e. a small number of variables set the remaining model variables. Figure 15 offers significant support for this assumption: observe how most of the improvement in costs and attainments were achieved after KEYS and KEYS2 made only a handful of decisions (often ten or fewer).

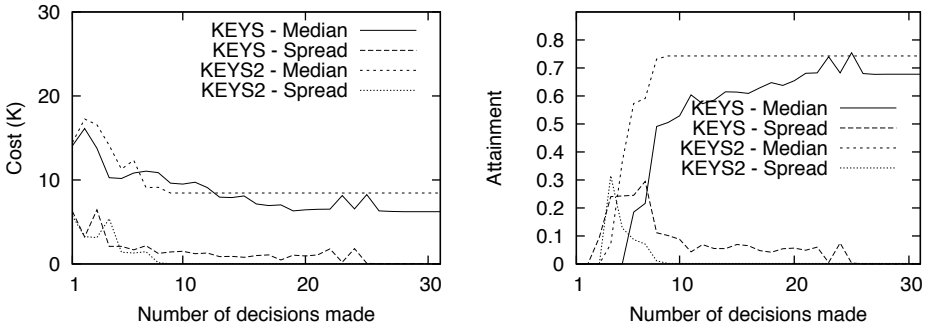

Figure 15a: Internal Decisions on Model 2.

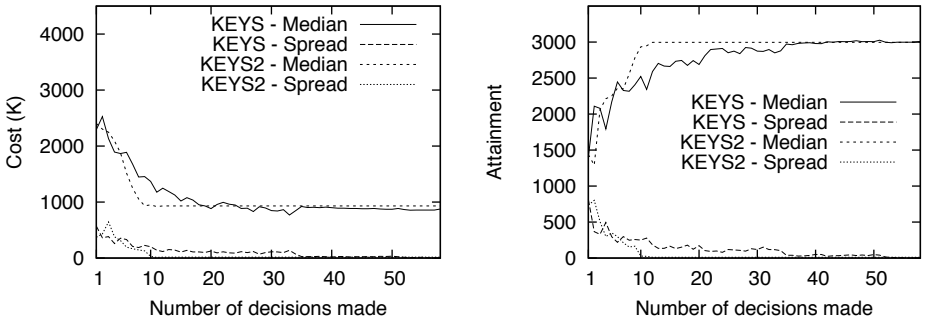

Figure 15b: Internal Decisions on Model 4.

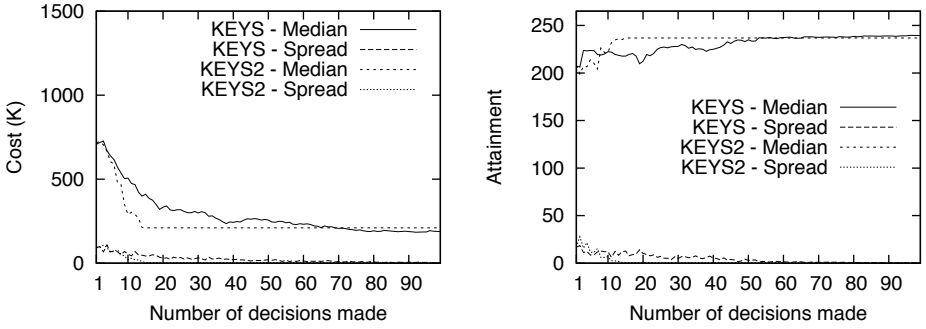

Figure 15c: Internal Decisions on Model 5.

Fig. 15. Median and spread of partial solutions learned by KEYS and KEYS2. X-axis shows the number of decisions made. "Median" shows the 50-th percentile of the measured value seen in 100 runs at each era. "Spread" shows the difference between the 75th and 50th percentile.

On another matter, it is insightful to reflect on the effectiveness of different algorithms for generating decision ordering diagrams. KEYS2 is the most direct and fastest method. As mentioned above, all of the required information can be collected during one execution. On the other hand, simulated annealing, ASTAR, and MaxWalkSat would require a post-processor to generate the diagrams:

- Given D possible decisions, At each era, KEYS and KEYS2 collects statistics on partial solutions where $1, 2, 3, ..|d|$ variables are fixed (where d is the set of decisions) while the remaining $D - d$ decisions are made at random.
- ASTAR, Simulated Annealing, and MaxFunWalk work with full solutions since at each step they offer settings to all $d_i \in D$ variables. In the current form, they cannot comment on partial solutions. Modified forms of these algorithms could add in extra instrumentation and extra post-processing to comment on partial solutions using methods like feature subset selection [28] or a sensitivity analysis [61].

7 Related Work

7.1 Early vs Later Life-cycle Requirements Engineering

The case studies presented in this paper come from the NASA Jet Propulsion Lab's Team X meetings. Team X conducts early life-cycle requirement discussions.

Once a system is running, released, and being maintained or extended, another problem is *release planning*; i.e. what features to add to the next N releases. To solve this problem, an inference engine must reason about how functionality extensions to current software can best satisfy outstanding stakeholder requirements. The challenge of release planning is that the benefits of added functionality must be weighed against the cost of implementing those extensions.

Several approaches have been applied to this problem including:

- The OPTIMIZE tool of Ngo-The and Ruhe [56], which combines linear programming with genetic programming to optimize release plans for software projects
- The weighted Pareto optimal genetic algorithm approach of Zhang et al. [70]

(See also the earlier comparison of exact vs greedy algorithms by Bagnall et al. [5]).

Without further experimentation, we cannot assert that KEYS2 will work as well on later life-cycle models (such as those used in release planning) as it did above (on the earlier life-cycle Team X models). However, at this time, we can see no reason why KEYS2 would not work as a non-linear optimizer of these later life-cycle models. This could be a productive area for future work.

7.2 Other Optimizers

As documented by the search-based SE literature [13,31,32,58] and Gu et al [27], there are many possible optimization methods. For example:

- Gradient descent methods assume that an objective function $F(X)$ is differentiable at any single point N. A Taylor-series approximation of $F(X)$ can be shown to decrease fastest if the negative gradient $(-\Delta F(N))$ is followed from point N.

– Sequential methods run on one CPU while parallel methods spread the work over a distributed CPU farm.
– Discrete methods assume model variables have a finite range (that may be quite small) while continuous methods assume numeric values with a very large (possibly infinite) range.
– The search-based SE literature prefers meta-heuristic methods like simulated annealing, genetic algorithms and tabu search.
– Some methods map discrete values true/false into a continuous range 1/0 and then use integer programming methods like CPLEX [53] to achieve results.
– Other methods find overlaps in goal expressions and generate a binary decision diagram (BDD) where parent nodes store the overlap of children nodes.

This list is hardly exhaustive: Gu et al. list hundreds of other methods and no single paper can experiment with them all. All the algorithms studied here are discrete and sequential. We are currently exploring parallel versions of our optimizers but, so far, the communication overhead outweighs the benefits of parallelism.

As to the general class of gradient descent methods, we do not use them since they assume the objective function being optimizing is essentially continuous. Any model with an "if" statement in it is not continuous since, at the "if" point, the program's behavior may become discontinuous. The requirements models studied here are discontinuous about every subset of every possible mitigation.

As to the more specific class of integer programming methods, we do not explore them here for two reasons. Coarfa et al. [14] found that integer programming-based approaches ran an order of magnitude slower than discrete methods like the MaxWalkSat and KEYS2 algorithms that we use. Similar results have been reported by Gu et.al where discrete methods ran one hundred times faster than integer programming [27].

Harman offers another reason to avoid integer programming methods. In his search-based SE manifest, Harman [31] argues that many SE problems are over-constrained and so there may exist no precise solution that covers all constraints. A complete solution over all variables is hence impossible and partial solution based on heuristic search methods are preferred. Such methods may not be complete; however, as Clarke et al remark, "...software engineers face problems which consist, not in finding *the* solution, but rather, in engineering an *acceptable* or *near-optimal solution* from a large number of alternatives." [13]

7.3 Models of Requirements Engineering

DDP is a ultra-lightweight modeling tool. The value of ultra-lightweight ontologies in early life cycle modeling is widely recognized. For example Mylopoulos' *soft-goal* graphs [54, 55] represent knowledge about non-functional requirements. Primitives in soft goal modeling include statements of partial influence such as *helps* and *hurts*. Another commonly used framework in the design rationale community is a "questions-options-criteria" (QOC) graph [64]. In QOC graphs:

– *Questions* suggest *options*. Deciding on one option can raise other questions;
– Options shown in a box denote *selected options*;

– Options are assessed by *criteria*;
– Criteria are gradual knowledge; i.e. they *tend/reject to support* options.

QOCs can succinctly summarize lengthy debates; e.g. the 480 sentences uttered in a debate on interface options can be displayed in a QOC graph on a single page [45]. Saaty's Analytic Hierarchy Process (AHP) [60] is a variant of QOC.

While DDP shares many of the design aspects of softgoals & QOC & AHP, it differs in its representations and inference method. As explained above around Equation 1, where as AHP and QOC and softgoals propagate influences over hierarchies, DDP propagate influences over matrices.

7.4 Formal Models of Requirements Engineering

Zave & Jackson [69] define requirements engineering as finding the specification S for the domain assumptions K that satisfies the given requirements R; i.e.

$$\text{find } S \text{ such that } S \vdash R \tag{7}$$

Jureta, Mylopoulous & Faulkner [41] (hereafter JMF) take issue with Equation 7, saying that it implicitly assume that K, S, R are precise and complete enough for the satisfaction relation to hold. More specifically, JMF complain that Equation 7 does not permit partial fulfillment of (some) non-functional requirements. Also, the Zave&Jackson definition does not allow any preference ordering of $specification_1$ over $specification_2$. JMF offer a replacement ontology where classical inference is replaced with operators that supports the generation and ranking of subsets of domain assumptions that lead to maximal (w.r.t. size) subsets of the possible goals, and softgoal quality criteria[5].

DDP reinterprets "$\vdash$" in Equation 7 as an inference across numeric quantities, rather than the inference over discrete logical variables suggested by Zave&Jackson. Hence, it can achieve the same goals as JMF (ranking of partial solutions with weighted goals) without requiring the JMF ontology.

7.5 Requirements Analysis Tools

There exist many powerful requirements analysis tools including continuous simulation (also called system dynamics) [1, 65], state-based simulation (including petri net and data flow approaches) [3,29,47], hybrid-simulation (combining discrete event simulation and systems dynamics) [19, 46, 63], logic-based and qualitative-based methods [7, chapter 20] [37], and rule-based simulations [52]. One can find these models being used in the requirements phase (i.e. the DDP tool described below), design refactoring using patterns [25], software integration [18], model-based security [40], and performance assessment [6]. Many researchers have proposed support environments to help explore the increasingly complex models that engineers are developing. Gray et

[5] According to JMF: "a salient characteristic of softgoals is that they cannot be satised to the ideal extent, not only because of subjectivity, but also because the ideal level of satisfaction is beyond the resources available to (and including) the system. It is therefore said that a softgoal is not satised, but satisced."

al [26] have developed the Constraint-Specification Aspect Weaver (C-Saw), which uses aspect-oriented approaches [24] to help engineers in the process of model transformation. Cai and Sullivan [9] describe a formal method and tool called *Simon* that "supports interactive construction of formal models, derives and displays design structure matrices... and supports simple design impact analysis." Other tools of note are lightweight formal methods such as ALLOY [38] and SCR [34] as well as UML tools that allow for the execution of life cycle specifications (e.g. CADENA [10]).

Many of the above tools were built to maximize the expressive power of the representation language or the constraint language used to express invariants. What distinguishes our work is that we are *willing to trade off representational or constraint expressiveness for faster runtimes*. There exists a class of ultra-lightweight model languages which, as we show above, can be processed very quickly. Any of the tools listed in the last paragraph are also candidate solutions to the problem explored in this paper, if it can be shown that their processing can generate tame and well-behaved decision ordering diagrams in a timely manner.

7.6 Other Work on "Keys"

Elsewhere [50], we have documented dozens of papers that have reported the keys effect (that a small number of variables set the rest) under different names including *narrows, master-variables, back doors,* and *feature subset selection*:

- Amarel [4] observed that search problems contain narrow sets of variables or collars that must be used in any solution. In such a search space, what matters is not so much how you get to these collars, but what decision you make when you get there. Amarel defined macros encoding paths between *narrows*, effectively permitting a search engine to jump between them.
- In a similar theoretical analysis, Menzies & Singh [50] computed the odds of a system selecting solutions to goals using complex, or simpler, sets of preconditions. In their simulations, they found that a system will naturally select for tiny sets of preconditions (a.k.a. the keys) at a very high probability.
- Numerous researchers have examined *feature subset selection*; i.e. what happens when a data miner deliberately ignores some of the variables in the training data. For example, Kohavi and John [44] showed in numerous datasets that as few as 20% of the variables are key - the remaining 80% of variables can be ignored without degrading a learner's classification accuracy.
- Williams et.al. [68] discuss how to use keys (which they call "back doors") to optimize search. Constraining these back doors also constrains the rest of the program. So, to quickly search a program, they suggest imposing some set values on the key variables. They showed that setting the keys can reduce the solution time of certain hard problems from exponential to polytime, provided that the keys can be cheaply located, an issue on which Williams et.al. are curiously silent.
- Crawford and Baker [16] compared the performance of a complete TABLEAU prover to a very simple randomized search engine called ISAMP. Both algorithms assign a value to one variable, then infer some consequence of that assignment with forward checking. If contradictions are detected, TABLEAU backtracks while

ISAMP simply starts over and re-assigns other variables randomly. Incredibly, ISAMP took *less* time than TABLEAU to find *more* solutions using just a small number of tries. Crawford and Baker hypothesized that a small set of *master variables* set the rest and that solutions are not uniformly distributed throughout the search space. TABLEAU's depth-first search sometimes drove the algorithm into regions containing no solutions. On the other hand, ISAMP's randomized sampling effectively searches in a smaller space.

In summary, the core assumption of our algorithms are supported in many domains.

8 Conclusion

Requirements tools such as the DDP tool (used at NASA for early lifecycle discussions), contain a shared group memory that stores all of the requirements, risks, and mitigations of each member of the group. Software tools can explore this shared memory to find consequences and interactions that may have been overlooked.

Studying that group memory is a non-linear optimization task: possible benefits must be traded off against the increased cost of applying various mitigations. Harman [31] cautions that solutions to non-linear problems may be "brittle" - small changes to the search results may dramatically alter the effectiveness of the solution. Hence, when reporting an analysis of this shared group memory, it is vitally important to comment on the robustness of the solution.

Decision ordering diagrams are a solution robustness assessment method. The diagrams rank all of the possible decisions from most-to-least influential. Each point x on the diagrams shows the effects on imposing the conjunction of decisions $1 \leq j \leq x$. These diagrams can comment on the robustness and neighborhood of solution $\{d_1..d_x\}$ using two operators:

1. By considering the variance of the performance statistics after applying $\{d_1..d_x\}$.
2. By comparing the results of using the first x decisions to that of using the first $x-1$ or $x+1$ actions.

Since the diagrams are sorted, this analysis of robustness and neighborhood takes, at most, time linear of the number of decisions. That is, theoretically, it takes linear time to *use* a decision ordering diagram (see §4.3).

Empirically, it take low-order polynomial time to *generate* a decision ordering diagram using KEYS2. This algorithm makes the "key" assumption (that a small group of variables set everything else) and uses Bayesian ranking mechanism to quickly find those keys. As discussed above in the *Related Work* section, this assumption holds over a wide range of models used in a wide range of domains. This "keys" assumption can be remarkably effective: our empirical results show that KEY2 can generate decision ordering diagrams faster than the other algorithms studied here. Better yet, curve fits to our empirical results show that KEYS runs in low-order polynomial time (of degree two) and so should scale to very large models.

Prior to this work, our two pre-experimental concerns were that:

– We would need to trade solution robustness against solution quality. More robust solutions may not have the highest quality.
– Demonstrating solution robustness requires multiple calls to an analysis procedure.

At least for the models studied here, neither concern was realized. KEYS2 generated the highest quality solutions (lowest cost, highest attainments) and did so more quickly than the other methods.

In §4.3 it was argued that decision ordering diagrams are useful when they are *timely* to generate while being *well-behaved* and *tame*. KEYS2's results are the most *timely* (fastest to generate) of all of the methods studied here. As to the other criteria, Figure 15 shows that KEYS2's decision ordering diagrams:

– Move smoothly to a plateau with only a small amount of "jitter";
– Have very low spreads, compared to the median results.

That is, at least for the models explored here, KEYS2 generated decision ordering diagrams they are both *well-behaved* and *tame*.

In summary, we recommend KEYS2 for generating decision ordering diagrams since, apart from the (slightly slower) KEYS algorithm, we are unaware of other search-based software engineering methods that enable such a rapid reflection of solution robustness.

9 Acknowledgments

This research was conducted at West Virginia University, the Jet Propulsion Laboratory under a contract with the National Aeronautics and Space administration, Alderson-Broaddus College, and Miami University. 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

References

1. T. Abdel-Hamid and S. Madnick. *Software Project Dynamics: An Integrated Approach*. Prentice-Hall Software Series, 1991.
2. F. A. Administration. System engineering manual version 3.1, section 4.6: Trade studies, 2006. Available from `http://www.faa.gov/about/office_org/headquarters_offices/ato/service_un%its/operations/sysengsaf/seman/SEM3.1/Section%204.6.pdf`.
3. M. Akhavi and W. Wilson. Dynamic simulation of software process models. In *Proceedings of the 5th Software Engineering Process Group National Meeting (Held at Costa Mesa, California, April 26 - 29)*. Software engineering Institute, Carnegie Mellon University, 1993.
4. S. Amarel. Program synthesis as a theory formation task: Problem representations and solution methods. In R. S. Michalski, J. G. Carbonell, and T. M. Mitchell, editors, *Machine Learning: An Artificial Intelligence Approach: Volume II*, pages 499–569. Kaufmann, Los Altos, CA, 1986.
5. A. Bagnall, V. Rayward-Smith, and I. Whittley. The next release problem. *Information and Software Technology*, 43(14), December 2001.

6. S. Balsamo, A. D. Marco, P. Inverardi, and M. Simeoni. Model-based performance prediction in software development: A survey. *IEEE Transactions on Software Engineering*, 30(5), May 2004.
7. I. Bratko. *Prolog Programming for Artificial Intelligence. (third edition).* Addison-Wesley, 2001.
8. T. Bylander, D. Allemang, M. Tanner, and J. Josephson. The Computational Complexity of Abduction. *Artificial Intelligence*, 49:25–60, 1991.
9. Y. Cai and K. J. Sullivan. Simon: modeling and analysis of design space structures. In *ASE '05: Proceedings of the 20th IEEE/ACM international Conference on Automated software engineering*, pages 329–332, New York, NY, USA, 2005. ACM Press.
10. A. Childs, J. Greenwald, G. Jung, M. Hoosier, and J. Hatcliff. Calm and cadena: Metamodeling for component-based product-line development. *IEEE Computer*, 39(2), Feburary 2006. Available from `http://projects.cis.ksu.edu/docman/view.php/7/129/CALM-Cadena-IEEE-Comp%uter-Feb-2006.pdf`.
11. A. Cimatti, E. Clarke, E. Giunchiglia, F. Giunchiglia, M. Pistore, M. Roveri, R. Sebastiani, and A. Tacchella. NuSMV Version 2: An OpenSource Tool for Symbolic Model Checking. In *Proc. International Conference on Computer-Aided Verification*, 2002.
12. R. Clark. Faster treatment learning, Computer Science, Portland State University. Master's thesis, 2005.
13. J. Clarke, J. Dolado, M. Harman, R. Hierons, B. Jones, M. Lumkin, B. Mitchell, S. Mancoridis, K. Rees, M. Roper, and M. Shepperd. Reformulating software engineering as a search problem. *IEE Proceedings-Software*, 150(3):161–175, 2003.
14. C. Coarfa, D. D. Demopoulos, A. San, M. Aguirre, D. Subramanian, and M. Y. Vardi. Random 3-sat: The plot thickens. In *In Principles and Practice of Constraint Programming*, pages 143–159, 2000. Availabe from `http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.42.3662`.
15. S. Cornford, M. Feather, and K. Hicks. DDP a tool for life-cycle risk management. In *IEEE Aerospace Conference, Big Sky, Montana*, pages 441–451, March 2001.
16. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.
17. J. DeKleer. An Assumption-Based TMS. *Artificial Intelligence*, 28:163–196, 1986.
18. P. Denno, M. P. Steves, D. Libes, and E. J. Barkmeyer. Model-drven integration using existing models. *IEEE Software*, 20(5):59–63, Sept.-Oct. 2003.
19. P. Donzelli and G. Iazeolla. Hybrid simulation modelling of the software process. *Journal of Systems and Software*, 59(3), December 2001.
20. 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`.
21. M. Feather, K. Hicks, R. Mackey, and S. Uckun. Guiding technology deployment decisions using a quantitative requirements analysis technique. In *IEEE International Conference on Requirements Engineering, Industrial Practice and Experience track Barcelona, Spain*, 2008.
22. 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`.
23. M. Feather, S. Uckun, and K. Hicks. Technology maturation of integrated system health management. In *Space Technology and Applications International Forum (STAIF-2008) Albuquerque, USA*, February 2008.
24. R. E. Filman. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.

25. R. France, S. Ghosh, E. Song, and D. Kim. A metamodeling approach to pattern-based moel refractoringt. *IEEE Software*, 20(5):52–58, Sept.-Oct. 2003.
26. J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering. *IEEE Computer*, 39(2):51–58, February 2006.
27. J. Gu, P. W. Purdom, J. Franco, and B. W. Wah. Algorithms for the satisfiability (sat) problem: A survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 19–152. American Mathematical Society, 1997.
28. M. Hall and G. Holmes. Benchmarking attribute selection techniques for discrete class data mining. *IEEE Transactions On Knowledge And Data Engineering*, 15(6):1437– 1447, 2003. Available from `http://www.cs.waikato.ac.nz/~mhall/HallHolmesTKDE.pdf`.
29. D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
30. M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering, ICSE'07*. 2007.
31. M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
32. M. Harman and J. Wegener. Getting results from search-based approaches to software engineering. In *ICSE '04: Proceedings of the 26th International Conference on Software Engineering*, pages 728–729, Washington, DC, USA, 2004. IEEE Computer Society.
33. P. Hart, N. Nilsson, and B. Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE Transactions on Systems Science and Cybernetics*, 4:100–107, 1968.
34. C. Heitmeyer. Software cost reduction. In J. J. Marciniak, editor, *Encyclopedia of Software Engineering*, January 2002. Available from `http://chacs.nrl.navy.mil/publications/CHACS/2002/2002heitmeyer-encse.p%df`.
35. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
36. Y. Hui, E. Prakash, and N. Chaudhari. Game ai: artificial intelligence for 3d path finding. In *TENCON 2004. 2004 IEEE Region 10 Conference*, volume 2, pages 306–309, 2004.
37. Y. Iwasaki. Qualitative physics. In P. C. A. Barr and E. Feigenbaum, editors, *The Handbook of Artificial Intelligence*, volume 4, pages 323–413. Addison Wesley, 1989.
38. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
39. 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`.
40. J. Jerjens and J. Fox. Tools for model-based security engineering. In *ICSE '06: Proceeding of the 28th international conference on Software engineering*, pages 819–822, New York, NY, USA, 2006. ACM Press.
41. I. Jureta, J. Mylopoulos, and S. Faulkner. Revisiting the core ontology and problem in requirements engineering. In *International Requirements Engineering, 2008. RE '08. 16th IEEE*, pages 71–80, Sept. 2008.
42. H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence and the Eighth Innovative Applications of Artificial Intelligence Conference*, pages 1194–1201, Menlo Park, Aug. 4–8 1996. AAAI Press / MIT Press. Available from `http://www.cc.gatech.edu/~jimmyd/summaries/kautz1996.ps`.
43. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
44. R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.

45. A. MacLean, R. Young, V. Bellotti, and T. Moran. Questions, options and criteria: Elements of design space analysis. In T. Moran and J. Carroll, editors, *Design Rationale: Concepts, Techniques, and Use*, pages 53–106. Lawerence Erlbaum Associates, 1996.
46. R. Martin and R. D. M. Application of a hybrid process simulation model to a software development project. *Journal of Systems and Software*, 59(3), 2001.
47. R. Martin and D. M. Raffo. A model of the software development process using both continuous and discrete models. *International Journal of Software Process Improvement and Practice*, June/July 2000.
48. T. Menzies, O. Jalali, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings PROMISE '08 (ICSE)*, 2008.
49. T. Menzies, J. Kiper, and M. Feather. Improved software engineering decision support through automatic argument reduction tools. In *SEDECS'2003: the 2nd International Workshop on Software Engineering Decision Support (part of SEKE2003)*, June 2003. Available from `http://menzies.us/pdf/03star1.pdf`.
50. T. Menzies and H. Singh. Many maybes mean (mostly) the same thing. In M. Madravio, editor, *Soft Computing in Software Engineering*. Springer-Verlag, 2003. Available from `http://menzies.us/pdf/03maybe.pdf`.
51. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. Equation of state calculations by fast computing machines. *J. Chem. Phys*, 21:1087–1092, 1953.
52. P. Mi and W. Scacchi. A knowledge-based environment for modeling and simulation software engineering processes. *IEEE Transactions on Knowledge and Data Engineering*, pages 283–294, September 1990.
53. H. Mittelmann. Recent benchmarks of optimization software. In *22nd Euorpean Conference on Operational Research*, 2007.
54. J. Mylopoulos, L. Cheng, and E. Yu. From object-oriented to goal-oriented requirements analysis. *Communications of the ACM*, 42(1):31–37, January 1999.
55. J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions of Software Engineering*, 18(6):483–497, June 1992.
56. A. Ngo-The and G. Ruhe. Optimized resource allocation for software release planning. *Software Engineering, IEEE Transactions on*, 35(1):109–123, Jan.-Feb. 2009.
57. J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
58. L. Rela. Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, 2004.
59. S. J. Russell, P. Norvig, J. F. Candy, J. M. Malik, and D. D. Edwards. *Artificial intelligence: a modern approach*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2003.
60. T. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill, 1980.
61. A. Saltelli, K. Chan, and E. Scott. *Sensitivity Analysis*. Wiley, 2000.
62. B. Selman, H. A. Kautz, and B. Cohen. Local search strategies for satisfiability testing. In M. Trick and D. S. Johnson, editors, *Proceedings of the Second DIMACS Challange on Cliques, Coloring, and Satisfiability*, Providence RI, 1993.
63. S. Setamanit, W. Wakeland, and D.Raffo. Using simulation to evaluate global software development task allocation strategies. *Software Process: Improvement and Practice, (Forthcoming)*, 2007.
64. S. B. Shum and N. Hammond. Argumentation-based design rationale: What use at what cost? *International Journal of Human-Computer Studies*, 40(4):603–652, 1994.
65. H. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill, 2000.

66. B. Stout. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, (7), 1997.
67. T. E. Uribe and M. E. Stickel. Ordered binary decision diagrams and the davis-putnam procedure. In *In Proc. of the 1st International Conference on Constraints in Computational Logics*, pages 34–49. Springer-Verlag, 1994.
68. R. Williams, C. Gomes, and B. Selman. Backdoors to typical case complexity. In *Proceedings of IJCAI 2003*, 2003. `http://www.cs.cornell.edu/gomes/FILES/backdoors.pdf`.
69. P. Zave and M. Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, 1997.
70. H. Zhang and X. Zhang. Comments on 'data mining static code attributes to learn defect predictors'. *IEEE Transactions on Software Engineering*, September 2007.

Obtaining our System

We have placed on-line all the materials required for other researchers to conduct further investigation into this problem. All the code, Makefiles, scripts, and so on used in this paper are available at `http://unbox.org/wisp/tags/ddpExperiment/install`. For security reasons, all the available JPL requirements models have been "sanitized"; i.e. all words replaced with anonymous variables.