Real-time Optimization of Requirements Models

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

[1] West Virginia University, Morgantown, WV, USA,
gregoryg@csee.wvu.edu, tim@menzies.us, ojalali@mix.wvu.edu,
[2] Jet Propulsion Laboratory, California Institute of Technology, Pasadena, CA, USA,
martin.s.feather@jpl.nasa.gov
[3] Dept. of Computer Science and Systems Analysis, Miami University, Oxford, OH, USA,
kiperjd@muohio.edu

Abstract. Early life cycle risk models can represent the requirements that a development group would want to achieve, the risks that could prevent these requirements from being met, and mitigations that could alleviate those risks. Our task is the selection of the *least expensive* set of mitigations that achieve the *highest attainment* of requirements.
As these risk 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.
One candidate technology for real-time requirements optimization is the KEYS2 search engine. KEYS2 uses a very simple (hence, very fast) novel Bayesian technique that identifies both the useful succinct sets of mitigations as well as cost-attainment tradeoffs for partial solutions. This paper reports experiments demonstrating that KEYS2 runs four orders of magnitude faster than our previous implementations and outperforms standard search algorithms including a classic stochastic search (simulated annealing), a state-of-the art local search (MaxWalkSat), and a standard graph search (A*).

1 Introduction

Model-based design is a well-accepted software engineering technique. According to Sendall and Kozacaynski, "using concepts closer to the problem domain ..." via modeling [90] can lead to increased productivity and reduced time-to-market. Hailpern and Tarr argue that model-based design "imposes structure and common vocabularies so that artifacts are useful for their main purpose in their particular stage in the life cycle" [6]. Model-based Software Engineering has become the norm at large organization such as Microsoft [37]; Lockheed Martin [96]; and the Object Management Group [12].

Such models reveal important combinations of options that might otherwise be missed. For example, at NASA, design teams use "Defect Detection and Prevention" (DDP) [18, 28] models to find the *least* costly project options that *most* increase the chance to obtain more requirements. DDP is used by teams of experts to display their (a) current understanding of requirements; (b) the known risks to requirements; as well as (c) the mitigations that might reduce the risks. A typical DDP session lasts hours to days. A dozen (or more) experts meet in a large room. In that room, experts either sit together for large group discussions or break apart into small groups around the room. In order to assure cohesiveness between all the experts and all the groups, a DDP model showing the total space of options is continuously updated and displayed on a screen at one end of the room. For more details on DDP, see §2.

Often, it is not possible for human experts to debate all N options within a DDP model. Many models contain dozens to hundreds of options and human beings can not rigorously debate all 2^N combinations of those options. Typically, humans focus on $W \ll N$ "what-if" scenarios. For example, at JPL, it is common in DDP sessions for experts to focus on (say) $W = 2..10$ binary choices within a total space of $N > 100$ choices. Note that this introduces a "blind-spot" into the requirements analysis of the $N - W$ issues that is not included in the experts' debates.

In order to probe those blind spots, we conduct optimization studies. For each of the 2^W scenarios, we impose the constraints of that scenario, then use an automatic optimizer to explore the other $N - W$ options. This exploration proposes settings to the other options that most increase the benefits of requirements coverage while minimizing their associated cost, for more details on that process, see §2. Each of the 2^W scenarios are then ranked via their net effect on benefits and cost. In an interesting number of cases, our optimization studies offer recommendations from the space of $N-W$ options in the model that the human experts have overlooked. For example:

- Experts may inappropriately focus on propulsion weight reduction; a better solution may be to use smarter controlling algorithms, thus requiring fewer thrusters.
- DDP optimization might suggest reducing the weight of a science payload by (say) smarter software to control the remote sensing, thus requiring fewer instruments.

This ability to explore human blind-spots in requirements models is a key feature of DDP that adds value to standard design practices.

To best support this blind-spot exploration, our DDP optimization should run fast enough that humans can get feedback on their models in a timely manner. Previously, we have proposed various technologies for DDP optimization [28, 30]. For example, after a Monte Carlo run of 1000 randomly selected options, rule learning can be used to identify which options were most associated with lower cost and higher attainment of requirements [30]. The constraints offered by these rules can then be imposed on the DDP models, followed by another round of Monte Carlo simulation and rule learning. For a DDP model with 100 variables, it takes ≈ 300 seconds before the rig terminates (i.e. no improvement is observed in round $i + 1$ after round i).

The premise of this paper is that our previous solutions to the DDP optimization problem are too slow. Ideally, we wish for requirements optimization to occur in "real-time"; i.e. to offer advice before an expert's attention wanders to other issues. Neilson [76, chp5] reports that "the basic advice regarding response times has been about the same for thirty years; i.e.

- 1.0 second is about the limit for the user's flow of thought to stay uninterrupted, even though the user will notice the delay
- 10 seconds is the limit for keeping a user's attention focused on the dialogue."

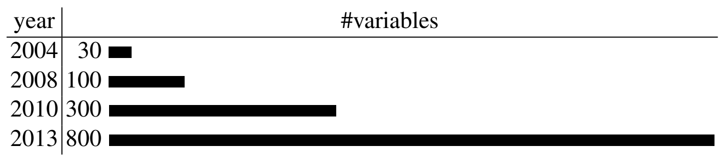

Fig. 1. Growth trends: number of variables in DDP's data dictionary.

In the following calculation, we will split the difference and demand five second response time for 2^W optimizations of $W \leq 10$ DDP scenarios. Figure 1 shows the growth rate (historical and projected) of variables within DDP models. Note that by 2013, it is expected that DDP models will be eight times larger than today. It is possible to estimate the required speed of our algorithms on contemporary computers, such that by the year 2013, DDP optimization will terminate in 5 seconds. Assuming a doubling of CPU speed every two years, then by 2013 our computers will run $2^{(2013-2008)/2} = 560\%$ faster. However, larger models have more interconnections, making them slower to process (DDP optimization time tends to grow exponentially with the number of variables).

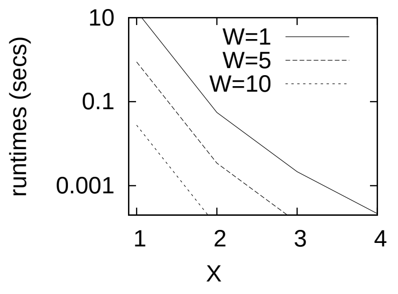

Fig. 2. Estimated time required on contemporary machines in order to handle W what-if scenarios DDP models in 2013 (when the models are assumed to be eight times larger than those currently seen at JPL). The x-axis represents a range of assumptions on the size of the exponential scale up factor X^8.

Figure 2 shows those calculations assuming that we are exploring 2^W scenarios in the space of $1 \leq W \leq 10$ what-ifs. We assume the larger models are slower to process by some amount X^8 where X is an exponential scale-up factor. Since we do not know the exact value of X, Figure 2 explores a range of possible values.

These plots show that our current runtimes are orders of magnitude too slow for real-time requirements optimization. If we run only one what-if query, the $W = 1$ curve shows that for even modest scale up facts ($X \geq 2$), we require runtimes on contemporary machines of 10^{-1} seconds; i.e. four orders of magnitude faster than the rule learning approach described above. Worse, handling multiple what-ifs (e.g. $W <= 10$) requires response times in the order of 10^{-2} to 10^{-3} secs.

The rest of this paper describes our progress towards achieving these runtimes:

– After a discussion of early life cycle ultra-lightweight modeling (such as DDP), we present various search-based software engineering techniques to optimize DDP models.

– Our new KEYS2 algorithm[4] will be discussed as well as various experiments that were conducted to compare these techniques.

In summary, KEYS2 is running in the order of 10^{-3} seconds, which makes it a candidate solution for real-time requirements optimization over 2^W scenarios of $W \leq 10$. However, further optimizations are desirable and our future work section offers some promising directions.

This paper improves over an earlier publication [69] in three significant ways:

– Previously, we explored one search engine (KEYS); this report compares that search engine against alternatives (KEYS2, Simulated Annealing, A*, MaxWalkSat);
– This paper demonstrates that, at least for the task of optimizing DDP models, KEYS does better than those state-of-the-art alternatives.
– This paper reports a new version of KEYS that runs two to four times faster than the previously reported version.

In the spirit of the PROMISE workshop series, all of the models and algorithms required to repeat our experiments are freely available on-line under an open-source license[5]. We welcome other researchers to download and experiment with our code. Also, we hope that this special issue (including this paper) encourages other researchers to offer their work in a similar manner (i.e. repeatable and/or refutable and/or improbable).

2 Early Life cycle Models

DDP models are an example of a class of early life cycle *ultra-lightweight* models. This section describes ultra-lightweight models and DDP.

2.1 Ultra-Lightweight Models

Informal ultra-lightweight sketches are a common technique for illustrating and sharing intuitions. For example, in early life cycle requirements engineering, software engineers often pepper their discussions with numerous hastily-drawn sketches. Kremer argues convincingly that such visual forms have numerous advantages for acquiring knowledge [56]. Other researchers take a similar view: “Pictures are superior to texts in a sense that they are abstract, instantly comprehensible, and universal.” [44].

Normally, hastily-drawn sketches are viewed as precursors to some other, more precise modeling techniques which necessitate further analysis. Such further formalization may not always be practical. The time and expertise required to build formal models does not accommodate the pace with which humans tend to revise their ideas.

[4] KEYS2 evolved as follows. Clark and Menzies [16] proposed a support-based Bayesian sampling method to optimizing rule learning. Jalali [69] speculated that a greedy hill-climbing version of that sampling method could *replace*, rather than merely *augment* rule learning. This resulted in the original KEYS algorithm [69]. Gay and Menzies (this paper) proposed an optimization to that process, called KEYS2, whereby the algorithm starts out greedy but then takes progressive longer steps into the search space.

[5] See `http://unbox.org/wisp/tags/ddpExperiment/install`

Also, such further formalization may not be advisable. Goel [35] studied the effects of diagram formalism on system design. In an *ill-structured diagram* (such as a free-hand sketch using pencil and paper), the denotation and type of each visual element is ambiguous. In a *well-structured diagram* (such as those created using MacDraw), each visual element clearly denotes one thing of one class only. Goel found that ill-structured tools generated more design variants — more drawings, more ideas, more re-use of old ideas — than well-structured tools.

The value of ultra-lightweight ontologies in early life cycle modeling is widely recognized, as witnessed by the numerous frameworks that support them. For example:

- Mylopoulos' *soft-goal* graphs [74, 75]. represent knowledge about non-functional requirements. Primitives in soft goal modeling include statements of partial influence such as *helps* and *hurts*.
- A common framework in the design rationale community is a "questions-options-criteria" (QOC) graph [92]. In QOC graphs:
  - *questions* suggest *options* and deciding on a certain 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 to support* or *tend to reject* options.

  QOCs can succinctly summarize lengthy debates; e.g. 480 sentences uttered in a debate between two analysts on interface options can be displayed in a QOC graph on a single page [59]. Saaty's Analytic Hierarchy Process (AHP) [86] is a variant of QOC.
- Another widely used framework is Quality Functional Deployment diagrams (QFD) [3] Where as AHP and QOC propagate influences over hierarchies, QDF (and DDP) propagate influences over matrices.

2.2 The Defect Detection and Prevention tool

From all the above work, we elect to study real-time requirements optimization in the context of DDP models (and this section describes DDP). The main reason for this decision is that one of us (Feather) can access larger real-world DDP models than models created using other frameworks (soft goals, QOC, AHP, QFD, etc). In future work, we intend to apply AI search to other early early life cycle ultra-lightweight models.

In a typical design session at the NASA Jet Propulsion Lab, a dozen (or more) designers refine and extend a single evolving design document. These designers are NASA's top experts in guidance, propulsion, communication, science, and other fields. Such an expert is a scarce resource who is needed for many tasks across the NASA organization. Hence, we must use these experts' time in the most cost-effective manner possible. Therefore, we seek tools that provide interactive design advice in a timely manner, ideally in time for the experts to assess multiple options during a single meeting. Such a tool would enable the teams to spend more time exploring a high-level trade space instead of laboriously elaborating one specific design in order to understand the consequences of various decisions.

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.
- *Part-of relations* structure the collections of requirements, risks and mitigations;

Fig. 3. DDP's ontology

The "Defect Detection and Prevention" (DDP) tool [18, 28] provides a succinct ontology for representing this design process. These models allow for the representation of the goals, risks, and risk-removing mitigations that belong to a specific project. DDP might be thought of as a more complicated form of QFD. Like in the QFD approach, DDP sets a series of requirements or goals. However, DDP sets itself apart with its basis- a quantitative, probabilistic approach inspired by risk assessment techniques. This approach is why we believe that DDP is useful for studying the requirements needs of a wide variety of technologies, including software, hardware and combinations of the two.

DDP is based on the following principles:

- *Earlier is Better:* The case for "earlier is better" is well established. Fark et al. [47] noted that "80% of all design decisions are taken in the first 20% of the design time." Design tools that do not operate early in the life cycle can only affect a small portion of the remaining decisions. Not only is earlier design critical, but rapid improvement of those designs is also critical. Fixing defect costs exponentially more the longer those defects remain in the system. Boehm and Papaccioa advise that reworking software is cheaper earlier in the life cycle "by factors of 50 to 200." [10]
- *Use Risk-driven SE:* In risk-driven SE [9], there are three primary modeling constructs. The engineering *requirements* are mapped to a series of *risks*, a representation of any of the factors that could prevent a goal from being reached. *Mitigations* are measures that may be implemented in order to alleviate, or at least reduce, risks. Hence, our DDP tool is a visual editor of a model with node types reflecting these three constructs. During interactive sessions, DDP projects its model on a screen where teams and designers review and extend the network. This information can influence the entire design process and may lead to a more efficient development cycle.

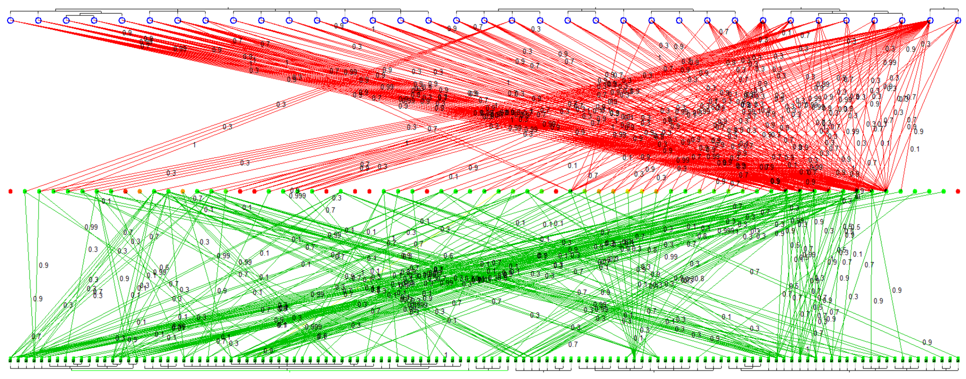

Fig. 4. An example of a model formed by the DDP tool. Red lines connect risks (middle) to requirements (top). Green lines connect mitigations (bottom) to the risks.

– *Use Value-based SE:* DDP is also a *value-based SE* [85] tool where the *benefits* of a design variant are weighed against their *cost*. Each mitigation in DDP has an associated cost so that DDP can guide and optimize decision making. For example, on a fixed budget, DDP can find the cheapest mitigations that retire the most risks and hence allow for the attainment of the most requirements. DDP can repeat this process for different cost levels to generate a cost-benefit trade-space diagram.

The process by which DDP is employed is 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.
– Assertions from the experts are expressed using the ontology of Figure 3.

The ontology must be 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 an intricate model of influences connecting project requirements to risks to possible mitigations; see Figure 4. Such models have been applied to over a dozen applications, including such advanced technologies 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.

Clear project improvements have been seen from these DDP sessions. Cost savings in at least two sessions have exceeded $1 million, while savings of over $100,000 have resulted in others. 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. The DDP tool can be used to document all of this information and the decision making process of these studies. Because of these reasons, 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.

It is important to note that DDP is not just a software design tool. Software and hardware are frequently designed together at JPL. The DDP tool is better viewed as a *software systems engineering* tool where the interactions between hardware and software, and the possible incompatibilities, can be quickly explored.

3 DDP Inputs/Outputs

The rest of this paper will discuss various algorithms designed to find an optimal solution to the DDP models. This section describes the models processed by these algorithms as well as the functions used to score the output.

3.1 DDP Model Format

A simple *knowledge compiler* exports the DDP models into a form accessible by our search engines. Knowledge compilation is a technique whereby certain information is compiled into a target language. These compiled models are used to rapidly answer a large number of queries while the main program is running [20, 88].

Knowledge compilation pushes some percentage of the computational overhead into the compilation phase. This cost is amortized over time as we increase the number of on-line queries. This is a very useful feature in our application since some of the algorithms that we use make thousands of calls to these DDP models. Previous work with knowledge compilation has focused on compilation languages utilizing CNF equations, state machines, or BDD [8, 20]. For this work, a procedural target language was sufficient. Hence, the DDP models were compiled into a structure used by the C programming language.

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

```
#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. 5. 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. 6. Details of Five DDP Models.

- 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 the impact of risks on requirements).

The knowledge compiler converts this trivial DDP model into `setupModel` and `model` functions similar to those in Figure 5. 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 6 compares this biggest model to four other real-world DDP models.

While not a linear process, knowledge compilation is not the bottleneck in our ability to handle real-time requirements optimization. Currently it takes about two seconds to compile a model with 50 requirements, 31 risks, and 58 mitigations. Note that:

- This compilation only has to happen once, after which we will run our 2^W what-if scenarios.
- These runtimes come from an unoptimized Visual Basic implementation which we can certainly significantly speed up.
- Usually, experts change a small portion of the model then run 2^W what-if scenarios to understand the impact of that change. An incremental knowledge compiler (that only updates changed portions) would run much faster than a full compilation of an entire DDP model.

Initially, it was hoped that knowledge compilation by itself would suffice for real-time requirements optimization. However, on experimentation, we found that it reduced our runtimes by only one to two orders of magnitude. While an impressive speed up, this is still two orders of magnitude too slow to to achieve the 10^{-3} seconds response time required for real-time requirements optimization of 2^{W} scenarios for $(W \leq 10)$ what-if queries. Hence, in this paper, we combined knowledge compilation with AI search methods.

Note that knowledge compilation is useful for more than just runtime optimization. Without it, the collaborative research that led to this paper would have been impossible. Knowledge compilation allows JPL to retain their proprietary information while at the same time, allowing outsiders to access these models. For our experiments, JPL ran the knowledge compiler and passed models like those shown in Figure 5 to West Virginia University. These models have been anonymized to remove proprietary information while still maintaining their computational nature. In this way, JPL could assure its clients that any project secrets would be safe while allowing outside researchers to perform valuable experiments.

3.2 Scoring Outputs

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:

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

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*.

Scoring Partial Solutions The search methods presented below divide into two groups.

- One group proposes settings to all variables, at each step of their search. For such *full* solutions, scoring is a simple matter: just call the DDP model (e.g. Figure 5) then execute Equation 1 on the returns *cost* and *attainment* values.
- Another group of search methods proposes settings to some subset of the variables. For such *partial solutions*, the scoring procedure must be modified. Given *fixed* settings to some of the variables, and *free* settings to the rest, call the DDP model and Equation 1 N times (each time with randomly selected settings to the free variables) and report the median of the generated scores.

In the following work, the following observation will be important. Scoring *partial* solutions requires N calls to the models while scoring *full* solutions requires only one call.

4 Algorithms

Discrete models (like DDP) are generally not suitable for numeric optimization. The non-continuous nature of these models leads to ‘cliffs” where the behavior of a system can change dramatically.

Optimization of discrete models can be implemented by AI search algorithms such as Simulated Annealing, A*, MaxWalkSat, KEYS, and KEYS2.

4.1 SimpleSA: Simulated Annealing

Simulated Annealing is a classic stochastic search algorithm. It was first described in 1953 [72] and refined in 1983 [54]. 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.

```
Procedure SimpleSA
MITIGATIONS:= set of mitigations
SCORE:= score of MITIGATIONS
while TIME < MAX_TIME && SCORE < MIN_SCORE //minScore is a constant score (threshold)
   find a NEIGHBOR close to MITIGATIONS
   NEIGHBOR_SCORE:= score of NEIGHBOR
   if NEIGHBOR_SCORE > SCORE
       MITIGATIONS:= NEIGHBOR
       SCORE:= NEIGHBOR_SCORE
   else if prob(SCORE, NEIGHBOR_SCORE, TIME, temp(TIME, MAX_TIME)) > RANDOM)
       MITIGATIONS:= NEIGHBOR
       SCORE:= NEIGHBOR_SCORE
   TIME++
end while
return MITIGATIONS
```

Fig. 7. Pseudocode for SimpleSA

For each round, SimpleSA “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, SimpleSA 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:

$$prob(w, x, y, temp(y, z)) = e^{((w-x)*\frac{y}{temp(y,z)})} \tag{2}$$

$$temp(y, z) = \frac{(z-y)}{z} \tag{3}$$

If the value of the `prob` function is greater than a randomly generated number, SimpleSA will move to that state anyways. This randomness is the very cornerstone of the Simulated 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 7, the algorithm will continue to operate until the number of tries is exhausted or a score meets the threshold requirement.

4.2 MaxFunWalk

The design of simulated annealing dates back to the 1950s. In order to benchmark our new search engine (KEYS) against a more recent 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 [52]. MaxWalkSat is a variant of WalkSat that applies weights to each clause in a conjunctive normal form equation [89]. 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.

We have adapted MaxWalkSat to the DDP problem as follows. Our MaxFunWalk algorithm is a variant of MaxWalkSat:

- Like MaxWalkSat, MaxFunWalk makes decisions about settings to variables.
- Unlike MaxWalkSat, MaxFunWalk scores those decisions by passing those settings to some function. In the case of DDP optimization, that function is the scoring function of Equation 1.

```
Procedure MaxFunWalk
for TRIES:=1 to MAX-TRIES
   SELECTION:=A randomly generate assignment of mitigations
   for CHANGED:=1 to MAX-CHANGES
       if SCORE satisfies THRESHOLD return
       CHOSEN:= a random selection of mitigations from SELECTION
       with probability P
           flip a random setting in CHOSEN
       with probability (P-1)
           flip a setting in CHOSEN that maximizes SCORE
   end for
end for
return BESTSCORE
```

Fig. 8. Pseudocode for MaxFunWalk

The MaxFunWalk procedure is shown in Figure 8. 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 subset C.
– P$> \alpha$: Local search is utilized. Each mitigation in C is tested until one is found that improves the current score.

The best setting of α is a domain-specific engineering decision. 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.

4.3 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 [42]. The algorithm has been proven to be optimal for a given scoring heuristic [21], and has seen widespread use in multiple fields [45, 79, 84, 94].

A* is a natural choice for DDP optimization since the scoring function described above is actually a Euclidean distance measure to the desired goal of maximum attainment and minimum costs. Hence, for H, we can make direct use of Equation 1.

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 determined by the model). For DDP models, we use Equation 1 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

```
Procedure ASTAR
CURRENT_POSITION:= Starting assignment of mitigations
CLOSED[0]:= Add starting position to closed list

while END:= false
   NEIGHBOR_LIST:=list of neighbors
   for each NEIGHBOR in NEIGHBOR_LIST
       if NEIGHBOR is not in CLOSED
           G:=distance from start
           H:=distance to goal
           F:=G+H
           if F<BEST_F
               BEST_NEIGHBOR:=NEIGHBOR
   end for
   CURRENT_POSITION:= BEST_NEIGHBOR
   CLOSED[++]:=Add new state to closed list
   if STUCK
       END:= true
end while
return CURRENT_POSITION
```

Fig. 9. Pseudocode for ASTAR

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.

4.4 KEYS

A*, MaxWalkSat, and simulated annealing are standard search engines. In our study, we benchmarked the performance of these standard search engines against our new method (KEYS) [69].

The premise of KEYS is that within the space of possible decisions, there exist a small number of key decisions that determine all others [68]. As discussed in the related work section, there is much evidence that keys can be found in many domains.

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.

What are "keys"? Before explaining the KEYS algorithm, we must first explore the general concept of "keys".

Within a model, there are chains of *reasons* that link inputs to the desired goals. As one might imagine, some of the links in the chain clash with others. Some of those clashes are most upstream; they are not dependent on other clashes. In the following chains of reasoning the clashes are $\{e, \neg e\}$, $\{g, \neg g\}$ & $\{j, \neg j\}$; the most upstream clashes are $\{e\neg e\}$, & $\{g\neg g\}$,

$$\begin{tikzcd}
 & a \arrow[r] & b \arrow[r] & c \arrow[r] & d \arrow[r] & e & \\
input_1 \arrow[r] & f \arrow[r] & g \arrow[r] & h \arrow[r] & i \arrow[r] & j \arrow[r] & goal \\
input_2 \arrow[r] & k \arrow[r] & \neg g \arrow[r] & l \arrow[r] & m \arrow[r] & \neg j \arrow[r] & goal \\
 & n \arrow[r] & o \arrow[r] & p \arrow[r] & q \arrow[r] & \neg e &
\end{tikzcd}$$

In order to optimize decision making about this model, we must first decide about these *upstream clashing reasons*. We refer to these decisions as *the collars* as they have the most impact on the rest of the model.

Returning to the above reasoning chains, any of $\{a, b, ..q\}$ are subject to discussion. However, most of this model is completely irrelevant to the task of $input_i \vdash goal$. For example, the $\{e, \neg e\}$ clash is unimportant to the decision making process as no reason uses e or $\neg e$. In the context of reaching $goal$ from $input_i$, the only important discussions are the clashes $\{g, \neg g, j, \neg j\}$. Further, since $\{j, \neg j\}$ are dependent on $\{g, \neg g\}$, then the core decision must be about variable g with two disputed values: true and false.

We call g the *collar* since it restricts the entire model. The collar may be internal to a model and may not be directly controllable. We refer to the controllable variables that can influence the collar as the $keys$ In the previous example, those keys are $input_i$.

Using the keys to set the $collars$ reduces the number of reachable states within the model. Formally, the reachable states reduce to the cross-product of all of the ranges of the collars. We call this the $clumping$ effect. Only a small fraction of the possible states are actually reachable. The effects of clumping can be quite dramatic. Without knowledge of these chains and the collar, the above model has $2^{20} > 1,000,000$ possible consistent states. However, in the context of $input_i \vdash goal$, those 1,000,000 states clumps to just the following two states: $\{input_1, f, g, h, i, j, goal\}$ or $\{input_2, k, \neg g, l, m, \neg j, goal\}$.

The KEYS Algorithm There are two main components to KEYS - 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 [69], the greedy search fixes one variable per era. This paper experimented with a newer version, called KEY2, that fixes an increasing number of variables as the search progresses (see below for details).

For both KEYS and KEYS2, each era e generates a set $< input, score >$ as follows:

```
1: $MaxTries$ times repeat:
     – $Selected[1 \ldots (e-1)]$ 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 $\beta\%$ "best" while the remainder are sent to
   "rest".
3: The mitigation values in the $input$ sets 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 10.

```
Procedure KEYS
while FIXED_MITIGATIONS != TOTAL_MITIGATIONS
   for I:=1 to 100
      SELECTED[1...(I-1)] = best decisions up to this step
      GUESSED = random settings to the remaining mitigations
      INPUT = SELECTED + GUESSED
      SCORES= SCORE(INPUT)
   end for
   for J:=1 to NUM_MITIGATIONS_TO_SET
      TOP_MITIGATION = BORE(SCORES)
      SELECTED[FIXED_MITIGATIONS++] = TOP_MITIGATION
   end for
end while
return SELECTED
```

Fig. 10. Pseudocode for KEYS

KEYS ranks mitigations by combining a novel support-based Bayesian ranking measure. BORE [16] (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 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:

$$\begin{aligned}
E &= (mitigation31 = false) \\
P(best) &= 1000/10000 = 0.1 \\
P(rest) &= 9000/10000 = 0.9 \\
freq(E|best) &= 10/1000 = 0.01 \\
freq(E|rest) &= 5/9000 = 0.00056 \\
like(best|E) &= freq(E|best) \cdot P(best) = 0.001 \\
like(rest|E) &= freq(E|rest) \cdot P(rest) = 0.000504 \\
P(best|E) &= \frac{like(best|E)}{like(best|E) + like(rest|E)} = 0.66
\end{aligned} \tag{4}$$

Previously [16], 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 4 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

$$P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \tag{5}$$

KEYS2 For each era, KEYS samples the DDP models and fixes the top $N = 1$ settings. The following observation suggested that $N = 1$ is, perhaps, an overly conservative search policy.

At least for the DDP models, we have observed that 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 10);
– Other variants of KEYS assigns larger values.

We experimented with several variants before settling on the following: in this variant, era i sets i settings. We call this variant "KEYS2". 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.

4.5 Scoring Costs

Return now to the issue of scoring *full* versus *partial* solutions, we note that:

– At each era, KEYS and KEYS2 generate partial solutions that fix $1, 2, 3, ..|V|$ variables (where V is the set of variables).
– Simulated Annealing, MaxWalkSat, and MaxFunWalk work with full solutions since, at each step, they offer settings to all $v \in V$ variables.
– ASTAR could produce either full or partial solutions. If implemented as a *full* solution generator, algorithm, each member of the closed list will contain a solution with fixed settings for all $v \in V$ variables. If implemented as a *partial* solution generator, then the open list will contain the current list of *free* variables and each step of the search will *fix* one more setting.

As mentioned above, such a partial solution generator will require N calls to the model in order to score the current solution; for example, KEYS and KEYS2 calls the models 100 times in each era. Since our concern is with runtimes, for this study, we used a full solution generator for ASTAR. Later in this paper, we will further discuss this implementation choice.

5 Results

Each of the above algorithms was tested on the five models of Figure 6. Final attainment and costs, runtimes, and (for KEYS/KEYS2), the convergence towards the final result were recorded. That information is shown below.

Note that:

- Models one and three are trivially small and we used them to debug our code. 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 [70] in detail. The largest, model 5 was optimized previously in [30]. As mentioned in the introduction, on contemporary machines, model 5 takes 300 seconds to optimize using our old, very slow, rule learning method.

5.1 Attainment and Costs

We ran each 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.

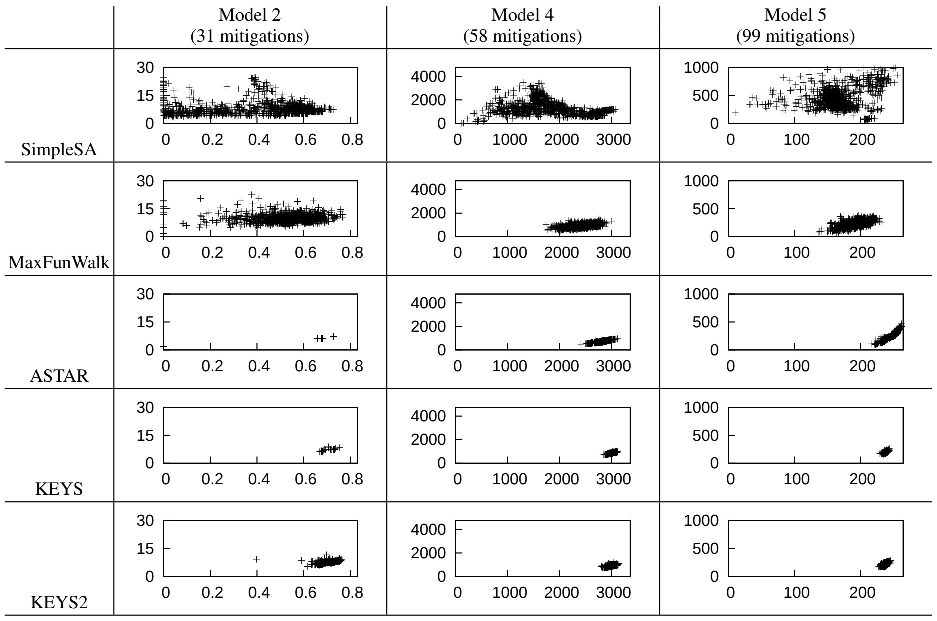

Fig. 11. 1000 results of running four algorithms on three models (12,000 runs in all). The y-axis shows cost and the x-axis shows attainment. The size of each model is measured in number of mitigations. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also better solutions exhibit less variance, i.e. are clumped tighter together.

The results are pictured in Figure 11. 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, i.e. are clumped tighter together.

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

- SimpleSA exhibit the worst variance, lowest attainments, and highest costs.
- MaxFunWalk is better than SimpleSA (less variance, lower costs, higher attainment) but it is clearly inferior to the other methods.

As to the others:

- On smaller models such as model2, ASTAR produces higher attainments and lower variance than the KEYS algorithms. However, this advantage disappears on the larger models.
- On larger models such as model4 and model5 KEYS and KEYS2 exhibit lower variance, lower costs, higher attainments than ASTAR.

	Model 2 (31 mitigations)	Model 4 (58 mitigations)	Model 5 (99 mitigations)
SimpleSA	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. 12. 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 6).

5.2 Runtime Analysis

Measured in terms of attainment and cost, there is little difference between KEYS and KEYS2. However, as shown by Figure 12, KEYS2 runs twice to three times as fast as KEYS. Interestingly, Figure 12 ranks two of the algorithms in a similar order to Figure 11:

- SimpleSA 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 in terms of runtimes, there is little to recommend KEYS2 over ASTAR. However, the two KEYS algorithms offer certain functionality which, if replicated in ASTAR, would make that algorithm much slower. To understand that functionality, we return to the issue of solution *stability* within the *full* versus *partial* solutions discussed in §3.2.

5.3 Stability and Partial Solutions

Software projects differ in many ways including the extent to which managers can control and monitor the development process:

- In small in-house projects, it may be possible to rigorously control and monitor all process options.
- In larger projects where much of the development is performed by contractors and sub-contractors (and sub-sub-contractors), it may be very difficult to monitor and control all process options. For example, in government contracts, it is common for contractors to minimize the amount of contact government personnel have with the contractors. Such contractors rigorously define a minimum set of monitoring and control points. Once enabled, it can be difficult (perhaps even impossible) to change that list.
- In any project, there is some expense associated with changing the current organization of a project.

Consequently, when discussing early life cycle requirements mitigations, it is useful to discover the *minimum* set of mitigations that most reduce cost and increase requirements attainment:

- Smaller solutions make fewer changes to a project than larger solutions and so may be cheaper and quicker to implement.
- For larger projects involving sub-contractors, such minimal sets can be used to define a contracts most informative set of monitoring and control points.

For all the above reasons, we prefer search engines that generate solutions recommending a subset of the available options. If a search engine recommends 10 out of the (e.g.) 90 possible mitigations then we need to know the stability of the costs/attainments associated with that recommendation. This is particularly true with discrete models like DDP since such models may be *brittle*; i.e. a single change may push the model into a new part of the state space with radically different properties.

KEYS and KEYS2 offer partial solutions and stability guarantees. Figure 13 shows the effects of the decisions made by KEYS and KEYS2. For KEYS and KEYS2, 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 sets one or more at a time). The lines in each of these plots show the median and spread seen in the 100 calls to the `model` function during each round. We define median as the 50th percentile and spread (the measure of deviation around the median) as (75th percentile - median).

The plots for KEYS and KEYS2 are nearly identical:

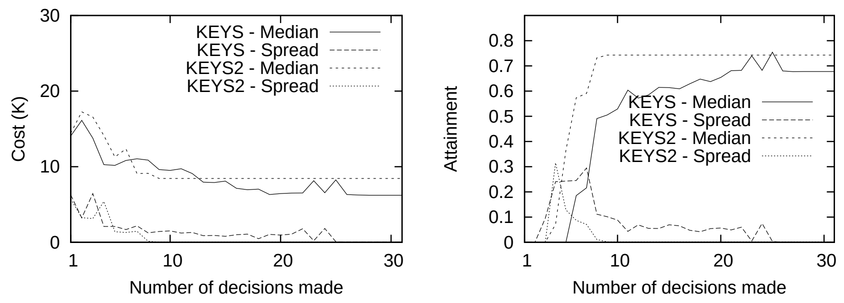

Figure 13a: Internal Decisions on Model 2.

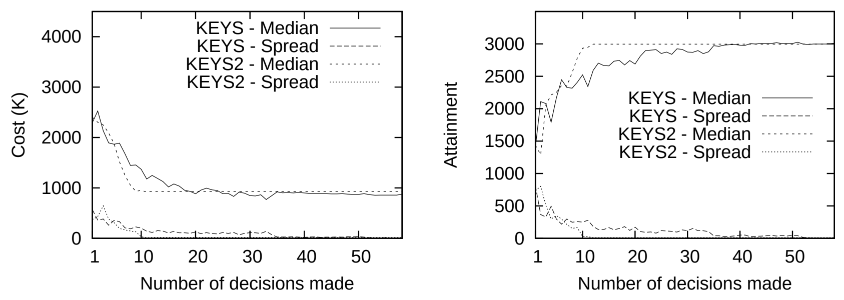

Figure 13b: Internal Decisions on Model 4.

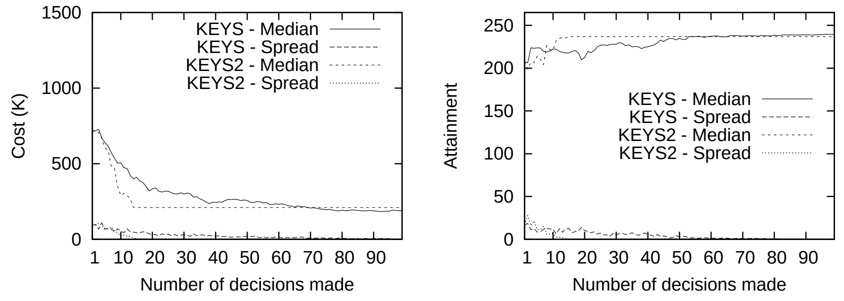

Figure 13c: Internal Decisions on Model 5.

Fig. 13. 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 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).

Since these results are so similar, and KEYS2 runs faster than KEYS, we recommend KEYS2 over KEYS.

Note that, for both KEYS and KEYS2, the cost improvements are dramatic for about 20 rounds, after which they tend to stabilize. Interestingly, as we can see in the plots, the spread is generally quite small compared to the median, particularly after 20 decisions. This indicates that our median estimates are good descriptions of the tendencies of the models. Software development managers can use Figure 13 to explore trade offs in how they adjust their projects. If they lack sufficient resources to implement all of KEYS2's recommendations, they can consider the merits of partial solutions. For example, in model4 and model5, most of the improvement comes making just 20 decisions. Hence, for these models, managers might decide to implement a succinct partial solution (i.e. all the recommendations $1 \leq x \leq 20$).

The version of ASTAR used in this study generated full solutions. That is, a manager seeking parsimonious solutions would not be supported by ASTAR. As mentioned above (end of §3), ASTAR could be modified to produce partial solutions but this would increase the runtimes of that algorithm since it would require multiple calls to the scoring function.

6 Related Work

§2 discussed the connection of DDP to other early life cycle modeling tools and §4 discussed KEYS in the context of alternative search engines. The rest of this section discusses other related work.

6.1 Qualitative Compartmental Models

Historically, this work began with Feldman & Compton [31, 32] who studied the validation of topoi (statements of gradual effects between model variables [23]), which they called *qualitative compartmental models*. Menzies worked on the optimization of their validation process and offered an implementation that was orders of magnitude faster than the system built by Feldman & Compton. However, he could not reduce the exponential upper-bound on the runtimes [62–64].

Assuming a certain restriction on topoi edge types, Cohen, Menzies, Waugh and Goss showed that the cost of checking temporal properties of a topoi-based simulation is bounded by a function of the number of time-ticks in the query [66, 67]. However, those restrictions are quite onerous. In this paper, we left the model in whatever form the experts wanted to use, and explored AI search engines for optimization.

6.2 Keys

KEYS and KEYS2 were designed under the assumption that that the behavior of a large system is determined by a very small number of *collar* variables. Recalling the above discussion in §4.4, we said that controllable inputs that effect the collars are called *keys*. For a system with collars, the state space within the model *clumps* to a small fraction of the possible reachable states.

Our reading of the literature is that $keys, collars, clumps$ have been discovered and rediscovered many times. Elsewhere [71], we have documented dozens of papers that have reported this effect under different names including *narrows, master-variables, back doors,* and *feature subset selection*:

- Amarel [5] 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.
- Druzdel [27] represented a model as the product of distributions of the model's variables. Given enough variance in the individual priors and conditional probabilities of the variables, the frequency of model states will exhibit a log-normal distribution. Such a system would show the clumping effect; a small fraction of the states would cover a large portion of the total probability space, with the remaining states being negligible. He studied one piece of software where the reached states were a vanishingly small fraction of the set of possible states. In one diagnosis system with a total of 525,312 possible internal states, he found that there were a total of 49 states that were reached 91% of the time and only one state was ever reached 52% of the time.
- In a similar theoretical analysis, Menzies & Singh [71] 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 [55] 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. [97] 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 [19] 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 KEYS and KEYS2 are supported in many domains. Hence, we hope in the future to apply KEYS and KEYS2 to more areas than just DDP-like models.

6.3 Model-Driven Software Engineering

DDP is a model-driven requirements engineering tool. Within the field of model-driven SE, DDP is a comparatively ultra-lightweight modeling approach; i.e. atomic propositions are divided into risks/ mitigations/ requirements and connected together as an acyclic forest. This section offers some brief notes on the boarder field of model-drive SE.

Anthropologists have argued that the ability to build abstract models is what gives homo sapiens their competitive edge. For almost as long as software engineering has existed, software engineers have developed models in order to gain insight about a concept. In his article, "What Models Mean" [87], Seidewitz describes the interactions and relationships between the concepts of a model. He asserts that "a model's meaning has two aspects: the model's relationship to what's being modeled and to other models derivable from it. Carefully considering both aspects can help us to understand how to use models to reason about the systems we build...". These are the aspects of a model's meaning that are focused on in *model-based* software engineering.

Model-based Software Engineering is becoming increasingly important. The Object Management Group (OMG) has recently adopted a *model-driven architecture* framework with goals of "portability, interoperability, and reusability through architectural separation of concerns." [77] OMG uses the established standards of UML, the Common Warehouse Metamodel, and the Meat-Object Factory to provide system interoperability in a vendor-neutral form [12]. Microsoft has been developing a *Software Factory* that utilizes models in order to automate the software development process [37]. At Lockheed Martin, engineers have developed an integrated modeling method called *Model Centric Software Development* that uses "automated generation of partial implementation artifacts," reverse engineering to integrate legacy assets, and model verification and checking" [96]

The importance of the model-driven approach is not limited to software design and UML. Many tools exist for modeling, some examples include distributed agent-based simulations [15], discrete-event simulations [41, 53, 57, 80, 81], continuous simulation (also called system dynamics) [1, 93], state-based simulation (including petri net and data flow approaches) [4, 38, 61], hybrid-simulation (combining discrete event simulation and systems dynamics) [26,60,91], logic-based and qualitative-based methods [11, chapter 20] [46], and rule-based simulations [73].

One can find these models being used in the requirements phase(i.e. the DDP tool described earlier), design refactoring using patterns [34], software integration [22], model-based security [49], and performance assessment [7].

Models are useful because humans can review, audit, and improve an explicit representation of their systems. However, as models grow in complexity, it becomes difficult for a manual analysis to reveal all their subtleties. Hence, many researchers have proposed support environments to help explore the increasingly complex models that engineers are developing. Gray et al [36] have developed the Constraint-Specification

Aspect Weaver (C-Saw), which uses aspect-oriented approaches [33] to help engineers in the process of model transformation. Cai and Sullivan [13] 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 [48] and SCR [43] as well as various UML tools that allow for the execution of life cycle specifications (e.g. the CADENA scenario editor [14]).

Algorithms like KEYS and KEYS2 are candidate tools for exploring and optimizing SE models, providing that there there exists some automatic oracle that can comment on the output of a particular run. Such a combination of model-based SE plus an AI optimizer would be an example of search-based software engineering.

6.4 Search-based Software Engineering

Search-based Software Engineering augments model-based SE with *meta-heuristic* techniques like genetic algorithms, simulated annealing, and others to explore a model. Such heuristic methods are not 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." [17]

Search-based SE is most often used to optimize software testing [50, 51, 78, 95], but it has had applications in numerous other areas. We have used search-based SE for requirements analysis [29, 30, 69]. Other researchers [39, 58] use genetic algorithms to examine ways of modularizing software [17] or develop effort estimators [2, 24, 25]. In all, Rela [82] lists 123 publications where search-based methods have been applied to the above applications as well as others such as automatic synthesis of software defect predictors, assisting in component design, developing multiprocessor schedules, re-engineering old systems into better ones, and searching for compiler optimizations.

To use a search-based approach, software engineers have to reformulate their problem by:

- Finding a *representation of the problem* that can be symbolically manipulated (e.g. simulated or mutated). Such representations always exist with model-based SE. In our work, DDP is the representation of the problem.
- Defining a *fitness function* (a.k.a. "utility function" or "objective function"); i.e. an "oracle" that scores a model configuration. Current model-based SE methods rarely offer such a function (exception: formal methods that generate temporal constraints). In our experience, generating such a fitness function is usually possible, albeit requiring days of work with the domain experts [65]. Our fitness function is the calculation of cost and attainment from Equation 1.
- Determining an appropriate set of *manipulation operators* to select future searches based on the prior searches [40]. Our manipulation operators are the various algorithms that we employ such as SimpleSA, ASTAR, MaxFunWalk, KEYS, KEYS2.

Inspired by the search-based SE literature, our initial experiments with requirements optimization used simulated annealing. As shown above, this simulated annealing algorithm is out-performed by every other algorithm studied here. Perhaps researchers in search-based SE might find it advantageous to look beyond standard search algorithms (even supposedly state-of-the-art algorithms like MaxWalkSat).

7 Conclusion

Real-time requirements optimization is the task of generating solutions from requirements models before an expert's attention wanders to other issues. Neilson states that such solutions must be available in one to ten seconds.

DDP is a requirements/risks/mitigation tool used by groups of experts at NASA's Jet Propulsion Laboratory. Our prior work on optimizing DDP models searched for subsets of the mitigations that most reduced cost and increased requirements attainment. Using rule learning technologies, our optimizers took hundreds of seconds to run. This is far too slow since it (a) it is expected that DDP models will grow much larger in the near future; and (b) human experts often want to run the DDP models 2^W times (once for each subset of W what-if options).

In order to handle such what-if queries on larger models, a back-of-the-envelope calculation (presented in the introduction) advised that that real-time requirements optimization of current DDP models running on contemporary machines must terminate in time 10^{-2} to 10^{-3} seconds. Two other goals for real-time optimization are:

– *Partial solutions*, since software project managers may be unable or unwilling to precisely control all aspects of a project; and
– *Stability results* commenting on the brittleness of that solution.

For real-time optimization of DDP-like requirements models, we recommend knowledge compilation and KEYS2:

– Knowledge compilation speeds up optimization by one to two orders of magnitude. While this is not enough by itself to support the DDP models of the kind we expect to see in the near future, it is certainly useful when combined with AI search algorithms.
– KEYS2 runs faster than KEYS and four orders of magnitude faster than our previous rule-learning based approach (for model5, 300 seconds from rule learning [30] to 0.038 seconds for KEYS2). That is, KEYS2 is fast enough for real-time requirements optimization. We attribute much of the success of KEYS2 to the presence of $keys$ in the DDP models, those variables that control everything else. These key variables were exploited by KEYS2 using two methods: BORE finds promising keys with high probabilistic support; and KEY2's greedy search sets the most promising key, before exploring the remaining options.
– As to other search algorithms, SimpleSA and MaxFunWalk run slower than KEYS2 and result in solutions with much larger variance. Also, unlike ASTAR, KEYS and KEYS2 offer stability results for partial solutions. As more and more variables get locked down, KEYS2 finds itself unable to make a decision that could throw off the results.

8 Future Work

DDP is an example of a class of ultra-lightweight early life cycle requirements engineering frameworks. In future work, we plan to explore real-time requirements optimization using other frameworks such as soft-goals, QOC, AHP, and QFD.

As to further improvements for DDP, in terms of attainment and cost, it may be hard to improve over the KEYS2 (and KEYS) results. Observe in Figure 11 how the results from these algorithms are very close to maximum attainment and minimum cost (the lower right-hard corner of these plots). Further improvements in real-time optimization of DDP models may be restricted to issues of speed and stability.

While KEYS2 is fast enough for real-time requirements optimization of $1 \leq W \leq 10$ what-ifs over DDP-like models, further optimizations would enable the processing of more what-ifs or more complex models. For example, our current implementation runs over binary variables and many problems require variables with larger ranges.

The stochastic nature of KEYS2 lends itself naturally to optimization by parallelism. We are currently exploring grid computing methods, with each node in the grid running its own version of KEYS2. In this scheme, at random time intervals, results are dispatched from the nodes to a centralized learner. This learner reports back to the grid on how to best bias future simulations. Preliminary analysis suggests that this approach may be quite promising.

Another possible project would be to improve ASTAR. If ASTAR could be given the stability guarantee of KEYS while maintaining its runtime advantage, we could potentially address far more complex models. For example, we could enhance ASTAR with the BORE heuristic found in KEYS2; e.g. BORE could be used as a sub-routine that biases ASTAR's selection of where to search next.

More generally, KEYS2 could be a useful sub-routine for improving the random selections made by other algorithms. Stochastic SAT solvers such as MaxWalkSat or the Markov Logic reasoning within ALCHEMY [83] could be biased by KEYS2 to find key variables within the CNF clauses. The effects of such an optimization could be dramatic- recall from the above discussion that Williams et.al. [97] reported that setting "back doors" (the keys) can reduce the solution time of hard problems from exponential time to polytime.

9 Acknowledgments

This research was conducted at West Virginia University, the Jet Propulsion Laboratory under a contract with the National Aeronautics and Space administration, 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. J. Aguilar-Ruiz, I. Ramos, J. Riquelme, and M. Toro. An evolutionary approach to estimating software development projects. *Information and Software Technology*, 43(14):875–882, December 2001.
3. Y. Akao. *Quality Function Deployment*. Productivity Press, Cambridge, Massachusetts, 1990.
4. 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.
5. 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.
6. b. hailpern and p. tarr. model-driven develpment: the good, the bad, and the ugly. *ibm systems journal*, 45(3):451–461, 2006.
7. 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.
8. 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.
9. B. Boehm. A spiral model of software development and enhancement. *Software Engineering Notes*, 11(4):22, 1986.
10. B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Trans. on Software Engineering*, 14(10):1462–1477, October 1988.
11. I. Bratko. *Prolog Programming for Artificial Intelligence. (third edition)*. Addison-Wesley, 2001.
12. A. Brown, S. Iyengar, and S. Johnston. A rational approach to model-driven development. *IBM Systems Journal*, 45(3):463–480, 2006.
13. 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.
14. 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`.
15. W. Clancey, P. Sachs, M. Sierhuis, and R. van Hoof. Brahms: Simulating practice for work systems design. In P. Compton, R. Mizoguchi, H. Motoda, and T. Menzies, editors, *Proceedings PKAW '96: Pacific Knowledge Acquisition Workshop*. Department of Artificial Intelligence, 1996.
16. R. Clark. Faster treatment learning, Computer Science, Portland State University. Master's thesis, 2005.
17. 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.
18. 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.
19. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.
20. 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`.

21. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality af a*. *J. ACM*, 32(3):505–536, 1985.
22. 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.
23. R. Dieng, O. Corby, and S. Lapalut. Acquisition and exploitation of gradual knowledge. *International Journal of Human-Computer Studies*, 42:465–499, 1995.
24. J. J. Dolado. A validation of the component-based method for software size estimation. *IEEE Transactions of Software Engineering*, 26(10):1006–1021, 2000.
25. J. J. Dolado. On the problem of the software cost function. *Information and Software Technology*, 43:61–72, 2001.
26. P. Donzelli and G. Iazeolla. Hybrid simulation modelling of the software process. *Journal of Systems and Software*, 59(3), December 2001.
27. M. Druzdzel. Some properties of joint probability distributions. In *Proceedings of the Tenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-94)*, pages 187–194, 1994. Available from `http://www.pitt.edu/AFShome/d/r/druzdzel/public/html/abstracts/uai94.ht%ml`.
28. 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`.
29. M. Feather and S. Cornfordi. Quantitative risk-based requirements reasoning. *Requirements Engineering Journal*, 8(4):248–265, 2003.
30. 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`.
31. B. Feldman, P. Compton, and G. Smythe. Hypothesis Testing: an Appropriate Task for Knowledge-Based Systems. In *4th AAAI-Sponsored Knowledge Acquisition for Knowledge-based Systems Workshop Banff, Canada*, 1989.
32. B. Feldman, P. Compton, and G. Smythe. Towards Hypothesis Testing: Justin, Prototype System Using Justification in Context. In *Proceedings of the Joint Australian Conference on Artificial Intelligence, AI '89*, pages 319–331, 1989.
33. R. E. Filman. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.
34. 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.
35. V. Goel. "ill-structured diagrams" for ill-structured problems. In *Proceedings of the AAAI Symposium on Diagrammatic Reasoning Stanford University, March 25-27*, pages 66–71, 1992.
36. J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering. *IEEE Computer*, 39(2):51–58, February 2006.
37. J. Greenfield and K. Short. *Software factories : assembling applications with patterns, models, frameworks, and tools*. Wiley Publishing, Indianapolis, IN, 2004.
38. D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
39. M. Harman, R. Hierons, and M. Proctor. A new representation and crossover operator for search-based optimization of software modularization. In *GECO 2002: Proceedings of the Genetic and Evolutionary Computation Conference*, pages 1351–1358. Morgan Kaufmann, July 2002.
40. M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
41. H. Harrell, L. Ghosh, and S. Bowden. *Simulation Using ProModel*. McGraw-Hill, 2000.

42. 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.
43. 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`.
44. M. Hirakawa and T. Ichikawa. Visual language studies - a perspective. *Software- Concepts and Tools*, pages 61–67, 1994.
45. 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.
46. 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.
47. M. G. C. H. S. H. J. K. M. L. T. S. T. S. M. S. J. Falk, J. Gladigau and J. Teich. System-codesigner the system-level hardware-software-co-design tool. In *Proceedings of Design Automation and Test in Europe*, April 2007.
48. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
49. 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.
50. B. Jones, D. Eyres, and H.-H. Sthamer. A strategy for using genetic algorithms to automate branch and fault-based testing. *Computer Journal*, 41(2):98–107, 1998.
51. B. Jones, H.-H. Sthamer, and D. Eyres. Automatic structural tsting using genetic algorithms. *Software Engineering Journal*, 11:299–306, 1996.
52. 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`.
53. D. Kelton, R. Sadowski, and D. Sadowski. *Simulation with Arena, second edition*. McGraw-Hill, 2002.
54. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
55. R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.
56. R. Kremer. Visual languages for knowledge representation. KAW'98: Eleventh Workshop on Knowledge Acquisition, Modeling and Management, Voyager Inn, Banff, Alberta, Canada, 1998. To appear.
57. A. Law and B. Kelton. *Simulation Modeling and Analysis*. McGraw Hill, 2000.
58. R. Lutz. Evolving good hierarchical decomposition of complex systems. *Journal of Systems Architecture*, 47:613–634, 2001.
59. 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.
60. 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.
61. 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.
62. T. Menzies. *Principles for Generalised Testing of Knowledge Bases*. PhD thesis, University of New South Wales, 1995. Ph.D. thesis. Available from `http://menzies.us/pdf/95thesis.pdf`.

63. T. Menzies. Applications of abduction: Knowledge level modeling. *International Journal of Human Computer Studies*, 45:305–355, 1996. Available from `http://menzies.us/pdf/96abkl.pdf`.
64. T. Menzies. On the practicality of abductive validation. In *ECAI '96*, 1996. Available from `http:/menzies.us/pdf/96ok.pdf`.
65. T. Menzies. Critical success metrics: Evaluation at the business-level. *International Journal of Human-Computer Studies, special issue on evaluation of KE techniques*, 51(4):783–799, October 1999. Available from `http://menzies.us/pdf/99csm.pdf`.
66. T. Menzies and R. Cohen. A graph-theoretic optimisation of temporal abductive validation. In *European Symposium on the Validation and Verification of Knowledge Based Systems, Leuven, Belgium*, 1997. Available from `http://menzies.us/pdf/97eurvav.pdf`.
67. T. Menzies, R. Cohen, S. Waugh, and S. Goss. Applications of abduction: Testing very long qualitative simulations. *IEEE Transactions of Data and Knowledge Engineering*, pages 1362–1375, November/December 2003. Available from `http://menzies.us/pdf/97iedge.pdf`.
68. T. Menzies, D.Owen, and J. Richardson. The strangest thing about software. *IEEE Computer*, 2007. `http://menzies.us/pdf/07strange.pdf`.
69. T. Menzies, O. Jalali, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings PROMISE '08 (ICSE)*, 2008.
70. 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`.
71. 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`.
72. N. Metropolis, A. Rosenbluth, M. Rosenbluth, A. Teller, and E. Teller. *J. Chem. Phys*, 21:1087–1092, 1953.
73. 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.
74. 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.
75. 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.
76. J. Nielson. *Usability Engineering*. Academic Press, 1993.
77. Object Management Group. *MDA Guide Version 1.0.1*, June 2003.
78. R. Pargas, M. Harrold, and R. R. Peck. Test-data generation using genetic algorithms. *Journal of Software Testing, Verification and Reliability*, 9:263–282, 1999.
79. J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
80. D. Raffo and T. Menzies. Evaluating the impact of a new technology using simulation: The case for mining software repositories. In *Proceedings of the 6th International Workshop on Software Process Simulation Modeling (ProSim'05)*, 2005.
81. D. Raffo and T. Menzies. Software project management using prompt: A hybrid metrics, modeling and utility framework. *Journal of Information, Software and Technology*, 47(15):1009–1017, December 2005.
82. L. Rela. Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, 2004.

83. M. Richardson and P. Domingos. Markov logic networks, February 206.
84. 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.
85. B. B. H. E. P. G. e. S. Biffl, A. Aurum. *Value-Based Software Engineering*. Springer, 2005.
86. T. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill, 1980.
87. E. Seidewitz. What models mean. *IEEE Software*, 20(5):26–32, Sept.-Oct. 2003.
88. 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`.
89. 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.
90. S. Sendall and W. Kozacaynski. Model transformation: The heart and soul of model-driven software development. *IEEE Software*, 20(5):42–45, Sept.-Oct. 2003.
91. S. Setamanit, W. Wakeland, and D.Raffo. Using simulation to evaluate global software development task allocation strategies. *Software Process: Improvement and Practice, (Forthcoming)*, 2007.
92. 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.
93. H. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill, 2000.
94. B. Stout. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, (7), 1997.
95. N. Tracey, J. Clarke, and K. Mander. Automated program flaw finding using simulated annealing. In *International Symposium on Software Testing and Analysis*, pages 73–81. ACM/SIGSOFT, March 1998.
96. D. Waddington and P. Lardieri. Model-centric software development. *IEEE Computer*, 39(2):28–29, February. 2006.
97. 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`.