Editorial Manager(tm) for Automated Software Engineering
Manuscript Draft

Manuscript Number: AUSE117

Title: Real-time Optimization of Requirements Models

Article Type: Manuscript

Keywords: requirements engineering, optimization, search

Corresponding Author: Mr. Tim Menzies,

Corresponding Author's Institution:

First Author: Gregory Gay

Order of Authors: Gregory Gay; Tim Menzies; Omid Jalali; Martin Feather; James Kiper

Manuscript
Click here to download Manuscript: paper-v4.pdf Click here to view linked References

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. As requirements models grow larger, so to does the need for faster requirements optimization methods, particularly when 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 KEYS2. KEYS2 uses a very simple (hence, very fast) 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

The room is crowded and everyone is talking at once. Time is limited: the people in that room are needed elsewhere, in just a few hours time. While the facilitator tries to guide the group to consensus, your PDA rings: its a text message from a friend across the room. She is suggesting an innovative redesign of the file system. You agree that it is a good idea and the two of you base all your future discussion on that redesign. Meanwhile, across the room, someone else's PDA rings and, unknown to you, two other people are agreeing on a contradictory change to the file system. This design clash is not detected till much later in the project, at which point it becomes very expensive to realign separate parts of the design based on conflicting assumptions.

This scenario is not imaginary. For example, at the "Team X" meetings at NASA's Jet Propulsion Laboratory, experts in science data collection, communication, guidance, etc to generate a "mission concept" document which may the basis of millions to billions of dollars of subsequent development work. There is no time for everyone to talk to everyone else and, all too often, experts on one side the room can make decisions that one hinder decisions *and the group is unaware of this conflict*.

This is not just a NASA problem. Rather, it is a a looming problem with all groupthink tools. As our tools grow better, and they will be used by larger groups who will

build more complex models. The problem of co-ordinating group discussions is challenging in the 21^{st} century net-enabled world where participates communicate via via multiple channels; e.g. talking to the whole room while texting some individuals across the room. In such a multiple channeled environment, small sub-groups can evolve opinions that are not shared with the rest of the group.

If miscommunication lead to inappropriate decisions, then this can be fatal to software projects. For example, it is very common for budgets to be determined after some requirements analysis. Once set, it can be difficult revise those budgets later in the project. Also, it can be very expensive to change poor decisions made early in the lifecycle. The longer a system is designed around the wrong set of assumptions, then more expensive it becomes to change those assumptions. Boehm & Papaccio advise that reworking software is far cheaper earlier in the life cycle than later "by factors of 50 to 200" [9]. Hence, when managing software projects, we must make the best decisions we can, as early as possible.

If requirements analysis is automatic and fast, then automatic methods can help a group maintain focus within multi-channeled meetings. Individuals will focus on the whole group is if the activity of the whole team commands their attention. For example, suppose a requirements analyzer finds a major problem or a novel better solution. Such a result would command the attention of the whole group, in which case everyone in the room would interrupt their current deliberations to focus on the new finding.

A premise of this approach is that requirements analyzers offer feedback on their models in "real time"; i.e. before an expert's attention wanders to other issues. Neilson [61, 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."

In this paper, we split the difference and demand five second response time for real-time requirements analysis.

This is a challenging problem. Based on current projections from JPL, we predict that by 2013, our requirements models will be ten times larger than today and contain nearly $N \approx 1000$ variables. This is an alarming growth rate since:

- Requirements optimization is often a non-linear problem where possible benefits must be traded off against the increase cost of implementing those benefits;
- In such non-linear problems, the number of combined options to consider in a model of size N can grow exponentially on model size to 2^N.
- Based on our JPL experience, we find that experts often explore their models in the context of W what-if scenarios where combinations, these 2^N options must be studied in the context of 2^W possible assumptions. That is, the space of options to explore is $2^W * 2^N$.
- Assuming that users explore just a handful of what-if queries (say, $W = 10$) models of size $N \leq 1000$, then the total search space to be explored in five seconds (or less) is $\geq 10^{300}$.

Extrapolating forwards in time, we can generate an estimate of how fast we need to be processing our *current models* on our *current hardware* in order to explore the models we expect to see in five years time. The appendix of this paper offers a "back of the envelope" calculation for that extrapolation. Based on that calculation, we assert that our current models must be terminating in 10^{-2} seconds if we wish to support five second response times for the kinds of real-time requirements problems we expect to see in the near future.

This paper shows that, at least for the case studies we explore here, that three techniques let us achieve runtimes of 10^{-2}. This is a significant improvement (10,000 times faster) than our prior results in this area [27] that used simulated annealing [47]. The three techniques are:

1. Like many before us [2, 19, 25, 50, 59, 60, 66, 70], we advocate the use of *ultra-lightweight requirement models*. Such ultra-lightweight models are the preferred options for very early life cycle models when the details required for more elaborate models are not yet available. Another advantage of such models is that, as shown in this article, there exists non-trivial ultra-lightweight models that can be processed fast enough to support real-time requirements engineering.
2. *Knowledge compilation* pushes some percentage of the query computational overhead into a compilation phase. This cost is amortized over time as we increase the number of on-line queries. Previous work with knowledge compilation has focused on compilation languages utilizing CNF equations, state machines, or BDD [7,21]. These forms may be needlessly complex for ultra-lightweight models. We show here that a simple procedural target language can suffice.
3. *Keys-based search:* Many models contain a small number of *key variables* that set the remaining variables [4, 20, 48, 55, 75]. For models contain keys, the complexity of the searching the entire model reduces to just the complexity of searching the key variables. The KEYS2 algorithm, proposed here, is a very fast method for finding the key variables.

1.1 Contributions of This Paper

The specific contribution is to define the problem of real-time requirements engineering. We offer materials that allow other researchers to work on this problem. All the code, Makefiles, scripts etc used in this paper are available on-line[4]. Also, we offer a baseline result. At least for the models explored in this paper, we can achieve optimizations in around 10^{-2} seconds. Such baseline results since they allow others in the field to demonstrate clear advances over (say) this paper.

A more general contribution to this paper is to demonstrate the advantages of keys-based search. Our keys-based search returns better quality solutions in faster time that other methods in widespread use (simulated annealing) or which are supposedly state of the art (MaxWalkSat). our keys-based search is remarkably simple to implement and finds solutions that constrain fewer variables that other methods (ASTAR) but which achieve similar, or slightly better scores. Also, the incremental keys-based search described in this paper offers extensive information about the space of effects *around* a

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

particular solution. Alternative search methods offer point solutions that are inappropriate when managers do not not have full control over which decisions are made, or when such control is prohibitively expensive. We hope these results motivates other researchers to experiment with the keys assumption.

Another important result from this paper is to comment on a standard search engine, used widely in the field of search-based software engineering (SBSE). Clark et al. [15] warn that simulated annealing may be quite computationally expensive. But those warning do not go further to say that, assuming simulated annealing terminates, then its results may be worse than those seen by other methods. More generally, apart from cautions on the runtime cost of simulated annealing, we are unaware of other cautionary notes on the relative value of the results of simulated annealing in the software engineering literature. Such cautions should be offered since, here, we show that the results of simulated annealing can be out-performed by a variety of alternatives search engines (some of these are quite simple to code such as KEYS). This is an exciting result since it means that current results from simulated annealing (e.g. [5,8,10,62,73]) could be greatly improved, just by switching to an alternate search engine.

2 Related Work

2.1 Other Requirements Tools

This work on real-time requirements analysis should not be interpreted as a rejection of other tools. There exist many powerful requirements analysis tools including continuous simulation (also called system dynamics) [1,71], state-based simulation (including petri net and data flow approaches) [3, 34, 52], hybrid-simulation (combining discrete event simulation and systems dynamics) [24,51,69], logic-based and qualitative-based methods [11, chapter 20] [43], and rule-based simulations [57]. One can find these models being used in the requirements phase (i.e. the DDP tool described below), design refactoring using patterns [30], software integration [23], model-based security [45], and performance assessment [6]. Many researchers have proposed support environments to help explore the increasingly complex models that engineers are developing. Gray et al [32] have developed the Constraint-Specification Aspect Weaver (C-Saw), which uses aspect-oriented approaches [29] to help engineers in the process of model transformation. Cai and Sullivan [12] 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 [44] and SCR [39] as well as various UML tools that allow for the execution of life cycle specifications (e.g. the CADENA scenario editor [13]).

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 constrain expressiveness for faster runtimes*. There exists a class of ultra-lightweight model languages which, as we show below, can be processed fast enough to support the real-time requirements engineering problem described above. 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 terminates in hundreths of a second.

2.2 Other Optimizers

As documented by the search-based SE literature [17,36,37,64] and Gu et al [33], there are many possible optimization methods. For example:

- 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 [58] 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 using integer programming methods, we do not explore them here for two reasons. Coarfa et al. [18] found that integer programming-based approaches ran two an order of magnitude slower than discrete methods like the MaxWalkSat and KEYS2 algorithm we use below. Similar results reported by Gu et.al where discrete methods ran 100 times faster than integer programming [33].

Harman offers another reason to avoid integer programming methods. In his search-based SE manifest, Harman [36] 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." [17]

2.3 Robust Solutions in SBSE

Another reason to avoid precise solutions that comment on all controllable variables is that such may be inappropriate for decision making and software projects. In many cases, managers do not or can not control all variables in a project:

- In any project, there is some expense associated with changing the current organization of a project. In this case, managers prefer smaller solutions to larger ones.

– In large projects, much of the development is performed by contractors and sub-contractors (and sub-sub-contractors). In such projects is is 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.

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 [35] comments that understanding the neighborhood our solutions is an open and pressing issue in 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."

It is trivial for our preferred method (KEYS and KEYS2) to offer robust information around partial solutions. As we shall see, such robust neighborhood information cannot be generated from other methods (e.g. simulated annealing) without extensive and time-consuming post-processing.

3 Ultra-lightweight Requirements Models

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 [49]. Other researchers take a similar view: "Pictures are superior to texts in a sense that they are abstract, instantly comprehensible, and universal." [40].

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.

Also, such further formalization may not be advisable. Goel [31] 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

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. 1. DDP's ontology

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 [59, 60]. 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 [70]. 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 to support* or *tend to reject* 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 [50]. Saaty's Analytic Hierarchy Process (AHP) [66] is a variant of QOC.
- Another widely used framework is Quality Functional Deployment diagrams (QFD) [2] Where as AHP and QOC propagate influences over hierarchies, QDF (and DDP) propagate influences over matrices.

Based on the above, Cornford and Feather designed the "Defect Detection and Prevention" (DDP) tool [19, 25]. DDP 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. Users of DDP explore combinations of mitigations that cost the least and support the most number of 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

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. 2. Sample DDP requirements, risks, mitigations.

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 1. The ontology must be lightweight 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. These sessions generate a set of requirements/ risks/ and candidate mitigations (e.g. see Figure 2) linked together into an intricate network (e.g. see Figure 3).

Sometimes, we are asked if requirements optimization 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 3, 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 3 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. Given the size of DDP models, and the rich set of connections between concepts, requirements optimization defeats manual analysis. Our task is to provide automatic (and rapid) support for this problem of *least cost, max effect mitigation selection*.

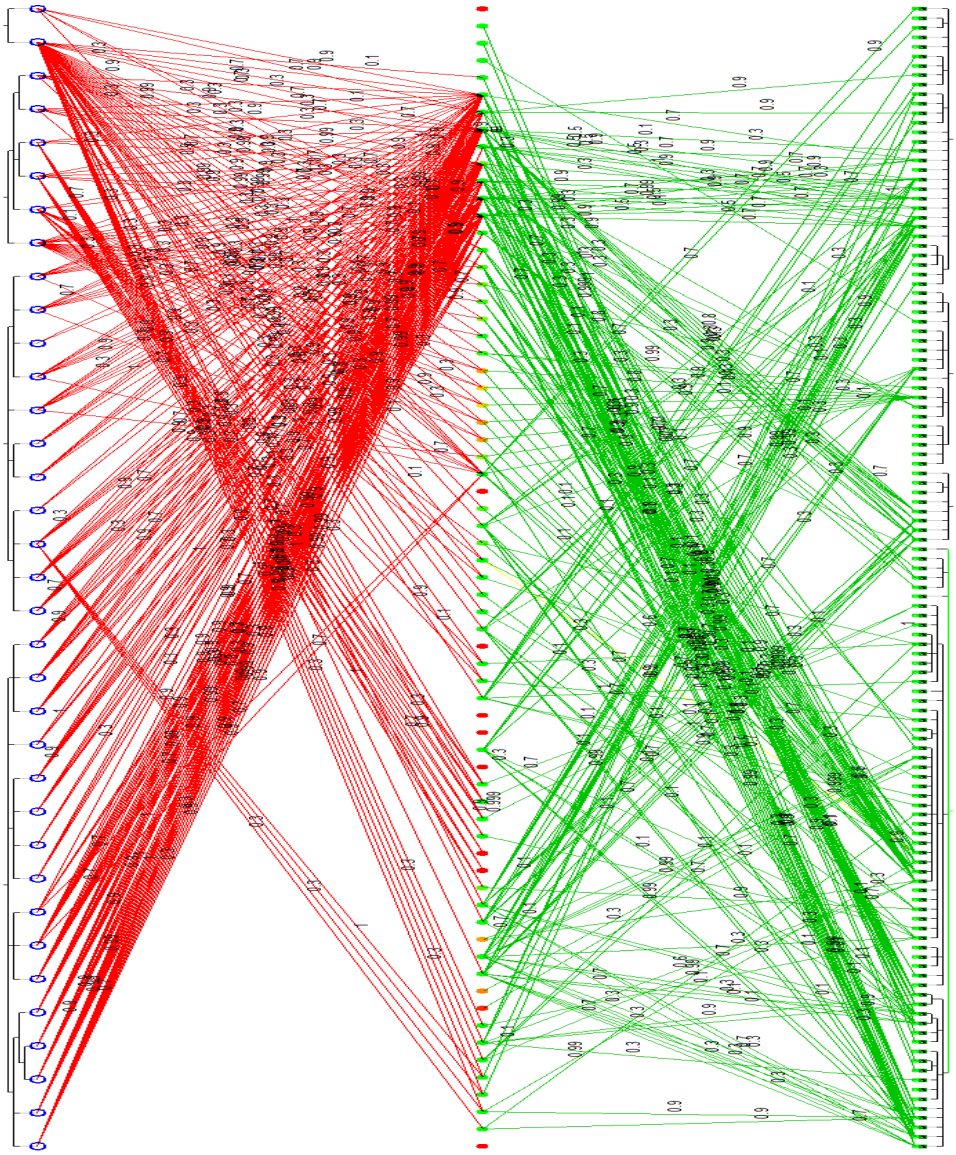

Fig. 3. An example of a model formed by the DDP tool. Red lines connect risks (middle) to requirements (left). Green lines connect mitigations (right) to the risks.

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. 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.

Our second reason to use DDP is that we can access numerous large 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 lead 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) [28];
- A study on the selection and planning of deployment of prototype software [26].

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 QOC or QFD, or a quantitative variant of Mylopoulos's qualitative soft goal graphs. In future work, we will extend our current implementation to those systems.

4 Knowledge Compilation

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 [21, 67].

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 [7, 21]. 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. 4. 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. 5. 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 4. 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 5 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^{-2} 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 4 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.

5 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:

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

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*.

The search methods presented below can be categorized many ways. However, for our purposes, the following will be insightful:

- 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 4) 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.

6 Keys-Based Search

Our proposed solution to real-time requirements optimization assumes that the behavior of a large system is determined by a very small number of *key* variables. For a system with collars, the state space within the model *keys* to a small fraction of the possible reachable states.

The following small example illustrates key variables. 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 $\{e\neg e\}$, & $\{g\neg g\}$,

$$\begin{array}{c} a \longrightarrow b \longrightarrow c \longrightarrow d \longrightarrow e \\ input_1 \longrightarrow f \longrightarrow g \longrightarrow h \longrightarrow i \longrightarrow j \longrightarrow goal \\ input_2 \longrightarrow k \rightarrow \neg g \longrightarrow l \longrightarrow m \rightarrow \neg j \longrightarrow goal \\ n \longrightarrow o \longrightarrow p \longrightarrow q \longrightarrow \neg e \end{array}$$

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 $\{input_2, k, \neg g, l, m, \neg j, goal\}$.

Our reading of the literature is that $keys, collars$ have been discovered and rediscovered many times. Elsewhere [55], 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 [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 [55] 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 [48] 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. [75] 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 [20] 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. 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.

6.1 The KEYS Algorithm, Version 1

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 [53], the greedy search fixes one variable per era. This paper experimented with a newer version, called KEYS2, 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 6.

```
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. 6. 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:

$$
\begin{aligned}
E &= (mitigation31 = false)\\
P(best) &= 1000/10000 = 0.1\\
P(rest) &= 9000/10000 = 0.9\\
freq(E|best) &= 10/1000 = 0.01\\
freq(E|rest) &= 5/9000 = 0.00056\\
like(best|E) &= freq(E|best) \cdot P(best) = 0.001\\
like(rest|E) &= freq(E|rest) \cdot P(rest) = 0.000504\\
P(best|E) &= \frac{like(best|E)}{like(best|E) + like(rest|E)} = 0.66
\end{aligned}
\tag{2}
$$

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 2 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

$$
P(best|E) * support(best|E) = \frac{like(best|E)^2}{like(best|E) + like(rest|E)} \tag{3}
$$

6.2 KEYS2: a Second Version of Keys

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 6);
- 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.

7 Other Algorithms

In order to assess KEYS and KEYS2, we benchmarked those algorithms against alternate algorithms. We selected the comparison algorithms with much care. Numerous researchers stress the difficulties associated with comparing radically different algorithms. For example, Uribe and Stickel [74] struggled valiantly 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, researchers in model checking researchers like Holzmann (pers. communication) eschew companions of tools like SPIN [41], which are search-based, with tools like NuSMV [14], which are BDD-based. Hence we take care to compare algorithms similar to KEYS.

In terms of the Gu et al. survey [33], our selected algorithms (simulated annealing, a-star and MaxWalkSat) share four properties with KEYS and KEYS2. They are each discrete, sequential, *unconstrained* algorithms (constrained algorithms work towards a pre-determine the number of possible solutions while unconstrained methods are allowed to adjust the goal space.)

Also, the selected algorithms have other properties that make them informative:

- Simulated annealing is the de facto standard in search-based software engineering, perhaps because it is so simple to implement. By comparing KEYS with SA, we can see how well our new method improves over alternatives in widespread use.
- SA is very old (first defined in the 1950s) while algorithms like MaxWalkSat are widely regarded as current state-of-the-art.
- ASTAR and KEYS2 offer full and partial solutions (respectively) over all or some of the model input variables. As we shall see, KEYS2's solutions are better than ASTAR. This lets us comment on what is lost by KEYS2's partial solutions (as we shall see, there is no lose associated with using KEYS2's partial solutions).

7.1 SimpleSA: Simulated Annealing

Simulated Annealing is a classic stochastic search algorithm. It was first described in 1953 [56] and refined in 1983 [47]. 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, 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:

$$prob(w, x, y, temp(y, z)) = e^{((w-x)*\frac{y}{temp(y,z)})} \tag{4}$$

$$temp(y, z) = \frac{(z - y)}{z} \tag{5}$$

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

```
1. Procedure SimpleSA
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. 7. Pseudocode for SimpleSA

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.

7.2 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 [46]. MaxWalkSat is a variant of WalkSat that applies weights to each clause in a conjunctive normal form equation [68]. 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.

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

```
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. 8. Pseudocode for MaxFunWalk

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.

7.3 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 [38]. The algorithm has been proven to be optimal for a given scoring heuristic [22], and has seen widespread use in multiple fields [42, 63, 65, 72].

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.

```
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. 9. Pseudocode for ASTAR

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 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.

7.4 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.

8 Results

Each of the above algorithms was tested on the five models of Figure 5. 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 [54] in detail. The largest, model 5 was optimized previously in [27]. As mentioned in the introduction, on contemporary machines, model 5 takes 300 seconds to optimize using our old, very slow, rule learning method.

8.1 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.

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, 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.

8.2 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 KEYS. Interestingly, Figure 11 ranks two of the algorithms in a similar order to Figure 10:

- 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.

Figure 12 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

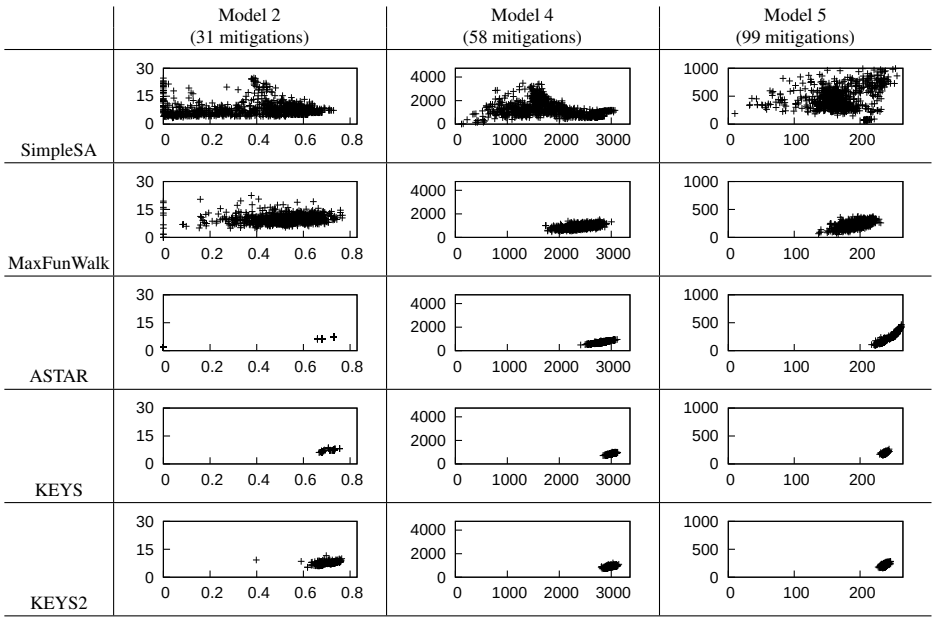

Fig. 10. 1000 results of running four algorithms on three models (12,000 runs in all). The y-axis shows cost and the x-axis shows attainment. The size of each model is measured in number of mitigations. Note that better solutions fall towards the bottom right of each plot; i.e. lower costs and higher attainment. Also better solutions exhibit less variance, i.e. are clumped tighter together.

	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. 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 5).

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). 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.

The plots for KEYS and KEYS2 are nearly identical:

- 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.

9 Discussion

Returning now to the issue of robust solutions for SBSE discussed in §2.3, one key feature of KEYS and KEYS2 is the ease with which they support *partial solutions* as well as *neighborhood* and *trade-off* information.

- Suppose a manager is advocating a policy based on the mitigations found by KEYS2 in some range $1..X$. Figure 12 lets them readily discuss the effects of solutions in the *neighborhood* of X. For example, the effects of applying (say) $x/2$ (half) or double $2x$ that solution is readily seen, just by glancing at Figure 12
- Neighborhood information allows managers 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 $1 \leq x \leq 20$).

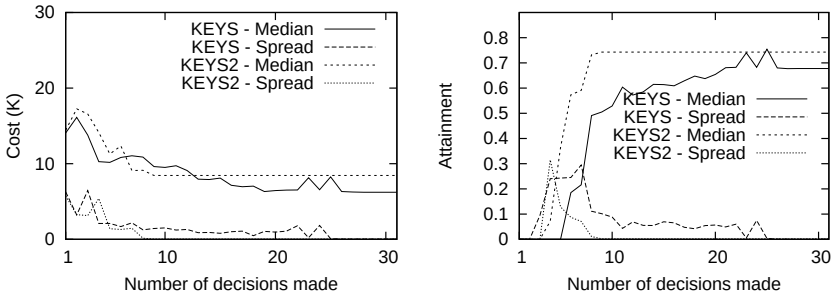

Figure 12a: Internal Decisions on Model 2.

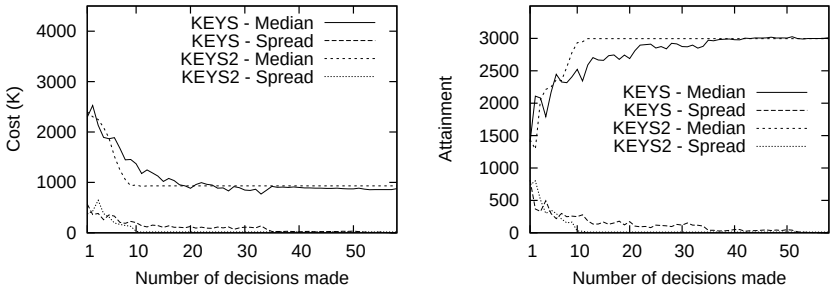

Figure 12b: Internal Decisions on Model 4.

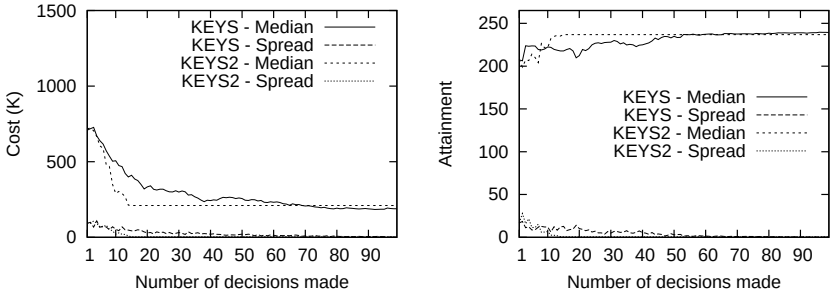

Figure 12c: Internal Decisions on Model 5.

Fig. 12. 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.

KEYS2 and KEYS generate neighborhood information as a side-effect of their processing. The other algorithms discussed here also generate neighborhood information, but at the expense of greatly increased runtimes. For example, as mentioned above (end of §6), 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.

The lesson of KEYS is that such multiple scoring is avoidable, at least for some requirements models. Historically, KEYS was originally designed as a *post-processor* to other algorithms. We proposed a greedy forward select search to prune solutions generated by other means. That order of that search was to have been informed by minimal instrumentation of other methods. Then, just as an experiment, we tried replacing (a) the other algorithms with a random input generator; and (b) replacing the minimal instrumentation with the Bayes sampling methods of Equation 3. As shown above, this approach ran much faster. That is, at least for the models we have explored so far, it may well be that the post-processor can replace the original processing.

10 Conclusion

As our requirements engineering tools grow better, and they will be used by larger groups who will build and debate increasingly complex models. In the net-enabled 21^{st} century, it is hard to keep a group's attention focused on the same set of issues.

We propose a model-based solution that runs fast enough to keep up with the group's dialogue and debates. We define *real-time requirement optimization* as the task of generating conclusions from requirements models before an expert's attention wanders to other issues. Our hope is that these conclusions will be so riveting (either startling, encouraging, or worrying) that the group will stop entering into side-debates, focusing instead on the conclusions from the requirements optimizer.

Based on a back-of-the-envelope calculation (presented in the appendix) we advised that that real-time requirements optimization of current models running on contemporary machines must terminate in time at least 10^{-2} seconds. The generated solution must have two properties:

- *Partial solutions*, since software project managers may be unable or unwilling to precisely control all aspects of a project; and
- *Robustness information* commenting on the brittleness of that solution within the space of nearby solutions.

The experiments of this paper recommend knowledge compilation and KEYS2 for real-time requirements optimization:

- Knowledge compilation speeds up optimization by one to two orders of magnitude. While this is not enough by itself to support (say) the JPL 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 [27] to 0.038 seconds for KEYS2). That is, KEYS2 is fast enough for real-time requirements optimization.
- KEYS2 is an incremental keys-based search algorithm. A log of its incremental findings can be presented in simple diagrams like Figure 12. Glancing at that diagram, it is simple to understand the trade space around the neighborhood of partial solutions.

We have shown that, at least for the DDP models, KEYS2 out-performs other methods. When compared to those of ASTAR, we see that ASTAR can over-specify a solution. KEYS2's partial solutions achieve the same, or better, goals that ASTAR and but, as shown by the trade space of Figure 12 can often do so using only part of the solutions proposed by ASTAR. As to other search algorithms, SimpleSA and MaxFunWalk run slower than KEYS2 and result in solutions with much larger variance.

We attribute the success of KEYS2 to the presence of $keys$ in our models; i.e. variables that control everything else. These key variables were exploited by KEYS2 using two methods: BORE finds promising keys with high probabilistic support; and KEYS2's greedy search sets the most promising key, before exploring the remaining options.

Elsewhere [55], we have documented dozens of papers that have reported the keys effect under different names including *narrows, master-variables, back doors,* and *feature subset selection*. That is, in theory, a keys-aware search like KEYS2 could optimize the processing of many other models. Our intent for this paper is to motivate other keys-based optimizations for other kinds of models.

11 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. Y. Akao. *Quality Function Deployment*. Productivity Press, Cambridge, Massachusetts, 1990.
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. S. M. B and S. Mancoridis. On the automatic modularization of software systems using the bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006.
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. 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.

8. L. Bin, L. Zhi-Shu, C. Yan-Hong, and L. Bao-Lin. Automatic test data generation tool based on genetic simulated annealing algorithm. In *Workshop of Computational Intelligence and Security Workshops, 2007, CISW 2007*, pages 183 – 186, 2007.
9. B. Boehm and P. Papaccio. Understanding and controlling software costs. *IEEE Trans. on Software Engineering*, 14(10):1462–1477, October 1988.
10. S. Bouktif, H. Sahraoui, and G. Antoniol. Simulated annealing for improving software quality prediction. In *GECCO '06: Proceedings of the 8th annual conference on Genetic and evolutionary computation*, pages 1893–1900, 2006.
11. I. Bratko. *Prolog Programming for Artificial Intelligence. (third edition)*. Addison-Wesley, 2001.
12. 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.
13. 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`.
14. 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.
15. J. Clark, J. J. Dolado, M. Harman, R. M. 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 on Software*, 150(3):161–175, 2003. Available from `http://www.brunel.ac.uk/~csstrmh/papers/sbse.ps`.
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. 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`.
19. 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.
20. J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *AAAI '94*, 1994.
21. 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`.
22. R. Dechter and J. Pearl. Generalized best-first search strategies and the optimality af a*. *J. ACM*, 32(3):505–536, 1985.
23. 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.
24. P. Donzelli and G. Iazeolla. Hybrid simulation modelling of the software process. *Journal of Systems and Software*, 59(3), December 2001.
25. 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`.
26. M. Feather, K. Hicks, R. Mackey, and S. Uckun. Guiding technology deployment decisions using a quantitative requirements analysis technique.

27. 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`.
28. 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. Available from `http://eis.jpl.nasa.gov/%7Emfeather/Publications/2008-STAIF-Feather-Uck%un-Hicks.pdf`.
29. R. E. Filman. *Aspect-Oriented Software Development*. Addison-Wesley, Boston, 2004.
30. 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.
31. 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.
32. J. Gray, Y. Lin, and J. Zhang. Automating change evolution in model-driven engineering. *IEEE Computer*, 39(2):51–58, February 2006.
33. 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.
34. D. Harel. Statemate: A working environment for the development of complex reactive systems. *IEEE Transactions on Software Engineering*, 16(4):403–414, April 1990.
35. M. Harman. The current state and future of search based software engineering. In *Future of Software Engineering, ICSE'07*. 2007.
36. M. Harman and B. Jones. Search-based software engineering. *Journal of Information and Software Technology*, 43:833–839, December 2001.
37. 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.
38. 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.
39. 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`.
40. M. Hirakawa and T. Ichikawa. Visual language studies - a perspective. *Software- Concepts and Tools*, pages 61–67, 1994.
41. G. Holzmann. The model checker SPIN. *IEEE Transactions on Software Engineering*, 23(5):279–295, May 1997.
42. 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.
43. 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.
44. D. Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.
45. 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.
46. 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`.

47. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.
48. R. Kohavi and G. H. John. Wrappers for feature subset selection. *Artificial Intelligence*, 97(1-2):273–324, 1997.
49. 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.
50. 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.
51. 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.
52. 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.
53. T. Menzies, O. Jalali, and M. Feather. Optimizing requirements decisions with keys. In *Proceedings PROMISE '08 (ICSE)*, 2008.
54. 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`.
55. 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`.
56. 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.
57. 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.
58. H. Mittelmann. Recent benchmarks of optimization software. In *22nd Euorpean Conference on Operational Research*, 2007.
59. 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.
60. 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.
61. J. Nielson. *Usability Engineering*. Academic Press, 1993.
62. H. Pan, M. Zheng, and X. Han. Particle swarm-simulated annealing fusion algorithm and its application in function optimization. pages 78–81, 2008.
63. J. Pearl. *Heuristics: intelligent search strategies for computer problem solving*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1984.
64. L. Rela. Evolutionary computing in search-based software engineering. Master's thesis, Lappeenranta University of Technology, 2004.
65. 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.
66. T. Saaty. *The Analytic Hierarchy Process: Planning, Priority Setting, Resource Allocation*. McGraw-Hill, 1980.
67. 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`.

68. 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.
69. S. Setamanit, W. Wakeland, and D.Raffo. Using simulation to evaluate global software development task allocation strategies. *Software Process: Improvement and Practice, (Forthcoming)*, 2007.
70. 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.
71. H. Sterman. *Business Dynamics: Systems Thinking and Modeling for a Complex World*. Irwin McGraw-Hill, 2000.
72. B. Stout. Smart moves: Intelligent pathfinding. *Game Developer Magazine*, (7), 1997.
73. 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.
74. 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.
75. 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`.

Appendix

This appendix shows a "back of the envelope" calculation on how fast trade space analysis needs to run on current machines in order to handle larger requirement models of the kind we expect to see in five years time. We will assume that those models are in the DDP format and that our goal is five seconds to complete a 2^W trade space analysis of $W \leq 10$ DDP scenarios.

Figure 13 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, DDP optimization time tends to grow exponentially with the number of variables since larger models have more interconnections.

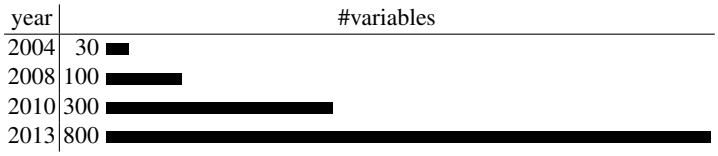

Fig. 13. Growth trends: number of variables in the data dictionaries of NASA's requirements models.

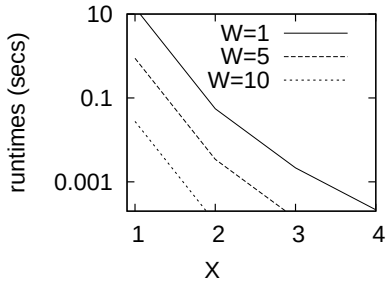

Fig. 14. Estimated time required on contemporary machines in order to handle W what-if scenarios of NASA requirements models in 2013 (when the models are assumed to be eight times larger than those currently seen at JPL). The x-axis represents a range of assumptions on the size of the exponential scale up factor X^8.

Figure 14 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 14 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 we used in 2002 [27]. Worse, handling multiple what-ifs (e.g. $W <= 10$) requires response times of at least of 10^{-2} seconds.