Making Sense of Requirements, Sooner

Tim Menzies, West Virginia University
Julian Richardson, Research Institute for Advanced Computer Science

Early requirements models can be built quickly using simulation to tease out key decisions.

Making decisions during early requirements formulations is like racing through a dark forest without a roadmap or compass. Often unclear are the available options or goals. Different participants in the formulation process have different points of view and frequently end up making decisions based on political rather than technical grounds. Yet high-quality decisions are still demanded in a timely manner. Worse, the cost of patching poor decisions is high—fixing bugs during the requirements phase is a thousand times cheaper than later in the lifecycle.

Using AI-based simulation tools can cut back the forest to reveal clear paths to possibilities and useful requirements. Researchers have long used modeling and simulation to analyze complex processes. In this approach, we describe the system's properties and behavior symbolically and numerically, then let the computer unleash its brute force to grind through the combinations.

Recently, developers have also applied modeling and simulation to analysis of complex decision spaces such as the EpiSims simulation. Built at Los Alamos National Laboratories, it analyzes the effects of public health decisions on the spread of diseases.

We have been developing our own model-based technique for making early requirements decisions. The technique uses two curious properties seen in many models, clumps and collars, that make it easier to search quickly through a seemingly vast range of options.

MODELING REQUIREMENTS OPTIONS

Clarifying the permissible requirements for the problem at hand is the first step toward seeing the paths in the requirements engineering forest. We call a permissible set of requirements a ***requirements candidate***. We use symbolic processing languages such as LISP and Prolog to define a succinct domain-specific language for representing requirements candidates, their component parts, and their interrelationships. Developers can use a simple rule language to specify the terms that are valid requirements. The chosen form for the term and rule languages is not important as long as it can be readily simulated or interpreted.

For example, candidate requirements for an online store might state that the store can have `U` simultaneous users (denoted `users(U)`), that some set of functions `F` is implemented in the online store (denoted `implemented(F)`), and that the runtime memory requirement for such a system with `U` users, implementing functions `F`, is `mem(M)`. A few rules specify constraints that define the permitted values of the term parameters `U`, `F`, and `M`:

```
store(U,F,M) if
  users(U) and implemented(F)
  and mem(U,F,M).

users(U) if 0<U<10

implemented(F)  if
  F=shop&pay or F=shop&drop

mem(U,F,M) if M=U*100000.
```

This description language can specify whether the candidate requirements are permissible. To enable our analysis, we must go further and specify how to construct a permissible requirements candidate. We can do this by extending the rule language to allow specification of an associated distribution for model parameters, for example:

```
users(U) if uniform(1,9,U)
```

`U` is equally likely to hold any of the integer values between 1 and 9 inclusive. We can mechanically generate permissible requirements candidates by randomly choosing one of the disjuncts when the model contains an `or`, and sampling from the specified distribution when the model contains a random variable.

We can generate a candidate system requirement `store(U,F,M)` by generating a maximum number of users, the functions implemented, and the runtime memory requirement. The first of these, `users(U)`, is drawn from a `uniform(1,9)` distribution

such as U=3. The second, `implemented(F)` nondeterministically, has either `F=shop&pay` or `F=shop&drop` (we randomly select `F=shop&pay`). The third, `mem(U,F,M)` is determined by `U` and `F`, here `M=U*100000=300000`—so `store(3,shop&pay,=300000)` is a requirements candidate.

The model's nondeterminism and randomness are important. The more precise our model of candidate requirements, the fewer choices we give ourselves in the requirements-formulation process. Nondeterminism lets us automatically explore more paths.

MODEL OBJECTIVES

Having described what constitutes a permissible set of requirements, we next clarify the objectives, determining not only objectives such as user number, functionality, or cost, but also their relative importance.

Scoring functions provide an effective way to encode the objectives and their relative importance. We define a scoring function as a mapping from requirements candidates to the integers. Just as we used the most convenient terms to describe the available options, we also define our own terminology for scoring candidate requirements. For example, we might define:

```
  score(store(U,F,M))
= U*A- cost(store(U,F,M))*B
```

The model's parameters A and B specify the relative importance of the maximum number of users and cost, while the scoring function `cost` evaluates system cost.

RUNNING THROUGH THE FOREST

Now we can generate a large number of candidate requirements and score them by sampling from the specified distributions. Determining good requirements candidates becomes a matter of generating enough samples and choosing the highest scoring. At first glance, this seems to be an intractable task because we have an exponential number of choices in the search space. A model encoding 20 binary choices has 2^{20} or approximately 1,000,000 states, but a model with 30 choices contains more than 1,000,000,000 possibilities. Fortunately, in practice we can exploit *clumps* or *collars* to make exploring search spaces feasible.

Clumps

A search space *clumps* when it remains in a few states at runtime. For example, Marek Druzdzel checked the number of states reached inside a diagnosis application for monitoring patients in intensive care. Although the software had 525,312 possible internal states, the application reached few of them at runtime: One of the states occurred 52 percent of the time, and 49 states appeared 91 percent of the time.

Druzdzel explored this curious effect ("Some Properties of Joint Probability Distributions," http://www.pitt.edu/~druzdzel/psfiles/uai94.pdf) at a 1994 conference on Uncertainty in Artificial Intelligence. His premise, which is hardly controversial, was that the probability distribution of a whole model's states is a product of the distribution of the model's parts. Using this premise, he showed that an asymmetry in the parts of a model can lead to a highly skewed log-normal distribution in the model's states. Thus, a small fraction of states can be expected to cover a large portion of the total probability space, with the remaining states having practically negligible probability. To put it another way, there was nothing unusual about Druzdzel's diagnosis system; we should always expect that software stays in a tiny part of its possible states.

Collars

In a 1968 paper, "On Representations of Problems of Reasoning about Actions" (*Readings in AI*, Webber & Nilsson, 1981, pp. 2-22), Saul Amarel observed that search spaces frequently contain tiny collars, which must be traversed in any solution. In such a search space, the decision you make when you reach the collars is much more important than how you travel between them. Since the route between collars is unimportant, Amarel defined macros that encoded paths between them in the search space, effectively permitting a search engine to jump between the collars.

With Harshinder Singh, we showed that collars are an expected property of models ("Many Maybes Mean Mostly the Same Thing," *Soft Computing in Software Engineering*, Springer-Verlag, 2003, pp. 125-150). If multiple paths lead to a goal, and those paths pass through collars (or mutually exclusive critical decisions), the most probable path becomes the one with the narrowest collars (that is, the one that uses the fewest critical decisions). For example, in one of the case studies explored with Singh, to ignore a pathway passing through a collar containing only three decisions, the alternate pathway must be 1,728 times as likely. This means that any stochastic search that reaches the goal states will naturally favor simple, or smaller, solutions.

Exploiting Clumps and Collars

Clumps and collars mean that seemingly complex systems will remain in a small number of states and a small number of critical "collar decisions" will switch the system between those states. Better yet, we need not search hard to find the collar decisions. By definition, if we stochastically sample a system, our pathways will naturally pass through the collars. To find the collar variables, we need to

- use an oracle to score the results of each simulation and
- keep frequency counts with high and low scores of how often variable ranges appear in runs.

A "candidate collar decision" is a variable range that appears far more frequently in highly scored runs than poorly scored runs. Ying Hu and one of the authors (Menzies) built the TAR3 learner that exploits this collar property in a stochastic search (Y. Hu, master's thesis, Dept. Electrical Eng., Univ. of British Columbia, 2003). The input to TAR3 is a table of simulation

results in which the values chosen for the term parameters plus the score assigned to that result represent each simulation result. TAR3 searches stochastically through these results for ranges of the model parameters that frequently select for highly scored simulation results. These ranges correspond to collars in the search space. To limit its search, TAR3 only looks for and recommends restrictions on the values of collar variables. TAR3's recommendations, therefore, have an appealing economy.

FINDING THE PATHS

Figure 1 shows the result of iterative simulation and learning applied to case studies described in a 2005 Cocomo Forum paper. The model combined the Cocomo effort/defect and risk models. The forest of requirements reflected different development options such as design for reuse, and process options such as which analysts and programmers to hire. We stochastically sampled from this space of options and asked our learner to find the fewest decisions that most reduced risk, effort, and defects. These decisions were then fed back into the model as constraints on the next round of simulations.

Initially in round 1, outcomes covered a large range of possible defects, risk, and effort. However, by round 5, that space had been severely constrained to a region of very few defects, low risk, and low to medium effort.

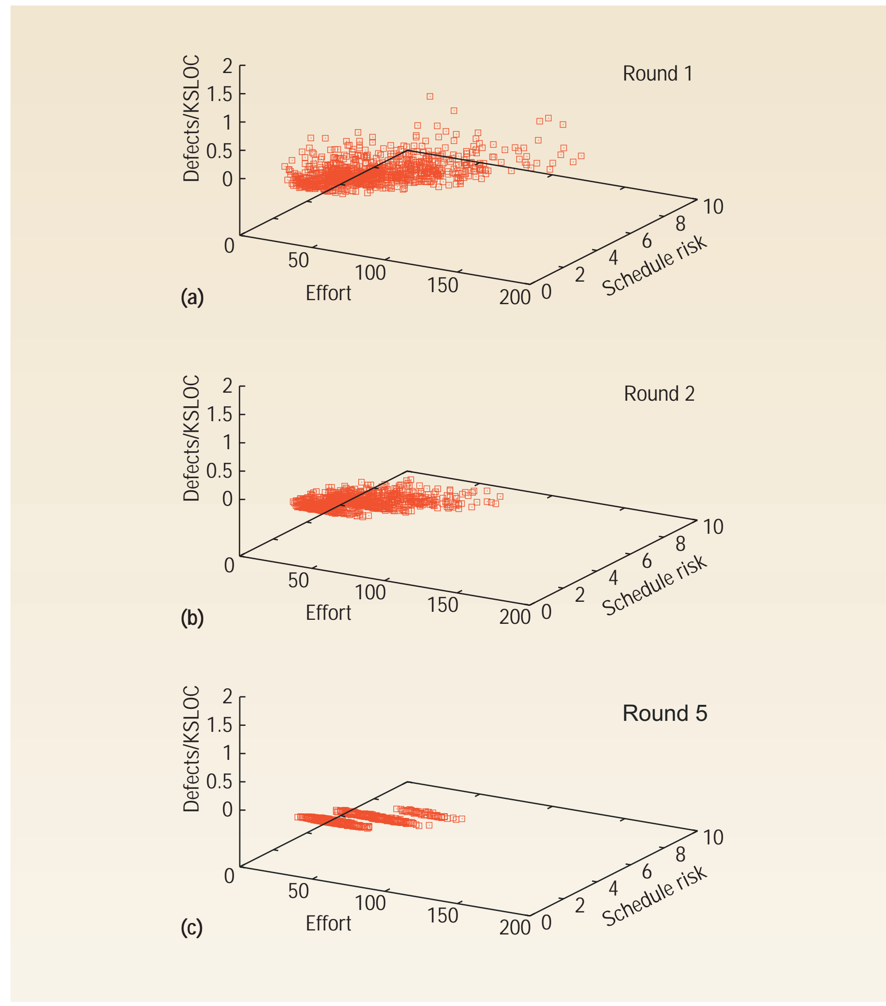

Figure 1. Iterative use of TAR3 in a complex space. Successive rounds of simulation and learning incrementally discover model constraints, which simultaneously optimize the output of three different submodels.

DISCUSSION

Our approach has three components:

- executable models of requirements options and design objectives,
- stochastic simulation of those models to generate a large number of candidate requirements, and
- machine learning to condense simulation logs into the key decisions that determine a good requirements candidate.

The framework outlined in this article is adaptable and widely applicable. Other examples include "Data Mining for Very Busy People," (T. Menzies and Y. Hu, *Computer*, Nov. 2003, pp. 22-29) and a study on near-earth autonomous rendezvous systems presented at the 2006 NASA Software Engineering Workshop (www. systemsandsoftwareweek.org/sew.html).

Early, imprecise life-cycle models generate quickly and let analysts explore more options, faster. Until now, it seemed that speed meant recklessness and that models written quickly were not suitable for understating requirements. However, exploiting clumps and collars can help find the paths to key requirements decisions. ■

***Tim Menzies** is an associate professor in the Lane Department of Computer Science and Electrical Engineering, West Virginia University. Contact him at tim@menzies.us.*

***Julian Richardson** is a principle research scientist at the Research Institute for Advanced Computer Science. Contact him at julianr@riacs.edu.*

Editor: Michael G. Hinchey, NASA Software Engineering Laboratory at NASA Goddard Space Flight Center and Loyola College in Maryland; michael.g.hinchey@nasa.gov