SCIENTIFIC AMERICAN

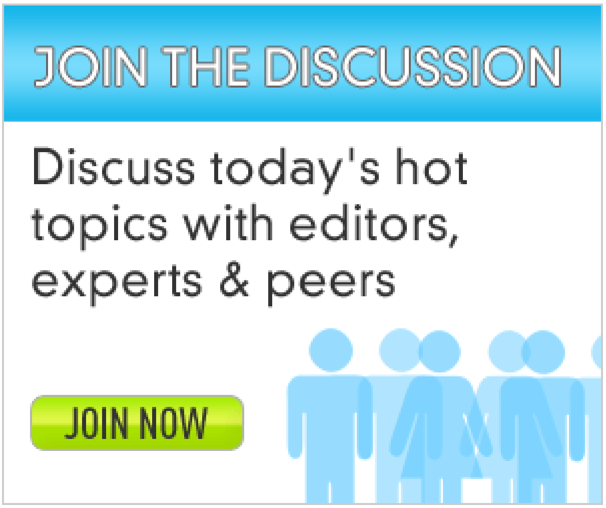
Scientific American Magazine - May 22, 2006

Dependable Software by Design

Computers fly our airliners and run most of the world's banking, communications, retail and manufacturing systems. Now powerful analysis tools will at last help software engineers ensure the reliability of their designs

By Daniel Jackson

An architectural marvel when it opened 11 years ago, the new Denver International Airport's high-tech jewel was to be its automated baggage handler. It would autonomously route luggage around 26 miles of conveyors for rapid, seamless delivery to planes and passengers. But software problems dogged the system, delaying the airport's opening by 16 months and adding hundreds of millions of dollars in cost overruns. Despite years of tweaking, it never ran reliably. Last summer airport managers finally pulled the plug--reverting to traditional manually loaded baggage carts and tugs with human drivers. The mechanized handler's designer, BAE Automated Systems, was liquidated, and United Airlines, its principal user, slipped into bankruptcy, in part because of the mess.



The high price of poor software design is paid daily by millions of frustrated users. Other notorious cases include costly debacles at the U.S. Internal Revenue Service (a failed $4-billion modernization effort in 1997, followed by an equally troubled $8-billion updating project); the Federal Bureau of Investigation (a $170-million virtual case-file management system was scrapped in 2005); and the Federal Aviation Administration (a lingering and still unsuccessful attempt to renovate its aging air-traffic control system).

Such massive failures occur because crucial design flaws are discovered too late. Only after programmers began building the code--the instructions a computer uses to execute a program--do they discover the inadequacy of their designs. Sometimes a fatal inconsistency or omission is at fault, but more often the overall design is vague and poorly thought out. As the code grows with the addition of piecemeal fixes, a detailed design structure indeed emerges--but it is a design full of special cases and loopholes, without coherent principles. As in a building, when the software's foundation is unsound, the resulting structure is unstable.

Managers involved in high-profile software blowouts could claim in their defense that they followed standard industry practices, and unfortunately they would be right. Developers rarely articulate their designs precisely and analyze them to check that they embody the desired properties. But with computers now flying airplanes, driving trains and cars, and running most of the financial, communications, trading and pro-duction machinery of the world, society has an urgent need to improve software dependability.

Now a new generation of software design tools is emerging. Their analysis engines are similar in principle to tools that engineers increasingly use to check computer hardware designs. A developer models a software design using a high-level (summary) coding notation and then applies a tool that explores billions of possible executions of the system, looking for unusual conditions that would cause it to behave in an unexpected way. This process catches subtle flaws in the design before it is even coded, but more important, it results in a design that is precise, robust and thoroughly exercised. One example of such a tool is Alloy, which my research group and I constructed. Alloy (which is freely available on the Web) has proved useful in applications as varied as avionics software, telephony, cryptographic systems and the design of machines used in cancer therapy.

Almost all grave software problems can be traced to conceptual mistakes made before programming started.

Alloy and related design-checking tools build on a quarter of a century of existing research into ways to prove mathematically whether programs are correct. But rather than requiring proofs to be done by hand, they employ automated reasoning techniques that treat a software design problem as a giant puzzle to be solved. These analyzers operate on designs, not program code, so they cannot guarantee that a program will not crash. But they potentially offer software engineers the first practical tools to ensure that designs are robust and free from conceptual flaws and thus provide a firm foundation on which to build reliable software systems.

Evaluating Designs
Bad software is not a new problem. Warnings of a software crisis go back to the 1960s and have only intensified as computers have been woven into the fabric of society [see "Software's Chronic Crisis," by W. Wayt Gibbs; *Scientific American*, September 1994].

Today most software typically is debugged and refined by testing. Human engineers run the program using a wide range of starting conditions (or inputs) to see if it operates as expected. Although the practice catches a raft of small flaws, it often overlooks faults in the basic design of the software. In some sense, these test procedures miss the (diseased) forest for the (rotting) trees.

What is worse, bugs "fixed" during the testing process often exacerbate design problems. As programmers debug the code and insert new features, the software invariably grows barnacles of complexity, creating more opportunities for errors and inefficient operation. This situation is reminiscent of the (incorrect) Ptolemaic theory of planetary motion first developed by the ancient Greeks. In the Middle Ages, as observations showed the predictions to be inaccurate, astronomers adjusted Ptolemy's system, which relied on epicycles. When that proved insufficient, they resorted to adding epicycles to the epicycles. Further fine-tuning over the centuries never solved the problem, because the initial concept was fatally flawed.

Similarly, bad software tends to get more and more complicated and less and less reliable, however much time and money are poured into improving it. It is well known that serious problems with software systems rarely arise from programming errors; almost all grave difficulties can be traced back to conceptual mistakes made before programming even started. In contrast, a small amount of modeling and analysis during the initial determination of requirements, specifications, or program design costs only a tiny fraction of the price tag of checking all the code but provides a large part of the benefit gained from an exhaustive analysis. Focusing on design early saves costly headaches down the road.

Design tools for software have been slow in coming because software does not obey physical laws. Because computer programs are in essence mathematical objects whose values are constructed from bits, software programs are discrete (particlelike) rather than continuous. A mechanical engineer can stress a component with a large force and assume that if it survives it will not fail when subjected to a slightly smaller force. When an object is subject to the (mostly continuous) principles of the physical world, a small change in one quantity generally produces a small change in another. Unfortunately, no such generalities apply to software: one cannot extrapolate between test cases. If one chunk of software works, that fact says nothing about the operations of a similar chunk of code; they are discrete and separate.

In the early days of computer science, researchers hoped that programmers might prove their codings were correct in the same way that mathematicians prove their theorems. With no way to automate the many steps involved, however, a human expert had to do much of the work. These so-called heavy-duty formal methods were impractical except for relatively modest but especially critical pieces of software, such as an algorithm for controlling railroad intersections.

More recently, researchers have adopted a very different approach, one that harnesses the power of today's faster processors to test every possible scenario. This method, known as model checking, is now used extensively to verify integrated-circuit designs. The idea is to simulate every possible sequence of states (the conditions of the system at specific times) that might arise in practice and to determine that none leads to a failure. For a microchip design, the number of states to evaluate is often huge: 10^{100} or more. The challenge is far more stringent for software. But clever encoding techniques (by which large sets of software states can be represented very compactly) make it possible to check every state by considering these large sets simultaneously.

Model checking alone regrettably cannot handle states with complex structures, which is characteristic of most software designs. My research -col-leagues and I have developed an approach that shares the same spirit yet employs a different mechanism. Like model checking, it considers all possible scenarios (although in truth, some bounds need to be introduced to keep the problem finite, because software is not restricted by the physical limitations imposed by hardware). Unlike model checking, however, our technique does not examine scenarios in their entirety, one at a time. Instead it searches for a bad scenario--one that results in failure--by filling in each state in an automated fashion, one bit at a time, in no particular order.

The process is in some sense comparable to a robotic arm fitting each piece of a jigsaw puzzle into place one by one until the completed image finally emerges. If that image corresponds to a bad scenario, Alloy would have done its job. Alloy thus treats design analysis as if it were a puzzle to be solved. Some other recently developed software model checkers work this way as well.

The Solution Is a Puzzle
To understand how Alloy solves software design puzzles, it helps to consider an old riddle: A farmer goes to market where he buys a fox, a goose and a bag of corn. On his way home, he has to carry his goods across a river by boat.

The skiff will hold only the man and one purchase at a time, however. Herein lies a problem: if left unsupervised, the fox would eat the goose and the goose would eat the corn. So how does the farmer get all of his goods to the far bank intact?

This variety of puzzle involves finding scenarios that satisfy a collection of constraints. Mentally we do this task by imagining a series of steps: The farmer transports the goose first; on the next trip, he takes the fox, whereupon he brings back the goose and then, leaving it behind, crosses with the corn; he then returns to fetch the goose. By checking whether each step satisfies the constraints, we ensure that each item remains safe.

A successful software design imposes a similar, though much more complicated, array of rules. To be useful, a design-checking tool must be able to find counterexamples: solutions to the puzzle that meet all the "good" constraints (and thus could occur when the program is run) and an additional "bad" constraint (and thus yield an unacceptable outcome). If any such counterexamples turn up, they will reveal flaws in the design. So whereas the puzzle solver is happy to find a solution to the "farmer's dilemma," a solution to a software design puzzle is bad news: it means that an undesirable scenario exists and the design is defective. In practice, the counterexample might not itself lead to any problems. It may instead reveal a discrepancy in how the designer originally characterized the unacceptable outcomes. Either way something needs to be fixed--the design or the designer's expectations.

The idea is to simulate every state that the software can take to determine that none leads to a failure.

The great difficulty in searching for counterexamples is that the number of potential scenarios in a software design of even moderate complexity is typically vast, but only a tiny proportion correspond to counterexamples. Imagine trying to plan who sits next to whom at a wedding reception. If all attendees get along, the solution is trivial. Throw in a few ex-spouses who require separation, and the problem gets trickier. Now consider the seating chart for Romeo and Juliet's reception. If there are 20 seats and any of 10 guests can sit in each, that makes 10^{20} possible combinations. Even checking a billion scenarios per second, a computer would take 3,000 years to explore them all.

In the 1980s, researchers identified problems of this form as a special class of problems that, in the worst case, can be solved only by enumerating all possible scenarios. But in the past decade, with new search strategies and algorithms and by building on ever increasing computational power, researchers have developed tools called SAT (satisfiability) solvers that can handle these problems fairly easily. Many are now freely available and can often solve problems with millions of constraints.

Importance of Abstraction
As its name suggests, Alloy melds two elements that help make software designs more robust. One is a new language that helps to elucidate the structure and behavior of the software design. The other is an automated analyzer (which incorporates a SAT solver) to hash through a multitude of possible scenarios.

The first step in applying Alloy is to create a model of the design: not the rough sketch or flowchart typical in software engineering but a precise model that spells out the "moving parts" and specific behaviors, both desired and undesired, of the system and its components. A software engineer first writes down definitions of the various kinds of objects in the design, then groups those objects into mathematical sets: collections of things that are alike in their structure and behavior (for example, the set of all Capulets) and linked by mathematical relations (such as the relation that associates guests sitting next to one another).

Next come facts that constrain these sets and relations. In a software design, the facts include the mechanism of the software system and assumptions about other components (say, statements about how human users are expected to behave). Some of these facts are simple assumptions--for example, that nobody is both a Capulet and a Montague and that every guest sits next to exactly two other guests. Some of them reflect the design itself: in our seating planner, for instance, the rule that each table, with the exception of the top table, is assigned either to one family or the other.

Alloy has uncovered serious deficiencies in published software designs.

Finally, there are assertions, which are constraints that are expected to follow from the facts. In our example, with the exception of Romeo and Juliet, no Capulet should be seated next to a Montague. The assertions say that the system can never get into certain undesirable states and that specific bad sequences of events can never occur.

The analyzer component of Alloy harnesses a SAT solver to search for counterexamples--possible scenarios of the software system that are permitted by its design but that fail a sanity check (which is accomplished by writing assertions that must be true if the model is correctly designed). In other words, the tool attempts to construct situations that satisfy the facts but violate a stated assertion. In our case, it would generate a seating plan in which a Capulet (other than Juliet) sits next to a Montague (other than Romeo) at the top table. To fix the seating rule, we can add a new fact: that Romeo and Juliet occupy the top table alone. Now Alloy would find no counterexample.

Together the declarations of the sets and relations, the facts, and the assertions make up an abstraction that captures the essence of the software design. Writing all this out makes the limitations of the design explicit and forces engineers to think hard about exactly which abstractions will work best. Bad abstraction choices lie at the root of many unnecessarily complicated or unreliable systems.

Systems that rely on software built on simple and robust abstractions should also be easier to use. Consider how e-ticket-ing simplified air travel, how universal product codes made shopping easier or how 800-number-based conference calls made teleconferencing more feasible. Each of these innovations stemmed from a transformation in the basic abstractions embodied in the underlying software.

The Road to Reliability
Tools akin to alloy are currently used primarily in research and in cutting-edge industrial settings. The technology has been employed to explore new architectures for telephone switching systems, to design avionics processors that are secure against hackers and to describe access-control policies for communications networks. We have used it to check widely used and robust software devices, such as protocols for finding printers on networks and tools for synchronizing files across machines.

In addition, Alloy has uncovered serious deficiencies in published software designs--such as a key management protocol that was supposed to enforce special-access rules based on membership in a group but turned out to grant access to former members who should have been rejected. It is noteworthy that many programmers who have used Alloy have been surprised by the number of flaws that the tool turns up in the designs for even their simplest applications.

It is most likely only a matter of time until tools resembling Alloy are adopted more widely in industry. Improvements in the underlying SAT solvers will make analysis tools faster and better able to handle very large systems. Meanwhile a new generation of software designers, educated in these methods, will incorporate them into their work. Modeling is growing in popularity, particularly among managers desperate to see some description of a software system's design beyond the code itself.

At some point, there may come a time when software becomes so essential to our day-to-day infrastructure that society will no longer tolerate bad software. As a result, governments may even establish inspection and licensing regulations that enforce high-quality program construction techniques. One day, perhaps, software systems will be truly robust, predictable and easy to use--by design.