What the Errors Tell Us

Margaret H. Hamilton, Hamilton Technologies

// *With a preventative paradigm, most errors aren't allowed into a system in the first place, just by the way the system is defined. With such an approach, the more reliable the system, the higher the productivity in its lifecycle.* //

IT WAS NOT quite the '60s. No school existed for learning how to build software. You were on your own. Courses, if any, were concerned only with becoming familiar with a set of available commands to tell the computer what to do (sometimes provided by a computer's manufacturer). It was difficult to understand why this and only this seemed to matter. Just knowing a set of English words would not demonstrate one's ability to write a good novel. Experience with the programming language needed on a particular project was in great demand.

Little did we know that what we were doing then would become known years later as "software engineering," when we were in the trenches building flight software for the Apollo missions,[1] and that software engineering would be celebrated in this issue of *IEEE Software* and at the 2018 International Conference on Software Engineering as having existed for at least half a century.[2]

Early Days

My first assignment was creating weather prediction software in hexadecimal on the LGP-30, for MIT's Edward Lorenz. Understanding the hardware's relationship to the software and how to use this knowledge to increase the software's performance was a priority.

Errors were dreaded, because debugging took forever. Setting up longer runs overnight, and spending more time up front (on the hexadecimal code) and less time on testing at the back end, helped. Still, more needed to be done. The "solution": instead of always having a new paper tape of machine code (in binary) generated by the computer from the hexadecimal program, the breakthrough was realizing changes could be made directly to the tape by poking a hole in it with a pencil to turn a 0 into a 1, or covering up a hole with Scotch Tape to turn a 1 into a 0. But this approach ("hacking") could be error-prone.

Another project was the SAGE (Semi-Automatic Ground Environment) air defense system at Lincoln Laboratories, where we developed software on the first AN/FSQ-7 computer, the XD-1, to search for unfriendly aircraft. It was especially important not to make an error because if you did, the computer would tell everyone. The machine was huge. When it crashed, we heard loud siren-like and foghorn-like sounds throughout a very large building. Operators and programmers would come running to find out whose program crashed. Since it belonged to the programmer standing in front of the console, it was no secret who the guilty one was. The location where the program halted could be found in a foot-long register on the console with its flashing lights—the only information we had to find out what caused the crash. The next step: write the register's contents on a piece of paper.

Given what it took to find the error, it was again reason to spend more time up front on the code. Keeping track of which program caused which crash was a challenge. My solution: take a Polaroid picture of each programmer posing next to his or her bug. The pictures became more creative as time went on. We all loved to listen to the sounds of one program. One time a computer

operator called at four in the morning and said, "Something terrible happened; your program no longer sounds like a seashore." I got in the car and rushed to work. We had found a new way to debug, using sound. I began to find more ways to understand what made a particular error or a class of errors happen as well as how to prevent it from happening in the future.

Apollo Onboard Flight Software

SAGE had its drama, especially when it came to errors. But this was only the beginning of what would come next: the Apollo onboard flight software project at MIT, under contract to NASA. The challenge was unique: build human-rated software, meaning astronauts' lives were at stake. Not only did it have to work, it had to work the first time. Not only did the software itself have to be ultrareliable, it also needed to be able to detect an error and recover from it in real time. It did not disappoint.

Each mission had its drama, but Apollo 11 was special. We had never landed on the moon before. Everything was going according to plan, until something totally unexpected happened. Just before the astronauts were about to land, the onboard Apollo Guidance Computer (AGC) became overtaxed. The software's priority displays (Display Interface Routines) of 1201 and 1202 alarms interrupted the astronaut's normal mission displays, warning them of an emergency, letting NASA's Mission Control understand what was happening and alerting the astronauts to place the rendezvous radar switch in the right position. The priority displays gave the astronauts a go/no-go decision (to land or not to land). It quickly became clear that the software was not only informing everyone there was a hardware-related problem but also compensating for it.[3] With minutes to spare, the decision was made to land. The rest is history. The Apollo 11 astronauts became the first humans to *walk* on the moon; our software became the first software to *run* on the moon. The software experience (designing it, developing it, and learning from it for future systems) was at least as exciting as the events surrounding the mission.

The task at hand: develop the Command Module (CM) and Lunar Module (LM) software for the AGC. This included the systems software, shared by and residing within both the CM and the LM, and the flight software's "glue" that defined, integrated, and managed the relationships between and among mission phases and routines. Updates to the software were continuously being submitted from hundreds of people over all the releases for each mission. Every change, with its reason for being documented, needed approval before being allowed into an official version. Everything needed to play together like a finely tuned orchestra, making sure there were no interface errors (data, timing, and priority conflicts) in the software and between the software and the other systems involved (including hardware, peopleware, and missionware).

The flight software was designed to be asynchronous so that higher-priority jobs could interrupt lower-priority jobs. This was accomplished by the developers' assigning a unique priority to every process, ensuring that all events in the software would take place in the correct order and at the right time relative to everything else going on. Steps taken earlier within the software to create solutions within an asynchronous environment became a basis for solutions within a distributed environment, once it became apparent that although only one process at a time executed in a multiprogramming environment, other processes in the same system—sleeping or waiting—existed in parallel with the executing process.

With this as a backdrop, the priority displays were created, changing the human–machine interface between the astronauts and the software from synchronous to asynchronous, where the flight software and the astronauts became parallel processes within a distributed system-of-systems environment. Such was the case with the systems-software error-detection-and-recovery programs. They included the system-wide *kill and recompute from a safe place* snapshot-and-rollback restart capabilities and the priority displays together with their human-in-the-loop capabilities (such as that of being able to warn the astronauts and replace their normal displays with priority displays).

This would not have been possible without an integrated system-of-systems (and teams) approach and innovative contributions made by other groups. The hardware team at MIT changed their hardware, and the mission-planning team in Houston changed their astronaut procedures. Both worked closely with us to accommodate the priority displays in both the CM and the LM, for any emergency throughout any mission. Mission Control was prepared. They knew what to do should the astronauts' displays be interrupted with priority displays.

In addition to the software developed by our team, "outside" code

could be submitted from other groups to become part of the flight software (e.g., from someone in the navigation analysis group). Once submitted to our team for approval, the code fell under our supervision. It was then "owned by" and updated by our team to become part of, and integrated with, the rest of the software. As such, it had to go through the strict rules required for all onboard flight software. This ensured that all the flight software modules and all aspects of these modules were completely integrated and that there would be no interface errors within, between, and among all modules, both during development and in real time.

When answers did not exist, we invented them. Dramatic events dictated change. Requirements were "thrown over the wall" from mission experts to software experts. To those people who weren't software experts, software "magically" appeared within the AGC, integrated and ready to go. What began as mission-related requirements for the software became more understood by everyone in the form of the application-oriented parts of the flight software that realized the mission requirements. Mission expertise moved on from mission experts to software experts and vice versa. Mission engineers and software engineers necessarily became interchangeable, as did their lifecycle phases—suggesting that a system is a system, no matter from which discipline things originated. From this perspective, system and software design issues became one and the same.

Understanding the subtleties of developing real-time asynchronous flight software was left up to the systems-software experts. Since there was still no school for learning such things, having this kind of responsibility necessitated our creating and evolving methods, standards, rules, tools, and processes for designing and developing the software, with a special emphasis on preventing errors. Although many errors were found during the software's preflight phases, no onboard flight software errors were known to have occurred during flight on any Apollo missions. An invaluable position within the team was the Assembly Control Supervisor (ACS), caretaker of the code submitted for a particular mission for the LM or the CM (e.g., one for Apollo 8's CM, another for Apollo 11's LM). Each ACS used the *Augekugel* (eyeball) method in his designated area, looking for interface errors and violations of coding rules, throughout all the official flight software releases and their interim updates.

Since it was not possible (certainly not practical) to test the software by flying actual missions, it was necessary to test it with a mix of hardware and digital simulations of every (and all aspects of an) Apollo mission. This included human-in-the-loop simulations (with real or simulated human interaction) and variations of real or simulated hardware and their integration, making sure a complete mission from start to finish would behave exactly as expected.

The Preventative Paradigm

One could not help but learn from these experiences. With initial funding from NASA and the US Department of Defense, we performed an empirical study of the Apollo effort. The subject of errors took on a life of its own. Opportunities to make errors were not lacking, not to mention every kind possible. I had the opportunity to have some responsibility in the making of many of these errors, without which we would not have been able to learn as much as we did. These errors sometimes occurred with great drama and fanfare, and often with a large-enough audience to never want such a thing to happen again!

What we learned was full of surprises. For want of a better term, the process evolved into a "theory of errors." A general systems theory was derived from the errors and what we learned from them. This theory continues to evolve on the basis of the lessons learned from Apollo and later projects. From its axioms we derived a set of allowable patterns that became the basis for the Universal Systems Language (USL) together with its automation[4] and preventative paradigm, "development before the fact."

Unlike with languages typically used in traditional systems, instead of telling the computer what to do, the user defines all the system's relationships (the *what*). A formalism for representing the mathematics of systems, USL is based on the axioms of control of the systems theory and formal rules for their application. Every system is defined in terms of Function Maps (FMaps) and Type Maps (TMaps).[5] FMaps define functions and their relationships to other functions; TMaps define types and their relationships to other types. FMaps are inherently integrated with TMaps. Figure 1 shows an example of a system defined in terms of FMaps and TMaps.

A USL system is defined from the very beginning to maximize the potential for its own reuse; reliable systems are defined in terms of reliable systems. Three universal primitive structures, derived from the axioms, and nonprimitive structures, derived

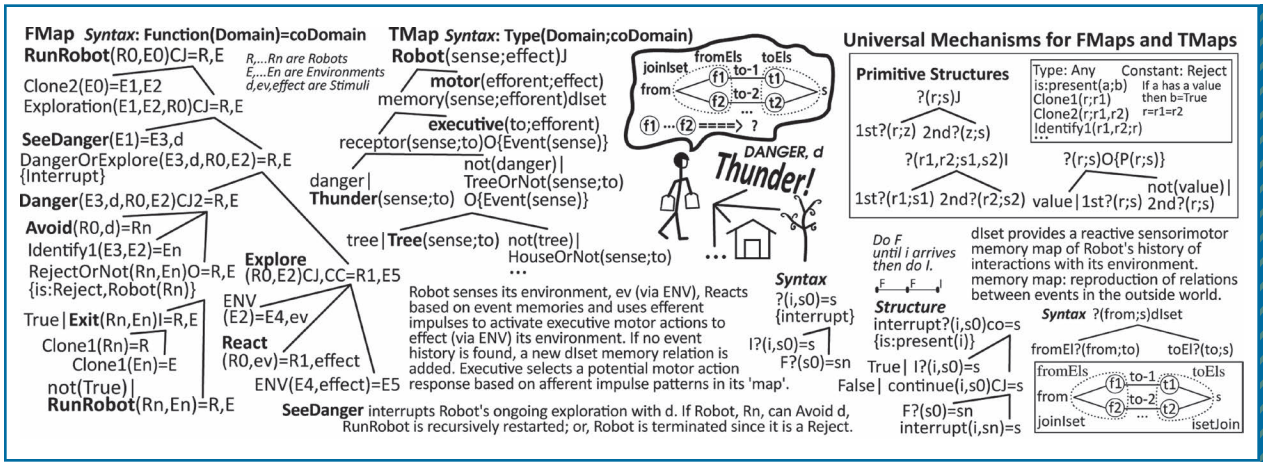

FIGURE 1. This robot exploration system is defined with the Function Map (FMap) **RunRobot** and the Type Map (TMap) **Robot**. FMaps represent the dynamic world of actions (doing) by capturing functional, temporal, and priority characteristics. TMaps represent the static world of objects (being) by capturing type, atemporal, and importance characteristics. Each map is defined in terms of the universal primitive structures: **Join** (**J**) for dependencies, **Include** (**I**) for independencies, and **Or** (**O**) for alternatives. The robot explores its environment, recording its experiences in a reactive sensorimotor memory map represented by a distributed independent set of relations, using the TMap structure **dIset**.[2]

ultimately from the primitive structures, specify each map. Each primitive function resides at an FMap leaf node and corresponds to a primitive operation of a TMap type. Primitive types, each defined by a set of primitive operations and axioms, reside at TMap leaf nodes (defining the system's application domain). Each primitive-type operation may be realized by an FMap on a lower layer of the system.

Correctness is accomplished by the very way a system is defined, by built-in language properties inherent in USL's grammar. A USL definition models both its application (e.g., an avionics or a cognitive system) and properties of control into its own lifecycle. Mathematical approaches are often known to be difficult to understand and limited in their use for nontrivial systems. Unlike other formal methods, USL extends traditional mathematics (e.g., mathematical logic) with a unique concept of control enabling it to support the definition and realization of a system, no matter its kind or size. The mathematical formalism is hidden by language mechanisms, derived in terms of that formalism, that are semantics-dependent but syntax-independent.

Properties of preventative systems are especially important from the perspective of a system's real-time, asynchronous (event-driven), and distributed behavior. Such a system inherently lends itself to being able to detect and automatically recover from errors in real time. Whereas on Apollo it was necessary for developers to explicitly schedule each process in the flight software and assign to it a unique priority, with USL this is no longer necessary because priority is inherent in the grammar of FMaps and TMaps. Functions (and types) within a system are totally ordered, since every function (and type) has a unique priority.

Along the way, it became clear that a system defined with USL has properties that inherently support its own development "before the fact," eliminating those things including tasks, procedures, rules, and tools that previously were necessary but now become no longer needed, such as analyzing the code by hand to ensure it conforms to the system's specification and design, and assigning unique priorities to (and scheduling) processes manually. We continue to discover new properties in USL-defined systems. One day, we realized that the root problem with traditional approaches is that they support users in "fixing up wrong things after the fact" rather than in "doing things in the right way in the first place." With USL's preventative paradigm, instead of looking for more ways to test for errors and continuing to test for errors late into the lifecycle, the majority of errors, including all interface errors (at least 75% of and the most subtle of all errors), are not allowed into a system in the first place, by the way it is defined.

Testing for nonexistent errors becomes an obsolete endeavor. Whereas most errors are found (if ever found) during the testing phase in traditional development, with USL, correct use of the language prevents ("eliminates") errors "before-the-fact" in a system's definition and its derivatives (e.g., its software, since software is automatically generated from its formal definition, inheriting all the properties of the definition from which it came). Instead of automation that supports the lifecycle, the lifecycle process itself can now be automated. Integration and traceability within a definition and from systems to software are seamless.

USL's automation hunts down errors from the incorrect use of USL. Much of the design and all the code (and documentation) for a given software system are automatically generated, or commands could be automatically generated for another kind of resource (e.g., a robot). Because of USL's open architecture, the automation can be configured to generate one of a possible set of implementations for a resource architecture (the *how*) of choice (e.g., a language, a database package, or the users' own legacy code). When an object type is changed, the status of all its functional uses (impacted by objects of that type) are demoted. The FMaps are then automatically reanalyzed by the analyzer, reestablishing the status of that type's uses.

Maintenance shares the same benefits. The developer never needs to change the code, since application changes are made to the USL definition—not to the code—and target resource architecture changes are made to the generator environment configuration—not to the code. Only the changed part of the system is regenerated and integrated with the rest of the application. Again, the system is automatically analyzed, generated, compiled, linked, and executed without manual intervention.

Many of the pressing software issues that existed in the earlier days still exist today. From our own work, we believe that this is largely because of the traditional paradigm. It has been around since the beginning and continues in force to this day.

Many well-known problems with the traditional paradigm need not exist with a preventative paradigm. Much of what seems counterintuitive with the traditional approach becomes intuitive with a preventative paradigm: the more reliable a system, the higher the productivity in its lifecycle. For each new property discovered that, in essence, comes along for the ride, there is the realization of something no longer needed as part of the system's own lifecycle. What works best for developing ultrareliable systems just happens to work best for systems in general, no matter the application.

Several kinds of systems have been developed with USL, including "the development process of a system" as a system itself. Just like the systems that are developed with USL, USL's automation is completely defined by itself (using USL), and it automatically generates itself. Looking toward systems of the future, university,[6,7] research,[8] government,[9] and commercial[10,11] organizations have conducted experiments throughout USL's evolution, comparing its approach with traditional approaches within diverse domains including formal methods,[7] object technologies and CASE (computer-aided software engineering),[12] domain analysis,[8] and model-driven development (e.g., UML,[10,11] SysML,[10] and Cleanroom[11]). These experiments have gone from system functional requirements through operational validated code, each refereed by third-party observers or by the agency sponsoring the competition.

USL has stood the test of time, especially when medium to large-scale systems are in the mix. In every case, reliability and productivity have been considered to be of highest importance.

The errors not only tell us how to build systems without them but also unexpectedly gave us a paradigm for the future. Educating people how to think and build systems in terms of the paradigm becomes the next challenge.

References

1. L. Snyder and R.L. Henry, *Fluency with Information Technology*, 7th ed., Pearson, 2018, pp. 173–176.
2. M. Hamilton, "The Language as a Software Engineer," keynote presented at 40th Int'l Conf. Software Eng. (ICSE 18), 2018; https://www.youtube.com/watch?v=ZbVOF0Uk5lU.
3. M. Hamilton, "Computer Got Loaded" (letter to the editor), *Datamation*, Mar. 1971; https://medium.com/@verne/margaret-hamilton-the-engineer-who-took-the-apollo-to-the-moon-7d550c73d3fa.
4. "System: do_all_taxes," Hamilton Technologies, 25 Aug. 2015; http://htius.com/Examples/tax_example/documentation/do_all_taxes.op.collector-full.
5. M. Hamilton and W. Hackler, "Universal Systems Language: Lessons Learned from Apollo," *Computer*, vol. 41, no. 12, 2008, pp. 34–43.

6. M. Hamilton and W. Hackler, "Universal Systems Language for Preventative Systems Engineering," *Proc. 5th Ann. Conf. Systems Eng. Research* (CSER 07), 2007, paper 36.
7. M. Ouyang and M.W. Golay, *An Integrated Formal Approach for Developing High Quality Software for Safety-Critical Systems*, tech. report MIT-ANP-TR-035, MIT, Sept. 1995.
8. B. Krut Jr., *Integrating 001 Tool Support in the Feature-Oriented Domain Analysis Methodology*, tech. report CMU/SEI-93-TR-11, Software Eng. Inst., Carnegie Mellon Univ., 1993.
9. *National Test Bed Software Engineering Tools Experiment—Final Report*, vol. 1, US Dept. of Defense Strategic Defense Initiative Org., Oct. 1992, Experiment Summary, Table 1, p. 9.
10. M. Hamilton and W. Hackler, "A Formal Universal Systems Semantics for SysML," *Proc. 17th Ann. Int'l Symp. Int'l Council Systems Eng.* (INCOSE 07), 2007, paper 8.3.2.
11. M. Hamilton, "Universal Systems Language (USL) and Its Automation, the 001 Tool Suite, for Designing and Building Systems and Software," presentation at IEEE Computer Society / Lockheed Martin Webinar Series, 27 Sept. 2012, slides 36–40.
12. M. Schindler, *Computer-Aided Software Design: Build Quality Software with CASE*, John Wiley & Sons, 1990.

ABOUT THE AUTHOR

MARGARET H. HAMILTON is CEO of Hamilton Technologies. Her research interests include ultrareliable systems, error detection and recovery, formal theory, languages, and OSs. She graduated in mathematics and philosophy from Earlham College. She was in charge of the onboard flight software for NASA's Apollo manned missions and was the director of the Software Engineering Division at MIT's Instrumentation Laboratory. She led empirical studies of Apollo and later efforts, resulting in her systems theory of control and the Universal Systems Language with its preventative paradigm. Her awards include the Augusta Ada Lovelace Award, NASA Exceptional Space Act Award, Earlham College Outstanding Alumni Award, Computer History Museum Fellow Award, and Presidential Medal of Freedom awarded by President Barack Obama. Contact her at mhh@htius.com.