Self-Sustaining Systems

Richard P. Gabriel
Distinguished Engineer, Sun Microsystems, Inc.

I have a small laboratory at Sun Microsystems which is just embarking on a research program on what we call "self-sustaining systems." A self-sustaining system is any combination of hardware and software which is able to notice errors, inefficiencies, and other problems in its own behavior and resources, and to repair them on the fly. We imagine that such systems will be quite large and that they might exhibit both hardware and software failures. This means that some of the failures will be intermittent or based on coincidences—of placement, of timing, and of naming.

We plan to explore a number of approaches, some inspired by good engineering principles, some inspired by biological mechanisms and mechanisms or principles from other disciplines, and some deriving from a reëxamination of accepted practices in design, coding, and language design. For example, we expect to examine the following:

- there is an existing practice in high-reliability systems, especially in telecom software and equipment, in real-time and embedded systems, and in distributed real-time and fault-tolerant systems, which use resource estimation, checkpointing, and rollback—among other techniques—to ensure proper operation. We plan to examine the literature, mostly in the form of software patterns and pattern languages to determine what can be used for self-sustainability.
- most programming languages are designed for running (correct) algorithms embodied in correct programs, the design center being the easy expression of precise calculations. But this is perhaps contrary to the needs of some parts of self-sustaining systems, which must be able to express the observation of running programs and alter their running behavior, and programs written to do this must be highly tolerant of errors of their own. Such robustness of an observational layer can perhaps be achieved by operating only on "safe" data or by being declarative or reactionary. We expect to look at programming constructs aimed at expressing temporal reasoning about and operations on programs more intuitively than is done now with low-level synchronization primitives.
- our intuition is that one key to self-sustaining programs is an observational, programmable layer which not only observes, interrupts, and alters, but which consists of decaying persistent memory. This intuition is hard to express. We imagine that this secondary layer is "soupy" and uses gradients to convey and combine information in an almost geometrical manner. For example, suppose there is a distributed system in which patterns of information flow vary over time. If a particular data structure is subject to frequent repair, it might be correlated with information flow which might persist or repeat, or which might be transitory. The traces of these events, if kept forever, might prove too complex to analyze; but with repetition, the pattern of causation might be reinforced and the amount of data to plow through would be greatly reduced. Another example is a tightly packed multiprocessor which can suffer heat-related failures when heavy-duty computations take place on them. Imagine an observational layer that is monitoring heat and memory or computation failure rates. If the observational memory were to hold data about the heat or failure rate and treat it like dissipating heat or dissolving chemicals, then adjacent memory and computational elements would "inherit" some of the characteristics of the problems and computations would not be scheduled onto them as frequently, thereby reducing errors by moving computations away from overheated areas.

  Part of this intuition is that the observational layers need to be simpler to program than the real computational layers, so that correctness of code running there is easier to show or if possible, irrelevant. Another part is that the stuff of which this layer is made must be fundamentally more robust or at least less fragile than the tightly wound stuff that makes up the base layer. One way to accomplish

this is to make this layer not computationally complete, but highly limited and based on rudimentary operations.

- after reading the work of Martin Rinard and his students at MIT, we believe that repairing deviations from "acceptable" behavior is a good approach for certain types of robustness. One way to think of this is that the supposed tight bounds on what would be considered correct behavior is often artificially too restrictive. For example, in triangle-based rendering, degenerate triangles causing division by 0 in some algorithms can probably skip expensive checking for degeneracy and use a quick exception-handling mechanism to return a fixed value, since all that might happen is that a handful of pixels in a gigantic image might have a slightly wrong color. What is acceptable to a self-sustaining system might be much more diverse (or forgiving) than a single execution path.
- many programming languages shortchange exception handling, treating them, well, like exceptions. In living systems, a large fraction of the mechanisms at various levels are concerned with preservation, conservation, and repair, while in an exceptionally careful software system, perhaps 5% of the mechanism are devoted to such things. We believe that many exception handling systems are not only unsuitable for programming self-sustainability, but the mechanisms themselves break modularity, causing additional errors due to programming mistakes

Self-sustainability and self-awareness appear to be related, perhaps only at the implementation level, but perhaps conceptually as well. A self-sustaining system performs some degree of reflection and can alter its own behavior based on what it sees, and this is what a self-aware system does.