Robust Design through Diversity

Gerald Jay Sussman
Matsushita Professor of Electrical Engineering
Massachusetts Institute of Technology

Computer Science is in deep trouble. Structured design is a failure. Systems, as currently engineered, are brittle and fragile. They cannot be easily adapted to new situations. Small changes in requirements entail large changes in the structure and configuration. Small errors in the programs that prescribe the behavior of the system can lead to large errors in the desired behavior. Indeed, current computational systems are unreasonably dependent on the correctness of the implementation, and they cannot be easily modified to account for errors in the design, errors in the specifications, or the inevitable evolution of the requirements for which the design was commissioned. (Just imagine what happens if you cut a random wire in your computer!) This problem is structural. This is not a complexity problem. It will not be solved by some form of modularity. We need new ideas. We need a new set of engineering principles that can be applied to effectively build flexible, robust, evolvable, and efficient systems.

In the design of any significant system there are many implementation plans proposed for every component at every level of detail. However, in the system that is finally delivered this diversity of plans is lost and usually only one unified plan is adopted and implemented. As in an ecological system, the loss of diversity in the traditional engineering process has serious consequences for robustness.

This fragility and inflexibility must not be allowed to continue. The systems of the future must be both flexible and reliable. They must be tolerant of bugs and must be adaptable to new conditions. To advance beyond the existing problems we must change, in a fundamental way, the nature of the language we use to describe computational systems. We must develop languages that prescribe the computational system as cooperating combinations of redundant processes.

From biology we learn that multiple strategies may be implemented in a single organism to achieve a greater collective effectiveness than any single approach. For example, cells maintain multiple metabolic pathways for the synthesis of essential metabolites or for the support of essential processes. For example, both aerobic and anaerobic pathways are maintained for the extraction of energy from sugar. The same cell may use either pathway, or both, depending on the availability of oxygen in its environment.

Suppose we have several independently implemented systems all designed to solve the same (imprecisely specified) general class of problems. Assume for the moment that each design is reasonably competent and actually correctly works for most of the problems that might be encountered in actual operation. We know that we can make a more robust and reliable system by combining the given systems into a larger system that redundantly uses each of the given systems and compares their results, choosing the best answer on every problem. If the combination system has independent ways of determining which answers are acceptable we are in very good shape. But even if we are reduced to voting, we get a system that can reliably cover a larger space of solutions. Furthermore, if such a system can automatically log all cases where one of the designs fails, the operational feedback can be used to improve the performance of the system that failed.

This redundant design strategy can be used at every level of detail. Every component of each subsystem can itself be so redundantly designed and the implementation can be structured to use the redundant designs. If the component pools are themselves shared among the subsystems, we get a controlled redundancy that is quite powerful. However, we can do even better. We can provide a mechanism for consistency checking of the intermediate results of the independently designed subsystems, even when no particular value in one subsystem exactly corresponds to a particular value in another subsystem. Thus the interaction between systems appears as a set of constraints that capture the nature of the interactions between the parts of the system.

For a simple example, suppose we have two subsystems that are intended to deliver the same result, but computed in completely different ways. Suppose that the designers agree that at some stage in one of the designs, the product of two of the variables in that design must be the same as the sum of two of the variables in the other design. There is no reason why this predicate should not be computed as soon as all of the four values it depends upon become available, thus providing consistency checking at runtime and powerful debugging information to the designers.

The ARPAnet is one of the few engineered systems in use that is robust in the way we desire. This robustness partly derives from the fact that network supports a diversity of mechanisms, and partly from the fact that the late-binding of the packet-routing strategy. The details of routing are locally determined, by ambient conditions of the network: there is no central control.

When we design a computational system for some range of tasks we are trying to specify a behavior. We specify the behavior by a finite description---the program. We can think of the program as a set of rules that describes how the state of the system is to evolve given the inputs, and what outputs are to be produced at each step. Of course, programming languages provide support for thinking about only a part of the state at a time: they allow us to separate control flow from data, and they allow us to take actions based on only a small part of the data at each step.

A computational system is very much a dynamical system, with a very complicated state space, and a program is very much like a system of (differential or difference) dynamical equations, describing the incremental evolution of the state. One thing we have learned about dynamical systems over the past hundred years is that only limited insights can be gleaned by manipulation of the dynamical equations. We have learned that it is powerful to examine the geometry of the set of all possible trajectories, the phase portrait, and to understand how the phase portrait changes with variations of the parameters of the dynamical equations. This picture is not brittle: the knowledge we obtain is structurally stable.

To support this focus on the development of interacting subsystems with multiply-redundant design requires the development of languages that allow description of the function and relationships between different parts of the overall system. These descriptions "let go" of the specific logic of individual processes to capture the interactions that are necessary for the redundancy and robustness of multiple processes. When stated in this way we see that it is the description of constraints between functional units of the system that are the essential parts of the collective description of the system. We propose to develop laguages that focus on describing these relationships/constraints among otherwise independent processes. We also propose to evaluate the relationship of such constraints to the robustness and flexibility of the system behavior as a whole.