

David F. Carlson
Copyright 1996, 2000, 2005
All Rights Reserved
Version 0.4

System Perspective for Problem Solving

A frequent problem we encounter with our clients is problem solving in a "system" environment. The "system" is the client's "system" -- network, hardware, software -- the works. There are people that "understand" systems -- those are the ones we try to hire -- and there are those that don't ever quite "get it". Many engineers that I've worked with have shown an ability to problem solve "in the small" -- a module or two of code -- but get bogged down by "large" systems.

There is a methodology to solving system problems which "system experts" use intuitively to do their jobs but which seems black art to the uninitiated. This paper attempts to document the mind-set of system level problem solving and how we can achieve linear convergence in an combinatorial solution space. The reader should have some knowledge of the architecture of their system-of-interest (subsystems, modules, processes, components, and interfaces).

Vignette One:

The Four Blind Men and the Elephant

The first blind man encounters the elephant's giant tusk, and thinks the unseen object is a mighty spear.

The second touches a massive leg, and understands it to be a tree trunk.

The third touches the rough skin of the beast's side, and concludes it to be a wall.

The fourth finds the tail, and knows it to be a rope.

The point of the story is that to their own perspective, each man is correct.

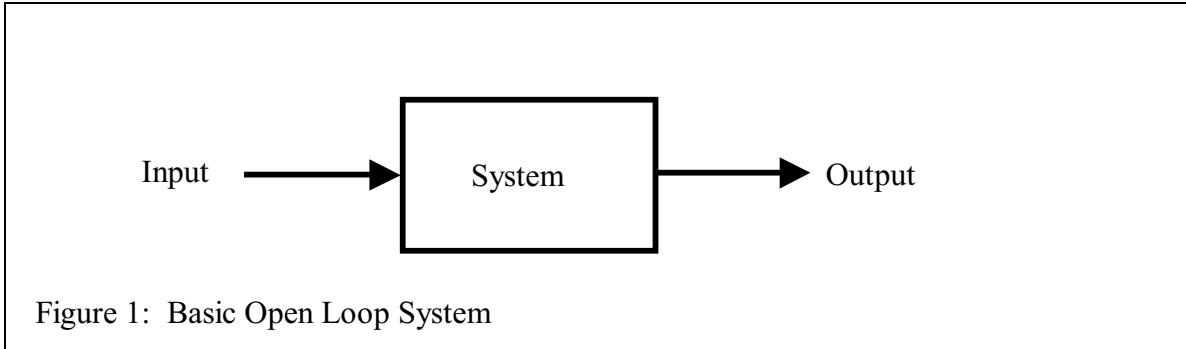
But each lacks the "systems" model -- **The Big Picture**tm.

In systems problems we often work with partial information. In particular, the people who "own" or control a sub-system component are quite often convinced they are touching a tree.

The system perspective will integrate sub-system domain knowledge to obtain a better view of the pacaderm.

Systems Model

It is useful to go back to system engineering kindergarten.

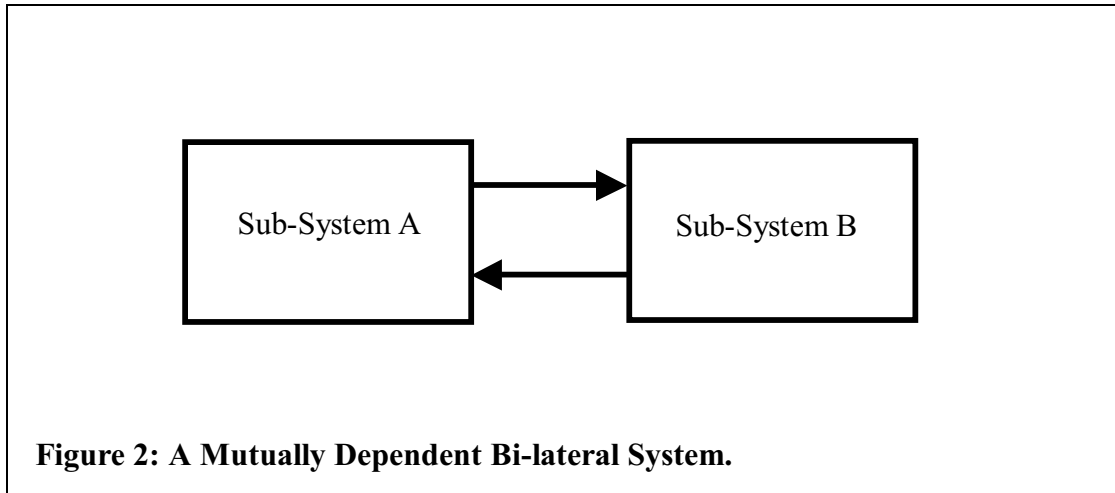


The most elementary building block of a "system" is the directed (ie., unidirectional) open loop process. Directed open loop processes are also known as unidirectional because the state of the input is unrelated to the output.

One can view this model as a data flow. In a data flow, the data-in gets chopped into some sort of data-out. In a control flow, the input is an exciting "event" and the output is a set of controls associated with the state transition caused by the input.

A good systems engineer should have a facility for understanding both data-driven systems models as well as control-driven systems model -- often expressed as state machines. Although useful, the relative merits of data-driven and control-driven systems models is out-of-scope for this paper.

A typical building block for more complex systems would include building blocks with bidirectional communication, As shown in Figure 2, systems that include bidirectional communications enable one system block to receive input and send responses to another system block.



A further step is to generalize the bi-lateral communication of N system blocks of 1..L inputs and 1..M outputs.

The systems task is to understand and accept the number of variables that affect the system behavior of interest can be decomposed to a finite number of subsystems with a finite number of inputs and outputs. (**Controlling** the input and **observing** the outputs is a **much** more difficult problem and out of the scope of this document.)

Why the Basics?

The key to managing a large system is to understand how to partition it to isolate the **area of interest**. Quite often, the **area of interest** is where the **bug** is.

The reason that partitioning is important is that "exhaustively" trying to "understand" or "test" every possibility only works for systems so trivial that a "systems person" would not be necessary at all.

Number of Binary "Variables"	Number of Combinations (2 ⁿ)
10	1024
32	4.3x10 ^{**9}
100	1.3x10 ^{**30}
1000	1.1x10 ^{**301}
Number of seconds in 1,000,000 years	3.2x10 ^{**13}

Even at a trial rate of 1 "guess" per second, a problem with only 17 independent binary variables would take a year. The project will not wait. And most systems problems have 100's or 1000's -- or 10's of thousands -- of "variables". So, if the systems experts cannot solve the problem "faster", the project will fail.

[Note: binary variables are a simplification of a complex systems problems where actual problems might use 32 or 64 bits to hold state information. The explosive growth for "real" problems is even worse than above!]

Partitioning solves the combinatorial math by "cutting" the state-space tree in "half". If theory, the 100 variable problem can be solved in 100 "correct" binary partitions rather than 10^{30} combinatorial "guesses".

Of course, this begs the questions "How does one make the correct partitions?"

Vignette Two

When the sound of thundering hoofs is heard,
think **HORSES**, not zebras.

(Unless one is in the Serengeti, in which case the reverse holds.)

Knowing How to Partition

Just how does one "know" how to partition? I haven't many college courses or websites devoted to the art/science of system partitioning. The concept for partitioning is to use knowledge and intuition also known as hunches to select strategic and hopefully fruitful places to "split" the problem. At each partition, a set of assertions are the the correctness of the input, output, timing, etc. will be made and tested on the "system-under-test".

A good "systems person" can use experience, prejudice (in a good sense) and guile to converge much faster than \log_2 to the "root cause" of the system behavior. The next section details a thought process to isolate the sound of horse hooves.

1. System Decomposition:

- What are the sub-systems of interest?
- How are they connected?
- What are their inputs and outputs?

If you lack the domain knowledge to understand the lay-of-the-land, obtain domain knowledge. Integrate domain knowledge into your system knowledge. It is better to not try to gain "complete" system knowledge in one chunk. Much of it will not be germane to the problem at hand. It is better to gain a small amount of insight into the sub-systems-of-interest, and then return to this step as necessary to refine the knowledge during the sub-system partitioning.

This step may be gained through discourse with engineers, formal documentation or (gasp) code inspection. It is very likely all three will be used to solve almost every problem.

2. What do we know? What do we trust? What do we NOT know?
 - a. Do we know/trust the "CPU" and "hardware"?
 - b. Do we know/trust the compiler?

Unless there is specific reason to suspect the CPU or the compiler, finding a Pentium floating point bug is definitely in the zebra category.

- c. Do we know/trust the OS or drivers or network protocol stack?

If the OS calls or network stack are "heavily-used" and "very stable" or if the problem does not seem to indicate an "OS" problem, partition these with the zebras.

- d. Do we know/trust the "inter-module" communication library?

If specific "heavily-used" and "very stable" sub-systems such as reading/writing to a disk drive or the inter-module communication library, are not directly implicated, partition them out...

- e. Do we know/trust the "input consumer" application task?
 - f. Do we know/trust the "output producer" application task?

If a particular sub-system interaction is suspect, what is known of that interaction? Can the producer/consumer be partitioned out of the rest of the module's interconnections? Is the input controllable? Is the output observable?

- g. Do we know a "bound" -- a point prior to this module the desired behavior of the inputs/outputs was correct?

If a "bound" is known, the sub-systems not in the input/output path of the "bounded" module can be partitioned out. The sets of problem bounds can partition a small cluster of sub-systems from a large whole.

- h. Do we have a subsystem that "detects" the "failure" or is the failure only in the human observable outputs?

If a sub-system is "detecting" a failure, whether it is the OS finding a memory access violation and halting the "offender" or a piece of application code finding something it "doesn't like" and logging an error, it is almost always the "other guy" that "caused" the failure. It is the nature of programs to validate their input -- to the OS, its input is the running program; to the application, it is the output of another sub-system. Although it is

possible the input validity checking is in error, it is 90% probable the input data is truly bad. A logged error is a good indication that a contract between a output producer and an input consumer has been violated. This narrows the initial partition significantly.

i. Do we know how to excite the system to produce the "failure"?

If the problem excitation is known, then the modules not involved in the handling of that set of excitations can be partitioned out.

j. Was the system behavior "correct" prior to the last set of modifications?

And if the last set of changes "broke" things, then the partitioning set should be just that set of changes. Everything else is partitioned out.

Applying the horse/zebra philosophy above, take a wide stroke first partition. Although we may eliminate only a handful of sub-system variables, we will likely eliminate more far more than half the search space.

k. Does the failure exist in other systems or is it unique to a single machine instance?

Most "single machine" failures from a large set of "identical machines" are as a result of marginal hardware or configuration. If configuration management exists, do a instance validation or configuration audit. If configuration management sufficient to validate a configuration instance does not exist, then you are **doomed**. Find something else to fill your interests. You are wasting your time doing systems work where basic systems configuration management does not flourish.

In the case of flaky hardware, use similar partitioning. Seek **common modes** -- where working subsystems share a common set of hardware such that it would be improbable that the wires, bus, chip that communicates to one device would lose its ability to communicate with another similar path. Use **differential modes**, unshared paths, to swap similar "known good" parts with "unknown" parts to increase the span of common mode spaces. The usual outcome is a failed/flaky board, sensor, actuator/motor, or wires. With modern crimping/snap connectors, wires (ie., cable assemblies) are often less reliable than solid state devices and circuit boards they interconnect.

Aphorism One

**When trying to find a needle in a haystack,
remove hay until all that remains is the needle.**

For each partition, devise a test to eliminate more of the space. Look at each of the possibilities above. Can a input excitation test case be constructed or "debug" code inserted to eliminate one side or the other of a "partition"?

Log₂ is Your Friend

Removing "half" the problem at a time (by testing each assertion) has the property of removing half the state-space of the system per test. This reduces the combinatorial complexity of the system to be linear as a function of state-variables instead of the naïve approach which would be power of two exponential in state-variables. This is the mathematical property known as a base-2 logarithm or Log₂ for short. The "trick" to large systems work is to strive for Log₂ convergence and to eschew exhaustive (combinatorial) closure rate.

What If the Partitioning Isn't Working?

Do not be afraid to recognize that the "partitioning" is wrong. The number of combinations ensures that at least half of the partitioning "guesses" are wrong. Learn from the mistakes -- they have still provided information. There are no wasted tests -- but failing to recognize a dead-end partition set is a waste of time.

Document your tested partition sets. This will save your sanity in the late evening or early morning.

When you suspect your current partition is wrong, re-partition "halfway" back to the last successful partition. This will still have the sought-after "closure" rate. A "failed" test is not a failure of recursive partitioning. It has provided information (and presumably insight) into the problem -- and we have learned that the asserted partitioning is incorrect. Only **failing to learn** something from each partitioning/experiment **is a failure**.

Conclusion

System problems are challenging to work -- and rewarding to solve. And although this article has dwelt mostly in the theory of Log₂ convergence and partitioning, the most important aspect to system problem solving is the people involved. People with vested interests in "their" pieces of the system. The problem solver must work with the sub-system engineers to gather information and to test theories (which are applied as the next set of partitions). Sometimes the sub-system engineers do not welcome your help. Some may even be threatened since they were not able to handle the problem. My advice is to use the very best people skills you have and to win allies with your ability to get the job done. They need your help even if they may not admit it, and by solving the problem, they will understand that everyone needs a helping hand sometimes.