# Complexity of verification

Every computation can ultimately be formulated as a boolean system. Let us begin, therefore, by asking how hard it is to verify a boolean expression.

Suppose our design problem is to design a device that adds two 2-bit binary numbers.

Here is a design to implement the adder, consisting of one equation for each bit of output, plus one additional equation defining a local subterm, the carry-bit from the 1s' place. ($\oplus$ stands for the *exclusive-or function*, $\wedge$ for "*and*" and $\vee$ for "*or*"):

$$\begin{aligned} s_0 &= a_0 \oplus b_0 \\ c_0 &= a_0 \wedge b_0 \\ s_1 &= c_0 \oplus a_1 \oplus a_2 \\ s_2 &= a_1 \wedge b_1 \end{aligned}$$

Maybe you can see that this implementation is incorrect, but let's pretend we don't.

To verify the implementation of the adder, we can evaluate it for a selected set of inputs, checking to see if it's three outputs are what they should be. To know what they should be, we need to know what numbers the inputs represent and whether the outputs represent the sum of those numbers. In this case we are asking whether

$$4s_2 + 2s_1 + s_0 = (2a_1 + a_0) + (2b_1 + b_0)$$

Here the binary (or truth) values $a_i$, $b_j$ and $s_k$ are re-interpteted as numbers, 1 for *true* and 0 for *false*.

Suppose we select the following inputs to check:

| *a* | | *b* | | | | *s* | | *verify* |
|---|---|---|---|---|---|---|---|---|
| $A_1$ | $A_0$ | $B_1$ | $B_0$ | $C_0$ | $S_2$ | $S_1$ | $S_0$ | $a + b \stackrel{?}{=} s$ |
| 0 | 0 | 0 | 1 | 0 | 0 | 0 | 1 | $0 + 1 \stackrel{?}{=} 1$ |
| 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 | $1 + 1 \stackrel{?}{=} 2$ |
| 0 | 1 | 1 | 0 | 0 | 0 | 1 | 1 | $1 + 2 \stackrel{?}{=} 3$ |
| 1 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | $2 + 0 \stackrel{?}{=} 2$ |
| 1 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | $3 + 1 \stackrel{?}{=} 4$ |

In each case, the outputs satisfy the specification of addition. However, we have not looked at all the possible inputs. Were we were to look at the case

$a = 2,\ b = 3$ the outputs we would see that the outputs are inconsistent with addition, because the correct implementation of $s_2$ should be:

$$s_2 = (a_1 \wedge b_1) \vee (c_0 \wedge (a_1 \vee b_1))$$

In the worse case we might need to evaluate all possible inputs, that is all 16 values for $a_1a_0b_1b_0$. It's not that bad for this particular problem, though:

(i) We might observe that the defining equations for each $s_k$ is *symetric* with respect to $a_0$ and $b_0$ and to $a_1$ and $b_1$, because all the boolean operations are commutative. On that basis, the tests $a_1a_0b_1b_0 = 1000$ and $a_1a_0b_1b_0 = 0010$ must produce the same outputs and are, therefore, redundant.

   *Question:* How many tests would this eliminate?

(ii) We can decompose the verification into two phases, the first using $a_0$ and $b_0$ to show that that $s_0$ and $c_0$ are correct; and the second using $c_0$, $a_1$, and $b_1$ to show that $s_1$ and $s_2$ are correct. The first phase has at most 4 cases and the second at most 8, for a total of 12 tests in the worst case.

Both of these improvements require insight into the design, and perhaps a time-consuming analysis of the verification task; and worse, raise the spectre of over-simplifying the problem. In both theory and practice, it is desirable to perform "black box" verification, where no details are available about how the implementation is done.

Unfortunately, this verification problem is inherently *exponential:*

Working Fact: A combinational boolean system with $N$ inputs and $M$ outputs requires $\mathcal{O}(M \cdot 2^N)$ computational effort, in the worst case, to exhaustively verify.

This is not simply a theoretical bound. There is no algorithm that can improve on this performance in all cases. Furthermore, common practical functions consume worst-case time to verify. To get some sense of what the numbers mean, here is a table showing the approximate time it would take to exhaustively verify an $N$-input combinational function, assuming one could perform

one million tests every second.

| $N$ | *time* |
|---|---|
| 0 | $*$ |
| 5 | $*$ |
| 10 | $*$ |
| 15 | $*$ |
| 20 | 1 sec. |
| 25 | 34 sec. |
| 30 | 18 min. |
| 35 | 10 hr. |
| 40 | 13 day |
| 45 | 1 yr. |

| $N$ | *time* |
|---|---|
| 50 | 36 yr. |
| 55 | $1,142$ yr. |
| 60 | $36,559$ yr. |
| 65 | $1,169,885$ yr. |
| 70 | $3.7 \times 10^7$ yr. |
| 75 | $1.2 \times 10^9$ yr. |
| 80 | $3.8 \times 10^{10}$ yr. |
| 85 | $1.2 \times 10^{12}$ yr. |
| 90 | $3.9 \times 10^{13}$ yr. |
| 95 | $1.2 \times 10^{15}$ yr. |
| 100 | $4.0 \times 10^{16}$ yr. |

Integrated circuits have hundreds of I/O pins. As we shall see later, The situation is even worse for *sequential systems*, which contain memories.

These numbers do not say that simulation is hopeless, but that it must be applied judiciously. They do say that brute-force black-box simulation is vulnerable to explosive performance costs and so must be used judiciously.

## Review of terminology

Let us pause here to review some terminology, drawing from the example of the 2-bit adder. The verification task involved relating two levels of description:

- A *specification*, describing the intended function of the design. In this case, the specification was given in several ways, the most explicit being the arithmetic equation

$$4s_2 + 2s_1 + s_0 = (2a_1 + a_0) + (2b_1 + b_0)$$

- An *implementation* describing how the specification would be attained. The implementation in this case is the system of boolean system

$$\begin{array}{rcl} s_0 & = & a_0 \oplus b_0 \\ c_0 & = & a_0 \wedge b_0 \\ s_1 & = & c_0 \oplus a_1 \oplus a_2 \\ s_2 & = & (a_1 \wedge b_1) \vee (c_0 \wedge (a_1 \vee b_1)) \end{array}$$

Both the specification and implementation are *expressions* and the task of verification is to compare them.

There can be many levels of description, and it is often the case that a given expression serves *both* as an implementation of the next higher (or more abstract) level and a specification for the next lower (or more concrete) level.

- An implementation *satisfies* a specification when a particular consistency relationship holds between the two. In our example, this relationship was mathematical equality between binary interpretations of the bits. Later, we will see circumstances where other relationships, such as logical implementation or behavioral equivalence, meet our needs.

- When we say "*verification*" we are referring to the equivalent of *exhaustive simulation*. In the case of the adder, this means doing all 16 trials, effectively. Automation of some kind is implied. Computer support usually performs the evaluations, keeps track of which trials have been done and which have not, and performs symbolic reasoning.

- The term *realization* refers to the design artifact, the physical product of design.

- We use the term "*testing*" to describe the execution of the realization on a sequence of inputs. Often, the purpose of testing is not to confirm that the device performs its logical function correctly but rather to determine if it contains any defects. In the fabrication process, some percentage of the circuits produced will malfunction, and these must be culled out of the product line.

  In software this distinction between verification and testing is often blurred. Testing for defects is no a significant activity since a software object is a digital copy. In hardware, it is easy to distinquish between the *description* of a design object and the circuit that realizes that description.

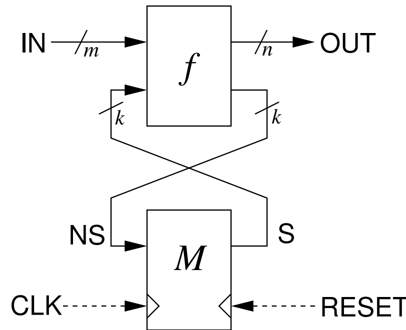
- We use terms like "requirement," "problem statement," "concept," "idea," to refer to the antecedent of a specification. The example began with the idea to "build a device that adds two 2-bit binary numbers." Since this English sentence is a written expression, one can argue that it is the original design specification. In practice, it is almost always the the case the first document describing design intent is written in a natural language.

  People often say that the difference between requirements and specifications is that the former describe *what* the design object is intended to do and the latter describes *how* that behavior is implemented (in successively greater detail). The fact is that almost all design descriptions do some of both.

## Verification of sequential systems

A *sequential system* contains memory in addition to combinational functions. It's behavior is not just a function of its inputs, but the current content of its memory, its *state*. In order to explore the complexity of sequential verification, we need a mathematical model of what these systems are. Sequential systems

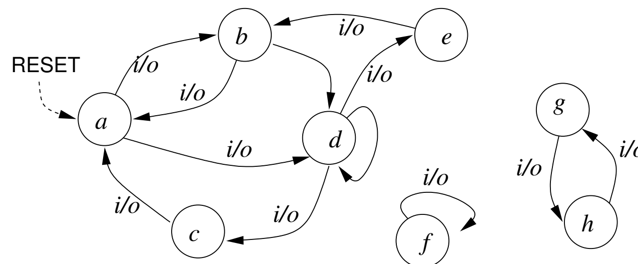
can be architecturally characterized as *finite state machines* (FSMs) having the structure:




The diagram contains two boxes, one representing the function of the machine and the other representing its memory. The combinational part, $f$ is an $n + k$ input, $m + k$ output boolean function. The memory, $M$ holds $k$ bits. There are two special inputs, a synchronizing signal CLK and a RESET signal. The FSM works in time like this:

- At the moment that CLK's value changes from 0 to 1, $M$ captures all $k$ bits present on NS. It holds these bits on S until the next time CLK "ticks."
- The combinational part $f$ computes its $k + m$ bit value from the values present on S and external inputs on IN. $m$ of the result bits make up output, OUT, and the rest are fed back to $M$ as NS.
- In a physical circuit, $f$ needs some time to reach electrical equilibium and stabalize its values. The purpose of $M$ is to hold the internal data steady until this happens. After sufficient time has elapsed for $f$ to perform its computation, CLK ticks again and the new values on NS are captured.
- A provision is needed to initialize the content of $M$. Regardless of what else is happening, when the RESET signal is asserted, a predetermined set of $k$ values is placed in $M$.

An mathematical model for digital behavior is a *finite-state automaton*:



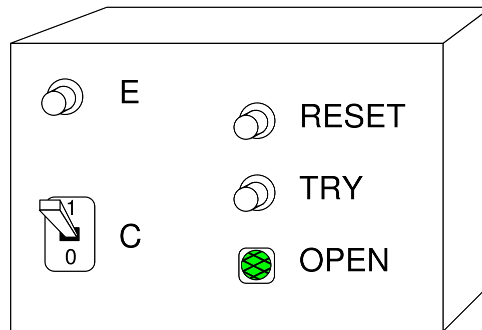

Each state of this structure represents one of the $2^k$ values that could be contained in memory $M$, above. Each edge is labeled by an expression specifying

an input, $i$, that causes that transition to be taken and an output, $o$ that is issued as it is taken. In this context an automata must statisfy the following properties:

- There are no "terminal" states or conditions. Each state must exactly one transition for every possible input. Unlike some other applications of automata, execution is non-terminating.
- There is a state for every possible $M$. As with states $f$, $g$ and $h$ above, it may be that some states cannot be reached from the *reset* state.
- There is one *reset*, or "start" state.
- We can allow for multiple transitions from some states for a given input. In such cases, we say the automaton is *nondeterministic*.

## Verifying sequential systems

Suppose you are given a device and told it is a combination lock. It looks like this:




**Inputs:**

- A RESET button. Whenever RESET is pushed, the device always goes to a predetermined state.
- A single toggle switch labeled C (COMBINAITON) for entering the bits of the combination.
- A pusbutton E (ENTER) telling the device when it to to read the next combination bit.
- A pushbutton TRY to be pushed when trying to open the lock.

**Outputs:** An OPEN signal that releases the physical lock. Think of it as a light that is on when the lock is open.

**Specification:** Here are the steps one must take to open the lock. Assume that the combination bits are: $c_1$, $c_2$, $c_3$.

| | | |
|---|---|---|
| 1. | Push RESET | |
| 2a. | Set C to $c_1$ | |
| 2b. | Push Enter | |
| 3a. | Set C to $c_2$ | The OPEN signal remains on until you hit RESET again. |
| 3b. | Push Enter | |
| 4a. | Set C to $c_3$ | |
| 4b. | Push Enter | |
| 5. | Push Try | |

**Other:** The device contains no more than 5 bits of memory.

## Demonstration by simulation

You are given a finite-state machine—or a black-box simulation model. Consider the problem of demonstrating that it correctly implements a 3-bit combination lock with combination `010`.

It is reasonable to assume that the RESET button works properly and that the device is a proper sequential system (or finite state machine).

Questions:

1. What properties must be demonstrated to establish that the device works correctly?

2. In the worst case, how many such steps will it take to verify all the necessary properties, assuming it contains exactly five bits of memory?

3. Are there features of the design behavior that you can exploit to improve on the worst case?

More specifically, assume your simulation procedure is a sequence of 3-part *steps* in which

- (a) you present the inputs;
- (b) the system takes a step; and
- (c) you observe the outputs, perhaps comparing them to an expected list of values or the outputs of a "specification model."

How many such steps would it take to demonstrate the correctness of the lock?

## Complexity of sequential verification

In the preceding combination-lock example is a pathological verification problem, because it involves verifying correctness for *both* valid *and* invalid opening sequences. The lock should open if *and only if* the operator enters a correct opening protocol. Furthermore, there is no bound on the duration between operator actions. It would be possible, for example, to include a "back door" for

opening the lock, for instance, if the operator hits TRY 57,001 times; or worse, by hitting TRY once, then again after exactly 57,001 clock cycles.

Thus, in the absence of additional information, verification by simulation is impossible. However, with knowledge of the number of state-bits *and* the effect of RESET, we can bound the number of steps needed.

(a) Given that the implementation has five state-bits, there are at most $2^5 = 32$ distinct states. Any execution path longer than 32 must contain a cycle, so we can limit each test to 32 steps.

(b) The "width" of the FSA is at most $2^4 = 16$ because there are just 4 bits of input. In fact, since one of the inputs is RESET the bound can be reduced to $2^3 = 8$.

(c) Thus the FSA has at most $32^8 = 1,099,511,627,776$ distinct, non-cyclic paths, so this is the number of 32-step sequences needed to exhaustively test the lock. For each sequence, we can use the specification to determine whether and when the OPEN output should be asserted.

(d) Of course, not all sequences would need to go all 32 steps, but this is an upper bound.

In general, then, exhaustive analysis of a finite-state machine with $n$ state bits and $m$ inputs requires at most

$$2^n \times (2^n)^{2^m} \in \mathcal{O}\big[(2^n)^{(2^m)}\big] \text{ simulation steps}$$

to exhaustively analyze. This hyper-exponential complexity is clearly infeasible for "large" FSMs.

In practice, however, sequential circuits with scores, even hundreds, of input & output bits can be (exhausively) verified, although not by simulation. By imposing a bound on the number of execution steps—typically just a few cycles—the verification problem can be reduced to a combinational *satisfiability* problem by "unrolling" the FSM, which can sometimes be solved using a *SAT* procedure.