Terminology

Levels of Description

The verification problem is one of comparing two or expressions describing a design at possibly different levels of detail. We typically call the higher, earlier or more abstract expression the *specification* and the lower, later or more concrete, expression the *implementation*.

The specification-implementation relationship is relative. The same expression can serve as an implementation at one level and as a specification for the next. In fact this is very common as the whole design process is taken into account.

An implementation adds detail to the design, describing more about how it is ultimately to be realized. The term *realization* is used for the final result of the design process—a circuit, object code, or other physical artifact of the design process. In the other direction, the initial statement of the design intent is sometimes called the *requirements* description. In principle, a requirements description says nothing about how the design is to be implemented, but only what the design is supposed to do. The distinction between "what" and "how" is often said to be the boundary line between requirements and specifications. In practice of course, the distinction is rarely so clear.

The "Implementation" Relationship

When one says that a implementation is "correct" one is just saying that it *satisfies*, or is in some sense consistent with, its specification. It is not very meaningful to say a program, standing alone, is correct; it must be correct with respect to some intended design goal.

Similarly, if implementation I satisfies specification S, a skeptic may ask, "Well, how did I know that S was correct in the first place?" Though naive, this question does reflect the very common experience of solving a poorly conceived or ambiguous problem. And it points out the fact that *design* is often a process of "negotiation" between evolving requirements and refining implementations.

In other words, the *satisfaction* relationship is rarely static. Once established, it is something to be maintained, like most other design entities. This is one of many reasons why automation plays a vital role in verification.

In very concrete instances, satisfaction reduces to logical or numerical equivalence. Generally, it is more like logical implication. This makes sense if you think of a design expression as describing the set of all its possible realizations. The more detailed the expression, the smaller the set. However, pure implication, $I \supset S$ is too weak, because it would allow a vacuous (false, or empty) I to implement anything. So "satisfaction" lies somewhere between equivalence and implication.

Property Specification versus Behavioral Specification

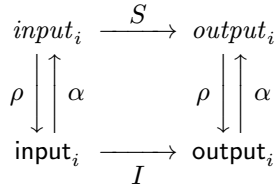
A specification sometimes centers on a statement describing a function, relation, or procedure for obtaining the intended outputs for a given set of inputs. Assertions about program correctness often take this form. For example, a program might be specified to sort the elements in a given list. *Functional specifications* of this kind may be augmented by constraints, such as a bound on the running time or the amount of memory allowed.

Another form of specification consists of a list of isolated *properties* that must be satisfied by the realization. This form is common in system specifications, where one may want to assure that a collection of processes doesn't deadlock, or responds with a certain priority.

Real specifications are usually a combination of behavioral and property aspects. The distinction lies in the kinds of tools applied to reason about a particular aspect, and often a collection of different tools are used in combination.

Verification

Verification is the process of ascertaining whether I *satisfies* S. Thus, in essence, it is the comparison of two design expressions, often, but not necessarily, at different levels of description. The prevailing practice of verification involves submitting a collection of sample inputs to both I and S and comparing the results. In all but the simplest cases this comparison involves some form of execution or interpretation of the two expressions. The diagram below gives a generalized diagram of the relationship.



The functions ρ and α represent transformations between concrete and abstract levels of representation. One or both of these transformations may be explicit in the verification process. Typically, a collection of representative inputs is $a_1,\ a_2,\ \ldots,\ a_n$ selected and the representations of both implementation inputs $\rho(a_1),\ \rho(a_2),\ \ldots,\ \rho(a_n)$ and expected implementation outputs $\rho(S(a_1)),\ \rho(S(a_2)),\ \ldots,\ \rho(S(a_n))$ are pre-calculated. Candidate implementation I is then simulated using these sample inputs and its outputs are compared for consistency against the $\rho(S(a_i))$.

Another way to perform this kind of verification is to randomly generate the $input_i$ or the input_i, simulate both I and S and compare the outputs.

The term *formal verification* has arisen to describe a verification process that involves a more direct comparison of S and I that is usually tantamount to a direct proof that I satisfies S for all possible inputs,

$$\text{For all inputs } a,\ I(\rho[a]) = \rho[S(a)]$$

Such a proof can take many forms, from a formal derivation in logic to a fully automated decision procedure, as we shall see.

Verification versus Testing

It has already been mentioned that verification is a comparison of two design *descriptions*. Testing, in contrast, involves looking at the actual behavior of the design realization. The difference is most readily seen in hardware, where the purposes for testing and verification are distinct.

The main purpose of verifying hardware is to assure that the design is free of logical errors. This is especially important because the cost of fabricating the realization is quite high, in terms of both money and time. Obviously, the verification is performed on a description of the eventual circuit, and the means of verification is either simulation or formal analysis. Once the circuit has been fabricated, the dominating concern is validating that the fabrication process introduced no physical flaws from contamination or miscalibration of the manufacturing equipment. The test inputs used to isolate such flaws are much different than those used to determine logical correctness.

While the distinction between verification and testing is more vague for software, it still exists. At one extreme, for example, we can perform a formal analysis of source code to derive conclusions about what it does without compiling it or executing the result. At the other, we could declare that embedded software cannot be "tested" until it has been installed in the physical environment in which it is to be deployed, since any other context merely represents a simulation of its actual working environment, say for the purpose of training or validation.

