Languages for Temporal Properties

Formal verification makes sense only when we have a formal specification that acts as the reference for verifying the correctness of a given design implementation. The notion of formal specifications is not new. Fundamentally we are aware that the functionality of all digital circuits may be formally expressed in terms of Boolean functions. For example, a half adder which receives two 1-bit inputs, $a$ and $b$ and produces two 1-bit outputs, namely the sum, $s$, and the carry $c$ may be specified completely by the Boolean functions:

\[
\begin{align*}
    s &= (a \land \neg b) \lor (\neg a \land b) \\
    c &= a \land b
\end{align*}
\]

Given an implementation of a half adder (say, in Verilog RTL) and the above equations, we can formally verify whether the RTL is correct. Typically this is done by translating both the RTL and the Boolean functions into some canonical representation of Boolean functions and then by checking whether the representations are isomorphic. There is a wide range of choices for Boolean function representation, including Binary Decision Diagrams (BDD) \[21\], Binary Moment Diagrams (BMD) \[22\], and ZBDDs \[82\]. There is also an arsenal of tools and libraries that support these representations, which facilitates the development of formal equivalence checking tools.

Given that the functionality of all digital circuits can be represented by Boolean functions, why do we need these new languages such as PSL and SVA? The reason is fundamental and is very significant towards understanding the basic tenets of formal property verification.

Let us consider the design of an arbiter having request lines $r_1$ and $r_2$, and grant lines $g_1$ and $g_2$. Suppose we specify the following properties to describe the functionality of the arbiter:

1. Whenever $r_1$ is raised, the arbiter must assert $g_1$ within the next two cycles.
2 Whenever $r_2$ is raised, the arbiter must eventually assert $g_2$.

3. The grant lines $g_1$ and $g_2$ are never asserted together.

Let us now compare the nature of this specification with the one for our half adder. In both cases it is possible to have more than one implementation, which satisfies the specification. Let us consider the following two implementations of the arbiter:

Implementation-1: The arbiter simply asserts $g_1$ and $g_2$ in alternate cycles – regardless of the status of the request lines.

Implementation-2: Whenever $r_1$ is raised, the arbiter asserts $g_1$ in the next cycle. In all other cycles, it asserts $g_2$.

Fig 2.1 shows the two implementations. It is obvious that these two implementations are not logically equivalent, that is, the Boolean functions representing their functionality are not identical. On the other hand, by specifying the Boolean functions for the sum and carry bits of the half adder, we have enforced that every implementation for the half adder must have the same Boolean functionality.

![Fig. 2.1. Two implementations of the arbiter](image)

At a high level of abstraction, the design intent is typically expressed in terms of several high-level correctness requirements. Specification of the exact Boolean functionality of the implementation may neither be practical, nor desirable at the high-level. Therefore properties allow us to express a more relaxed version of the specification, covering the critical correctness requirements of the design, but leaving room for design optimization by not specifying the exact Boolean functionality. Recent experience shows that specifying formal properties at the higher levels of the design flow of large and complex chips is both feasible and beneficial – it helps in capturing the essential elements of the design intent in an accurate and non-ambiguous way.

We have not yet justified the need for new languages for formal property specification. One may argue that partial specifications of the Boolean func-