On Applying Model Checking in Formal Verification

Håkan Hjort
Cadence Design Systems
Gothenburg, Sweden
hhjort@cadence.com

Abstract—Use of Hardware model checking in the EDA industry is widespread and now considered an essential part of verification. While there are many papers, and books, about SAT, SMT and Symbolic model checking, often very little is written about how these methods can be applied. Choices made when modeling systems can have large impacts on applicability and scalability. There is generally no formal semantics defined for the hardware design languages, nor for the intermediate representations in common use. As unsatisfactory as it may be, industry conventions and behaviour exhibited by real hardware have instead been the guides. In this tutorial we will give an overview of some of the steps needed to apply hardware model checking in an EDA tool. We will touch on synthesis, hierarchy flattening, gate lowering, driver resolution, issues with discrete/synchronous time models, feedback loops and environment constraints, input rating and initialisation/reset.

Design compilation, also known as elaboration and (quick) synthesis, is used to create a gate netlist from a hardware description language, commonly System Verilog. When done for implementation this often leverages any semantic freedom in order to create a more efficient implementation. In contrast, for verification we prefer to preserve all possible behaviour of any valid implementation choice. Assertions (properties) are normally handled similarly and translated to an automata representation that is then implemented by a gate netlist.

The gate netlist is a hierarchical representation of gates and their connections (to wires). Removal of hierarchy can largely be done replicating the logic. Most gate types represent combinatorial functions, these can be kept as is, or lowered to smaller subset of gate functions (such as in And-Inverter graphs). The state holding gates, (Flip-)Flops (edge sensitive) and Latches (level sensitive) require some more care to model their (as)synchronous behaviour.

Special care is also needed to model Tri-state gates (and weak drivers), which can either drive a value on their output or hold it isolated. Verilog wire uses a domain with 4-values 0,1,X,Z where Z is high-impedance / not-driving. Resolving the drivers means replacing the gates that drive a common wire with a model for the resolved logic value (and possibly checks for invalid/bad combinations).

It is common to have configurations, modes of operation and/or parts that should not be validated. Forcing some inputs to a fixed value is referred to as environment constraints. Mode complex constraints are instead normally considered part of the verification setup and handled as SV assumptions. The fixed values can be propagated into the gates to remove parts that become constant or disconnected.

For power and performance reasons it is common that designs are multi-clocked, or that clocks are gated (can be turned off and on). To have a global synchronous model for verification we need to reduce these multi-clock systems to a single global system (or tool) clock. This is often handled by mux-feedback added to the flops/latches along with logic generating the condition for the muxes. Inputs to the netlist may also have constraints at which rate/phase they can change. Rated inputs are free to take any value but only at certain points, clock generators follow a periodic pattern.

The use of a zero-delay timing model, meaning combinatorial gate output the function of their inputs without any delay, can give rise to problems when there are feedback loops in the netlist. Causing contradictions when a net would have two (or more) values, had there some delay in propagating the values through gates. There are 5 kinds of loops we can occur, through flops (data and clock), through latches (data and enable) and those only going through combinatorial gates. The ones going through flop data are benign, as its effect is mediated by the clock. The others need to be ruled out, or handled by modeling. Introducing some (fractional)-delay/steps seems an attractive approach, but establishing a bound on the number steps needed is challenging (and for some, no bound exists).

Initialisation, also referred to as reset, is commonly done by applying sequence of values to a subset of inputs. This aims to get the design from an arbitrary unknown state into a set of states from which it will have predictable behaviour. Part of the design flops might have asynchronous reset, others can receive values on the data input from other flops and inputs, yet others might be left uninitialised. Automating the computation of an (over-)approximation of the reset states will provide more information to the constructed model checking problem.