Chapter 2

Design and verification of digital systems

Before diving into the discussion of the various verification techniques, we are going to review how digital ICs are developed. During its development, a digital design goes through multiple transformations from the original set of specifications to the final product. Each of these transformations corresponds, coarsely, to a different description of the system, which is incrementally more detailed and which has its own specific semantics and set of primitives. This chapter provides a high level overview of this design flow in the first two sections. We then review the mathematical background (Section 2.3) and cover the basic circuit structure and finite state machine definitions (Section 2.4) that are required to present the core algorithms involved in verification.

The remaining sections presents the algorithms that are at the core of the current technology in design verification. Section 2.5 covers the approach of *compiled level logic simulation*. This technique was first introduced in the late 80's and it is still today the industry's mainstream verification approach. Section 2.6 provides and overview of formal verification and of a few of its more successful solutions; within this context Section 2.7 focuses on providing a more detailed presentation of *symbolic simulation*, since this technique will be at the basis of the novel solutions introduced by this thesis.

2.1 The design flow

Figure 2.1 presents a conceptual design flow from specifications to final product. The flow in the figure shows a top-down approach that is very simplfied: as we discuss later in this section, the reality of an industrial development is much more complex, involving many iterations through various portions of this flow, until the final design converges to a form that meets the specification requirements of functionality, area, timing, power and cost. The design specifications are generally presented as a document describing a set of functionalities that the final solution will have to provide and a set constraints that it must satisfy. In this context, the *functional design* is the initial process of deriving a potential and realizable solution from this design specifications and requirements. This is sometimes referred to as modeling and includes such activities as hardware/software tradeoffs and a micro-architecture design.

Because of the large scale of the problem, the development of a functional design is usually carried out using a hierarchical approach, so that a single designer can concentrate on a portion of the model at any given time. Thus, the architectural description provides a partition of the design in distinct modules, each of which contributes a specific functionality to the overall design. These modules have well defined input/output interfaces and protocols for communicating with the other components of the design. Among the results of this design phase is a high level functional description, often a software program in C or similar programming language, that simulates the behavior of the design with the accuracy of one clock cycle and reflects the module partition. It is used for performance analysis and also as a reference model to verify the behavior of the more detailed designs developed in the following stages.

From the functional design model, the hardware design team proceeds to the *Register Transfer Level (RTL) design* phase. During this phase, the architectural description is further refined: memory element and functional components of each model are designed using an Hardware Description Languages (HDL). This phase also sees the development of the clocking system of the design and architectural trade-offs such as speed/power.

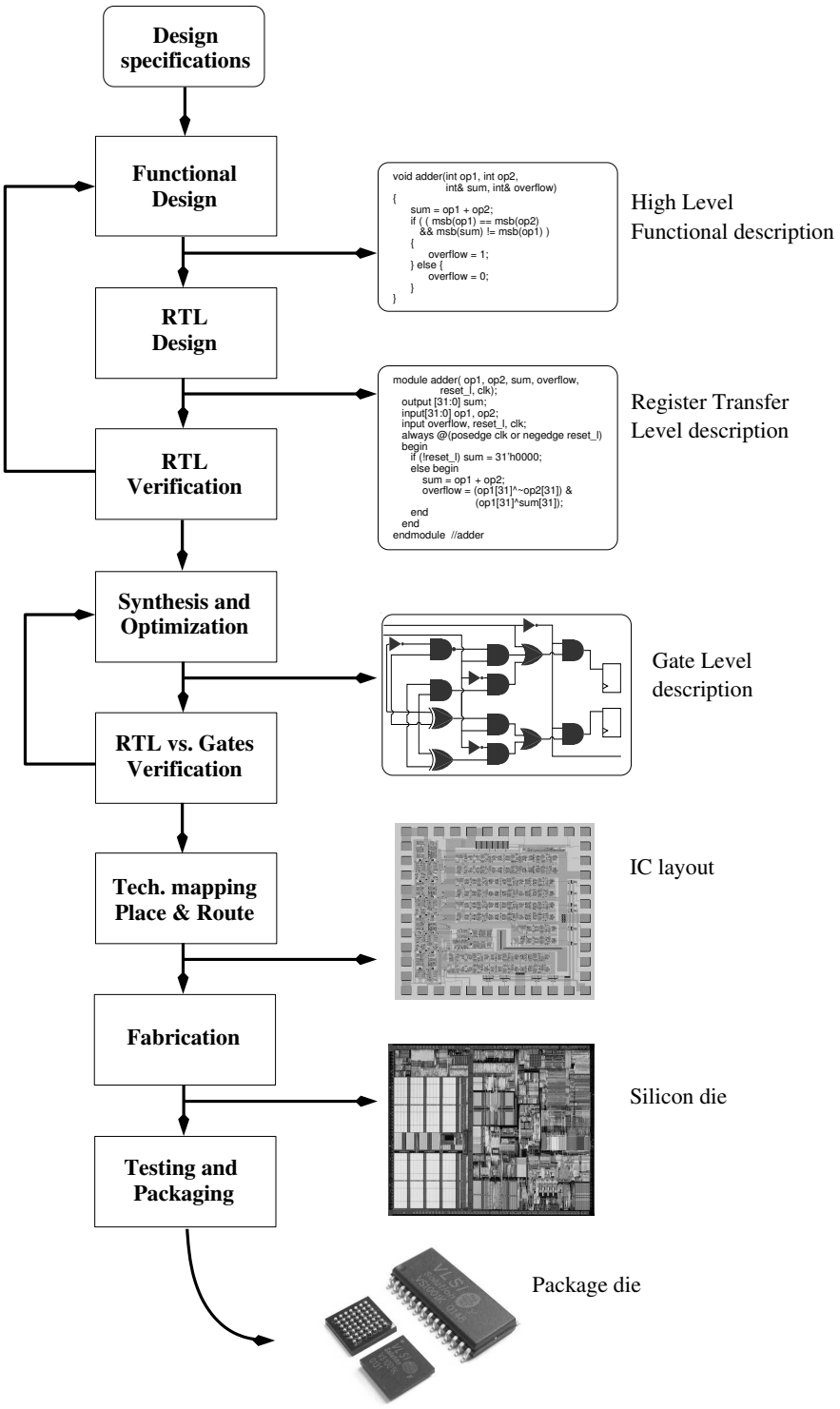

Figure 2.1: Design flow of a digital system

With the RTL design, the functional design of our digital system ends and its verification begins. *RTL verification* consists of acquiring a reasonable confidence that a circuit will function correctly, under the assumption that no manufacturing fault is present. The underlying motivation is to remove all possible design errors before proceeding to the expensive chip manufacturing. Each time functional errors are found the model needs to be modified to reflect the proper behavior. During RTL verification, the verification team develops various techniques and numerous suites of tests to check that the design behavior corresponds to the initial specifications. When that is not the case, the functional design model needs to be modified to provide the correct behavior specified and the RTL design updated consequently. It is also possible that the RTL verification phase reveals incongruences or overlooked aspects in the original set of specifications and this latter one needs to be updated instead.

In the diagram of Figure 2.1, RTL verification appears as one isolated phase of the design flow. However, in practical designs, the verification of the RTL model is carried on in parallel with the other design activities and it often lasts until chip layout. An overview of the verification methodologies that are common in today's industrial developments is presented in the next section.

The next design phase consists of the *Synthesis and optimization* of the RTL design. The overall result of this phase is to generate a detailed model of a circuits which is optimized based on the design constraints. For instance a design could be optimized for power consumption or the size of its final realization (IC area) or for the ease of testability of the final product. The detailed model produced at this point describes the design in terms of its basic logic components, such as *AND*, *OR*, *NOT* or *XOR* and memory elements. Optimizing the netlist, or gate level description, for constraints such as timing and power requirements is an increasingly challenging activity for current developments and it usually involves multiple iterations of trial-and-error attempts before it converges to a solution that satisfies both these requirements. Such optimizations may in turn introduce functional errors that require additional RTL verification.

While all the design phases, up to this point, have minimal support from Computer Aided Design (CAD) software tools and are almost entirely hand crafted by the design and verification team,

starting from synthesis and optimization, most of the activities are semi-automatic or at least heavily supported by CAD tools. Automating the RTL verification phase, is the next challenge that the CAD industry is facing in providing full support for digital systems development.

The synthesized model needs to be verified. The objective of *RTL versus gates verification*, or equivalency checking, is to guarantee that no errors have been introduced during the synthesis phase. It is an automatic activity requiring minimal human interaction that compares the pre-synthesis RTL description to the post-synthesis gate level description in order to guarantee the functional equivalence of the two models.

At this point, it is possible to proceed to *technology mapping* and *placement and routing*. The result is a description of the circuit in terms of a geometrical layout used for the fabrication process. Finally the design is *fabricated*, and the microchips are *tested and packaged*.

This design flow is obviously a very ideal, conceptual case. For instance, usually there are many iterations of synthesis, due to changes in the specification or to the discovery of flaws during RTL verification. Each of the new synthesized version of the design needs to be put through again all the subsequent design phases. One of the main challenges faced by design teams, for instance, is in satisfying the ever increasing market pressure to produce designs with faster and faster clock cycles. These tight timing specifications force engineering teams to push the limits of their designs by optimizing them at every level: architectural, in the components choice and sizing, and in placement and routing. Achieving timing closure, that is, developing a design that satisfies the timing constraints set in the specifications while still operating correctly and consistently, most often requires optimizations that go beyond the abilities of automatic synthesis tools and forces the engineers to intervene manually, at least in some critical portions of the design. Often, it is only possible to check if a design has met the specification requirements after the final layout has been produced. If these requirements are not met, the engineering team comes up with alternative optimizations or architectural changes and creates a new model that needs to go through the complete design flow.

2.2 RTL verification

As we observed in the previous section, the correctness of a digital circuit is a major consideration in the design of digital systems. Given the extremely high and increasing costs of manufacturing microchips, the consequences of flaws going unnoticed in system designs until after the production phase, would be very expensive. At the same time, RTL verification is still one the most challenging activities in digital system development: as of today, it is still carried on mostly with ad-hoc tests, scripts and often even tools developed by the design and verification teams specifically for the current design. In the best case, these verification infrastructure development can be amortized among a family of designs with similar architecture and functionality. Moreover, verification methodology still lacks any standard or even a commonly accepted approach, with the consequence that each hardware engineering team has its own distinct verification practices which often change with subsequent designs by the same team, due to the insufficient "correctness confidence level" that any of the current approaches provide. Given this scenario, it is easy to see why many digital IC development teams report that more than 70% of the design time and engineering resources are spent in verification, and why verification is thus the bottleneck in the time-to-market for integrated circuit development [5].

The workhorse of the industrial approach to verification is *functional validation*. The functional model of a design is simulated with meaningful input stimuli and the output is checked for the expected behavior. The model used for simulation is the RTL description The simulation involves applying patterns of test data at the inputs of the model, then using the simulation software or machine to compute the simulated values at the outputs and finally checking the correctness of the values obtained.

Validation is generally carried on at two levels: module level and chip level. The first verifies each module of the design independently. It involves producing entire suites of *stand alone tests*, each of which checks the proper behavior of one specific aspect or functionality of that module. Each test includes a set of input patterns to stimulate the module and a portion that verifies that the

output of the module corresponds to what is expected. The design of these tests is generally very time consuming, since each of them has to be handcrafted by the verification engineering team. Moreover their reusability is very limited because they are specific to each module. In general, the verification team develops a separate test suite for each functionality described in the original design specification document. Recently, a few CAD tools have become available to support functional validation: they mainly provide more powerful and compact language primitives to describe the test patterns and to check the outputs of the module, thus saving some test development time [34, 37, 5].

During chip level validation, the design is verified as a whole. Often this is done after a fair confidence is obtained about the correctness of each single module, and the focus is mainly in verifying the proper interaction between modules. This phase, while more compute intensive, has the advantage of being carried on in a more automatic fashion. In fact, input test patterns are often randomly generated, with the only constraint of being compatible with what the specification document define to be the proper input format to the design. During chip level validation it is usually possible to simulate both the RTL and a high level description of the design simultaneously and check that the outputs of the two systems and the values stored in their memory elements match one to one, at the end of each clock cycle.

The quality of all these verification efforts is usually analytically evaluated in terms of coverage: a measure of the fraction of the design that has been verified [43, 50]. Functional validation can provide only partial coverage because of its approach; the objective is thus to maximize coverage for the design under test. Various measures of coverage are in use: for instance *line coverage* counts the lines of the RTL description that have been activated during simulation. Another type is *state coverage* which measures the number of all the possible configurations of a design that have been simulated, that is, validated. This measure is particularly valuable when an estimate of the size of the total state space of the design is available: in this situation the designer can use state coverage to quantify the fraction of the design that she has verified.

With the increasing complexity of industrial designs, the fraction of the design space that the functional validation approach can explore is becoming vanishingly small, and it is showing more

and more that it is an inadequate solution to the verification problem. Since only one state and one input combination of the design under test are visited during each step of simulation, it is obvious that neither of the above approaches can keep up with the exponential growth in circuit complexity.

Because of the limitations of functional validation, new alternative techniques have received increasing interest. The common trait of these techniques is the attempt to provide some type of mathematical proof that a design is correct, thus guaranteeing that some aspect or property of the circuit behavior holds under every circumstance, and thus its validity is not limited only to the set of test patterns that have been checked. These techniques go under the name of *formal verification* and have been studied mostly in academic research settings for the past 25 years. Formal verification constitutes a major paradigm shift in solving the verification problem. As Figure 2.2 shows, with logic simulation we probe the system with a few handcrafted stimuli, while with formal verification we show the correctness of a design by providing analytical proofs that the system is compatible with each of the specifications. Compared to a functional validation approach, this is equivalent to simulating a design with all possible input stimuli and thus to providing 100% coverage.

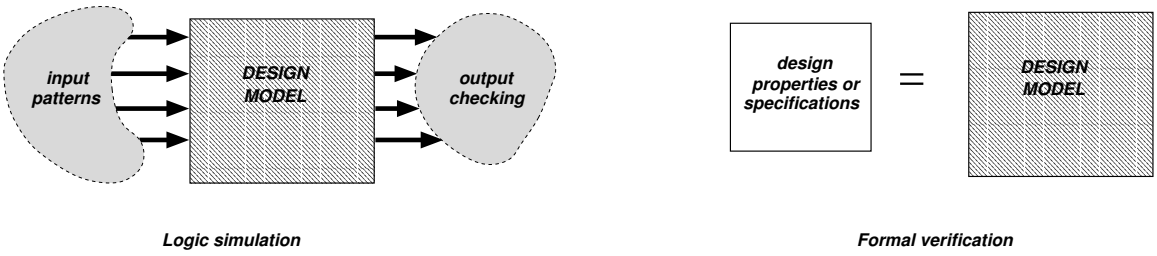

Figure 2.2: Approaches to verification

It is obvious that the promise of such thorough verification, makes formal verification a very appealing approach. While on one hand the solution to the verification problem seems to lie with formal verification approaches, on the other hand, these techniques have been unable to tackle industrial designs due to the complexity of the underlying algorithms, and thus have been applicable only to smaller components. They have been used in industrial development projects only at an experimental level, and so far they generally have not been part of the mainstream verification

methodology.

We now overview some of the fundamental methods for validation and verification to set the stage for the new techniques presented in this thesis. Before diving into this, we briefly review some mathematical concepts and the models and abstractions of digital systems used by these techniques.

2.3 Boolean variables and functions and their representation

We review here a few basic notions on Boolean algebra to set the stage for the following presentation.

Let $\mathcal{B}$ denote the Boolean set $\{0,1\}$. A symbolic variable is a variable defined over $\mathcal{B}$. A logic function is a mapping $\mathbf{F}:\mathcal{B}^n \rightarrow \mathcal{B}^m$. Hereafter, lower-case and upper-case letters will denote logic variables and functions, respectively. We will be mostly concerned with **scalar** functions $F(x_1,\cdots,x_n):\mathcal{B}^n \rightarrow \mathcal{B}$. We use boldface to indicate vector-valued functions. The i^{th} component of a vector function $\mathbf{F}$ is indicated by F_i.

The *1-cofactor* of a function F w.r.t. a variable x_i is the function F_{x_i} obtained by substituting 1 for x_i in F. Similarly, the *0-cofactor*, $F_{\overline{x_i}}$, is obtained by substituting 0 for x_i in F.

Definition 2.1. *Let $F:\mathcal{B}^n \rightarrow \mathcal{B}$ denote a non-constant Boolean function of n variables $x_1,\cdots,x_n$. We say that F* **depends** *on x_i if $F_{x_i} \neq F_{\overline{x_i}}$. We call* **support** *of F, indicated by $\mathcal{S}(F)$, the set of Boolean variables F depends on. In the most general case when F is a multiple output function, we say that $\mathbf{F}:\mathcal{B}^m \rightarrow \mathcal{B}^n$ depends on a variable x_i, if at least one of its components F_i depends on it.*

The **size** of $\mathcal{S}(F)$ is the number of its elements, and it is indicated by $|\mathcal{S}(F)|$. Two functions F,G are said to have **disjoint support** if they share no support variables, *i.e.* $\mathcal{S}(F)\cap\mathcal{S}(G)=\emptyset$.

Definition 2.2. *The* **range** *of a function $\mathbf{F}:\mathcal{B}^n \rightarrow \mathcal{B}^m$ is the set of m-tuples that can be asserted by $\mathbf{F}$, and it will be denoted by $\mathcal{R}(\mathbf{F})$:*

$$\mathcal{R}(\mathbf{F}) = \{y \in \mathcal{B}^m | \exists x \in \mathcal{B}^n, \mathbf{F}(x) = y\}$$

For scalar functions the range reduces to $\mathcal{R}(F) = \mathcal{B}$ *for all except the two constant functions* 0 *and* 1.

An operation between Boolean functions that will be needed in the following presentation is that of *generalized cofactor*:

Definition 2.3. *Given two functions* F *and* G*, the generalized cofactor of* F *w.r.t.* G *is the function* F_G *such that for each input combination satisfying* G *the outputs of* F *and* F_G *are identical.*

Notice that in general there are multiple possible functions F_G satisfying the definition of generalized cofactor. Moreover, if F and G have disjoint supports, than one possible solution for F_G is the function F itself.

A special class of functions that will be used frequently is that of *characteristic functions*. They are scalar functions that represent sets implicitly: They are asserted if and only if their input value belongs to the set represented.

Definition 2.4. *Given a set* $\mathcal{V} \subset \mathcal{B}^n$*, whose elements are Boolean vectors, its characteristic function* $\chi_V(x) : \mathcal{B}^n \to \mathcal{B}$ *is defined as:*

$$\chi_V(x) = \begin{cases} 1 & \textit{when} \quad x \in \mathcal{V} \\ 0 & \textit{otherwise} \end{cases} \tag{2.1}$$

When sets are represented by their characteristic function, the operations of set intersection, union and complementation correspond to *AND*, *OR* and *NOT* respectively, on their corresponding functions. This observation will be very useful in the following presentation.

2.3.1 Binary Decision Diagrams

Binary Decision Diagrams (BDDs) [17, 19] are a compact and efficient way of representing and manipulating symbolic Boolean functions. Because of this, BDDs are a key component of all symbolic techniques for verification. They form a canonical representation, making the testing of functional properties such as satisfiability and equivalence straightforward.

BDDs are rooted *directed acyclic graphs* that satisfy a few restrictions for canonicity and compactness. Each path from root to leaves, in the graph, correspond to an evaluation of the Boolean function for a specific assignment of its input variables.

Example 2.1. *Figure 2.3.a represents the BDD for the function* $F = (\overline{x} + \overline{y})pq$. *Given any assignment to the four input variables it is possible to find the value of the function by following the corresponding path from the root* F *to a leaf. At each node, the* 0 *edge (dashed) is chosen is the variable has a value* 0*, similarly for the* 1 *edge.*

Figure 2.3.b represents the BDD for the function $G = w \oplus x \oplus y \oplus z$. *Observe that the number of BDD nodes needed to represent XOR functions with BDDs, is* $2 \cdot$*#vars. At the same time, other canonical representations, such as truth tables or sum of minterms require a number of terms that is exponential with respect to the number of variables in the function's support.*

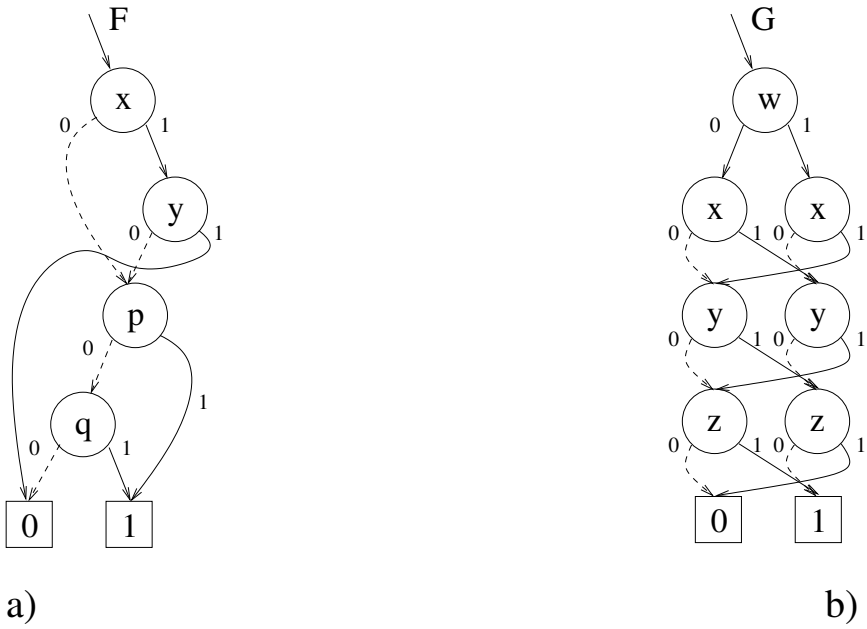

Figure 2.3: Binary Decision Diagrams

For a given ordering of the variables, it was shown in [17] that a function has a unique BDD representation. Therefore, checking the identity of two functions reduces to checking for BDD identity, which is accomplished in constant time.

The following definition formalizes the structure of BDDs:

Definition 2.5. *A BDD is a DAG with two sink nodes labeled "0" and "1" representing the Boolean functions* **0** *and* **1**. *Each non-sink node is labeled with a Boolean variable x_i and has two out-edges labeled 0 and 1. Each non-sink node represents the Boolean function $\overline{x_i}F_0 + x_iF_1$, where F_0 and F_1 are the cofactors w.r.t. x and are represented by the BDDs rooted at the 0 and 1 edges respectively.*

Moreover, a BDD satisfies two additional constraints:

1. *There is a complete (but otherwise arbitrary) ordering of the input variables and every path from source to sink in the BDD visits the input variables according to this ordering.*

2. *Each node represents a distinct logic function, that is, there is no duplicate representation of the same function.*

A common optimization in implementing BDDs is the use of *complement edges* [13]. A complement edge indicates that the connected function is to be interpreted as the complement of the ordinary function. When using complement edges, BDDs have only one sink node "1", whereas the sink node "0" is represented as the complement of "1".

Boolean operations can be easily implemented as graph algorithms on the BDD data structure by simple recursive routines making Boolean function manipulation straightforward when using a BDD representation.

A critical aspect that contributes to the wide acceptance of BDDs for representing Boolean functions is that in most applications the amount of memory required for BDDs remains manageable. The number of nodes that are part of a BDD, also called the BDD size, is proportional to the amount of memory required, and thus the peak BDD size is a commonly used measure to estimate the amount of memory required by a specific computation involving Boolean expressions. However, the variable order chosen can affect the size of a BDD. It has been shown that for some type of functions the size of a BDD can vary from linear to exponential based on the variable order. A lot of research has been done in finding algorithms that can provide a good variable order. While finding the optimal order is an intractable problem, many heuristics have been suggested to find sufficiently

good orders, from static approaches based on the underlying logic network structure in [52, 32], to dynamic techniques that change the variable order whenever the size of the BDD grows beyond a threshold [58, 12].

Binary Decision Diagrams are used extensively in symbolic simulation, one of the more successful formal verification methods. The most critical drawback of this method is its high demand on memory resources, which are mostly used for BDD representation and manipulation. This thesis introduces novel techniques that transform the Boolean functions involved in symbolic simulations through parameterization. The result of the parameterization is to produce new functions that have a more compact BDD representation, while preserving the same results of the original symbolic exploration. The reduced size of the BDDs involved in simulation translates to a lower demand of memory resources, and thus it increases the size of IC designs that can be effectively tackled by this formal verification approach.

2.4 Models for design verification

The verification techniques that we present in this thesis rely on a structural gate-level network description of the digital system, generally obtained from the logic synthesis phase of the design process. In the most general case, such networks are sequential, meaning that they contain storage elements like flipflops or banks of registers. Such circuits store state information about the system, thus the output at any point in time depends not only on the current input but also on historical values of the input. State transition models are a common abstraction to describe the functionality of a design. In this section we review both their graph representation and the corresponding mathematical model.

2.4.1 Structural network model

A digital circuit can be modeled as a network of ideal combinational logic gates and a set of memory elements to store the circuit state. The combinational logic gates we use are: *AND*, *OR*, *NOT* or

XOR. Figure 2.4 reproduces the graphic symbol we use for each of these types.

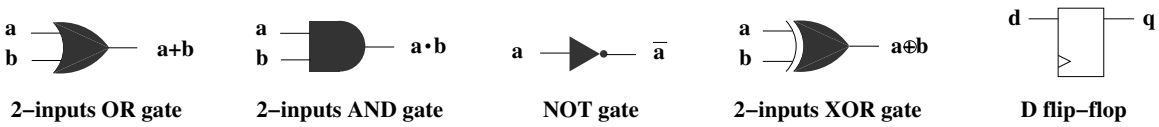

Figure 2.4: Graphic symbols for some basic logic gates

A synchronous sequential network has a set of primary inputs and a set of primary outputs. We make the assumption that the combinational logic elements are ideal, that is that there is no delay in the propagation of the value across the combinational portion of the network. Figure 2.5 represents such a model for a general network.

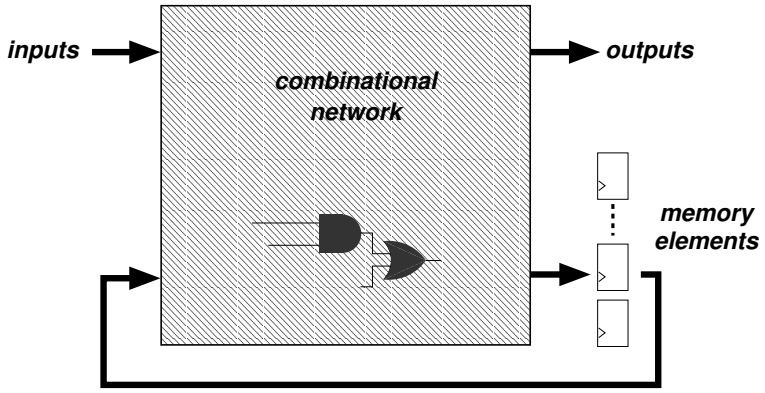

Figure 2.5: Structural network model

We also assume that there is a single clock signal to latch all the memory elements. In the most general case where a design has multiple clocks, the system can still be modeled by an equivalent network with a single global clock and appropriate logic transformations to the inputs of the memory elements.

Example 2.2. *Figure 2.6 is an example of a structural network model for a 3-bits up-down counter with reset. The inputs to the system are the* reset *and the* count *signals. The outputs are 3 bits representing the current value of the counter. The clock input is assumed implicitly. This system*

has four memory elements that store the current counter value and if the counter is counting up or down. At each clock tick the system updates the values of the counter if the count signal is high. The value is incremented until it reaches the maximum value seven, after, it is decremented down to zero. Whenever the reset signal is held high the counter is reset to zero.

The dotted perimeter in the figure indicates the combinational portion of the circuit's schematic.

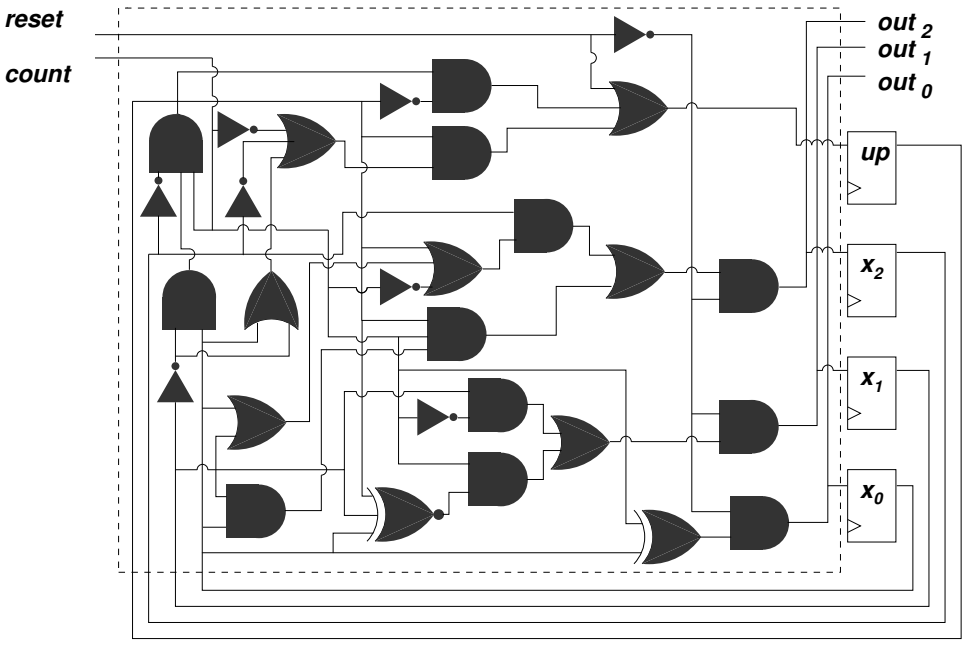

Figure 2.6: Network model of a 3-bits up/down counter with reset

2.4.2 State diagrams

A representation that can be used to describe the functional behavior of a sequential digital system is a Finite State Machine model.

Such model can be represented through state diagrams. A *state diagram* is a labeled directed graph where each node represents a possible configuration of the circuit. The arcs connecting the nodes represent changes from one state to the next and are annotated by the input which would cause such transition in a single clock cycle. State diagrams present only the functionality of the design,

while the details of the implementation are not considered and any implementation satisfying this state diagram will perform the function described. State diagrams also contain the required outputs at each state and/or at each transition. In a Mealy state diagram, the outputs are associated to each transition arc, while in a Moore state diagram outputs are specified with the nodes/states of the diagram. The initial state is marked in a distinct way to indicate the starting configuration of the system.

Example 2.3. *Figure 2.7 represents the Moore state diagram corresponding to the counter of Example 2.2. Each state indicates the value stored in the three flip-flops x_0, x_1, x_2 in bold and in the up/down flip-flop under it. All the arcs are marked with the input signal required to perform that transition. Notice also that the initial state is indicated with a double circle.*

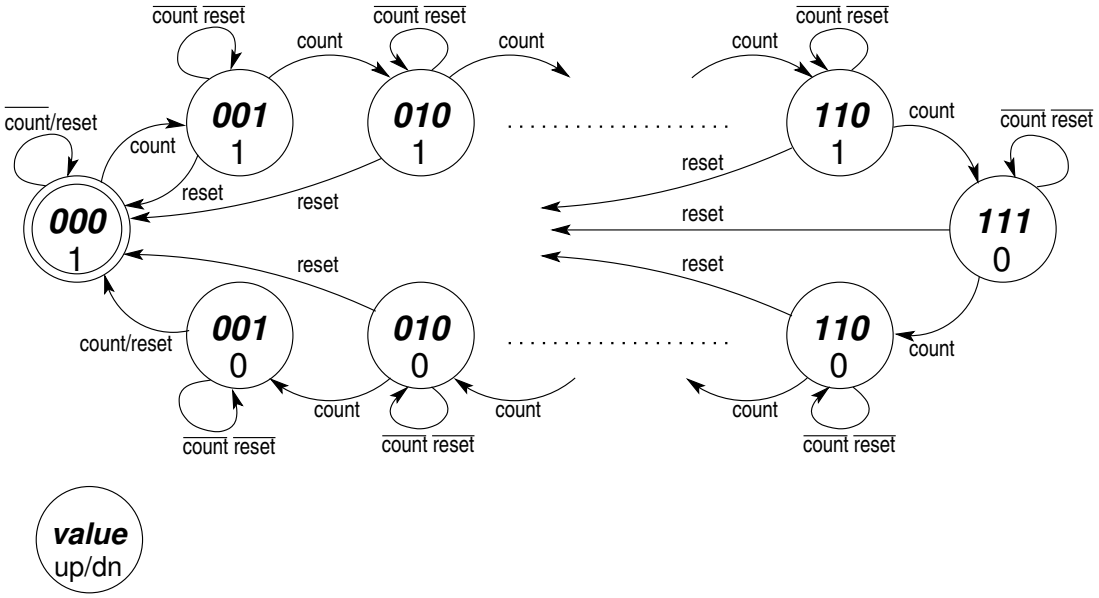

Figure 2.7: State diagram of a 3-bit up/down counter

In the most general case the number of configurations, or different states a system can be in, is much smaller than the number of possible values that its memory elements can assume.

Example 2.4. *Figure 2.8 represents the Finite State Machine of a 3-bits counter 1-hot encoded. Notice that even if the state is encoded using three bits, only the three configurations* $001, 010, 100$

are possible for the circuit. Such configurations are said to be reachable *from the Initial State. The remaining five configuration* $000, 011, 101, 110, 111$ *are said to be* unreachable*, since the circuit will never be in any of these states during normal operation.*

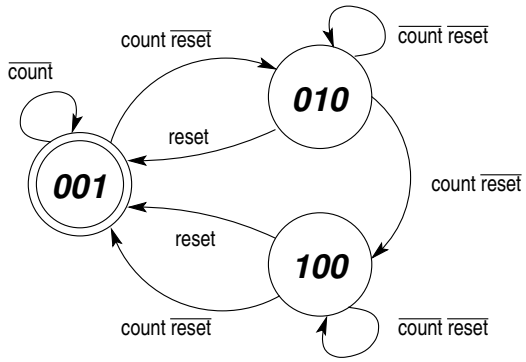

Figure 2.8: State diagram of a 1-hot encoded 3-bit counter

2.4.3 Mathematical model of Finite State Machines

An alternative way of describing a Finite State Machine is through a mathematical description of the set of states and the rules to perform transitions between states. In mathematical terms, a completely specified, deterministic Finite State Machine (FSM) is defined by a 6-tuple:

$$\mathcal{M} = (I, O, S, \delta, S_0, \lambda)$$

where:

- I is an ordered set $(i_1, \cdots, i_m)$ of Boolean input symbols,
- O is an ordered set $(o_1, \ldots, o_p)$ of Boolean output symbols,
- S is an ordered set $(s_1, \ldots, s_n)$ of Boolean state symbols,
- δ is the next-state function: $\delta : S \times I : \mathcal{B}^{n+m} \rightarrow S : \mathcal{B}^n$,
- λ is the output function $\lambda : S \times I : \mathcal{B}^{n+m} \rightarrow O : \mathcal{B}^p$,
- and S_0 is an *initial assignment* of the state symbols.

The definition above is for a Mealy type FSM, for a Moore type FSM the output function λ simplifies to: $\lambda : S : \mathcal{B}^n \rightarrow O : \mathcal{B}^p$.

Example 2.5. *The mathematical description of the FSM of Example 2.4 is the following:*

- $I = \{count, reset\}$,
- $O = \{x_0, x_1, x_2\}$,
- $S = \{001, 010, 100\}$,
- $\delta =$

count reset	*001*	*010*	*100*
0 0	*001*	*010*	*100*
0 1	*001*	*001*	*001*
1 0	*010*	*100*	*001*
1 1	*010*	*001*	*001*

- $\lambda = \{001 \rightarrow 001, 010 \rightarrow 010, 100 \rightarrow 100\}$,
- $S_0 = \{001\}$.

While the state diagram representation is often much more intuitive, the mathematical model gives us a mean of having a formal description of a FSM or, equivalently, of the behavior of a sequential system. The formal mathematical description is also much more compact, making it possible to describe even very complex systems for which a state diagram would be unmanageable.

2.5 Functional validation

The most common approach to functional validation involves the use of a logic simulator software. A commonly deployed architecture is based on the levelized compiled code logic simulator approach by Barzilai and Hansen [4, 33, 66].

Their algorithm starts from a gate level description of a digital system and chooses an order for the gates based on their distance from the primary inputs – in fact, any order compatible with

this partial ordering is valid. The name levelized of the algorithm is due precisely to this initial ordering by levels of the gates. The algorithm then builds an internal representation in assembly language where each gate corresponds to a single assembly instruction. The order of the gates and, equivalently, of the instructions, guarantees that the values for the instruction's inputs are ready when the program counter reaches that specific instruction. This assembly block constitutes the internal representation of the circuit in the simulator.

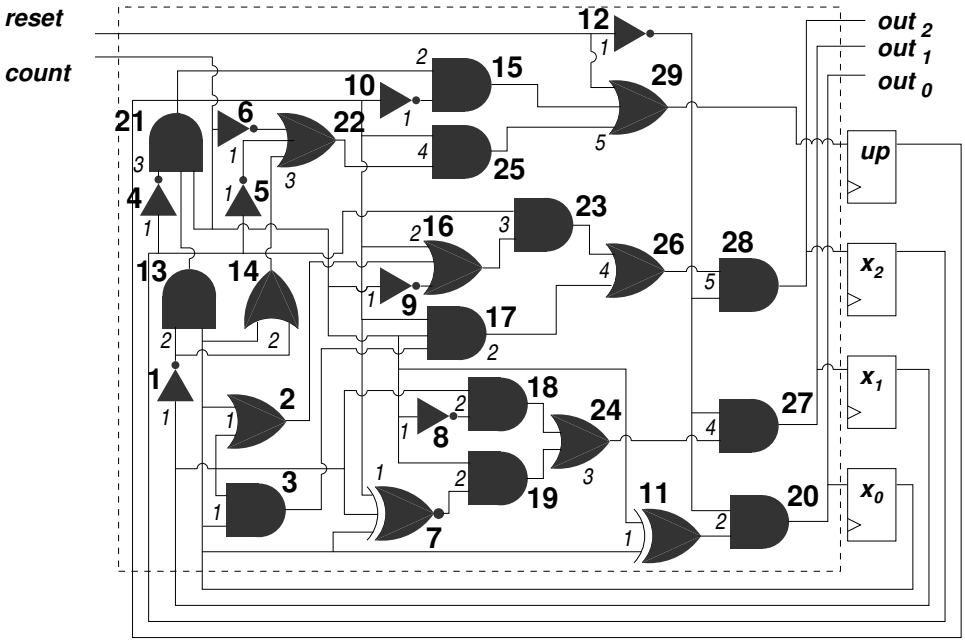

Figure 2.9: Compiled logic simulator

Example 2.6. *Figure 2.9 reproduces the gate level representation of the counter we used in Example 2.2. Each combinational gate has been assigned a level number (in italic in the picture) based on its distance from the inputs of the design. Subsequently, gates have been numbered sequentially (bold numbers) compatibly with this partial order. From this diagram it is possible to write the corresponding assembly block:*

```
1. r1 = NOT(x1)
2. r2 = OR(x0, x1)
```

```
3. r3 = AND(x0, x1)
.. ..
11. r11 = XOR(x0, count)
12. r12 = NOT(reset)
13. r13 = AND(r1, x0)
.. ..
27. r27 = AND(r24, r12)
28. r28 = AND(r26, r12)
.. ..
```

Note that there is a 1-to-1 correspondence between each instruction in the assembly block and each gate in the logic network.

The assembly compiler can then take care of mapping the virtual registers of the source code to the physical registers set available on the specific simulating host.

Multiple inputs gates can be easily handled by composing their functionality through multiple operations. For instance, with reference to Example 2.6, the 3-input *XNOR* of gate **7**, can be translated as:

```
7.     r7tmp = XOR(up, x1)
7bis. r7     = XNOR(r7tmp, x0)
```

At this point, simulation is performed by providing an input test vector, executing the assembly block and reading the output values computed. Such output values can be written to a separate file to be further inspected later to verify the correctness of the results:

Notice that, in first approximation, each of the assembly instructions can be executed in one CPU clock cycle of the host computer, thus providing a very high performance simulation. Moreover, this algorithm scales linearly with the length of the test vector and with the circuit complexity. The high performance and linear scalability of logic simulation are the properties that make this approach to functional validation widely accepted in industry.

```
Logic_Simulator(network_model) {
   assign(present_state_signals, reset_state_pattern);
   while (input_pattern != empty) {
      assign(input_signals, input_pattern);
      CIRCUIT_ASSEMBLY;
      output_values = read(output_signals);
      state_values  = read(next_state_signals);
      write_simulation_output(output_values);
      assign (present_state_signals, state_values);
      next input_pattern;
   }
}
```

Figure 2.10: Logic simulation - pseudocode

The model just described is called a cycle-based simulator, since values are simulated on a cycle by cycle basis. Another family of simulators are event-driven simulators: the key difference is that each gate is simulated only when there is a change of the values at its inputs. This alternative scheduling approach makes possible to achieve a finer time granularity in the simulation, and to simulate events that occur between clock cycles.

Various commercial tools are available that use one or both of the approaches described above, and that have proven to have the robustness and scalability to handle the complexity of designs being developed today. Such commercial tools are also very flexible: Practical cycle-based simulators allow for circuits with multiple clocks and the ability to mix cycle-based and event-based simulation to optimize performance [31]. When deployed in a digital system development context, simulators constitute the core engine of the functional validation process. However, the development of meaningful test sequences is the bulk of the time spent in verification. Generally the test stimuli are organized so that distinct sequences cover different aspects of the design functionalities. Each test sequence needs to be hand crafted by verification engineers. The simulated output values then are checked again by visual inspection. Both these activities require an increasing amount of engineering resources.

As mentioned before, some support in such development is available from specialized programming languages that make it possible for the verification engineer to use powerful primitives to create stimuli for the design, and to create procedures to automatically check the correctness of the output values [34, 49]. These test programs are then compiled and executed side by side with the simulation, exchanging data with it at every time step. Another module that is often run in parallel to the simulator or as a post-processing tool is a coverage engine: it collects analytical data on the portions of the circuit that has been exercised.

Since designs are developed and changed on a daily basis, it is typical to make use of verification farms – thousands of computers running logic simulators – where the test suites are run every day for weeks at a time.

Another common validation methodology approach in industry is pseudo-random simulation. Pseudo-random simulation is mostly used to provide chip level validation and to complement Stand Alone Testing validation at the module level. This approach involves running logic simulation with stimulus generated randomly, but within specific constraints. For instance, a constraint could specify that the reset sequence is only initiated 1% of time. Or it could specify some high level flow of the randomly generated test, while leaving the specific vectors to be randomly determined [2, 26, 69]. The major advantage of pseudo-random simulation is that the burden on the engineering team for test development is greatly reduced. However, since there is a very limited control on the direction of the design state exploration, it is hard to achieve a high coverage with this approach and to avoid producing just many similar redundant tests that have limited incremental usefulness.

Pseudo-random simulation is also often run using emulators, which conceptually are hardware implementations of logic simulators. Usually they use configurable hardware architectures, based on FPGAs (Floating Point Gate Arrays) or specialized reconfigurable components that are configured to reproduce the gate level description of the design to be validated [56, 35, 25]. While emulators can perform one to two order of magnitude faster than software based simulation, they constitute a very expensive solution because of the high raw cost of acquiring them and the time consuming process of configuring them for a specific design, which usually requires several weeks

of engineering effort. Because of these reasons, they are mostly used for IC designs with a large market.

Even if design houses put so much effort in developing tests for their designs and in maximizing the amount of simulation in order to achieve thorough coverage, simulation can only stimulate a small portion of the entire design and thus can potentially miss a subtle design error that might only surface trouble under a particular set of rare conditions.

2.6 Formal verification

On the other side of the verification spectrum are formal verification techniques. These methods have the potential to provide a quantum leap in the coverage achievable on a design, thus improving significantly the quality of verification. Formal verification attempts to establish universal properties about the design, independent of any particular set of inputs. By doing so, the possibility of letting corner situations go untested in a design is removed. A formal verification system uses rigorous, formalized reasoning to prove statements that are valid for all feasible input sequences. Formal verification techniques promise to complement simulation because they can generalize and abstract the behavior of the design.

Almost all verification techniques can be roughly classified in one of two categories: model-based or proof-theoretic. *Model-based techniques* usually rely on a brute-force exploration of the whole solution space using symbolic techniques and finite state machines representations. The main successful results of these methods are based on *symbolic state traversal* algorithms which allow the full exploration of digital systems up to a few hundreds latches. At the root of state traversal approaches is some type of implicit or explicit representation of all the states of a systems that have been visited up to a certain step of the traversal. Since there is an exponential relationship between the number of states and the number of memory elements in a system, it is easy to see how the complexity of these algorithms grows exponentially with the number of memory elements in a system. This problem is called the *state explosion problem* and it's the main reason for the very

limited applicability of the method. At the same time, the approach has the advantage of being fully automatic.

An alternative approach that belongs to the model based category is *symbolic simulation*. This method verifies a set of scalar tests with a single symbolic vector. Symbolic functions are assigned to the inputs and propagated through the circuit to the outputs. This method has the advantage that large input spaces can be covered *in parallel* with a single symbolic sweep of the circuit. Again, the bottleneck of this approach lies in the explosion of symbolic functions representations.

Symbolic approaches are also at the base of equivalency checking, another verification technique. In equivalence checking, the goal is to prove that two different network models provide the same functionality. In recent years this problem has found solutions that are scalable to industrial size circuits, thus achieving full industrial acceptance. Although checking the equivalence of two circuits has the advantage that it does not require to face the state explosion problem, nevertheless, there is hope that symbolic techniques will be the basis for viable industrial-level solutions to formal verification.

The other family of techniques, *proof-theoretic methods*, are based on abstractions and hierarchical methods to prove the correctness of a system [39, 42]. Verification in this framework uses theorem prover software to provide support in reasoning and deriving proofs about the specifications and the developed model of a design. They use a variety of logic representations. The design complexity that a theorem prover can handle is unlimited. However, currently available theorem provers require significant human guidance: even with a state-of-the-art theorem prover, proving that a model satisfies a specification is a very hand-driven process. Thus, this approach is still impractical for most industrial applications.

2.6.1 Symbolic Finite State Machine traversal

One approach used in formal verification is to focus on a property of a circuit and prove that this property holds for any configuration of the circuit that is reachable from its initial state. For instance, such property could specify that if the system is properly initialized, it never deadlocks. Or, in the

case of pipelined microprocessor, one property could be that any issued instruction completes within a finite number of clock cycles. The proof of properties such as these, require, first of all, to construct a global state graph representing the combined behavior of all the components of the system. After this, each state of the graph needs to be inspected to check if the property holds for that state. Many problems in formal hardware verification are based on reachable state computation of Finite State Machines (FSMs). A *reachable state* is just one that is reachable for some input sequence from a given set of possible initial states (see Example 2.4). This type of computation uses a symbolic breadth-first approach to visit all reachable states, also called reachability analysis. This approach, described below, has been published in seminal papers by Madre, Coudert and Berthet [27] and later in [64, 21].

In the context of FSMs, reachable states computations are based on implicit traversal of the state diagram (Section 2.4.2). The key step of the traversal is in computing the *image* of a given set of states in the diagram, that is, computing the set of states that can be reached from the present state with one single transition (following one edge in the diagram).

Example 2.7. *Consider the state diagram of Figure 2.7. The image of the one state set* $\{000-1\}$ *is* $\{000-1, 001-1\}$ *since there is one edge connecting the state* $000-1$ *to both these states. The image of the set* $\{110-1, 111-0\}$ *is* $\{000-1, 110-1, 111-0, 110-0\}$.

Definition 2.6. *Given a FSM* $\mathcal{M}$ *and a set of states R, its image is the set of states that can be reached by one step of the state machine. With reference to the model definition of Section 2.4.3, the image is:*

$$Img(\mathcal{M}, R) = \{s' | s' = \delta(s,i), s \in R, i \in I\}$$

It is also possible to convert the next state function $\delta()$ into a *transition relation* $TR(s,s')$, which holds when there is some input i such that $\delta(s,x) = s'$. This relation is defined by existentially

quantifying the inputs from $\delta()$:

$$TR(s,s') = \exists i \left[\bigwedge_{k=1}^{n} \delta_k(s,i) \equiv s'_k \right]$$

where δ_k represents the transition function for the kth bit. The transition relation can be represented by a corresponding characteristic function – see Definition 2.4 – $\chi_T R$ which equals 1 when $TR(s,s')$ holds.

We can then define the image of a pair $\langle \mathcal{M}, R \rangle$ using characteristic functions. Given a set of states R with characteristic function χ_R, its *image under transition relation TR* is the set *Img* having characteristic function:

$$\chi_{Img}(s') = \exists s \ \ (\chi_{TR}(s,s') \cdot \chi_R(s))$$

As we mentioned before, image computation is the key step of reachability analysis, the operation of finding the set of all the states of the FSM that are reachable. Reachability analysis involves determining the set of states that can be reached by a FSM, after an arbitrary number of transitions, given that it starts from an initial state S_0. The set of reachable states can be computed by a *symbolic breadth-first traversal* where all operations are performed on the characteristic functions. During each iteration, the procedure starts from a set of newly encountered states `from` and performs an image computation on the set to determine the new set of states `to` that can be reached in one transition from the states in `from`. The states to include in the `from` set are simply the states in the `to` set of the previous step that have not already been used in a previous image computation. Since the FSM that is being traversed has a finite number of states and transitions, these iterations will eventually reach a point where no new states are encountered. That point is called the *fixpoint*. The final accumulated set of states `reached` represents the set of all reachable states from the initial state S_0. In practice, all the sets involved in the computation are represented by their characteristic functions.

```
Machine_Traversal( FSM M ) {
   from = new = reached = initial_state;
   while (new ≠ ∅) {
      to = Img(transition_relation, from);
      new = To \ reached;
      reached = reached ∪ new;
      from = new;
   }
   return reached;
}
```

Figure 2.11: FSM state traversal - pseudocode

In general, the number of iterations required to achieve this fixpoint could be linear in the number of states of the FSM, and thus exponential in the number of memory elements of the system.

Symbolic traversal is at the core of the *symbolic model checking* approach to verification. The basic idea underlying this method is to use BDDs (Section 2.3.1) to represent all the functions involved in the validation and the set of states that have been visited during the exploration. The primary limitation of this approach is that the BDDs that need to be constructed can grow extremely large, exhausting the memory resources of the simulation host machine and/or causing severe performance degradation. Moreover, each image computation operation can take too long. The solution (exact or approximate) to these bottlenecks is still the subject of intense current research, in particular various solutions have been proposed that try to contain the size of the BDDs involved [57, 24] and to reduce the complexity of performing the image computation operation [23, 54].

Finally, another limitation of symbolic traversal is that it is not very informative from a design debugging standpoint: If a bug is found, it is nontrivial to construct an input trace that exposes it.

2.7 Symbolic Simulation

An alternative approach to symbolic state traversal is symbolic simulation. As described in Section 2.5, a logic simulator uses a gate-level representation of a circuit and perform the simulation by manipulating the Boolean scalar values, 0 and 1. Symbolic simulation differ from logic simulation

because it builds Boolean expressions rather than scalar values, as a result of circuit simulation. Consider the two *OR* gates in Figure 2.12.

Figure 2.12: Logic and symbolic simulation

On the left side, in performing logic simulation, the two input values 0 and 1 are evaluated and the result of the simulation produces a value 1 for the output node of the gate. On the right side of the figure, we perform a symbolic simulation of the same gate. The inputs are the two symbolic variable a and b, and the result placed at the output node is the Boolean expression $a+b$.

This approach is very powerful is two ways. First, at the completion of the symbolic simulation we have a Boolean expression that represents the full functionality of the circuit (in this example, a tiny one-gate circuit). This expression can be compared and verified against a formal specification of the desired outputs. Because of the quickly increasing complexity of these expressions, this comparison is only feasible for very small designs. Second, symbolic simulation can be seen as a way of applying multiple test vectors in parallel to a logic simulator, each symbolic variable representing implicitly both scalar values 0 and 1 and thus multiplying by a factor of two the number of equivalent vectors that are being simulated. For instance, the symbolic simulation of Figure 2.12 is implicitly applying four test vectors in parallel corresponding to $\{a = 0, b = 0\}$, $\{a = 1, b = 0\}$, $\{a = 0, b = 1\}$, $\{a = 1, b = 1\}$. This use of symbolic simulation is interesting also because it can be easily integrated in a logic simulation methodology where the amount of parallelism in the test vector can be tuned to the resources of the host.

2.7.1 The algorithm

The key idea of symbolic simulation consists of the use of mathematical techniques for letting symbols represent arbitrary input values for a circuit. One of the first works in this area is by King in 1976 [48], where he proposes a method of symbolic execution to verify software programs. More recently, in 1987, Bryant introduced in [20] a method for the symbolic simulation of CMOS designs. The algorithm presented in that work uses BDDs – see the previous Section 2.3.1 – as the underlying mathematical technique to represent the values associated with the nodes of the circuit.

In symbolic simulation, the state space of a synchronous circuit is explored iteratively by means symbolic expressions. The iterative exploration is performed with reference to a gate level description of the digital design. At each step of simulation each input signal and present state signal is assigned a Boolean expression; these expressions can be generally complex or extremely simple, such as simple Boolean variables or even constant values. The simulation proceeds by deriving the appropriate Boolean expression for each internal signal of the combinational portion of the network, based on the expressions at the inputs of each logic gate and the functionality of the gate. It is straightforward to see an analogy with the logic simulation approach described in Section 2.5, where we would operate on the same gate level description model for the design, but the inputs would be assigned to constant values instead of Boolean expressions and the internal operations would be in the binary domain.

In detail, the algorithm operates as follows: At time step 0, the gate level network model is initialized with the initial assignment S_0 for the each of the state signals and with a set of Boolean variables $IN_{@0} = \{i_{1@0}, \cdots, i_{m@0}\}$ for the combinational input signals. symbols. During each time step, the Boolean expressions corresponding to the primary outputs and the next state signals are computed in terms of the expressions at the inputs of the network. To do this, a Boolean expression is computed at each gate's output node based on the gate's functionality. Gates are evaluated in an order compatible with their distance from the input nodes, similarly to what is done in compiled level logic simulation. At the end of each step, Boolean expression are obtained for the primary outputs. The expressions computed for the memory elements' inputs are fed back to the state inputs

of the circuit for the next step of simulation.

```
Symbolic_Simulator(network_model) {
   assign(present_state_signals, reset_state_pattern);
   for (step = 0; step < MAX_SIMULATION_STEPS; step+1 ) {
      input_symbols = create_boolean_variables (m, step);
      assign(input_signals, input_symbols);
      foreach (gate) in (combinational_netlist) {
         compute_boolean_expression (gate);
      }
      output_symbols = read(output_signals);
      state_symbols  = read(next_state_signals);
      check_simulation_output(output_symbols);
      assign (present_state_signals, state_symbols);
   }
}
```

Figure 2.13: Symbolic simulation algorithm - pseudocode

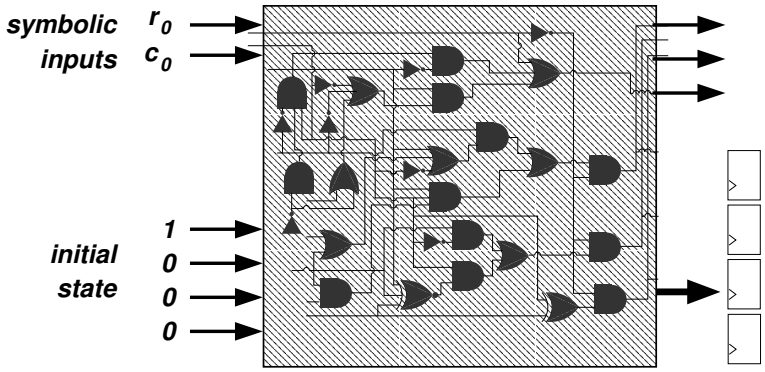

Figure 2.14: Symbolic simulation for Example 2.8 - Initialization phase

Example 2.8. *We want to symbolically simulate the counter circuit of Example 2.2. To set up the simulation we configure the state lines with the initial state* $\{000-1\}$ *as indicated in Figure 2.7. Then, we use two symbolic variables* r_0 *and* c_0 *for the two input lines.*

At this point the simulation proceeds computing a symbolic expression for each gate output node. Using the labels of Figure 2.9, we show some of the expressions for the first step of simulation:

`1.` 1 `2.` 0 ...

11.	c_0	12.	$\overline{r_0}$	13.	0
...		19.	0	20.	$\overline{r_0}c_0$

At the end of the first step, the expressions for outputs and flip-flop's inputs are:

$$\begin{aligned} out_0 &= x_0 = \overline{r_0}c_0 \\ out_1 &= x_1 = out_2 = x_2 = 0 \\ up &= 1 \end{aligned}$$

The expressions computed for the memory elements are used to set the state lines for the next simulation step, while the input lines will be set with new symbolic variables as indicated in Figure 2.15. At completion of the second simulation step, we obtain the following expressions:

$$\begin{aligned} out_0 &= x_0 = ((\overline{r_0}c_0) \oplus c_1)\overline{r_1} \\ out_1 &= x_1 = \overline{r_0 r_1}c_0 c_1 \\ out_2 &= x_2 = 0 \\ up &= 1 \end{aligned}$$

Note how the expressions involved in the simulation become increasingly complex at each time step.

Notice that new Boolean variables are created at every simulation step, one for each of the combinational primary inputs of the network. Thus, the expression obtained for the outputs and the state signals at the end of each step k will be functions of variables in $\{IN_{@0}, \cdots, IN_{@k}\}$. The vector of Boolean functions obtained for the state symbols $\mathbf{ST}_{@\mathbf{k}} : \mathcal{B}^{mk} \rightarrow \mathcal{B}^n$ represents all the states that can be visited by the circuit at step k. The state symbols $\mathbf{ST}_{@\mathbf{k}}$ represent the states at step k in implicit form, that is, any distinct assignment to the symbolic variables $\{IN_{@0}, \cdots, IN_{@k}\}$ will evaluate the state expressions to a valid state vector that is reachable in k steps from the initial state S_0. Viceversa, each state that is k steps away from the initial state corresponds to a least one

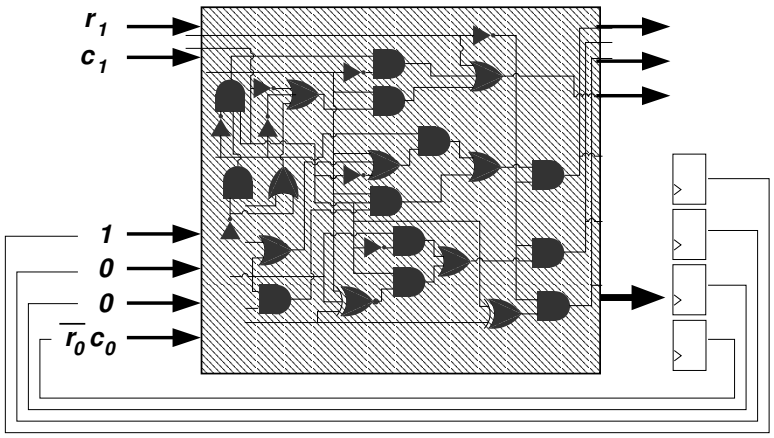

Figure 2.15: Symbolic simulation for Example 2.8 - Simulation Step 2

evaluation of the symbolic variables.

The procedure just described is equivalent to propagating the symbolic expressions through a time-unrolled version of the circuit, where the combinational portion is duplicated as many times as there are simulation steps. Figure 2.16 shows this iterative model and input and output nodes for the symbolic variables and expressions, respectively.

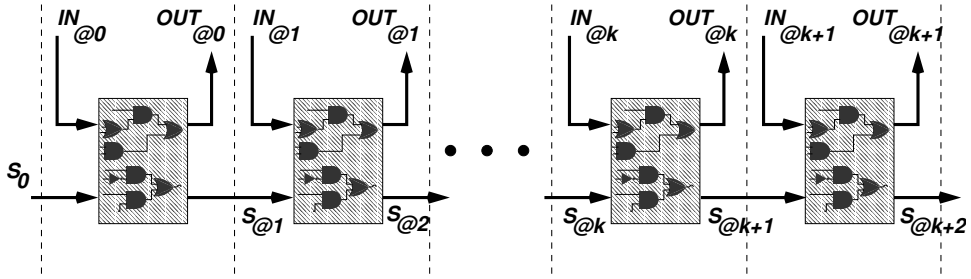

Figure 2.16: Iterative model of symbolic simulation

Design errors are found by checking at every step that the functions $\mathbf{OUT}_{@\mathbf{k}} : \{in_{@0}, \cdots, in_{@k}\} \rightarrow \mathcal{B}^p$ represents a set of legal values for the outputs of the circuit. When an illegal output combination is found, $\mathbf{OUT}_{@\mathbf{k}}$ reports all the possible input combinations that expose it. To generate a test sequence that exposes the error, is sufficient to find an evaluation of all the symbolic variables that simultaneously satisfy the error condition. If the correct output vector at time k is given by the

Boolean vector $C \in \mathcal{B}^p$, all the assignments that satisfy the expression Err:

$$Err = \bigwedge_{i=1}^{p} (\mathbf{OUT}_{@\mathbf{k},i} = C_i)$$

are valid test sequences that expose the design error. This approach can be easily generalized to verify more complex properties where the output signals have to satisfy complex relations expressed in terms of the input symbols.

2.7.2 The challenge in symbolic simulation

While theoretically the simulation can proceed indefinitely, the representation of the Boolean expressions involved eventually requires memory resources beyond those available in the simulation host. Similarly to the state traversal algorithm, the bottleneck of this approach lies in the explosion of the BDD representations. Various techniques have been suggested to approximate the functions represented in order to contain the size of the BDDs within reasonable limits.

In [67], Wilson presents a technique that imposes a hard limit on the size of the BDDs involved in the simulation. Whenever this limit is reached, one or more of the symbolic variables are evaluated to a constant, so that the Boolean expressions, and consequently, their BDD representations, can be simplified. The simulation step is then re-simulated with the other constant value for each of the simplified symbolic variable, until complete expressions are obtained for the outputs of the network.

A different approach is chosen by Bergmann in [6], where the design is abstracted, so that its size can be within reach of a simulator. Different abstractions are chosen based on coverage holes – areas of the design that had not been explored yet – with the objective that each new abstraction will improve the current coverage.

In [36], Bertacco *et al.* attempt to overcome the limitations of the single techniques presented in this chapter by using collaborative engines: symbolic simulation interacts with logic simulation in achieving the most coverage of the design within boundaries of time and memory usage, while

symbolic state traversal is used with abstraction techniques to prove some portions of the design's state space as non reachable, and thus prove that could they cannot be covered by the simulation engines. The result is an integrated software tool that supports the designer in "classifying" the state space of the IC design into reachable and unreachable and produces efficient and compact tests to visit the reachable portion of a design.

Even if some of these efforts provide a major contribution in making the symbolic simulation approach much more attractive for use in industrial settings, there is still a lot to be conquered and the functional verification of digital systems remains a challenge for every hardware engineering team. The core observation underlying the work in this thesis is that symbolic simulation traverses the states of a digital system carrying across each simulation step much more information than it is needed to verify the system. In fact, it uses complex Boolean expressions to describe each of the states that can be visited during each simulation step and how they relate to the input symbolic variables. However, we need much less information in order to achieve our objective to be able to identify which states can be visited at each time step. In fact, for this latter goal, any encoding of the set of states reached is sufficient. Thus, the remainder of this thesis develops techniques and theoretical work to discover new efficient encodings of the state sets involved in simulation. These new encodings, or parameterization, have more compact representations than the original ones, and thus allow for a memory efficient symbolic simulation, that presents much better robustness and scalability characteristics.