Chapter 2

CONSTRAINED RANDOM SIMULATION

Constrained random simulation addresses two major problems of the traditional testbench approach: being procedural as opposed to declarative and being enumerative as opposed to comprehensive. Test cases and coverage metrics are planned according to functional specifications, which are more declarative than procedural. Converting a declarative specification into test generation procedures is generally a synthesis process, one that is not particularly suitable for manual handling. Additionally when errors are made, the debugging is difficult since the specification is not executable.

The enumerative approach of traditional testbenches is also time-consuming and error-prone. Although some degree of enumeration, from coverage metrics to functionalities to be tested, is unavoidable in simulation, a simple-minded exercise of all the functionalities will not do the job because the most elusive bugs reveal themselves only under such convoluted scenarios as a series of cross states among several state machines that are set up only by a particular transaction sequence. Besides writing as many directed tests as possible, the only reasonable solution is to employ random simulation. However, the random simulation capability offered by existing HDLs is very limited.

Constrained random simulation attempts to address all of the above issues. First, as constraints are declarative, they are closer to specifications in that they mimic written language. Furthermore, because constraints are executable in the sense that stimuli are generated directly from constraints, manual construction of a testbench is eliminated. In other words, the use of constraints moves testbench writing to a higher abstraction level, one more closely resembling that of the specification. This alleviates the effort and pitfalls associated with writing a procedural-level testbench. This benefit closely resembles the advantage of logic synthesis, that is RTL to gate: higher productivity with fewer errors.

Second, constraint solving allows for flexible randomization so that case enumeration can be drastically reduced or eliminated altogether. Third, constraints can be easily converted to monitors and used at the result checking stage. This generator/monitor duality of constraints is especially useful in hierarchical verification where interface constraints for one design block are checked as properties when verifying the neighboring blocks or blocks at a higher level of hierarchy.

Failures in simulation can reveal bugs both in the specification and in the constraint-based testbench. It is therefore possible to start simulation with a fully verified testbench. Finally, in constrained random simulation, feedback from functional coverage can be used to direct constraint solving and randomization towards the not-yet-explored behavior, bringing into reality the so called "reactive testbench."

In the remainder of this chapter we discuss the main concepts of constrained random simulation, and demonstrate these concepts by describing a real constrained random simulation tool, the *Simgen* [YSP+99, YSPM01] system developed at Motorola.

2.1 Constraints for Test Generation

Constraints are formal and unambiguous specifications of design behaviors and aspects. In the context of constrained random simulation, constraints define what input combinations can be applied and when.

There are basically two types of constraints in constrained random simulation: the environment constraints and constraints used as test directives. The former defines the interface protocol, which must be strictly followed. The latter are used on top of the environment constraints to steer the simulation to the desired test scenarios, the so-called "corner cases." In other words, constrained random simulation is meaningful only if the test stimuli meet certain requirements of the environment, and can become more interesting with the help of additional test directives.

A constraint can be as simple as a Boolean formula defined over design signals. As an example, consider a typical assumption about bus interfaces: the "transaction start" input (ts) is asserted only if the design is in the "address idle" state. This can be captured in the following constraint:

```
ts -> (addr_state == Addr_Idle)
```

When the constraint formula gets too complicated, it may be a good idea to define intermediate subformulas upon which the constraint formula is defined. The use of macros and functions enables the sharing of common subformulas.

Auxiliary variables can be used to remember past states to constrain the sequential behavior of the inputs. The following statement from the PCI bus protocol specification is such an example.

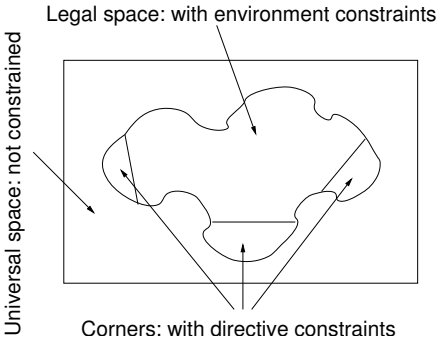

Figure 2.1. Environment and directive constraints.

> Once a master has asserted Irdy#, it cannot change Irdy# or Frame# until the current data phase completes regardless of the state of Trdy#.

This statement corresponds to the constraint below:

```
Irdy# && !data_complete -> Irdy#==prev_Irdy# && Frame#==prev_Frame#.
```

The auxiliary variables prev_Idry# and prev_Frame# take the values of Irdy# and Frame#, respectively, from the last clock cycle.

A class of constraints — constraints that specify bundles of sequences — can efficiently be expressed in grammar rules similar to the ones used in program language parsers. SystemVerilog allows constraints written in Backus Naur Form (BNF) to do just that. For example, the sequences

```
add popf mov
add pushf mov
dec popf mov
dec pushf move
```

can be captured with the BNF:

```
sequence : top middle bottom;
top : add | dec;
middle: popf | pushf;
bottom: mov;
```

In the most flexible form, a constraint can be facilitated by an auxiliary state machine to model complex sequential behavior.

EXAMPLE 2.1 We construct the constraint for the scenario:

> If a *request* is sent (asserted), then it will be sent continuously until after an *acknowledge* is sent.

It is understood that the *request* is an input variable and the *acknowledge* is a state variable. The scenario will be specified with a state machine and a

(Boolean) constraint expression. The state machine monitors a violation of the above scenario: a *request* is sent and then withdrawn without receiving an *acknowledge* first.

```
always @ (posedge clk) begin
    if (reset)
        state <= START;
    else begin
        case (state)
        START:
            if (req)
                state <= CHECK;
        CHECK:
            if (!req)
                state <= ERROR;
            else if (ack)
                state <= START;
        ERROR:
            state <= ERROR;
        endcase
    end
end v
```

And the constraint

```
state != ERROR
```

assures that input req is asserted in state CHECK, because this is when req was sent but /small ack has not been received, which is exactly the scenario we are trying to specify. □

For commonly encountered constraint scenarios, the above example being one, the use of templates can save a great deal of effort. The most powerful templates are from the assertion languages. For example, the Open Verification Library (OVL) provides a set of Verilog/VHDL monitors (in state machines) that may be directly used as properties or constraints. The scenario described in the previous example can be captured with the following OVL constraint:

```
assert_window #(0, 1) handshake (clk, !reset, req, req, ack);
```

In another assertion language, the Property Specification Language (PSL), the above is written as:

```
assume always req -> req Until_ ack @ clk;
```

Unlike OVL, PSL is not a library of simulatable monitors. To use the above constraint in simulation, it must first be converted to a state machine, either implicitly if the simulator supports PSL, or explicitly otherwise.

To summarize, a constraint definition can be broken into two parts: the auxiliary modeling logic, and the Boolean expression. The modeling logic is either made up of macros, functions, auxiliary variables, and state machines,

that are directly based on HDL or C/C++, or is constructed from assertion language constructs. The Boolean expression is simply a formula over signals from the design and the modeling logic. The expression evaluates to true or false. Assertion languages and constraint languages that specialize in modeling constraints are the topics of the next two chapters.

2.2 Constraining Design Behaviors

Constraints work on a DUV through synchronous composition: simply connecting the corresponding signals in the modeling logic of constraints to those of the DUV. The result is called the *product machine* of the constraints and the DUV. The direction of flow divides the signals into two groups: the signals that leave the DUV are called the state signals, whose values are set by the DUV during simulation, and the signals that flow into the DUV are called inputs. The goal of constraint solving is to provide the input assignment based on the state signal values. Figure 2.2 illustrates a constraint/DUV product machine in

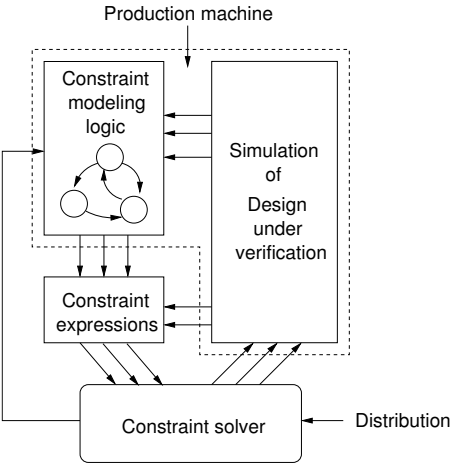

Figure 2.2. Constraint solving.

simulation. The role of constraints can be seen as a pruner of the "raw" design behavior under unrestricted input. Test generation with constraints can be considered a process of pruning and randomization, as in the following steps:

1. *Simulation*: Apply the input from the current constraint solution, simulate, and update the current state of the product machine. (Do nothing at the beginning of the simulation when no input is available.)

2. *Constraint solving*: Propagate the current state assignment of the product machine to the constraints and "solve" the resulting constraints, that is, find

an assignment to the variables in the constraints that satisfies all the constraints simultaneously. The solution space represents the possible inputs allowed under the current state. The states that can only be reached through the prohibited inputs are therefore pruned.

3. *Randomization*: There are two possible outcomes in solving the constraints:

   a. There is no solution, which we call an *over-constraint*. In simulation, this means not a single input is allowed at the current state. The current state is therefore called a *dead-end state*. Simulation has to be aborted upon reaching a dead-end state.

   b. There is at least one solution. One input is randomly picked, according to some user specified probability distribution.

In this flow, constrained random generation is reduced to a generic constraint satisfaction problem of Boolean formulas. The propagation of state assignment can be seen as a dynamic parameterization of the original constraints.

We have shown how constraints of various types may be expressed, and how constrained random generation is formed as the constraint satisfaction problem. Several issues remain to be covered: efficient constraint solving to enhance simulation throughput, randomization, and over-constraint debugging. In the sections that follow, an overview of these issues is given and questions are raised. Detailed discussions will be given in later chapters.

2.3 Constraint Solving

Constraint solving deals with the Constraint Satisfaction Problem (CSP). A general CSP consists of

- a set of variables,
- a non-empty domain for each of these variables,
- a set of constraints restricting the values of these variables, and
- optionally, a cost function that measures the quality of assignments to these variables.

Constraint solving is to find an assignment to all of the variables that simultaneously satisfies all the constraints, or to determine the non-existence of such an assignment. With many important applications in EDA, CSPs originated and are still the subject of intense research in artificial intelligence, operational research, and database queries.

Depending on the domains being continuous or discrete, and finite or infinite, and depending on the constraints being linear of non-linear, the general CSP definition can be refined to many familiar special cases. We give a brief overview

of special CSPs commonly encountered in EDA. Almost all are computationally intractable; refer to [GJ79] for more details.

- *Linear Programming (LP)* LP solves a set of linear constraints over reals and optimizes the solution according to a cost function. A special case of IP, the *Integer Linear Programming* (ILP), gives solutions in integers. Many digital logic optimization problems can be modeled as ILP, for example, Pipeline Resource Scheduling.

- *Propositional Satisfiability (SAT)* and *Automatic Test Pattern Generation (ATPG)* Both SAT and ATPG are search-based methods for solving propositional formulas in the Boolean domain. SAT works off a set of clauses that are equivalent to a gate-level design, whereas ATPG works off the design itself.

- *Binary Decision Diagrams (BDD)* BDD is a graph-based data structure that represents a Boolean function. For a fixed variable ordering in the graph, BDDs for logically equivalent functions are isomorphic. Solving a constraint is the process of constructing the corresponding BDD. The resulting BDD represents all solutions of the constraint.

- *Boolean Unification (BU) and Synthesis* A Boolean constraint can be solved via the classical BU method [W. 88, MN89]. The solution is an array of functions, each of which generates the value of a variable in the constraint. This solution method can be viewed as synthesizing a constraint into a vector function.

- *Constraint Logic Programming (CLP)* CLP combines constraint solving and logic programming. A CLP tool may solve problems ranging from linear and non-linear constraints, to propositional logics, and even temporal logic. The underlying constraint solvers can be logic-based (e.g., SAT), mixed LP and ILP, and interval arithmetic, over the real, finite, and Boolean domains.

When used in constrained random simulation, ILP, CP, SAT, and ATPG are classified as *on-line* approaches in the sense that the solution happens during simulation. BDD and BU are off-line because the solution happens when the BDDs and solution functions are constructed, usually before simulation. The trade-off is that the on-line approaches take exponential time in the worst case, while the off-line approaches use exponential space in the worst case. As we will see later, these exponential complexities can often be avoided with heuristic data structures and algorithms.

With the move to a higher model level, constraints are routinely expressed at the word-level instead of at the Boolean level. For example, transaction requests and grant signals have as many scalar variables as there are bus masters.

However, they are referenced as single word-level variables. While ILP and CP are natural word-level solvers, the others are originally designed to work with Booleans. The most efficient approach, however, is the combination of both.

2.4 Efficiency of Constraint Solving

One important characteristic of simulation is that it scales up to large designs. This is its main advantage over static verification. Complex designs, however, come with a large number of constraints. Even though constraints are concise, the complexity of constraint solving can easily be multiplied by the hundreds or thousands of constraints and variables when verifying a unit- or even a block-level design.

Fast constraint solving is critical to high simulation throughput and, therefore, good coverage. Enhancement of constraint solving efficiency can come from many sources. We will discuss these enhancements in later chapters; examples include:

- Choosing the right solver and improving its capacity. For example, bit-level versus word-level.

- Simplifying the problem with the partitioning, decomposition, and extraction of special constraints.

- Changing the problem with prioritized constraints and soft constraints.

2.5 Randomization

Besides giving the correct inputs, giving "good" inputs that are more likely to exercise interesting scenarios is a top priority for a constrained random generator. Choosing good inputs from correct ones is not as easy as it sounds. There are a few questions that must be asked when determining good inputs.

First of all, what is a good choice? In most cases, a good choice means that every possible input gets an equal chance to be selected, making the simulation effort spread over the entire input space. This fairness assumes that every input has the same weight in filling up the coverage chart, which is not always the case. For example, in generating data inputs to arithmetic operations, it is better to give much higher weights to the minimum and maximum values than to the ones in between. This ensures that boundary conditions are sufficiently exercised.

Second, given the desired distribution or weighting, how do you implement it? This should be achieved under the reasonable assumption that constraint solving and randomization are done simultaneously.

Related to these questions, how does one to make randomization adaptive to simulation?

Finally, how does one maintain *random stability* so that simulation traces may be repeated in identical settings? This is very critical to debugging.

2.6 Constraint Diagnosis

The problem of unsatisfiable constraints, or an over-constraint, has serious consequences in both simulation and formal verification. Over-constraints stall the progress in simulation and lead to vacuous proofs in formal verification. Constraint diagnosis identifies the source of and helps to fix over-constraints.

2.7 A Constrained Random Simulation Tool

In this section, we discuss a real-world tool, *Simgen* [YSP+99]. *Simgen* stands for simulation generation. It provides a constrained random simulation framework. *Simgen* is written in C++ and interfaces with a Verilog simulator through PLI. It recognizes a small set of syntax for defining constraints, randomization, and simulation control.

Simgen also works with a formal verification tool and an assertion language in an integrated dynamic and static verification environment. Line and branch coverage are given for the assertions, and state and transition coverage are given for any state machines the user specifies. Figure 2.3 gives a flow chart of *Simgen* and the integrated verification environment.

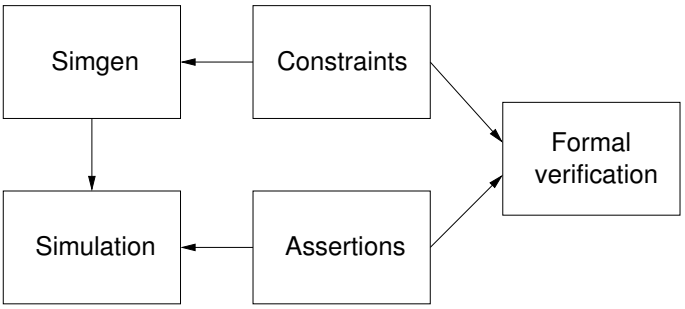

Figure 2.3. *Simgen*, simulation, and formal verification.

2.7.1 The Language

The *Simgen* constraint and randomization language follows the conventions of Verilog system calls. However, constraints and randomization are handled in preprocessing by *Simgen*'s Verilog compiler. Only the simulation controls are really system calls.

A constraint is declared with the syntax

```
$constraint (name, expr);
```

where name is a label for the constraint, and expr a Verilog Boolean expression that is asserted by the constraint. The name provides a handle to turn the constraint on and off statically (before simulation).

By default, a constraint is applied at every edge of a fast virtual clock. The user can also specify at what clock the constraint should be applied using

```
$constraint @ (posedge|negedge clock) (name, expr);
```

Clocks can be generated with clock constraints

```
$clock_constraint (name, expr);
```

which usually have the need to call the facility function

```
$prev(var)
```

to fetch the value of variable var from the last virtual clock cycle.

Randomization can be done in two ways: assigning weight to input variables, or to Boolean expressions.

```
$setprob1 (var, weight); $setprob0 (var, weight);
```

The above statements define how likely the input variable var will take the value 1 or 0, respectively. The likelihood is assigned to weight, which is a real number or a Verilog expression that evaluates to a real number between 0 and 1. Here is an example:

```
$setprob1(x, state == IDLE ? 0.9 : 0.5);
```

The syntax

```
$dist {
    expr1: weight1;
    expr2: weight2;
    ...
    exprN: weightN;
}
```

defines a weighted distribution where all the expressions are Boolean and mutually exclusive.

The simulation control constructs are

```
$simgen_stable;
$simgen_start;
$simgen_finish;
```

where the first system call is used to detect the stability of the simulator (usually the end of initialization), the second starts the *Simgen*-driven simulation, and the third performs post-processing.

Simgen uses one random generator. Its seed can be set with the following syntax.

```
$simgen_seed (seed);
```

Simulations can be reproduced if they start with the same seed and if the other settings remain the same.

2.7.2 BDD-based Constraint Solving

Constraints are first compiled into BDDs and the conjoined. A special BDD traversal algorithm is used at simulation time to generate vectors. The generator has the following properties:

1. If the constraints are satisfiable, then the generator will produce an input with no backtracking.
2. Without user-specified weighting, the generated inputs follow the uniform distribution.

Constraint solving in *Simgen* is preceded by several passes of optimizations, including extraction and simplification, partitioning, and decomposition [YKAP02, YPAA03].

2.7.3 Prioritized Constraints

To further enhance its constraint solving capacity, *Simgen* introduced a new concept called *prioritized constraints*. Prioritized constraints are basically normal constraints assigned with priority levels. The higher priority constraints are solved first, and the solution is propagated to the lower priority constraints before they are solved.

One characteristic of this approach is that input variables are not solved simultaneously. Rather, they are solved in an order implied by the hierarchy of the constraints. This alters the semantics of the original constraints by making them tighter. However, situations of sound prioritization abound. To take a simple example, clock constraints should always be solved before other constraints. Also, constraints regarding the transaction type can safely be solved before the constraints concerning transaction attributes.

http://www.springer.com/978-0-387-25947-5

Constraint-Based Verification
Yuan, J.; Pixley, C.; Aziz, A.
2006, XII, 254 p. 72 illus., Hardcover
ISBN: 978-0-387-25947-5