Chapter 3

Cycle-Based Symbolic Simulation

This chapter introduces our first technique to address the robustness and scalability limitations of the traditional symbolic simulation approach. We present an algorithm that can be applied to much more complex designs using a bounded amount of memory resources. The main focus of this algorithm is to achieve as much breadth of traversal as possible while maintaining the advantages of logic simulation, namely scalability and limited memory requirements. In the best situation, this approach achieves the same breadth of traversal as a pure symbolic simulation algorithm, but the breadth of the traversal can be reduced if that is required to minimize memory usage. Thus, Cycle-Based Symbolic Simulation, or CBSS, can be viewed as a hybrid approach that exploits the tradeoffs between symbolic search and logic simulation. Before diving into the presentation of this new algorithm, we discuss the motivation for this direction of work, namely the use of parameterization in symbolic simulation and its advantages for a reduced memory profile.

3.1 Parametric transformations

The central observation underlying the work of this thesis is that the expressions involved in a symbolic simulation exploration carry more information than the algorithm uses. At the end of each step, the Boolean expressions representing the state signals are fed back to the sequential inputs of the gate level network and used for the next simulation step. As we pointed out in Section 2.7.1, at the end of a generic step k, these expressions represent implicitly all the states that are reachable by the design in k steps from an initial state S_0. We observe now, that this implicit description of the set of states $\mathbf{S_{@k}}$ is most often redundant. Since the only information that needs to be transfered across simulation steps is the set of states that have been reached in the previous step, it is generally possible to define a more compact encoding of this set description, that is, a new *parameterization* of the state set. We can then use this new implicit description for the next simulation step. Figure 3.1 shows where the transformation takes place in the simulation flow.

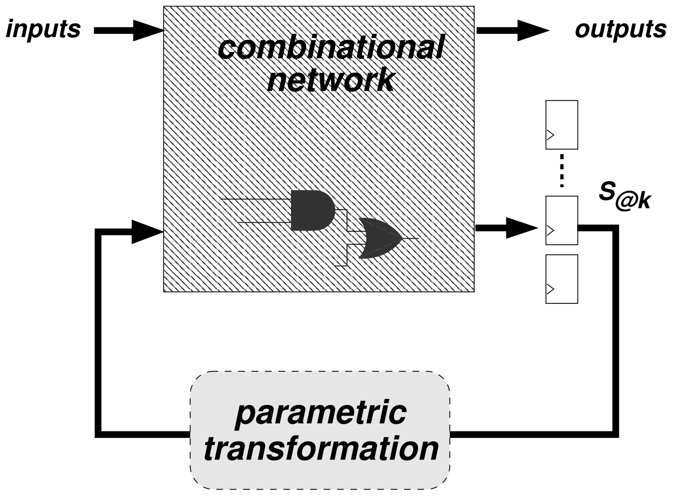

Figure 3.1: Parameterization of the state vector during symbolic simulation

Consequently, if we can define a new encoding that uses compact BDDs and transforms the expressions defining the set of reached states at the end of every simulation step based on this new parameterization, then we can maintain a low memory profile across the process and thus achieve better scalability and robustness in simulation.

Example 3.1. *Consider once again the counter of Example 2.2. When we perform the first step of symbolic simulation on this design, with reference to Figure 3.2 - step 1, we obtain the following* *vector of Boolean expressions for the next state functions:*

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

By varying the values associated with each of the variables in the expressions, that is, performing all the assignments $\{00, 01, 10, 11\}$ *for the pair of variables* r_0 *and* c_0*, we obtain an explicit list of all the states that can be reached in one step of symbolic simulation. For this example, such state set is* $\{1000, 1001\}$. *It's easy to see that this set can be more simply encoded as* $\{100p_0\}$*, where* p_0 *is a new Boolean parameter. Note that the new parameterization uses only one Boolean variable instead of two.*

We can now use this new simpler representation of the state set for the second step of simulation and obtain the expressions reported in Figure 3.2 - step 2 after simulating the combinational portion of the network. The new state expressions depend now on three variables: r_1*,* c_1 *and* p_0*, the parameter. Again, by evaluating the expressions for each possible assignment to the Boolean variables, we only obtain three distinct states:* $\{1000, 1001, 1010\}$. *These three states can be more efficiently encoded using only two parameters as:*

$$\begin{aligned} up &= 1 \\ x_2 &= 0 \\ x_1 &= p_0 \\ x_0 &= \overline{p_0} \cdot p_1 \end{aligned}$$

As even this small example shows, most often symbolic simulation produces expressions that are not an efficient encoding of the state set spanned by the traversal. In order to exploit the compact

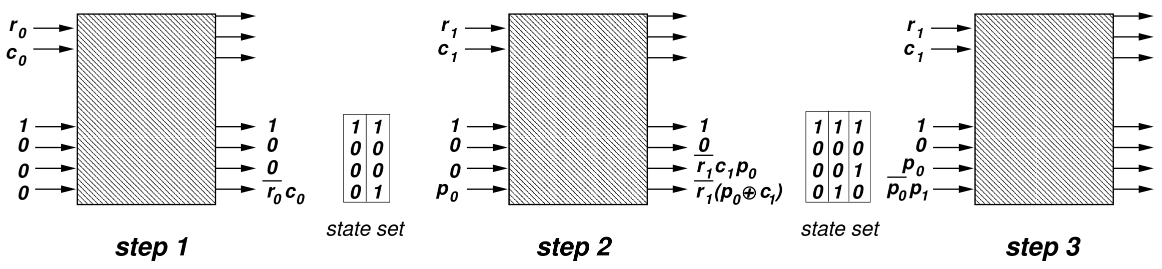

Figure 3.2: Three steps of symbolic simulation for the counter of Example 2.2 and possible parameterizations of the reached state sets

memory representations allowed by parameterization, we need to find an efficient algorithm that can discover good parameterizations automatically.

3.2 Parameterizations in symbolic simulation

Cycle-Based Symbolic Simulation is a hybrid approach in the sense that the values that are propagated through the network can be both symbolic expressions or constant Boolean values. Section 2.3.1 showed that BDDs can be used to represent both efficiently.

Our algorithm adds a *parameterization phase* at the end of each simulation step to basic symbolic simulation, as indicated in Figure 3.1. This parameterization transforms the state vector BDDs into compact BDDs that use only a small amount of memory resources. It is possible that the set of parametric BDDs produced spans only a subset of the original state set. This under-approximation may occur as a trade-off between accuracy of the traversal (that is, producing an exact parameterization) and complexity of the expressions produced (which we want to keep at a minimum). However, even when we settle for representing a subset of the state set, this set is chosen to maximize the amount of states represented for the amount of memory used.

Previous work has used parameterization techniques in connection with FSM traversal or symbolic simulation to reduce the memory requirements of the algorithms. Often, user interaction is required to suggest a relation among different signals of the systems under verification which can be exploited for parameterizing the simulation. For instance, in [38], the authors exploit the dependencies among state variables to simplify the traversal of a FSM. such dependencies need to be suggested by the designer and are verified for correctness during the simulation of the system. [41] presents a range of techniques to parameterize relations provided by the user. The work in [65] automatically discovers dependencies among state variables during FSM traversal. Detecting such dependencies requires checking all the state variables at each step of the traversal and transforming both the reached set and the transition relation accordingly during every step of the traversal.

Another research direction related to symbolic simulation and parameterizations focuses on partitioning the search exploration based on circuit related constraints and then performing multiple simulation for each element of the partition. In this context, parameterization techniques have been used to express the conditions of each subcase of the partitioned constraints. In particular, Jain *et al.* considered in [40] a variety of Boolean formula representations for the constraints and proposed a method to obtain parametric solutions. Aagaard *et al.* [1] introduced an alternative method where the case splitting on the constraints is based on Shannon decomposition.

In contrast, the focus of the algorithm presented here is to be fully automatic in a symbolic simulation context and to introduce a parameterization that is efficiently computed and produces very compact results in terms of memory requirements. We compare our approach to logic simulation and show that while each simulation step is more time consuming, since it manipulates Boolean expressions instead of constant values, it also produces the equivalent of multiple logic simulations test vector in a single pass. Experimental results report a quantitative analysis of the performance of this new approach to logic simulation.

3.3 The CBSS algorithm

Cycle-Based Symbolic Simulation is initialized by setting the state of the circuit to the initial constant vector S_0 (see Section 2.4.3 for a definition of S_0). Each of the combinational input signals is assigned a distinct symbolic variable $IN_{@0} = \{i_{1@0}, \cdots, i_{m@0}\}$. The simulation proceeds by computing the Boolean expressions corresponding to each node in the combinational portion of the network, as in the basic symbolic simulation algorithm. At the end of a simulation step, the expressions representing the next-state functions undergo a parametric transformation. During this parameterization, a minimal number of inputs could be set to constants. The objective of the selection is to maximize the breadth of the traversal, while keeping the representation of the state set compact through use of Boolean expressions with a small BDD.

During the simulation, we do not compute a `reached` set as in symbolic state traversal (Section 2.6.1). This computation is one of the main causes of reduced scalability of symbolic state traversal. Its main advantage is to maintain a history of states previously visited in the traversal, which is central to 1) discover when all the reachable states have been visited and the traversal is complete and to 2) select a set of states to use in the next simulation step, possibly with a compact representation. However, the simulation approach we present here targets circuits whose size is beyond the capability of symbolic state traversal. In general we don't expect to complete the simulation within a few hundreds steps, as it is generally the case for the type of designs that symbolic state traversal approaches. Moreover, we use a novel parameterization algorithm that does not require `reached` set information.

After parameterization, the newly generated functions are used as present state for the next state of simulation. Figure 3.3 shows how the algorithm just described corresponds to the iterative model for symbolic simulation. Notice that now we have added two new blocks to those in Figure 2.16. The outputs of the *parametric transformation (PAR-TRF)* block are: 1) the parametrized state vector that is fed to the present state in the next step of simulation and 2) a set of parametric equations that relate the newly created parameters $p_{@k}$ to the set of combinational inputs $\{IN_{@0}, \cdots, IN_{@k}\}$.

The parametric representation of frontier sets that we adopted can be constructed and manipulated very efficiently. The selection of which inputs to tie and to what value is based on the ease of construction of this representation. Alternatively, the value selection can be left to the user or to the tool: by evaluating to constant symbolic variables selectively, it is possible to symbolically simulate

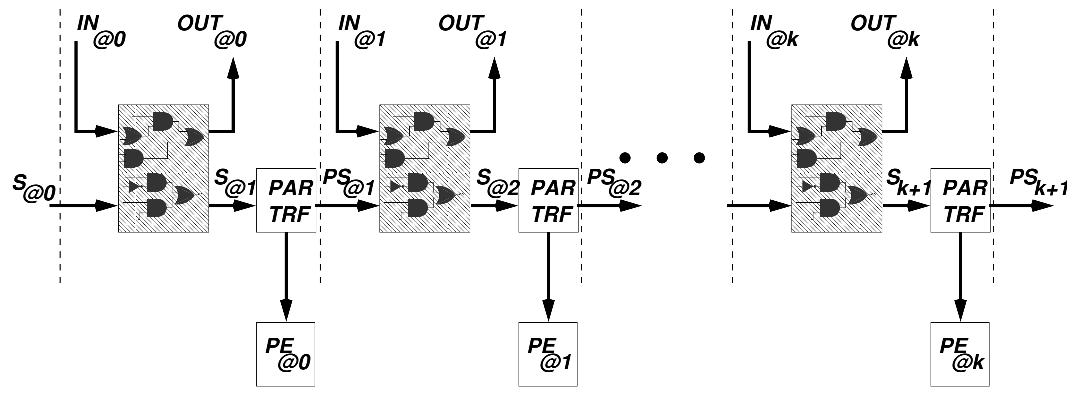

Figure 3.3: Cycle-Based symbolic simulation flow

any neighborhood of an input trace generated by the test bench.

3.4 The parameterization phase

The parameterization technique is based the following observation. In symbolic FSM traversal, the next state function δ can be, in general, complex. The next state functions of symbolic simulation at time step 0, $\mathbf{S_1}$, can be derived from δ as:

$$\mathbf{S_1}(i_{@0}) : I : \mathcal{B}^m \to S : \mathcal{B}^n = \{\delta(s,i) | s \in S_0, I \equiv IN_{@0}\}, \tag{3.1}$$

that is, by evaluating the state variables to the initial state values and substituting the combinational input variables with the input variables of time step 0. Often, because the state variables are evaluated to constant, the resulting components of $\mathbf{S_1}$ are very simple, such as constants, copies of an input, or complement of an input. Moreover, an input variable may be copied into several components of $\mathbf{S_1}$: there are then functional dependencies among the various state bits. We use these functional dependencies to obtain a simplified representation $\mathbf{PS_1}$ of $\mathbf{S_1}$. At each time step k, we produce a simple, parameterized representation of the next state functions to use for the next step $k+1$. By always presenting a simple set of functions at the present state signals of the network we are able to generate next state functions $\mathbf{S_{@k}}$ that are always simpler and more compact than the

```
CBSS(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);
      /* the next line also writes out the parametric equations */
      parametric_state_set = parameterize(state_symbols, step);
      assign (present_state_signals, par_state_set);
   }
}
```

Figure 3.4: The CBSS algorithm - pseudocode

ones involved in the pure symbolic simulation algorithm.

In practice, we never explicitly build the next function δ. Rather, at each clock tick k, we use the functional dependencies among the components of the next state function $\mathbf{S}_{\mathbf{@k}}$ at time k to build a parameterized version, $\mathbf{PS}_{\mathbf{@k}}$, for time $k+1$. If, in spite of our efforts, $\mathbf{PS}_{\mathbf{@k}}$ becomes too complex to be represented with BDDs within our memory budget, a few symbolic variables are tied to constant values to simplify it.

Notice that the parametric representation allows us to avoid the computation and representation of the global next state functions of the circuit as in symbolic state traversal, thereby avoiding a lengthy simulation set-up time.

3.4.1 Using functional dependencies

We discover and exploit functional dependencies using a parametric representation of the next state set. Figure 3.3 illustrates the approach. We introduce some *intermediate* variables p_i. At a generic clock tick k, we inspect the BDDs of $\mathbf{S_{@k}}$ and build a function $\mathbf{PS_{@k}}$ such that (see Definition 2.2):

$$\mathcal{R}(\mathbf{PS_{@k}}) = \mathcal{R}(\mathbf{S_{@k}}). \tag{3.2}$$

In practice, we will settle for a $\mathbf{PS_{@k}}$ such that 1) the number of parameter variables p is small, and 2) $\mathcal{R}(\mathbf{PS_{@k}})$ is a "large" and easily identifiable subset of $\mathcal{R}(\mathbf{S_{@k}})$:

$$\mathcal{R}(\mathbf{PS_{@k}}) \subseteq \mathcal{R}(\mathbf{S_{@k}}). \tag{3.3}$$

The set $\mathbf{PS_{@k}}$ that we generate has cardinality that is 2^p, where p is the number of parameters we introduce during the parameterization phase. The diagram in Figure 3.5 shows the relation between the whole state space of the system, $\mathbf{S_{@k}}$ and $\mathbf{PS_{@k}}$.

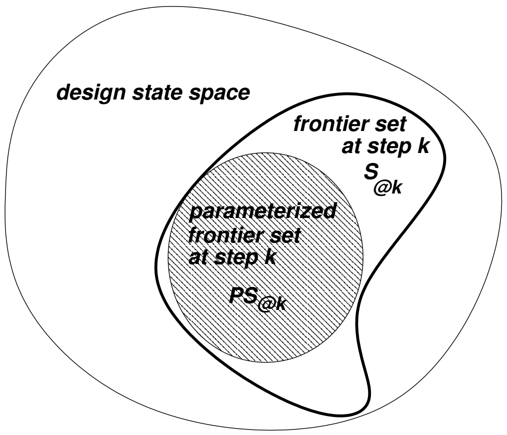

Figure 3.5: The parameterized frontier subset $\mathbf{PS_{@k}}$

Section 3.4.2 provides the details on $\mathbf{PS_{@k}}$ and its construction. The BDD of the next state functions for step $k+1$ is then built by simulation of the combinational portion of the circuit. In terms of the δ function, this corresponds to:

$$\mathbf{S_{k+1}}(i_{@k+1}) : I : \mathcal{B}^m \to S : \mathcal{B}^n = \{\delta(s,i) | s \in \mathbf{PS_{@k}}, I \equiv IN_{@k+1}\} \tag{3.4}$$

and a new $\mathbf{PS_{k+1}}$ constructed by parameterization. Notice that the state variables are effectively replaced by the parametric variables p_i.

In addition, we build a second mapping $\mathbf{PE_{@k}}$. This second mapping expresses each p_i as a function of inputs and intermediates at the previous tick. $\mathbf{PE_{@k}}$ should also be "simple", for the following reason. Suppose an error is discovered at time k. There is then an assignment of primary inputs and intermediates at time k that exposes the bug. We need to be able to map the assignment of intermediates to an assignment of inputs and intermediates at time $k-1$, and then iteratively back to primary inputs at time $k-2,\cdots,0$.

The parametric transformation develops in two phases: the first phase identifies *simple* variables, while the second phase parameterizes *unbound* functions. The pseudocode of the function is shown in Figure 3.6. It guarantees that $\mathcal{R}(\mathbf{S_k})$ can be parameterized in linear time. If this is not the case, it identifies variables for assignment, and cofactors $\mathbf{S_k}$ accordingly. The actual constant values used for the assignment could correspond to the values provided in a testbench for the design, if this is available. For instance, if at the third step of CBSS simulation we need to evaluate to a constant the variable corresponding to input x, we could extract the value assigned at input x in the testbench at the third step of logic simulation. By choosing values based on this criteria, we guarantee that our CBSS algorithm produces a design exploration that includes the search corresponding to logic simulation run on the same testbench. For instance, if there is a testbench that drives the design to a specific corner case to check it, CBSS can not only check that specific configuration of the system, but also cover a set of additional configurations that are "close" to the target one in the FSM model.

Whenever a testbench is not available, we can still automatically produce a random value for the variable assignment. This choice will drive the design through a random walk of the state space.

The pseudocode of the parameterization phase is shown in Figure 3.6. The details of functions `find_simple_complex_var`, `find_shared_eqclasses`, and `remap` are described in the following sections. Function `assign_&_cofactor` simply takes a vector of expressions and a set of variables, assigns a value to each of the variables in the set and partially evaluates each expression based on these values.

```
parameterize(state_equations, step) {
   <simple, complex> = find_simple_complex_var(state_equations);
   state_equations = assign_&_cofactor(state_equations, complex);
   state_equations = remap(state_equations, simple);
   append_param_equations(simple, step);
   <classes, shared> = find_shared_eqclasses(state_equations);
   state_equations = assign_&_cofactor(state_equations, shared);
   state_equations = remap(state_equations, classes);
   append_param_equations(classes, step);
   return state_equations;
}
```

Figure 3.6: `parameterize` function - pseudocode

3.4.2 How to classify the components of the state vector

We show how to quickly identify a function $\mathbf{PS}_{@\mathbf{k}}$ such that $\mathcal{R}(\mathbf{PS}_{@\mathbf{k}})$ is a "large" subset of $\mathcal{R}(\mathbf{S}_{@\mathbf{k}})$. The set of transformations presented in the next two sections can be applied to any Boolean vector function. For purposes of readability, in the following definitions we will refer to the generic function $\mathbf{V} : \mathcal{B}^n \rightarrow \mathcal{B}^m$. As explained above, the CBSS algorithm applies such transformation to the next state vector $\mathbf{S}_{@\mathbf{k}}$.

Definition 3.1. *A variable x is termed* **simple** *if there is a component $\mathbf{V}_i$ of $\mathbf{V}$ such that $\mathcal{S}(\mathbf{V}_i) = \{x\}$. Given a function $\mathbf{V}$, let Si denote the set of simple variables. A component $\mathbf{V}_i$ is termed* **simple** *if $\mathcal{S}(\mathbf{V}_i) \subseteq Si$.*

Definition 3.2. *Let again Si denote the set of simple variables. A non-constant component function $\mathbf{V}_i$ is termed* **complex** *if:*

1. *$\mathcal{S}(\mathbf{V}_i) \cap Si \neq \emptyset$ and*

2. *$\mathcal{S}(\mathbf{V}_i) \cap \overline{Si} \neq \emptyset$.*

For a complex function $\mathbf{V}_i$, a variable belonging to $\mathcal{S}(\mathbf{V}_i) \cap \overline{Si}$ is also termed **complex**.

Definition 3.3. *A function is* **unbound** *if it is neither simple nor complex. Two components $\mathbf{V}_i$ and $\mathbf{V}_j$ of $\mathbf{V}$ are termed* **equivalent** *if they are unbound and either $\mathbf{V}_i = \mathbf{V}_j$ or $\mathbf{V}_i = \overline{\mathbf{V}_j}$ holds.*

Definition 3.4. *Given an equivalence class ε of functions with reference to the previous definition, we indicate with $\mathcal{S}(\varepsilon)$ the set of variables belonging to the support of any function in ε. A variable $x \in \mathcal{S}(\mathbf{V})$ is said to be* **bound** *if it belongs only to the support of a single equivalence class of* $\mathbf{V}$*. It is termed* **shared** *if it belongs to more than one class.*

Example 3.2. *Consider the following function $\mathbf{S_{@k}}$:*

$$\mathbf{S_{@k}}(x,y) : \mathcal{B}^2 \rightarrow \mathcal{B}^7 = (x,\overline{x},y,0,f(x,y),g(x,y),\overline{y})$$

Its components are only: 1) constants, 2) functions of a single variable, or 3) functions of variables also appearing as single variables in other components (that is, simple functions).

In this situation, an exact parametric description is obtained by replacing x and y with two parameters:

$$\mathbf{PS_{@k}} = (p_0,\overline{p_0},p_1,0,f(p_0,p_1),g(p_0,p_1),\overline{p_1})$$

Notice that $\mathbf{PE_{@k}}$ is just a data-transfer: $p_0 = x$, $p_1 = y$.

Suppose now that $\mathbf{PS_{@k}}$ consists only of simple and complex functions. By assigning a value to complex variables, other complex variables may become simple:

Example 3.3. *Consider*

$$\mathbf{S_{@k}}(q,r,s,x,y) = (x,y,x+y+q+r,s+xq).$$

$\mathbf{S_{@k}}_{,0}$ and $\mathbf{S_{@k}}_{,1}$ are simple. $\mathbf{S_{@k}}_{,2}$ and $\mathbf{S_{@k}}_{,3}$ are complex, as variables q, r and s are complex. If we assign q and r as $q = 0$ and $r = 1$, component $\mathbf{S_{@k}}_{,3}$ become simple and $\mathbf{S_{@k}}$ can have a simple parametric representation:

$$\mathbf{PS_{@k}}(p_0,p_1,p_2) = (p_0,p_1,1,p_2).$$

Simple and complex variables (and functions) are identified in a two-pass scan of the BDDs of $\mathbf{S_{@k}}$. Figure 3.7 shows the pseudocode for identifying them. We assume that initially, all component functions are labeled `UNBOUND`. The first `foreach` loop finds the support of each component of $\mathbf{S_{@k}}$ and identifies simple variables. The second `foreach` loop identifies complex variables and places them in `Co`. It also classifies the functions whose support is all contained in `Si` as simple.

```
find_simple_complex_var(state_equations) {
   Si = Co = ∅;
   foreach (eq) in (state_equations) {
      if (support_size(eq) == 1) {
         Si = Si ∪ support(eq);
         assign_type(eq, SIMPLE);
      }
   }
   foreach (eq) in (state_equations) {
      if (support(eq) ∩ Si ≠ ∅) {
         csupp = support(eq) \ Si;
         if (csupp ≠ ∅) {
            Co = Co ∪ csupp;
            assign_type(eq, COMPLEX);
         } else {
            assign_type(eq, SIMPLE);
         }
      }
   }
   return <Si, Co>;
}
```

Figure 3.7: Classifying simple and complex variables - pseudocode

After complex variables are identified and removed, each component of $\mathbf{S_{@k}}$ is labeled as either `SIMPLE` or `UNBOUND`. Unbound functions have no support variables in `Si`.

We then examine unbound functions. The simplest case occurs when one such function has support disjoint from all other components. For example, in Eq. 3.5 below:

$$\mathbf{S_{@k}} = (f(p,q), x, y, g(x,y)). \tag{3.5}$$

the first component is unbound and has support disjoint from all others. The component can be replaced by an independent intermediate variable:

$$\mathbf{PS}_{@\mathbf{k}} = (p_0, p_1, p_2, g(p_1, p_2))$$

where

$$p_0 = f(p,q); \quad p_1 = x; \quad p_2 = y.$$

Consider now the more general situation:

$$\mathbf{S}_{@\mathbf{k}} = (f(p,q), \overline{f(p,q)}, x, y).$$

The first and second component of $\mathbf{S}_{@\mathbf{k}}$ can be replaced by p_0, $\overline{p_0}$ respectively.

Definition 3.4 partitions the set of unbound functions in $\mathbf{S}_{@\mathbf{k}}$ into equivalence classes. These classes can be discovered in a single scan of the array $\mathbf{S}_{@\mathbf{k}}$. If a value is assigned to all shared variables, then the support of each equivalence class will contain only bound variables, so that each class can be replaced by an independent parameter.

Example 3.4. *Consider*

$$\mathbf{S}_{@\mathbf{k}} = (x+y+z, \bar{x}\bar{y}\bar{z}, \bar{z}w, \bar{z}w).$$

By assigning the shared variable $z = 0$, the components of $\mathbf{S}_{@\mathbf{k}}$ become:

$$\mathbf{S}_{@\mathbf{k},z=0} = (x+y, \overline{x+y}, w, w)$$

A parametric representation of $\mathcal{R}(\mathbf{S}_{@\mathbf{k}})$ is then

$$\mathbf{PS}_{@\mathbf{k}} = (p_0, \overline{p_0}, p_1, p_1) \tag{3.6}$$

where $p_0 = x + y$ and $p_1 = w$.

Figure 3.8 shows the algorithm for finding shared variables. We first group the `UNBOUND` state expressions into equivalence classes. Then, we consider each variable in the support of these expressions, check if it belongs to one or more equivalence classes and tag it consequently.

```
find_shared_eqclasses(state_equations) {
   Sh = EC = ∅;
   foreach (eq) in (state_equations) {
      if (function_type(eq) == UNBOUND) {
         class = find_or_make_new_class(eq, EC);
         EC = EC ∪ class;
         foreach (x) in (support(eq)) {
            if (tag(x) == empty) tag(x) = class;
            else if (tag(x) ≠ class) tag(x) = shared;
         }
      }
   }
X   foreach (class) in (EC) {
      foreach (x) in (support(class)) {
         if (tag(x) == shared) Sh = Sh ∪ {x};
      }
   }
   return <Sh, EC>;
}
```

Figure 3.8: Classifying shared variables - pseudocode

3.4.3 The `remap` function

`remap` generates the new parameters for $\mathbf{PS}_{@\mathbf{k}}$ based on the results of the previous two routines. The first call remaps the variables in the *simple* set. Each of these variables is simply substituted by a new parameter variable in the state expressions with a single traversal of each of the BDDs. The second call remaps each equivalence class to a parameter. This operation is even simpler, since it just requires to represent each state equation with a single parameter based on the equivalence class it belongs to. The maximum numbers of parameters needed by the two calls is bounded by the number of memory elements in the design to simulate. In fact, a new parameter is only assigned to Boolean expressions that occur at least once as a complete state equation. Thus, after parameterization, for each parameter, there is at least one equation whose expression is simply the parameter variable.

Example 3.5. *Suppose you are given a system to simulate with ten memory elements and eight inputs. After the first cycle of symbolic simulation, we obtain the following expressions for the state equations, where each combinational input was assigned a distinct Boolean variable literal a to h:*

$$\begin{array}{llll} s_0 = a & s_3 = ab & s_6 = d+e+f & s_8 = f+g \\ s_1 = a & s_4 = abc & s_7 = \overline{d}\overline{e}\overline{f} & s_9 = hg \\ s_2 = b & s_5 = b+c & & \end{array}$$

At first, all the equations are assigned the type `UNBOUND`*. With the first pass through the state equations, we detect the simple variables: a and b and we assign the type* `SIMPLE` *to s_0, s_1 and s_2. The second pass detects that s_3 is also simple, and classifies variable c and equations s_4 and s_5* `COMPLEX`*. After evaluating variable c to* 0 *and remapping the simple variables, we obtain:*

$$\begin{array}{llll} s_0 = p_0 & s_3 = p_0p_1 & s_6 = d+e+f & s_8 = f+g \\ s_1 = p_0 & s_4 = 0 & s_7 = \overline{d}\overline{e}\overline{f} & s_9 = hg \\ s_2 = p_1 & s_5 = p_1 & & \end{array}$$

At this point, we need to identify the equivalence classes for the remaining unbound functions. We find three equivalence classes: $\varepsilon_1 = \{s_6, s_7\}$, $\varepsilon_2 = \{s_8\}$, $\varepsilon_3 = \{s_9\}$. Variables d and e are tagged with ε_1, h is tagged with ε_2 and f and g are `shared`*. Consequently, we need to evaluate these last two variables to a constant value. We choose* 0 *for f and* 1 *for g. The set of equations at this point is:*

$$\begin{array}{llll} s_0 = p_0 & s_3 = p_0p_1 & s_6 = d+e & s_8 = 1 \\ s_1 = p_0 & s_4 = 0 & s_7 = \overline{d}\overline{e} & s_9 = h \\ s_2 = p_1 & s_5 = p_1 & & \end{array}$$

and after remapping the unbound functions using one parameter for each equivalence class, we obtain:

$$\begin{array}{llll} s_0 = p_0 & s_3 = p_0 p_1 & s_6 = p_2 & s_8 = 1 \\ s_1 = p_0 & s_4 = 0 & s_7 = \overline{p_2} & s_9 = p_3 \\ s_2 = p_1 & s_5 = p_1 & & \end{array}$$

Notice that the function in class ε_2 *was reduced to a constant, thus we did not need to use a parameter to remap it. This final set of equation is our new parameterized state vector. The* $\mathbf{PE}_{@\mathbf{k}}$ *equations are:*

$$p_0 = a \qquad p_1 = b \qquad p_2 = d + e \qquad p_3 = h$$

Note that the number of parameters that are needed during each parameterization is always $\leq n$ where n is the number of state elements in the design. This is easy to derive based on the fact that for each parameter p_i there is at least one parameterized state equation $\mathbf{P}S_{@k,j}$ such that $\mathbf{P}S_{@k,j} = p_i$.

3.5 Implementation and complexity

In implementing the algorithm we made some observations that made possible to use the Boolean variables needed for the simulation efficiently. Since, in general, BDD packages can allow only a limited number of variables, this has also an impact on how many steps of simulation we can run. First, since we know that the number of parameters is bounded by the number of memory elements, we simply reserved an equivalent number of variables in the BDD manager for parameterization.

Second, we noticed that at the end of each parameterization step, the state equations do not depend on the combinational input variables any longer, but only on the parameters. Thus, we can reuse the same set of Boolean variables for the combinational inputs at every step of simulation. It follows that CBSS only needs a constant number of Boolean variables, equal to the number of inputs plus the number of states of the design to simulate. In contrast, a basic symbolic simulator requires a new Boolean variable for each combinational input signal needs at each simulation step. Thus, a symbolic simulator needs a number of Boolean variables that depends on the length of the simulation and is equal to the number of combinational input signals times the number of simulation steps.

During simulation, the parametric equations **PE** at each step can be stored in BDD form. Since the variables used for these equations are the same involved in the simulation, sharing among the BDD nodes is possible and the additional memory required for these equations is not significant.

Moreover, while remapping the simple variables, we assign them in ascending variable order and we choose the parameters to reflect the same order, so that corresponding BDDs do not need to be recomputed, but can be simply duplicated and relabeled in a single pass. A more optimized approach would simply dynamically classify which variables are inputs and which are parameters, then, without modifying the BDDs at all, simple variables would just be reclassified as parameters at the next step of simulation and an equivalent number of parameters would become input variables to assign to the input signals.

The complexity of the algorithm can be computed considering each phase separately. We use here n for the number of states in the design, and **#BDD** for the size of the BDDs of the state equations:

- **simple variables** can be identified in a single pass of the state equations - $O(n)$.
- **complex variables** can be identified in another single pass of the state equations. We also need to cofactor each state equation w.r.t. to the complex variables, this can be done with a specialized cofactor routine that traverses each BDD once - $O(n \times \#\mathbf{BDD})$.
- **remapping simple variables** as we mentioned above can be done with a single pass of the state equations' BDDs - $O(\#\mathbf{BDD})$.
- **equivalence classes** can again be identified in a single pass of the state equations - $O(n)$.

- **shared variables** require similar treatment than complex variables, leading to the same worst case complexity - $O(n \times \#\mathbf{BDD})$.
- **remapping unbound functions** requires only assigning the proper parameter variable to each equivalence class - $O(n)$.

3.6 Experimental results

The CBSS algorithm was implemented in a C++ program and tested on the largest sequential circuits from the Logic Synthesis Benchmarks suite [68] and the ISCAS'89 Benchmark Circuits [16], including their 1993 additions. Table 3.1 reports results on all but the smallest testbenches of the two suites (we excluded from the table the circuits with less then 20 memory elements). The testbenches are grouped by benchmark suite. The experiments were run on a Linux PC equipped with a Pentium 4 processor running at 2.7Ghz and 2GB of memory and 512Kb of cache. As the underlying ROBDD package we used the CUDD package by Somenzi, [29], for which we set a reordering threshold of 200,000 nodes. We evaluated the simulator by running it for 5,000 symbolic simulation cycles on each testbench: at the end of each symbolic simulation step we would run our parameterization algorithm to simplify the state functions and then proceed to the next step. For the purpose of evaluating the performance of the approach, we chose a random Boolean value whenever we needed to evaluate complex and shared variables to constant. However, in a real-world context it is possible to choose the values based on the test stimulus, if one is available. For each circuit, the table reports first a few relevant metrics: the number of inputs In, outputs Out, memory elements FF, and internal network gates Gates.

The next three columns report the results of the parameterizations. The values are the average over the 5,000 steps of simulation. Our objective is to evaluate how many symbolic parameters we could find and the average number of states we could reach at each simulation step. To this end, the first of this group of columns, Param, reports the average number of symbolic parameters that we

Circuit	In	Out	FF	Gates	Parameterization			Time (s)		Efficiency	Memory (KB)	
					Params	Ass.d	Symbols	CBSS	Logic	ratio	CBSS	Logic
Logic Synthesis ‘91 - FSM tests												
ex1	9	19	20	622	0	0	9	0.69	0.04	29.68	4647	312
s1423	17	5	74	830	1.04	12.91	5.14	2.04	0.06	1.04	5818	320
s838	35	2	32	596	0.57	1.52	34.04	0.93	0.04	$7.62\cdot10^{8}$	4690	312
s953	16	23	29	658	1.15	6.1	11.05	1.36	0.04	62.19	5060	-
Logic Synthesis ‘91 - Addition ‘93												
bigkey	262	197	224	9211	0	228	34	163.49	0.55	$5.78\cdot10^{7}$	38255	516
clma	382	82	33	24482	1	0	383	75.77	1.5	$3.90\cdot10^{113}$	5078	836
dsip	228	197	224	3893	0	228	0	135.35	0.28	0	21289	404
mm9a	12	9	27	639	3.02	2	13.02	1.24	0.04	267.95	4658	-
mm9b	12	9	26	786	0	11.99	0.01	2.23	0.05	0.02	5339	-
mult16b	17	1	30	284	5.83	10.88	11.96	1.76	0.01	22.59	5563	308
mult32a	33	1	32	715	0.21	32.36	0.85	22.23	0.04	0	15980	-
s38417	28	106	1465	23771	47.5	19.66	55.83	190.55	1.67	$5.62\cdot10^{14}$	40613	956
s38584	38	304	1426	20281	7.48	25.71	19.77	488.99	1.35	2468.29	45244	864
s5378	35	49	163	3232	14.88	26.17	23.7	14.79	0.22	$2.03\cdot10^{5}$	13859	384
s838	34	1	32	618	0.5	1	33.5	0.85	0.04	$5.72\cdot10^{8}$	4690	-
s9234	36	39	135	3019	16.96	10.48	42.48	7.56	0.21	$1.70\cdot10^{11}$	5093	372
sbc	40	56	27	1143	2.92	22.22	20.7	3.76	0.07	$3.16\cdot10^{4}$	6066	324
ISCAS ‘89 - FSM tests												
s13207.1	62	152	638	9539	56.52	15.12	103.4	48.95	0.69	$1.89.\cdot10^{29}$	24684	568
s13207	31	121	669	9539	14.75	4.66	41.09	41.59	0.69	$3.88\cdot10^{10}$	9710	568
s1423	17	5	74	830	1.05	12.88	5.17	1.94	0.06	1.11	5834	-
s15850.1	77	150	534	11316	29.7	40.07	66.63	52.45	0.78	$1.69\cdot10^{18}$	35961	600
s15850	14	87	597	11316	4.39	2.78	15.61	35.69	0.74	$1.03\cdot10^{3}$	9386	604
s35932	35	320	1728	23085	1	35	1	194.66	1.67	0.02	38938	968
s38417	28	106	1636	27648	48.27	19.81	56.46	190.75	1.94	$1.01\cdot10^{15}$	39558	1068
s38584.1	38	304	1426	24619	7.45	25.75	19.71	475.62	1.68	3025.16	49685	972
s38584	12	278	1452	24619	6.26	6	12.27	271.83	1.65	29.88	45271	968
s5378	35	49	179	3973	14.86	26.13	23.73	14.95	0.06	$5.59\cdot10^{4}$	14226	-
s838	34	1	32	626	0.5	1	33.5	0.85	0.05	$7.15\cdot10^{8}$	4690	-
s9234.1	36	39	211	6585	18.06	19.51	34.55	16.61	0.43	$6.51\cdot10^{8}$	7965	464
s9234	19	22	228	6585	1.18	6.9	13.28	14.09	0.43	303.55	4964	464
s953	16	23	29	658	1.17	6.16	11.01	1.32	0.04	62.32	5029	312
ISCAS ‘89 - Addition ‘93												
prolog	36	73	136	1845	29.13	24	41.13	10.04	0.03	$7.20\cdot10^{9}$	9117	-
s1269	18	10	37	771	1.83	12.99	6.84	3.22	0.05	1.78	6019	312
s1512	29	21	57	990	9.85	5.93	32.92	2.29	0.06	$2.13\cdot10^{8}$	4960	324
s3271	26	14	116	2166	6.3	26	6.3	18.73	0.15	0.63	8295	352
s3330	40	73	132	2020	29.11	24.33	44.79	12.01	0.13	$3.28\cdot10^{11}$	9069	352
s3384	43	26	183	1734	53.32	17.99	78.33	10.6	0.14	$5.02\cdot10^{21}$	13529	352
s4863	49	16	104	2492	7.72	25.75	30.97	18	0.03	$3.50\cdot10^{6}$	8812	-
s6669	83	55	239	3272	77.86	68.73	92.13	275.47	0.04	$7.87\cdot10^{23}$	47362	388
s938	34	1	32	626	0.5	1	33.5	0.89	0.05	$6.82\cdot10^{8}$	4690	312
s967	16	23	29	677	1.25	6.13	11.12	1.41	0.05	78.98	5077	-

Table 3.1: Cycle Based Symbolic Simulation results

generate during a parameterization phase. For our second objective, we used the following reasoning: if we never evaluated a variable to constant, the number of symbols we had at each step would be given by the number of inputs symbols plus the number of parameters. However, since at every step some variables maybe be assigned to constant, we need to keep this into account by subtracting this amount from the number of live symbols that we carry across simulation steps. The average number of states that we reach at each step is then given by 2 to the power of this value, since, after parameterization each symbol doubles the number of states spanned by the parameterized state functions. The table shows the results we obtained with this evaluation: the second column of the group indicates the average number of symbolic variables that we assigned to a constant because they were classified as complex or shared variables, and the third column counts the number of live symbols as just described: Symbols = Param + IN - Ass.d. The actual size of the average state set visited at every step is 2^{Symbols}. This latter value also represents the average number of logic simulation equivalent traces that we carry on in parallel at every step.

The reminder of the table compares the results we obtained with CBSS to the performance of a compiled-level logic simulator. We built a logic simulator as described in Section 2.5 and we simulated again each of the testbenches for 5,000 cycles, providing a random stimuli to each circuit's inputs at each step. The two columns labeled Time compare the execution time for the CBSS simulation to the one for the logic simulator. We did not take into account the time spent compiling the circuit's netlist into assembly code for logic simulation. However, we measured this time and it was not transcurable: above 200s for the seven biggest benchmarks and above 1s for most of the testbenches. As the table indicates, once the compilation was completed, logic simulation could execute quite fast. As for the CBSS execution times, we point out that variable reordering was only triggered by the test *s6669* of the ISCAS suite, and thus it was not a factor for all the other benchmarks. Column Efficiency compares the performance of CBSS to logic simulation in terms of traces simulated per second of execution. Its value is computed as the ratio 2^{Symbols}·(Time-logic/Time-CBSS). It represents the number of traces visited by CBSS in the time of executing one logic simulation trace. A value of 1 in this column indicates that CBSS is providing the same performance as a compiled-level logic simulator; when the value is less than 1, the logic simulator is more efficient; otherwise CBSS is providing "Efficiency" times better performance than a logic simulator. Note that most of the testbenches show an efficiency of 10-20 orders of magnitude over logic simulation, and this is particularly true for the most complex designs. Our intuition is that the more complex designs have more inputs and more memory elements that increase the possibility of discovering good parameterizations for the state vectors. For instance, the two variations of *s13207* in the ISCAS suite, provide very different efficiency results: the second one, having only half the inputs, can generate many fewer Symbols on average and thus it achieves lower efficiency. When the parameterization can only produce a small number of Symbols because of the high percentage of complex and shared variables, the extra time spent by CBSS in manipulating Boolean expressions makes this approach less attractive compared to logic simulation. This is the case mostly for the smaller designs, because of their limited potential for parameterizations.

Finally, the last two columns compare the memory profile of the two approaches. Even the smallest designs require a minumum of 4-5 KB to start the CUDD package in CBSS. However, the memory profiles are only moderately sensitive to the size of the design. As for the logic simulation memory column, we were able to collect the memory profile of the simulator only for the medium to large designs of the suites, and we report a '-' for the testbenches for which we could not gather this data. This last column can be used to gain an insight on the impact of design size over the memory profile of logic simulation, which can then be compared to the corresponding one for CBSS.

Overall we see that a high average number Symbols is key to a high efficiency over logic simulation. In general, testbenches that contain *highly sequential* components (such as counters) have a lower potential for good parameterizations: if the state bits of a counter take constant value at some point in time, that is, they are represented by constants, then they will be represented by constants also at the next clock tick. On the other hand, other circuits are more data-path intensive, they contain several large data-transfer or arithmetic operations, and in this cases it is easier to assign state bits independently, hence the larger number of parameter variables.

3.7 Conclusion

CBSS was published in [10]. This algorithm has shown to improve the scalability of symbolic simulation by providing a quick and memory friendly parameterization technique for the state equations. It can find quickly a large subset of the frontier set which can be represented very efficiently. The experimental results shows that in most cases we can achieve 10-20 orders of magnitude or more better efficiency over a compiled logic simulator.

However, in a few cases we noticed that many variables in the support of the state vector are complex or shared and need to be evaluated to constant. In those cases the performance is no longer competitive with logic simulation and the breadth of the state exploration is limited. In order to improve on the quality of the parameterization, we need to explore better techniques to represent the state vector through parameters. To this end, the next chapter introduces the theory of disjoint support decomposition of Boolean functions. This theory will be exploited in Chapter 6 to present an algorithm that can perform an exact parameterization of the state vector, thus guaranteeing to achieve the same maximal search breadth of symbolic simulation.