Automated Equivalence Checking of Switch Level Circuits

Simon Jolly
FourSticks Pty. Ltd.
2/259 Glen Osmond Road,
Frewville, South Australia, 5063.
61-8-83385500
sjolly@foursticks.com

Atanas Parashkevov
Motorola Inc.
2 Second Avenue, Mawson Lakes,
South Australia, 5095.
61-8-81683687
aparashk@asc.corp.mot.com

Tim McDougall
Motorola Inc.
2 Second Avenue, Mawson Lakes,
South Australia, 5095.
61-8-81683709
tmcdouga@asc.corp.mot.com

ABSTRACT
A chip that is required to meet strict operating criteria in terms of speed, power, or area is commonly custom designed at the switch level. Traditional techniques for verifying these designs, based on simulation, are expensive in terms of resources and cannot completely guarantee correct operation. Formal verification methods, on the other hand, provide for a complete proof of correctness, and require less effort to setup. This paper presents Motorola's Switch Level Verification (SLV) tool, which employs detailed switch level analysis to model the behavior of MOS transistors and obtain an equivalent RTL model. This tool has been used for equivalence checking at the switch level for several years within Motorola for the PowerPC, M*Core and DSP custom blocks. We focus on the novel techniques employed in SLV, particularly in the areas of pre-charged and sequential logic analysis, and provide details on the automated and integrated equivalence checking flow in which the tool is used.

Categories and Subject Descriptors
J.6 [**Computer-Aided Engineering**]: Computer-Aided Design.

General Terms
Algorithms, Design, Verification.

Keywords
Custom design, switch level analysis, equivalence checking, formal verification, MOS circuits, VLSI design.

1. INTRODUCTION
To keep up with the challenges poised by the constantly increasing digital circuit complexity, several levels of abstraction in circuit representation are typically utilized. Register-Transfer Level (RTL) describes a circuit at a high level of Boolean functions and data flow within the circuit. Gate level representation provides a structural (schematic) description of a circuit as an interconnection of basic blocks having a known and relatively simple Boolean functionality. Switch level representation contains an interconnection of switches (transistors) and gates that implement the desired functionality of a circuit.
RTL is often the preferred abstraction level for most functional design activities because it provides the highest level of productivity. RTL models serve as a reference for design implementation. However, any RTL design has to be translated into an equivalent switch level design as a necessary step prior to the fabrication of a physical chip. This translation can be performed using synthesis flows and tools such as Synopsys Design Compiler [17]. When a chip has to meet stringent operating requirements (e.g. speed or power) certain parts of the chip may be manually implemented and carefully tuned at the switch level. This practice is commonly referred to as custom design.
When the reference RTL view and the switch level view of the same digital circuit is created independently, there is a clear verification problem: it has to be ascertained that the two views have the same functionality. Traditional verification techniques such as simulation or emulation do not provide a complete guarantee of correctness and can be very expensive in terms of necessary resources. Formal methods such as combinational equivalence checking [12] can be applied to complement traditional approaches and provide a proof of equivalence between two functional descriptions of a circuit. RTL and gate level models are both suitable functional representations and thus equivalence checking tools can operate directly on them. A switch level model, however, is a purely structural representation. Therefore, to enable the use of equivalence checking at the switch level, the actual functionality of the circuit has to be obtained first. This is achieved by applying switch level analysis techniques [2][3].
A novel switch level analysis platform called the SLV (Switch Level Verification) tool has been developed at Motorola over the last four years and used within an automated formal equivalence checking flow. Given a switch level model, SLV can produce an equivalent RTL model in a hardware description language such as Verilog [9] suitable as an input to an equivalence-checking tool. SLV has been successfully applied to other flows, e.g. in the automated generation of test models from switch level designs [16].
The design of the algorithms behind SLV has been driven by a number of design goals and constraints. Firstly, the algorithms of choice had to cover the unique blend of design styles in use at Motorola. Secondly, a switch level model had to be analyzed with the bare minimum of additional design annotation by verification engineers. Thirdly, the analysis had to be as rigorous as possible—deriving a wrong RTL model is, in practice, worse than not completing the analysis at all. Finally, the generated RTL had to be a compact, human-friendly functional representation that preserves the original structure of the circuit. Satisfying these requirements in a practical tool requires a number of novel techniques. Two key contributions to switch level analysis—handling pre-charged logic and structural loops—are the main subjects of this paper.

2. PRELIMINARIES
The choice of semantics for the components in a switch level model affects the range of circuit behaviors that can be accurately modeled. SLV employs the semantics of the Verilog hardware description language (HDL) which was chosen as the primary language for the tool. There are, however, several significant characteristics that can be modeled in the Verilog HDL that are ignored by SLV. In the absence of complete formally specified semantics for Verilog, semantics have been implied from a reference HDL simulator.
At the switch level, a MOS transistor is represented as a switch. Switches usually have two switched terminals and a control terminal that correspond respectively to transistor source, drain and gate terminals. SLV models numerous refinements to this switch model, with the most fundamental distinction being between NMOS and PMOS device models. PMOS switches close when logical 0 is applied to the control terminal, and NMOS switches close when logical 1 is applied to the control terminal. Other refinements include variations on the number of controlling

terminals and resistance between the channel terminals. The flow of a signal between the switched terminals is always assumed to be bi-directional and of zero delay.

Gates are devices that model logic relationships between signals. Gates have a set of input and output terminals and define a mapping from the logic value at each input terminal to the logic value at each output terminal. The set of gate types modeled by SLV and the driving strength of the signal output by gates is defined in [9]. In SLV, gates have zero delay.

Nets serve as interconnection points in a circuit. They are assumed to have no resistance, inductance or delay associated with them. Nets are also assumed to have no associated capacitance, except in the analysis of pre-charge logic. Three classes of nets serve special purposes in a circuit:

- Supplies: Circuits are connected to a power supply with two terminals. In the switch level model, these terminals provide a source of constant logical level 0 and 1.
- Ports: Circuits have an interface that is comprised of input, output and inout port connections. It is assumed that the inputs are Boolean and that they are stable through each period of clocks.
- Clocks: Clocks are a special kind of input port periodically alternating their value. There are two types of clocks: pre-charge clocks are used in the analysis of pre-charge logic; master clocks are used in the analysis of sequential behavior.

The channel terminals of switches in a circuit form closely connected partitions called channel-connected components (CCC) [2]. A CCC comprises of a set of switches, a set of gates and a corresponding set of nets that are used to connect these components. The gate terminals of CCC switches and the input nets of CCC gates form the set of input nets that belong to a CCC (called controlling nets in [12]). According to basic laws of electronics, the logic values of all nets in a CCC can be determined from the logic values of the inputs of that CCC.

The input nets of a CCC are further divided into two distinct sub-sets. Internal CCC input nets are driven by the CCC itself and represent feedback (self-dependencies) within a CCC. External CCC input nets are either driven by another CCC or are top-level input ports. CCC outputs are a set of nets belonging to a CCC which serve as external inputs to other CCCs or as circuit output or inout ports.

Each net in the circuit can assume one of the following logic values:

- 0, when there is a path of conducting switches from this net to a source of 0, and no path of conducting switches from this net to a source of 1 that has a greater or equal driving strength than this path. Under these conditions, it is said that this net is *pulled down*.
- 1, when there is a path of conducting switches from this net to a source of 1 and no path of conducting switches from this net to a source of 0 that has a greater or equal driving strength than this path. Under these conditions, it is said that this net is *pulled up*.
- X, when there is a path of conducting switches from this net to both a source of 0 and a source of 1 that have equal driving strengths. Under these conditions, it is said that the net is pulled up and down simultaneously. A net in state X is a part of a short-circuit (DC) path.
- Z, when there is no path of conducting switches to either a source of 0 or a source of 1. We call such nets *floating*.

The primary function of the SLV tool is to determine the logic values of circuit output and inout ports as functions of the logic values applied at the circuit input and inout ports. The evaluation processes result in the logic functions of a net being determined and stored in the following forms:

- The global pull-up and pull-down functions are in terms of the circuit input and inout ports and any state-storing nets in the circuit. These Boolean functions describe the conditions under which paths of switch devices drive the net to a source of logic 1 or 0 respectively. This form is used for many electrical design checks.
- The local pull-up and pull-down functions are in terms of the external inputs to the Channel Connected Component (CCC) containing the net. These Boolean functions describe the conditions under which the net is driven by a path of switch devices to a source of logic 1 or 0 respectively within the CCC. This form is generally used in the production of output.

3. OUR EQUIVALENCE CHECKING FLOW

3.1 Basic Switch Level Analysis

The computation of the pull-up and pull-down functions of a net (referred to as net evaluation) is based on an explicit path enumeration technique [12] that operates within a CCC. It is well known that the evaluation of a net in a CCC using explicit path enumeration can potentially require the exploration of a number of paths exponential with the number of switches in the CCC [2] [12]. SLV avoids this problem by utilizing a series parallel CCC compression algorithm that reduces the circuit graph of the CCC switch network by alternatively applying parallel and then serial compression to the network until no further compression is possible [13]. The explicit path enumeration is also accompanied by the identification of false and self-dependent paths as identified by Brzozowski and Yoeli [6].

The semantics of Verilog discrete signal strengths [9] are incorporated into the evaluation of pull-up and pull-down functions for nets in a two step process:

1. A driving strength is calculated for each path of switches during explicit path enumeration. This driving strength is a function of the driving strength of the net driving the path and the number of resistive switches in the path. Separate pull-up and pull-down functions are maintained for each driving strength during this phase of the analysis.
2. The pull-up and pull-down functions for each driving strength are combined to create single pull-up and pull-down functions for the net.

Supplies are defined to have a driving strength of Supply0/1. Circuit input and inout ports have a driving strength of Strong0/1. The driving strengths associated with a gate output are Strong0/1, unless otherwise specified for a particular gate instance in the input model.

The strength of a signal arriving at a net via a path of switch devices from a supply net, an input port, an inout port, or gate output is calculated by applying the following rules for strength reduction for each of the switches in the path:

1. A non-resistive switch device in the path of switches passes a strength through its channel unchanged, except that a Supply strength is reduced to a Strong strength.
2. A resistive switch device reduces the strength of the signal passing through its channel in accordance with the rules in [9].

For each path that SLV identifies during explicit path enumeration, its driving strength is calculated according to these rules. Separate pull-up and pull-down function pairs are maintained for each of the defined signal strengths. At the completion of explicit path enumeration in the CCC, these functions are combined to form single pull-up and pull-down functions. For simplicity, consider a system with functions of three driving strengths, where a1, b1, and c1 represent the pull-up functions with decreasing strength, and a0, b0, and c0 represent the pull-down functions with decreasing strength. The combined pull-up and pull-down functions of a net 'out' are then given by:

```
out.pull-up = a1 | (~a0.~a1).b1 | (~a0.~a1.~b0.~b1).c1
out.pull-down = a0 | (~a0.~a1).b0 | (~a0.~a1.~b0.~b1).c0
```

which simplifies to:

```
out.pull-up = a1 | ~a0.b1 | ~a0.~b0.c1
out.pull-down = a0 | ~a1.b0 | ~a1.~b1.c0
```

3.2 Analysis of Pre-charged Logic

A pre-charge (or dynamic) net has two distinct phases of operation: the pre-charge phase and the evaluate phase. The pre-charge phase is assumed to be the first phase of the dynamic clocking scheme, during which the pre-charge net is unconditionally charged to either logical 1 or logical 0. The evaluate phase is assumed to be the last phase of the dynamic clocking scheme, during which the pre-charge net is conditionally discharged, so that it maintains its pre-charge logic level if the required conditions for discharge are not satisfied. Any dynamic clock phases between the pre-charge and evaluate phases are assumed to be transient, and do not contribute to the required output model.

Figure 1 illustrates a typical example of pre-charge logic in which the net labeled 'out' is a pre-charge net. During the pre-charge phase, the dynamic clock input 'clk' is logical 0. The capacitor 'C' is pre-charged to logical 1 since there are no conducting paths to a supply of logical 0 and there is a conducting path to a supply of logical 1. In the evaluate phase, the dynamic clock input is logical 1. If ‘a’ and ‘b’ are both logical 1, then there is a path to a supply of logical 0 and ‘C’ is discharged. Otherwise ‘C’ maintains a logical 1. This implements a NAND logic gate.

The presence of logic that is dependent on dynamic clocks in the discharging network, and which prevents discharge paths from conducting in the pre-charge phase of the clocking scheme is referred to as ‘footed’ pre-charge logic. The example shown in Figure 1 illustrates footed pre-charge logic due to the NMOS transistor that prevents the discharge path in the pre-charge phase when ‘clk’ is logical 0. The absence of footing logic is referred to as ‘non-footed’ pre-charge logic.

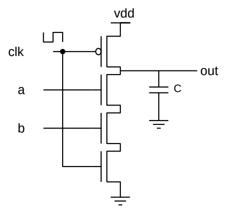

Figure 1 Example of a dynamic nand gate

The major tasks in the analysis of pre-charge logic are to identify nets that are pre-charged and analyzed under a dynamic clocking scheme (termed ‘pre-charge nets’), and, for each pre-charge net, to derive the pre-charge functionality and obtain the static behavior.

3.2.1 Identifying Pre-charge Nets

There are two elements of the switch level model that are specific to the analysis of pre-charge logic, being net capacitance and dynamic clock waveforms. All nets that are connected to a transistor gate or the input to a logic gate are assumed to have a capacitance and are capable of holding a logical value. An alternative to capacitors are weak state-holding structural loops that maintain the logical level of a net while it is not driven by a stronger signal.

A dynamic clock waveform defines the clocking scheme for the circuit and is specified by the user. SLV only requires the user to define the dynamic clock inputs of the top-level block in the design hierarchy as it is capable of handling gated clock designs and propagates the clocking scheme throughout the logic in the clock tree during the circuit analysis.

In the initial stages of CCC evaluation, a CCC is marked as being dynamic if it has one or more dynamic clock external inputs. The dependence of this CCC on a dynamic clock input is used to suggest the presence of pre-charge logic. The nets within a dynamic CCC are subsequently examined and can be classified as either a pre-charge net, or a dynamic clock (indicating presence of gated clock logic in this CCC).

A net is labeled as a dynamic clock if either the net is specified as such and associated with a waveform by the user, or if a dynamic CCC drives the net and it is not a pre-charge net and its global functions are dependent on a dynamic clock BDD variable (see Table 1).

A net in a dynamic CCC is labeled as a pre-charge net if the net may float (i.e. it is not driven to a logical 1 or 0) in the evaluate phase of the dynamic clocking scheme and:

1. The net has level restoring logic indicated by a sequential loop; or
2. The local functions of the net are dependent on the capacitance of the net (see Table 1).

The SLV tool has a user selectable ability to ignore net capacitance and only classify pre-charge nets based on condition 1 above, which is useful in the analysis of a particular design style.

In order to model net capacitance in a dynamic CCC, each capacitor C_i in the CCC is associated with two new independent local variables, V_{Ci}^1 and V_{Ci}^0 respectively, representing the pull-up and pull-down drive of the capacitor. The subsequent analysis of each dynamic CCC output derives local pull-up and local pull-down functions that may be dependent on both CCC inputs and any V_{Ci} of the CCC. Immediately after explicit path enumeration the type of a net can be inferred from the dependence of the local functions on V_{Ci}, and the dependence of the global functions on dynamic clock input variables, as shown in Table 1. In this table C_a represents the capacitor variables of the net that is currently being evaluated. Note that bad charge sharing is defined to occur when a net with capacitance is locally dependent on the capacitance of any other net in the same CCC, and the two nets are pre-charged to opposing values.

Table 1 Net Classification in Dynamic CCCs

Local functions depend on V_{Ca}	Local functions depend on V_{Ci} other than V_{Ca}	Net can float in the evaluate phase	Net Classification
Yes	Yes	Yes	This is a pre-charge net. There is the potential for bad charge sharing if capacitors in the local functions are pre-charged to opposite values.
Yes	No	Yes	This is a pre-charge net.
All other combinations			The net is a dynamic clock if its global functions are dependent on a dynamic clock BDD variable.

In order to avoid interfering with subsequent processing steps, once this net classification is complete, logical 0 is substituted for each V_{Ci} in the local pull-up and pull-down functions. Following this, the local pull-up and pull-down functions become what they would be for a non-dynamic CCC. If the net has not been marked as a pre-charge net then no further processing is required, otherwise the local and global functions of the pre-charge net are modified to model the static behavior.

3.2.2 Modifying the Functions of a Pre-charge Net

One of two distinct modifications is made to a pre-charge net’s local and global functions, depending on whether the net is pre-charged to logical 1 or logical 0. Therefore, as a first step in modifying the functions of a pre-charge net, its global functions are used to determine if it is pre-charged to logical 1 or logical 0. This is achieved by substituting the actual value of all dynamic clock inputs in the pre-charge phase into the net’s global functions. This result indicates either pull-up or pull-down pre-charge as shown in Table 2.

The heuristic presented in Table 2 determines the pre-charge behavior for a wide variety of dynamic design styles including gated clocks, footed or non-footed pre-charge logic, pre-charge paths containing resistive transistors or logic gates with weak driving strengths, and pre-charge level restoring logic.

Table 2 Heuristic for Determining Pre-charge Behavior

Resulting global pull-up	Resulting global pull-down	Additional Conditions	Interpretation
Constant 1	Not constant 1	N/A	Pull-up pre-charge
Not constant 0	Constant 0	N/A	Pull-up pre-charge
Constant 0	Not constant 0	N/A	Pull-down pre-charge
Not Constant 1	Constant 1	N/A	Pull-down pre-charge
Modified	Unchanged	Resulting global pull-up and pull-down are complementary	Resistive pull-up pre-charge
Unchanged	Modified	Resulting global pull-up and pull-down are complementary	Resistive pull-down pre-charge
Modified	Modified	Resulting global pull-up and pull-down are complementary and the strongest driving path is a pull-up path	Resistive pull-down pre-charge
Modified	Modified	Resulting global pull-up and pull-down are complementary and the strongest driving path is a pull-down path	Resistive pull-up pre-charge
All other combinations			The net is not a pre-charge net.

Based on the pre-charge behavior, the local and global functions of a pre-charge net are modified to model the static footed behavior of the net. The first step in this process is the determination of the global pre-charging function, which is the Boolean function representing the global variable condition that results in the pre-charge behavior of the net. The technique for determining the global pre-charging function uses the pre-charge behavior from Table 2 and is shown in Table 3. In Table 3 ‘gpu’ and ‘gpd’ respectively represent the global pull-up and pull-down functions of the pre-charge net, 'Universal' represents the universal quantification operation, and 'Constrain' represents the BDD constraining function. The local pre-charging function is determined by translation from the global pre-charging function.

Table 3 Determining the Global Pre-charging Function

Pre-charge Value	Resistively Pre-charged	Global Pre-charging Function
1	Yes	gpu.Constrain(~gpd).Universal(<All BDD variables not representing dynamic clocks>)
1	No	gpu.Universal(<All BDD variables not representing dynamic clocks>)
0	Yes	gpd.Constrain(~gpu).Universal(<All BDD variables not representing dynamic clocks>)
0	No	gpd.Universal(<All BDD variables not representing dynamic clocks>)

The functionality of footing logic is added to the local and global functions of the pre-charge net by the following operations:

Pre-charged to logic 1: pull-dn=pull-dn&~(pre-charging-function)
Pre-charged to logic 0: pull-up=pull-up&~(pre-charging-function)

The footing logic is first added to the global functions of the pre-charge net, and non-footed dynamic logic is implemented if this translation modifies the global functions of the pre-charge net. A similar translation is only applied to the local pull-down function of the pre-charge net if non-footed logic is being modeled.

The final step in modeling the footed static behavior of the pre-charge net is to apply a translation that represents the level-restoring behavior of the net when no discharge paths conduct. If the pre-charge net is pre-charged to logical 1, then its local and global functions are modified according to the following scheme:

pull-up = pull-up | ~(pull-down function)
pull-down = ~(pull-up function)

Similarly, if this net is pre-charged to 0, then its local and global functions are modified as follows:

pull-down = pull-down | ~(pull-up function)
pull-up = ~(pull-down function)

SLV provides two modes of modes of output representation for pre-charge nets, the first includes explicit references to the dynamic clocks in the behavior of pre-charge nets, and the second makes no reference to dynamic clocks. This allows users to select the mode that suits their design style.

3.3 Analysis of Structural Loops

Current generation equivalence checking tools rely on a close structural correspondence between the circuits being compared. In practice, a one-to-one mapping between the state points (latches) of the two circuits is required to identify the cones of logic that need to be compared. To apply these tools to a switch level block, one has to determine its sequential behavior by reliably identifying the state points as intended by the logic designer and not introducing any additional ones.

Traditional switch level symbolic analysis derives a gate level model that contains fine-grained sequential behavior introduced by a special simulation clock that is not part of the original design, and associated unit-delay gates [2][4]. Therefore, this gate level model is not suitable for comparison with an RTL model as is; further processing is required to abstract the detailed timing information and obtain an FSM representation suitable for formal verification [10][11]. However, the algorithms performing this abstraction rely on symbolic reachability analysis, which is likely to limit their capacity. Thus, some commercial tool vendors are relying on pattern-matching techniques to identify the latches in a design [7][14], while others require users to pinpoint the state points in the input design [12].

From a practical point of view, sequential behavior in circuits is implemented either by structural loops (latches, keepers and half-keepers), or by storing electrical charge (dynamic logic, dynamic memory arrays). Since the verification of dynamic memory arrays is not an application SLV is currently targeted at, this section focuses on the analysis of structural loops in the switch level circuit.

The primary objective of structural loop analysis is to determine whether a loop is combinational, sequential, or oscillatory. A secondary requirement is the ability to produce an RTL description for the loop in case it is not found to be oscillatory. Our approach to this extends [12][15].

Firstly, we consider structural loops involving two or more CCCs (called external CCC loops). Graph-based algorithms are used to traverse the connectivity of the CCCs in the switch level circuit from the outputs to the inputs. Any closed loops are broken by inserting two new independent boolean variables v_0 and v_1 representing, correspondingly, the pull-down and pull-up functions of the net N closing the loop (referred to as a *shadow* net) [12][15]. Note that selecting, as a shadow net, any of the nets on the boundaries between CCCs that form it will break a loop. To aid in the subsequent equivalence-checking step, SLV employs a set of heuristics guiding the choice of a shadow net.

During the analysis of the logic cone driving a shadow net, the variables v_0 and v_1 are used to resolve the circular dependencies in the logic and the global pull-down function F_0 and pull-up function F_1 are computed. Examining the dependencies of F_0 and F_1 on v_0 and v_1 performs the loop analysis. We introduce a function depend(F, v) that evaluates to 0 when $F|_{v=1}=F|_{v=0}$ and to 1 otherwise. Then, a so-called loop signature LS is computed as follows:

$$LS = \text{depend}(F_1, v_1)*8 + \text{depend}(F_1, v_0)*4 + \text{depend}(F_0, v_1)*2 + \text{depend}(F_0, v_0)$$

By definition, LS has an integer value between 0 and 15 characterizing the self-dependencies in the corresponding structural loop. For example, LS=0 implies that the loop is purely combinational (this has been previously noted in [12]). Similarly, LS=9 implies that the loop is likely to be sequential, although a further check is required to ensure that there is no risk of uncontrolled oscillation of that loop (see below).

Table 4 Loop Signature Interpretation

LS	Inferred type of loop / Action
0	Combinational loop; the pair of boolean functions does not depend on either of the temporary boolean variables.
1	Sequential loop implementing a half-latch that can keep a value of 0 but not a value of 1.
8	Sequential loop implementing a half-latch that can keep a value of 1 but not a value of 0.
9	Sequential loop implementing a latch.
other	F_0 and F_1 are simplified under the assumption that the shadow net can only take boolean values; LS is re-computed.

The complete set of LS values and their interpretations are provided in Table 4. In some cases, the value of LS does not uniquely identify the nature of the structural loop. To avoid that, the value of LS is re-computed after F_0 and F_1 are simplified under the assumption that the shadow net can only take Boolean values. This is achieved by substituting any occurrence of v_0 in F_1 by $\sim v_1$, and any occurrence of v_1 in F_0 by $\sim v_0$.

This step is, of course, based on an assumption that is later discharged by verifying that the net cannot have X or Z value. Note that the re-computed loop signature is guaranteed to have a value of 0, 1, 8, or 9.

When the structural loop under analysis is found to be combinational, the only remaining task is to substitute v_0 with F_0 and v_1 with F_1 in the global functions of all nets that depend on the temporary Boolean variables. However, if the loop is inferred to be of sequential nature, the extra steps of functional decomposition and a check for oscillatory behavior are required. The functions F_0 and F_1 are decomposed into sub-functions as follows:

$$F_0 = X_0 \mid Y_0.v_0 \mid Z_0.\sim v_0$$
$$F_1 = X_1 \mid Y_1.v_1 \mid Z_1.\sim v_1$$

where functions X_0, Y_0, Z_0 do not depend on v_0 and do not share common cubes, and, similarly, functions X_1, Y_1, Z_1 do not depend on v_1 and do not share common cubes. These functions have the following interpretations:

- X_0 (X_1) covers the case when F_0 (F_1) is independent of v_0 (v_1). Since the loop is sequential, this is the condition under which the loop's environment "writes" a value into the loop's latch;
- Y_0 (Y_1) covers the case when F_0 (F_1) take their previous value v_0 (v_1). In terms of structural loop behavior this is the condition under which the loop's latch maintains its stored value;

- Z_0 (Z_1) covers the case when F_0 (F_1) take the negation of their previous values ~v_0 (~v_1). Thus, if $Z_0 \neq 0$ or $Z_1 \neq 0$ the structural loop exhibits oscillatory (unstable) behavior. This clearly points to a design error and prevents the algorithm from obtaining a functionally equivalent RTL description of the input switch level circuit.

The above decomposition of F_0 and F_1 not only allows us to identify unwanted oscillatory loops, but also provides input to the final step of RTL output generation for the loop. One way of describing the loop behavior in the Verilog language (assuming X_0 and X_1 do not share common cubes) is:

```
reg N;
always @(<list of nets that X0 and X1 depend on>)
    If (X0 | X1)
            N = X1;
```

In practice, further analysis is applied to identify asynchronous and synchronous set/reset signals, clocks and data inputs and produce a much more detailed and human-friendly style of RTL description. Loops that implement latches are distinguished from loops that implement keepers based on a functional dependence of X_0 and X_1 on one or more of the master clock inputs defined by the user, and different styles of output are generated. Other types of loop behavior analysis on the decomposed F_0 and F_1 are possible and actually employed in SLV. For example, X_0=0 and X_1=0 implies the presence of a latch that cannot be written to and, therefore, can be regarded as a source of unknown value (logical X). Such conditions are reported to the user.

A separate technique has been developed for the analysis of structural loops that occur within a single CCC (internal CCC loops). Although some specific but common cases such as Brzozowski-Yoeli loops [6] can be dealt with using rather simple techniques, the design goals of SLV and the requirements of its user community required a more general solution to this problem. In our presentation of the internal CCC loop resolution technique, we consider a CCC with m external inputs with functions $e_1, e_2, \ldots, e_m$, and n>0 internal inputs. Again, we notionally break the loops at the internal inputs by introducing a pair of temporary variables—u_k for the local pull-up function and d_k for the local pull-down function at the k-th input—one for each internal input.

Next, a system of boolean equations B is formed. The actual local pull-up (U_k) and pull-down (D_k) functions for all internal inputs are obtained in terms of e_i, u_i, and d_i. Then, for any given internal input k, the following equations must hold when the signals at the inputs of the CCC are stable:

$u_k = U_k$

$d_k = D_k$

The 2n equations obtained in this way form the system of equations B. In essence, B captures the conditions under which all nets in the CCC hold a stable value. Solving B with respect to u_k and d_k using a method such as Gaussian elimination can lead to one of the following results:

- If B has no solutions, then the CCC cannot reach a stable state, and it is therefore established that at least one of the internal CCC loops results in oscillation. This is a design error and no RTL output is produced;
- If B has exactly one solution, then u_k and d_k are uniquely determined in terms of e_i. Therefore, all structural loops in the CCC are of combinational nature and the local functions of all nets inside the CCC can be obtained by substituting the temporary boolean variables u_k and d_k with their solutions in B;
- If B has more than one solution, then at least one of the internal CCC loops is sequential. Further analysis is required to handle this case.

Two observations are important in order to understand our approach to handling the latter of the three cases above. Firstly, even if B does not have a unique solution, it is quite possible that u_k and d_k do have a unique solution for a given k. Thus, we can rule out the possibility of a certain internal input being a state storing net without having to solve B. Secondly, u_k and d_k not having a unique solution for a given k implies that either (1) the k-th internal input is a state storing net, or (2) the k-th internal input is combinationally dependent on at least one state storing point inside the CCC.

The analysis of the case when B has more than one solution proceeds in an iterative fashion:

1. The set of pairs of temporary boolean variables u_k and d_k that have unique solutions are identified, and their nets are marked as non-state storing ones. Substituting the identified pairs of boolean variables with their unique solutions modifies the system of equations B. This step reduces the number of the remaining free temporary variables in B.
2. If there are no more free variables in B, all internal inputs have been categorized, and there is no more work to do.
3. One of the remaining unresolved nets is picked up and selected as a state storing one. The system of equations B is modified to make the corresponding temporary boolean variables be parameters rather than unknown entities in B. This step makes B more constrained with respect to the remaining free temporary variables. The three steps above are repeated starting with point 1. above.

3.4 Tool Integration and Flow Automation

The compareS2RTL tool encapsulates an automated flow for performing formal equivalence checks between two designs, each design being at the switch, gate, or RTL abstraction level. The compareS2RTL tool controls the flow of data between each tool that it employs, providing a single generic interface to the user that allows different back-end tools or modules to be plugged-in with minimal impact to the development environment. Typically the compareS2RTL tool is applied to a switch level design and an RTL design; the compareS2RTL flow for this is illustrated in Figure 2.

The compareS2RTL tool employs SLV to create an RTL model from the switch level design. The equivalence check is subsequently performed between the extracted RTL model and the RTL design using a functional equivalence checker such as Tuxedo-LEC [18]. A sequential model checker is applied if the equivalence checker did not determine that the two designs are equivalent, which can often be a result of differences by the tool in mapping sequential elements between the designs.

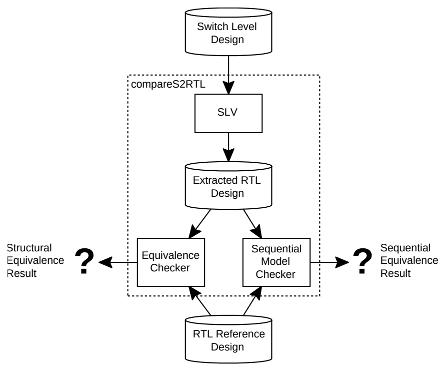

Figure 2 Equivalence Checking using compareS2RTL

The compareS2RTL tool is very versatile, allowing the user full control of which tools and options are used. It provides a single interface for the user to become familiar with and automatically controls the passage of information between tools. Another benefit is that the compareS2RTL tool automatically passes BDD variable ordering information between tools, resulting in fewer reordering operations and improved performance.

4. CONTRIBUTIONS AND RELATED WORK

The main contribution of this paper is the description of two novel switch level analysis algorithms. The first algorithm provides automated identification and analysis of pre-charge logic in the presence of complex logic structures such as gated clocks, non-footed logic, and charge sharing. The other algorithm is concerned with structural loop analysis and classifies loops as combinational, sequential, or oscillatory (unstable).

Many of the fundamental technologies for symbolic analysis of switch level circuits grew out of efforts to increase the speed and capacity of switch level simulation. Bryant pioneered several important techniques in this field, including novel methods for switch level modeling [2], CCC analysis [3][4] and the representation of boolean logic with reduced ordered binary decision diagrams [5]. While this research clearly demonstrated the utility of switch level analysis, the development of software to generate an output model suitable for formal equivalence checking was not its primary objective.

A tool called Anamos reads a switch level model in a Spice-like language and generates an equivalent gate level model [2][4]. The gate level modeling style employs fine-grained sequential behavior introduced by a special simulation clock that is not part of the original design and unit-delay gates to represent sequential behavior. This gate level model is suitable for simulation but not appropriate for comparison with an RTL model as is. In contrast to this, the techniques for dynamic logic and structural loop analysis in SLV ensure the preservation of the original state storing points in the input design and produce a netlist that is suitable as input to equivalence checking tools.

Several commercial and academic tools and tool flows are based on the switch level analysis engine implemented in Anamos. Significantly, experiments have attempted to implement algorithms that process the output of Anamos with the objective of transforming it into a more usable form of HDL [10][11]. These algorithms rely on symbolic reachability analysis, which is likely to limit their capacity. A major advantage of our approach to structural loop analysis is the fact that each loop is analyzed in isolation avoiding such computationally intensive. This has a positive effect on the size of circuits that can be handled. This increase in tool capacity with respect to the background art has been achieved without a compromise in accuracy or rigor of analysis, and these algorithms have exposed subtle custom design problems on a regular basis.

Verity [12] implements efficient algorithms for verifying the behavior of switch level models with static combinational and simple dynamic logic. This has seen the tool used with great success in commercial custom VLSI design settings. However, Verity cannot perform analysis of structural loops and depends on user input to resolve these. Unlike SLV, Verity cannot generate an RTL representation of a switch level design, which limits this tool’s utility to formal verification flows only.

Pattern matching techniques have also been used for switch level circuit extraction [7][14]. Tools employing this approach repetitively attempt to match parts of the input design and replace them with RTL components from the library. Unfortunately, there is no guarantee that the component library is both correct and complete. The switch level analysis in SLV, on the other hand, is purely functional and does not rely on pre-defined structures and patterns, which translates in an ability to handle a wide variety of design styles.

5. EXPERIMENTAL RESULTS

SLV has been in production use across Motorola for several years and has been applied to the verification of key PowerPC, M*Core and DSP custom blocks. As a generic switch level analysis platform, SLV has also successfully targeted a number of additional applications—library characterization, legacy design re-engineering, and generation of gate level test views from switch level logic [16].

Table 5 Benchmark Results from Running SLV

Design	Transistors	Loops	Latches	Dyn. Cells	CPU Time (s)	Memory (MB)
1	18	2	2	0	0.1	3.5
2	157	0	0	12	0.1	3.5
3	5615	174	158	16	0.7	9.3
4	5814	395	0	395	11.7	19
5	7329	557	0	557	21.0	47
6	37549	1304	1270	0	11.7	35
7	89667	3771	3456	0	111.0	35

Experimental results for SLV are presented in Table 5. SLV was executed for all benchmarks on a Sun Blade 1000 with 1024 MB of RAM. The benchmarks range from simple flip-flops and adders to complex Motorola proprietary designs. Table 5 demonstrates the ability of SLV to analyze a representative set of switch-level designs including a mixture of structural loops, dynamic, and sequential logic.

6. CONCLUSIONS

This paper presented a complete framework for formal verification of designs at the switch level. The algorithms that make SLV include novel contributions to the state of the art in the areas of pre-charged logic analysis and structural loop and sequential logic analysis. These improvements made possible the satisfaction of all high-level requirements set for SLV from the very beginning of the project—coverage of a wide set of design styles, minimal change and added overhead for established custom design practices, robustness, and user-friendly RTL output comparable to manually written models.

One area for improvement in SLV is increasing the size of the blocks that can be handled in a single run. One approach that looks particularly promising is utilizing a non-canonical representation for boolean functions such as Boolean Expression Diagrams [1], coupled with a joint use of BDD- and SAT-based [8] decision techniques.

7. ACKNOWLEDGMENTS

We would like to acknowledge the importance of the contribution of Vladimir Grudinin, Neil Boroky, and Craig Biggs at various stages of the development of SLV. This work would not be possible without the immense support of the Design Verification group at Motorola by Carl Pixley, Michael Burns, Jun Yuan, and Kurt Shultz.

8. REFERENCES

[1] Andersen, H.R., and Hulgaard, H. Boolean Expression Diagrams. LICS’97, pp. 88-98, 1997.

[2] Bryant, R.E. Boolean analysis of MOS circuits. IEEE Transactions on CAD-ICS, vol. 6, no. 4, pp. 634-649, July 1987.

[3] Bryant, R.E. Algorithmic aspects of symbolic switch network analysis. IEEE Transactions on CAD-ICS, vol. 6, no. 4, pp. 618-633, July 1987.

[4] Bryant, R.E. Extraction of gate level models from transistor circuits by four-valued symbolic analysis. Fujitsu Labs Ltd., April 1991

[5] Bryant, R.E. Graph-based algorithms for Boolean function manipulation. IEEE Transactions on Computers, vol. C-35, pp.677-691, August 1986.

[6] Brzozowski, J.A., and Yoeli, M. Combinational Static CMOS Networks. Integration, the VLSI Journal, 5, pp. 103-122, 1987.

[7] Groupe Bull. Generalized recognition of gates: User Guide. Version 06, April 1996.

[8] Gu, J., Purdom, P.W., Franco, J., and Wah, B.W. Algorithms for the satisfiability (SAT) problem: a survey. DIMACS Series in Discrete Mathematics and Theoretical Computer Science, American Mathematical Society, pp. 19-152, 1997.

[9] IEEE. 1364-1995, Verilog Language Reference Manual. 1995.

[10] Jain, S., Bryant, R.E., and Jain, A. Automatic clock abstraction from sequential circuits. DAC’95, pp. 707-711, 1995.

[11] Kam, T., and Subrahmanyam, P.A. Comparing layouts with HDL models: a formal verification technique. IEEE Transactions on CAD-ICS, vol. 14, no. 4, pp. 503-509, April 1995.

[12] Kuehlmann, A., Srinivasan, A., and LaPotin, D.P. Verity—a formal verification program for custom CMOS circuits. IBM Journal of Research and Development, vol. 39, no 1,2, January/March 1995.

[13] Kundu, S. GateMaker: A Transistor to Gate Level Model Extractor for Simulation, Automatic Test Pattern Generation and Verification. IEEE International Test Conference, pp. 372-381, 1998.

[14] Laurentin, M., Greiner, A., and Marbot, R. DESB, a functional abstractor for CMOS VLSI circuits. EDAC’92, pp. 22-27, 1992.

[15] Malik, S. Analysis of cyclic combinational circuits. ICCAD’93, pp. 618-625, 1993.

[16] McDougall, T., Parashkevov, A., Jolly, S., Zhu, J., Zeng, J., Pyron, C., and Abadir, M. SLV/ATPG: An Automated Flow for Test Model Generation from Switch Level Custom Circuits. Submitted to MTV ‘02

[17] Synopsys. Synopsys Online Documentation. Version 2001.08, August 2001.

[18] Verplex Systems. Logical Equivalence Checker Reference Manual version 3.0. August 2001.