Modeling and Verification of Industrial Flash Memories

Sandip Ray[1], Jayanta Bhadra[2], Thomas Portlock[2], Ronald Syzdek[2]
[1]Department of Computer Science, University of Texas at Austin, Austin, TX, USA.
[2]Freescale Semiconductor Inc., Austin, TX, USA.
[1]E-mail: sandip@cs.utexas.edu

Abstract—We present a method to abstract, formalize, and verify industrial flash memory implementations. Flash memories contain specialized transistors, *e.g.*, floating gate and split gate devices, which preclude the use of traditional switch-level abstractions for their verification. We circumvent this problem through behavioral abstractions, which allow formalization of the behaviors of the design as interacting state machines. Behavioral abstractions are agnostic to transistor type, making them suitable for formalizing flash memories. We have verified industrial flash memory implementations based on both floating gate and split gate technologies. Our work provides the first formal functional verification results for industrial flash memories.

Keywords—equivalence checking, formal analysis, simulation, SPICE, theorem proving

I. Introduction

The goal of memory verification is to check that the transistor network implementing the memory corresponds to the high-level view of a state machine that stores and retrieves data at addressed locations. Memories account for more than half of a modern microprocessor design, both in real estate and in transistor count. Furthermore, modern memory implementations are complex artifacts, with subtle pipelining, power optimization, and area compression features. Consequently, the functional verification of memory cores is a crucial component of the verification of a microprocessor or SoC design.

Since transistors are inherently analog artifacts, comprehensive verification of a transistor network requires extensive analog (SPICE) simulations. The size and complexity of a memory core precludes analog simulation on the entire core. Consequently, a memory verification tool flow typically consists of two components. First, SPICE simulations performed at the level of a bitcell and associated logic blocks, to check that the bitcell performs according to specification. Such simulations are extensive and detailed, covering various process corners and operating conditions, but can only be carried out at the level of *single bitcells*. Second, fast high-level or RTL simulations are performed to check that the *entire memory core* operates correctly when embedded within a larger SoC block. To facilitate this, the implementation of the memory core itself is abstracted to a RTL or a high-level (C/C++) model that represents its interface to the surrounding block.

A consequence of this two-stage approach is a “verification gap”: given that the individual bitcells operate correctly, how can we ensure that the network of bitcells implements the abstraction representing its interface? For SRAM memories, this gap is bridged by switch-level analysis [1] which abstracts the network as a graph of switches. Unfortunately, switch-level analysis cannot be extended to flash and non-volatile memories that depend on analog effects of transistors for proper functionality. Indeed, in the industrial verification tool flow, there is currently no way to compare a bitcell-level implementation of a non-volatile memory with the high-level specification of its interface. Consequently, incorrect composition of memory bitcells with surrounding SoC logic can cause subtle design errors which are difficult to detect or diagnose.

In previous work [2], [3], the idea of behavioral abstraction was proposed to ameliorate the above problem. The idea is to abstract the behavior of the memory core as a composition of interacting state machines; each individual state machine component represents behaviors of bitcell-level structures that are validated by SPICE. The approach is agnostic to transistor type, making it suitable for non-volatile memories, *e.g.*, flash. Furthermore, the correspondence with SPICE models facilitates corroboration of the abstractions with readily available simulation data, and identification of corner case bugs potentially missed during analog simulation.

In this paper, we develop framework, based on behavioral abstraction, to verify industrial flash memory designs. We verify designs with both floating gate and split gate technologies and for NOR and NAND configurations. Our framework handles the intricacies of industrial designs, *e.g.*, split gate represents the cutting edge in flash technology and we verify a split gate design from a leading semiconductor company. Our work thus provides a comprehensive methodology to bridge the verification gap mentioned above for non-volatile memories. Indeed, to the best of our knowledge, our research provides the first results on formal functional verification of industrial non-volatile memories.

The rest of the paper is organized as follows. Section **2** provides the necessary background, explaining the inadequacies of switch-level abstractions in modeling flash memories and describing the role of behavioral abstractions to circumvent those deficiencies. Section **3** describes our behavioral models of flash memories. Section **4** discusses our verification framework. Section **5** provides an estimate the effort involved, both in the development of the framework and in its subsequent application on concrete industrial design verification. We discuss related work in Section **6**, and conclude in Section **7**. The paper is self-contained, but familiarity with flash designs will be helpful in appreciating the discussions. The verification was done with the ACL2 theorem prover [4], an industrial-strength theorem prover for a first-order logic of recursive functions. However, the paper does not assume familiarity with ACL2; we eschew ACL2's Lisp syntax and use standard mathematical notations throughout the presentation.

II. Background

A. Deficiencies in Switch-level Models

Current industrial practice uses switch-level models to verify memory designs. In particular, switch-level analyzers such as ANAMOS [5] and its variants [6], [7] represent the state of the art in the abstraction of SRAM memories built with CMOS transistors. The key insight for such models is that transistors can be effectively abstracted as on-off switches from the perspective of their use in designs (such as memories) that implement digital behavior. Consequently, switch-level models abstract a transistor network as a collection of nodes connected by transistor switches. Each node has state 0, 1, or X; each switch has state "open", "closed", or "indeterminate"; state transitions are specified by switch equations. The models capture a number of transistor characteristics, *e.g.*, bidirectionality, signal strengths, etc. Switch-level analyzers partition a network into *channel connected subcomponents* and analyze each component separately to construct switch equations.

Even for SRAM designs, analog effects ignored by switch-level analyzers may have pronounced impact. For instance, the transistor strength assignment procedure in ANAMOS produces significant mismatch with SPICE models for networks with closely matching but different strengths [8]. The traditional response to such discrepancies has been to design more and more elaborate analyzers [6]; however, this does not solve the fundamental problem of representing inherently analog behaviors with equations in a discrete algebra.

The accuracy problem is prohibitive for non-volatile memory designs because analog effects are not only present but *exploited* for correct bitcell operations. Consider the a flash bitcell that includes both CMOS and floating gate (FG) transistors. It contains, in addition to conventional *drain* (D), *gate* (G) and *source* (S) terminals, a *floating gate* (F) — a polysilicon layer in the oxide between the gate and the substrate that is disconnected from both S and D (Fig. 1). By controlling the stored charge in the capacitive coupling between G, F, and the substrate, the threshold voltage V_{th} (the minimum voltage to turn on the device) is regulated dynamically to design a bitcell with a single FG transistor: a low threshold voltage (V_{th}^L) represents logic 1 and high threshold voltage (V_{th}^H) represents logic 0. However, the capacitive coupling breaks the view of a transistor network as a graph of switches, precluding switch-level analysis.

B. Behavioral Abstractions

The rationale for behavioral models is that custom memories are designed *not* as an ad hoc transistor network

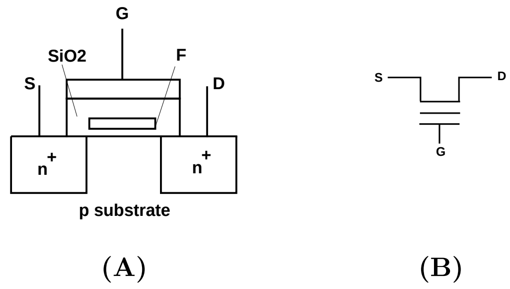

Fig. 1. (A) The structure of an FG Transistor. (B) Schematic used to represent an FG transistor.

but by carefully interconnecting components (*e.g.*, bitcells, sense amplifiers, etc.) with well-understood behavior within the operation. range. In Motorola's design flow, components operate over a limited sequence of certified stimulus patterns, each of which is validated by SPICE simulation across operating conditions [8]; any pattern not validated is illegal. Thus, memory verification must ensure that (1) each component performs within operating conditions, and (2) assuming each component operates correctly, their composition implements the high-level specification of the memory core.

Behavioral abstractions formalize this insight. Instead of viewing the memory as merely a graph of transistors, we model the behavior of the individual components *within the operating constraints* as specified by SPICE simulation. Since operating constraints specify a partial behavior, the models are *constrained* rather than total. More precisely, the behavior of each component is formalized as a guarded state machine, with guards specifying the operating constraints for each transition; if the guards are violated, the behavior is unspecified. For instance, the behavior of a bitcell under *read* operation is defined only when its environment produces the stimuli sequence to select its word line followed by appropriate gate voltage; nothing is known when these conditions are violated. The behavior of the memory core is formalized by an interactive composition of state machines for the individual components.

The approach, albeit simple, provides several benefits. First, it is agnostic to transistor type: the same approach works for both SRAM and flash memories. Second, since the formal constraints exactly correspond to conditions used in SPICE simulations, models can be validated with readily available SPICE data; thus, unlike switch-level abstractions, our models preserve correspondence with analog behavior while obviating the need to formalize low-level analog effects (*e.g.*, capacitive coupling). Third, since the abstractions are state machines, traditional functional verification techniques can handle these models. Finally, nondeterminism in the formalization naturally captures effects of unsuccessful electrical operations inherent in analog behavior.

III. Flash Formalization

A. Floating Gate Bitcells

In order to explain behavioral models of an FG bitcell, we briefly summarize its operations. Cappelletti *et al.* [9] provide a detailed treatment of FG operations. The three main operations are *read*, *program* (writing 0), and *erase* (writing 1). Note that the operations are inherently analog. A low-level model (*e.g.*, one derived from solid-state physics) reflecting all facets of the behavior is intractable for verification.

Read: For the selected bitcell, one applies a voltage v ($V_{th}^L < v < V_{th}^H$) at G which is driven by the selected word line, while keeping other word lines at ground. If the cell has logic 0, the transistor does not turn on and no current flows to the associated sense amplifier; otherwise the bitcell turns on and current is detected, reading a 1.

Program: The *Channel Hot-Electron Injection* procedure injects negative charge into the FG, raising its V_{th} to V_{th}^H. Then there is a verification phase to ensure that V_{th} has been appreciably raised; this is done by "reading" the cell with a gate voltage v ($> V_{th}^H$). A result of 0 for the read indicates successful programming; otherwise programming is iterated until it succeeds or a specified number of attempts have been made, signalling failure in the latter case.

Erase: Erasing is based on removal of stored charge by *Fowler-Nordheim tunneling.* This involves (i) *raising* the V_{th}s of the bitcells in the sector to V_{th}^H by programming, (ii) charge removal to lower the V_{th} to V_{th}^L, and finally, (iii) normalization, which employs *soft programming* to increase the V_{th} of the cells that have fallen below V_{th}^L.

The description underlines the analog nature inherent in flash operations. A low-level model (*e.g.*, one derived from solid-state physics) reflecting all facets of this behavior is intractable for verification. Additional factors to account for in abstracting flash memories include (i) multiple voltage levels, (ii) charge injection and removal, and (iii) complex sense amplifier activity to compare various current values.

Behavioral models formalize these operations by capturing the *effect* of electrical operations while abstracting the physical reasons. For instance, reading requires the following stimuli: (1) precharge (2) applying a voltage v, ($V_{th}^L < v < V_{th}^H$) at the gate G for the bitcell; each stimulus must also permit appropriate setup time. Fig. 2 shows a simple state machine for this sequence. The state machine is only partially specified, stipulating deterministic behavior only under this stimuli sequence; the behavior is unspecified otherwise. The output behavior for the state machine is the "activation" of the sense amplifier, that is, a current flow if and only if the bitcell has the value 1. Correspondingly, the state machine for sense amplifier (not shown) is enabled by the output transition of the bitcell state machine (providing values for the bitlines *bl* and *blb*); its output behavior updates the read buffer to produce the value 1 if and only if a current has been detected on its input.

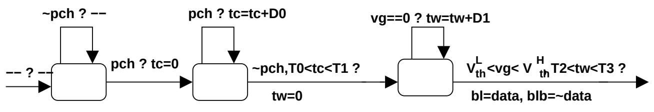

Fig. 2. A 3-state machine representing a simplified behavioral model for bitcell read operation. The notation "$x?y$" for a transition indicates that the transition has guard x produces a change of state variable y. The notation "$--?y$" represents that the transition is always enabled; the notation "$x?--$" represents that the transition produces no change in any state variable. Here t_c and t_w are auxiliary state variables representing timing; D_0, T_0, T_1, T_2, and T_3 are parameters representing timing constraints (cf. Section **3**-D). The output behavior is given by the state variables changed in the output transition (*e.g.*, *bl* and *blb*).

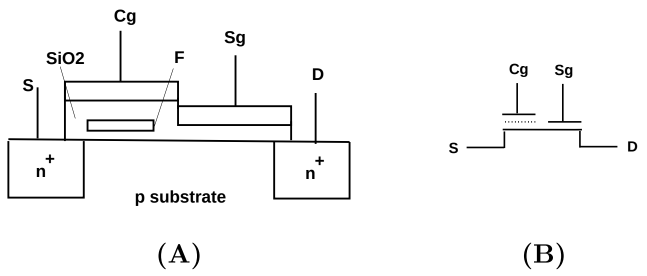

Fig. 3. (A) The structure of an SG Transistor. (B) Schematic used to represent an SG transistor.

The state transitions for *program* and *erase* operations are more involved, but similar in spirit. These operations additionally require formalization of uncertainties. Recall that programming might involve several unsuccessful iterations of electron injection. To model this nondeterminism, the behavioral model for *program* operation includes an additional "oracle stimulus", which determines if the current iteration succeeds. If the oracle stipulates the iteration to be unsuccessful, then the model constrains the threshold voltage V_{th} to be lower than V_{th}^H; the verification phase detects this failure and iterates until programming is successful or the iteration upper-bound is reached. Nondeterminism in the *erase* operation is modeled analogously. Modeling nondeterminism with oracle is standard in formal verification of reactive systems [10], [11]; since behavioral models are formalized as state machines, we can apply the same technique to memory modeling.

B. Split Gate Bitcells

Split gate or SG transistors (Fig. 3) represent the cutting edge of the flash design technology [12]. The gate (G) is "split" into *control gate* (C_g) and *select gate* (S_g). This facilitates power reduction over FG bitcell as follows: Channel-Hot Electron Injection in FG relies on electron drift for charge injection; the additional control in SG provided by the split gate permits creation of an external electric field to *force* electron migration, increasing injection efficiency and lowering programming current. Furthermore, the read voltage is lower.

SG operations are performed as follows. The bitcell is initially in *idle* state, with S_g and S actively pulled low and C_g driven to a bias voltage V_r required for reading.

Read: When the read address is available in the word line, S_g is driven high and the bit line is biased to read level. The sense amplifier compares the bitcell current I_c under V_r against a known, generated reference current I_r to determine the conduction state of the bitcell. If $I_c > I_r$ then the bitcell has value 1, otherwise it has value 0. The bitcell is then returned to the *idle* state.

Program: Once the write address is available, C_g is driven to *program mode value* V_p. Then S_g is driven high and the bit line is selected. Depending on the data value, a program current pulse I_p, provided by a data input buffer, is applied to the bit line for microseconds. The current changes the bias on the bit line, which is consequently actively pulled low. I_p causes charge injection into the channel of the selected bitcell, where it tunnels onto the floating gate. Once programming is complete, the bias current is removed and the bit line returns to the "pre-current" bias value. The bitcell is then returned to an *idle* state, and a verification is performed through a read operation to confirm whether programming was successful. If the operation was unsuccessful, the process is iterated until programming succeeds or a specified upper bound is reached.

Erase: As with the FG bitcell, erasing an SG bitcell is also a sector-based operation where the sector to be erased is determined by the applied address. Once an address is available, a voltage V_e (called the *erase mode value*), is applied to the C_g of the bitcells in the selected sector for a duration of milliseconds, keeping all the other terminals low. Consequently, charge is removed from the floating gate, turning the cells conductive. Once erase is complete, C_g is pulled to V_r, ending the erase operation and returning the cells to the idle state. The process is then followed by a read to confirm that the erase operation was successful. If the erase was unsuccessful, the process is iterated until erase succeeds or a specified upper bound is reached.

Electrical justifications of SG operations are more complex than FG, since they involve manipulating charge through two independent terminals C_g and S_g. However, the justifications are irrelevant for behavioral models, and the *effect* of each operation can be formalized as a state machine (albeit more elaborate than FG). Consider an iteration of the *program* operation representing successful programming. The state machine for this iteration simply specifies correct programming under the following sequence of input stimuli: (1) driving of C_g to V_p, (2) driving of S_g to high, (3) selection of the bit line, and (4) application of current pulse I_p; the behavior is unspecified otherwise. As with FG bitcell, uncertainty in program and erase operations is modeled by an oracle stimulus.

C. Modeling Full Memory Cores

A memory core is constructed by connecting individual bitcell-level components in a specified configuration. Fig. 4 shows the interconnection for the NOR configuration. The interconnection includes both analog and digital components.

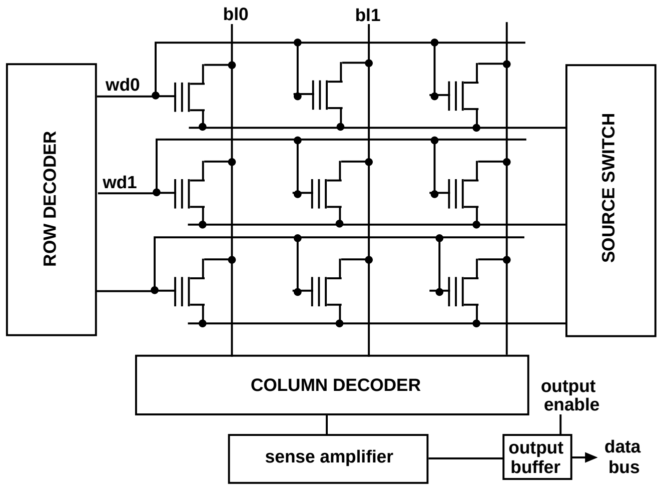

Fig. 4. Implementation of a NOR Flash Configuration with FG bitcells.

The behavior of an interconnection of state machines is formalized as an *interactive composition.* Suppose the interconnection specifies that the component $\mathcal{C}$ receives stimuli from $\mathcal{E}_1, \ldots \mathcal{E}_k$. Then the model of the interconnection is the composition in which the output behaviors of models of $\mathcal{E}_1, \ldots \mathcal{E}_k$ are composed with the input behavior of $\mathcal{C}$.

Interactive compositions are common in formal models of sequential digital designs [13], [14], [15]; extending them to memories requires state machine models of (1) analog components, and (2) combinational digital components (*e.g.*, decoder). Behavioral abstractions formalize analog components; a state machine for a combinational digital component is derived by augmenting its functional model with timing constraints characterizing the delay units for the input to propagate to output. Note that a formal model of flash memory must include the timing constraints of digital blocks, since correct bitcell functionality depends on these components producing input stimuli within specified time intervals. For instance, SG programming requires current pulse I_p for charge injection; correct programming thus depends on the data input buffer and associated logic to guarantee adequate pulse duration.

The approach to modeling a flash core as an interactive composition of state machines is completely general; once the behaviors of the individual components are formalized, construction of this model reduces to a mechanical process of comprehending the interconnection, independent of the behavioral characteristics of the components themselves. Indeed, the models of memory cores that we analyze are automatically generated by a tool, which takes (1) a pre-computed library of state machines modeling individual components, and (2) a formal description of interconnection. Consider the two flash configurations shown in Fig. 5. In spite of the fact that the configurations are implemented with different bitcell types, the same tool generates the formal abstractions of both cores and the NOR flash in Fig. 4. Since the configurations in Fig. 4 and 5(A) use the same bitcell type, the tool uses the same library of behav-

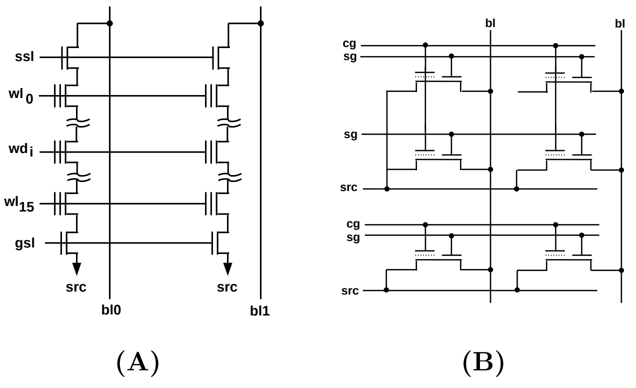

Fig. 5. (A) Implementation of a NAND Configuration with FG bitcells. (B) Implementation of a NOR Configuration with SG bitcells

ioral components; only the description of interconnection changes. Since the configuration of Fig. 5(B) uses of a different bitcell type, the tool uses a different component library.

D. Parameterization

We have developed a library of behavioral models for flash components. Note however, that different flash implementations (even with the same bitcell type and configuration) use different operating conditions. For instance, it is common for implementations of the NOR configuration in Fig. 5(B) to use different values of I_p, V_p, V_e, etc., to optimize for different design metrics. To facilitate reuse of verification collateral, our models are formalized as *parameterized* state machines with constrained predicates specifying the operating conditions for different parameters. The rationale is that the correctness (rather than efficiency) of the design depends typically on the *relation* between parameters rather than the concrete values. For instance, one (timing) constraint for the program operation of SG bitcell is that the application duration of I_p must lie within the application duration of high voltage at S_g; this is formalized by modeling the two durations as constrained functions specified to satisfy the interval containment requirement. The bitcell models are parameterized with respect to (1) timing constraints, (2) transistor threshold voltage, (3) bias current value, (4) operating voltage etc; at the level of the entire memory, we parameterize the core size. In addition to reuse of verification results, parameterization facilitates focus on the design requirements that are relevant to functional correctness while abstracting others.

IV. Verification

Given a state machine representing the behavior of a memory core, the verification goal is to relate all its executions with the high-level specification. The specification is an abstract state machine derived from the interface of the core used for functional verification of the surrounding SoC blocks: it supports *read*, *program*, and *erase*, together with *core enable* that controls operations on the entire core, *write protect* that regulates programming bitcells, etc.

The key component of the verification is the notion of correspondence used to relate the implementation and the specification. Our notion is based on *stuttering trace containment* [16], [17] up to a refinement map. A refinement map enables us to appropriately view implementation states as specification states [18]: for memories, it projects bitcell states in the behavioral model to an association list that maps addresses to data values. We use stuttering to formalize the fact that several transitions of the implementation may correspond to one transition of the specification. The verification goal then is to prove the following theorem.

Theorem 1: Let *impl* be the transition function[1] for the state machine representing the implementation of a flash memory, and let *spec* be the state transition function for the state state machine representing the interface of the memory to the surrounding SoC block. Let *rep* be a refinement map from the visible observations of the states of *impl* to those of *spec*. Then for every execution of *impl*, there exists an execution of *spec* that has the same observable behavior (defined by *rep*) up to stuttering.

To establish Theorem 1, it is sufficient to exhibit functions *inv*, *commit*, and *pick*, such that (i) *inv* is an implementation invariant and (ii) the following formulas are provable [2]:

1. $\forall s, i : \mathit{inv}(s) \wedge \neg \mathit{commit}(s, i) \Rightarrow$
   $\mathit{rep}(\mathit{impl}(s, i)) = \mathit{rep}(s)$
2. $\forall s, i : \mathit{inv}(s) \wedge \mathit{commit}(s, i) \Rightarrow$
   $\mathit{rep}(\mathit{impl}(s, i)) = \mathit{spec}(\mathit{rep}(s), \mathit{pick}(s, i))$

Fig. 6 pictorially depicts how the above conditions guarantee execution correspondence specified by Theorem 1. Here *commit* stipulates whether the specification matches an implementation transition or stutters; *pick* provides the specification stimulus for a matching transition. The formulas state that for each transition of the implementation, the specification either has a matching transition or stutters. Note that the notion of stuttering trace containment itself is very general, and can be used to compare executions of two reactive systems at different levels of abstraction. The rules are adapted from Manolios' rules for stuttering simulation [11] by restricting stuttering to be one-sided: the restriction is viable since one step of the specification corresponds to several steps of the implementation, but not vice versa. A key advantage of the rules is that they only involve single steps of the two systems, thus obviating expensive fixed-point computations. Additionally, we verify finiteness of stuttering by exhibiting a well-founded ranking function that decreases along stuttering steps. This proof is trivial since timing constraints upper-bound the completion of each operation by the number of delay units in one cycle of the system clock(s) driving memory operations.

[1]As is conventional in ACL2, we model the state transition as a function instead of a relation. Non-determinism is modeled as environmental stimuli. Thus *impl* and *spec* are binary functions that take a state and an environmental stimulus and produce the state after one transition.

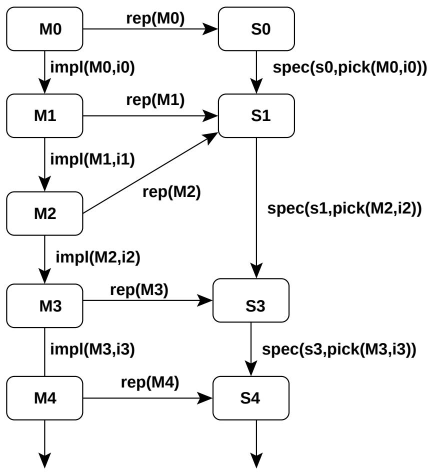

Fig. 6. Pictorial representation of conditions guaranteeing stuttering trace containment. Here $M_0, M, \ldots$ represent the states of the implementation and $S_0, S_1, \ldots$ represent states of the specification. The vertical arrows represent transitions of the corresponding state machines and horizontal arrows represent the refinement map specified by the function *rep*. Here we assume $\neg \mathit{commit}(M_1, i_1)$; thus, the transition $\mathit{impl}(M_1, i_1)$ represents a stuttering step.

A. Verifying Parameterized Models

Each flash implementation is a composition of a number of parameterized state machines. Verification of arbitrary parameterized state machines is intractable in general [19]; however, the special structure of behavioral models ameliorates verification complexity in the following manner. Observe that the following definitions of *commit* and *pick* work for any memory configuration:
1. *commit* holds only for transitions that *complete* a specific operation (*e.g.*, read, program, or erase).
2. *pick* selects the type of operation being completed and the associated addresses for the operation.
Verification thus reduces to the definition and proof of the invariant *inv*. To mechanize the latter process, recall that by construction, a behavioral component operates correctly under the operating conditions (guards) associated with its transitions. Thus it is sufficient to prove as invariant the requirement that each component always operates within the constraints specified by its guard. More precisely, let $C_1, \ldots, C_k$ be the behavioral models for the individual components, and let $G_m(s)$ be the guard for model C_m in state s, $(1 \le m \le k)$. Then we define *inv* as follows.

$$\mathit{inv}(s) \triangleq \bigwedge_{m=1}^{k} G_m(s)$$

The predicate stipulates that each component always operates within its operating condition. We decompose the proof of invariance of *inv* into the following proof obligation for each component m, $(1 \le m \le k)$, which states that the guards for the components at state s together guarantee the guard for C_m one transition from s.

$$\forall s, i : \mathit{inv}(s) \Rightarrow G_m(\mathit{impl}(s, i))$$

Verification thus reduces to an assume-guarantee reasoning over components, *i.e.*, proving that the guard for C_m is satisfied by the stimuli generated by the components surrounding C_m. Note that the obligations together imply that the following formula is a theorem, which justifies that *inv* is an inductive invariant (*i.e.*, holds along each transition of the implementation).[2]

$$\forall s, i : \mathit{inv}(s) \Rightarrow \mathit{inv}(\mathit{impl}(s, i))$$

TABLE I
Statistics on Parameterized Flash Verification

Flash type	Components	Lemmas needed	Proof time (secs)
FG NOR	33	120	248.05
FG NAND	33	143	223.04
SG NOR	67	275	435.23

B. Concretization

Given the generic proof of correctness of parameterized configurations, verification of a flash implementation based on these configurations reduces to mechanically instantiating the generic proof. In particular, it is sufficient to check *by execution* that the concrete instance satisfies the parameterized constraints: no further reasoning is necessary. In addition to facilitating reuse of verification collateral, the generic model thus provides a "template" for flash implementations. Note that *any* implementation is functionally correct as long as the constraints are satisfied. Thus the designer can freely fine-tune the parameters for different design metrics within the bounds specified by the constraints without affecting correctness. The approach can also identify subtle design errors that are difficult to detect by other means (cf. Section **5**).

V. Results

We created parameterized models of FG (NOR and NAND) designs, and SG (NOR) designs, derived generic proofs of correctness of each parameterized model using the ACL2 theorem prover, and used these models to verify a number of concrete flash implementations. Both generic and concrete proofs were done on a workstation with 2GHz Intel Pentium processor with 3GB of memory. The models incorporate transistor-level complexities of industrial designs, *e.g.*, the SG designs are taken directly from an industrial flash implementation.

Table I gives the statistics of the verification effort for parameterized models. For each flash type, we list the number of behavioral components involved, the number of lemmas needed to complete the proofs with ACL2, and the time required by ACL2 to process the proof. The FG NOR and NAND flash designs make use of the same set of (33) behavioral models for the individual components, although the interconnections (and hence the proofs) are different. The number of lemmas include only those that

[2]Additionally we prove that *inv* is true for initial states. This is trivial by execution.

had to be crafted for these verifications and do not include pre-existing ACL2 libraries. Most of the lemmas involved arithmetic theorems to discharge specifying timing constraints. Some of the lemmas designed during the verification of FG flash were reused for SG. Notice that the proofs take only a few minutes to replay. This is in stark contrast with traditional equivalence checking methods [5], [7] that can take hours for verifying industrial memories. The reason is that the models are parameterized, and the verification makes use of deduction rather than state exploration, ameliorating state explosion. However, a downside of the deductive approach is that some of the lemmas can take significant manual effort to craft. On the other hand, the effort is amortized by reuse of the proof over a number of concrete verification runs, as explained below.

Table II gives the verification statistics for the concrete flash implementations verified. All the times reported are in seconds. For each memory size we verified three different implementations. For each implementation, verification involves functional instantiation of the generic proof above; this entails checking that the concrete values of the design parameters satisfy all the constraints in the corresponding generic models. The number of constraints, and hence the verification time, increases with memory size. However, Table II indicates that the increase is linear. Note that in contrast, model checking time for traditional SRAM memories typically increases exponentially with memory size. Verification times for SG implementations are higher than FG implementations because of the more complex bitcell structure in SG, which induces more complex constraints. However, the verification times are still small compared to typical equivalence checking times for standard memory verification, *e.g.*, the verification of a 2MB SG memory completes in less than 35 minutes. This is possible since the generic proof discharges the verification of the parameterized implementation once and for all, reducing the verification of concrete implementation to the much more tractable problem of checking constraints on concrete parameters. Note that the the instantiation can be non-trivial since the constraints include arbitrary first-order predicates over the design states. However, as mentioned in Section **4**-B, no deduction (and hence manual effort) is necessary; the instantiated constraints can be discharged by execution.

One key strength of our approach is the ability to identify subtle design errors that are difficult to detect by other means. As an example, we inserted a bug in an SG design, where the magnitude of the current I_p provided by the data input buffer was insufficient for effective charge injection. The bug was immediately detected while attempting to instantiate the generic SG proof with this design: the input constraints on bitcell operation were not satisfied by the stimulus from the data input buffer. Note that the bug is at the *interface* of two components. Since SPICE simulation targets individual components, such bugs typically go undetected [7], [8].

VI. Related Work

Despite its importance, we are aware of no effort other than ours on formal functional verification of flash memories. Formalization of transistor circuits has chiefly focused on developing switch-level analyzers such as SLS [20], MOSSIM-II [1], and ANAMOS [5]. This enabled verification of regular CMOS-based digital circuits [5], [7].

Our approach is inspired by work on abstracting SRAM memories through parameterized regular expressions [8]; this work suggested the key insight of formalizing a memory as a composition of well-defined components rather than an arbitrary graph of transistors. However, the regular expressions did not correspond directly to SPICE simulation; it is also tricky to write succinct regular expressions for flash designs. Modeling components as partial state machines with expressive constraints overcomes both these deficiencies.

There has been recent interest in formal verification of analog designs [21], although we are not aware of any approach at abstracting analog circuits through parameterized behavioral models. There has been recent applications of equivalence checking on analog designs by digitization of infinite analog state space [22], [23], [24], and model checking by extending temporal logic to capture analog properties [25], [26]. Finally, the PROSYD (`prosyd.org`) project provides an assertion-based run-time monitoring tool supporting STL or PSL properties in analog circuits. This tool has been applied on simulation traces from a flash memory [27].

VII. Conclusion and Future Work

We have presented a framework for formal verification of flash memories. Our fundamental insight is that although the electrical characteristics of FG and SG transistors are complex, their behavior within the operating conditions and can be effectively formalized as state machines. Consequently, instead of abstracting a memory as a graph of transistors, as done by traditional switch-level analysis, we focus on formalizing the behavior of well-defined design components within their operating conditions. This permits us to circumvent the complex problem of effectively abstracting inherently analog characteristics with equations in a discrete algebra. To our knowledge, our approach provides the first platform for formal verification of industrial flash designs. A key feature of our approach is the parameterization of the state machines and operating constraints; this permits our formalization to target the design facets that are actually relevant to functional correctness, and also facilitates reuse of verification collateral over a range of concrete implementations. Furthermore, since memory components are modeled as state machines, traditional simulation and verification tool flows can be easily adapted to handle these models. Thus, functional verification of digital components can be hierarchically composed with flash models for full SoC verification. Finally, the direct correspondence between components used for analog simulation and the behavioral models facilitates corroboration of models with readily available SPICE simulation data. Note,

TABLE II
Verification Statistics for Concrete Flash Designs

	128K			256K			512K			1M			2M		
FG NOR	67.16	75.04	73.56	120.54	123.41	131.24	200.31	209.32	260.32	382.32	412.34	435.51	576.19	588.45	600.17
FG NAND	64.32	69.17	81.23	145.43	151.28	160.22	202.19	231.18	232.41	345.53	372.21	386.19	548.28	534.25	580.29
SG NOR	145.31	156.42	163.41	312.23	327.15	348.39	663.24	679.26	682.11	1129.15	1157.08	1192.43	1823.17	1912.06	1924.18

however, that the approach can only be applied to designs constructed by interconnection of components with well-defined behavioral characteristics; in particular, we cannot abstract an arbitrary transistor netlist.

In future work, we plan to extend the approach to verification of multi-level flash, as well as other non-volatile memories such as MRAM and FeRAM [12]. Since behavioral models are agnostic to transistor type, such extension merely requires developing bitcell-level behavioral models of these structures. We are also contemplating the use of learning techniques to *synthesize* such behavioral models from SPICE simulation data. There has been progress on learning techniques to estimate state machines [28], which may be applied for such synthesis.

Acknowledgements

Support for this work was provided in part by the Semiconductor Research Corporation under Grant 08-TJ-1849. Additionally, Sandip Ray is supported by the National Science Foundation under Grants CCF-0916772 and CNS-0910913.

References

[1] R. E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems," *IEEE Trans. on Computers*, vol. C-33, no. 2, pp. 160–177, Feb. 1984.

[2] S. Ray and J. Bhadra, "A Mechanized Refinement Framework for Analysis of Custom Memories," in *Proceedings of the 7th International Conference on Formal Methods in Computer-Aided Design (FMCAD 2007)*, J. Baumgartner and M. Sheeran, Eds. Austin, TX: IEEE Computer Society, Nov. 2007, pp. 239–242.

[3] ——, "Abstracting and Verifying Flash Memories," in *Proceedings of the 9th Non-Volatile Memory Technology Symposium (NVMTS 2008)*, K. Campbell, Ed. Pacific Grove, CA: IEEE, Nov. 2008, pp. 100–104.

[4] M. Kaufmann, P. Manolios, and J. S. Moore, *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers, 2000.

[5] R. E. Bryant, D. Beatty, K. Brace, K. Cho, and T. Sheffler, "COSMOS: A Compiled Simulator for MOS Circuits," in *Proceedings of 24th Design Automation Conference*. ACM/IEEE, 1987, pp. 9–16.

[6] P. Agrawal, "Automatic Modeling of Switch-Level Networks Using Partial Orders," *IEEE Transactions on Computer-Aided Design*, vol. 9, no. 7, pp. 696–707, 1990.

[7] N. Krishnamurthy, A. K. Martin, M. S. Abadir, and J. A. Abraham, "Validating PowerPC$^{\mathrm{TM}}$ Microprocessor Custom Memories," *IEEE Design & Test of Computers*, vol. 17, no. 4, pp. 61–76, 2000.

[8] J. Bhadra, A. K. Martin, and J. A. Abraham, "A Formal Framework for Verification of Embedded Custom Memories of the Motorola MPC7450 Microprocessor," *Formal Methods in Systems Design*, vol. 27, no. 1-2, pp. 67–112, 2005.

[9] P. Cappalletti, C. Golla, P. Olivo, and E. Zanoni, Eds., *Flash Memories*. Kluwer Academic Publishers, 1999.

[10] J. Sawada and W. A. Hunt, Jr., "Verification of FM9801: An Out-of-Order Microprocessor Model with Speculative Execution, Exceptions, and Program-Modifying Capability," *Formal Methods in Systems Design*, vol. 20, no. 2, pp. 187–222, 2002.

[11] P. Manolios, "Mechanical Verification of Reactive Systems," Ph.D. dissertation, The University of Texas at Austin, 2001.

[12] E. J. Prinz, "The Zen of Nonvolatile Memories," in *Proceedings of the 43rd Design Automation Conference (DAC 2006)*. San Francisco, CA: ACM, 2006, pp. 815–820.

[13] M. J. C. Gordon, "The Semantic Challenges of Verilog HDL," in *10th Annual IEEE Symposium on Logic in Computer Science (LICS 1995)*, 1995, pp. 136–145.

[14] D. Russinoff, "A Formalization of a Subset of VHDL in the Boyer-Moore Logic," *Formal Methods in Systems Design*, vol. 7, no. 1/2, pp. 7–25, 1995.

[15] B. Brock and W. A. Hunt, Jr., "The Dual-Eval Hardware Description Language," *Formal Methods in Systems Design*, vol. 11, no. 1, pp. 71–104, 1997.

[16] R. Milner, *Communication and Concurrency*. Prentice-Hall, 1990.

[17] D. Park, "Concurrency and Automata on Infinite Sequences," in *Proceedings of the 5th GI-Conference on Theoretical Computer Science*, ser. LNCS, vol. 104. Springer-Verlag, 1981, pp. 167–183.

[18] M. Abadi and L. Lamport, "The Existence of Refinement Mappings," *Theoretical Computer Science*, vol. 82, no. 2, pp. 253–284, May 1991.

[19] K. R. Apt and D. Kozen, "Limits for Automatic Verification of Finite-state Concurrent Systems," *Information Processing Letters*, vol. 15, pp. 307–307, 1986.

[20] Z. Barzilai, D. K. Beece, L. M. Hiusman, V. S. Iyegar, and G. M. Silberman, "SLS – a Fast Switch Level Simulator for Verification and Fault Coverage Analysis," in *Proceedings of 23rd Design Automation Conference*, 1986, pp. 164–170.

[21] M. Zaki, S. Tahar, and G. Bois, "Formal Verification of analog and mixed signal designs: A survey," *Microelectronics Journal*, vol. 39, pp. 1395–1404, Dec. 2008.

[22] A. Balivada, Y. Hoskote, and J. Abraham, "Verification of transient response of linear analog circuits," in *Proceedings of IEEE VLSI Test Symposium*, 1995, pp. 42–47.

[23] L. Hedrich and E. Barke, "A formal approach to nonlinear analog circuit verification," in *ICCAD*, 1995, pp. 123–127.

[24] A. Salem, "Semi-formal verification of VHDL-AMS descriptions," in *Intl. Symp. on Circuits and Systems*, 2002, pp. 123–127.

[25] R. Kurshan and K. McMillan, "Analysis of digital circuits through symbolic reduction," *IEEE Transactions on Computer-Aided Design*, vol. 10, no. 11, pp. 1350–1371, Nov. 1991.

[26] S. Little, N. Seegmiller, D. Walter, C. Myers, and T. Yoneda, "Verification of Analog and Mixed-signal Circuits Using Timed hybrid Petri Nets," in *Automated Technology for Verification and Analysis*, ser. LNCS, no. 3299, 2004, pp. 426–440.

[27] D. Nickovic, O. Maler, A. Fedeli, P. Daglio, and D. Lena, "Analog Case Study, PROSYD Deliverable D3.4/2," Jan. 2007.

[28] A. Gupta and E. M. Clarke, "Reconsidering CEGAR: Learning Good Abstractions without Refinement," in *Proceedings of 23rd Inernational Conference on Computer Design (ICCD 2005)*, 2005, pp. 591–598.