SAND2011-6508C

Complete Formal Verification of Stateful Designs for High Consequence Systems: A Case Study

Kevin Hulin
Jonsson School of Engineering and Computer Science
The University of Texas at Dallas
Richardson, Texas
USA
kjh061000@utdallas.edu

Yalin Hu
Sandia National Laboratories
Livermore, California
USA
yhu@sandia.gov

Abstract. The term formal verification refers to the application of mathematical methods to determining the validity of a system implementation with respect to a set of specifications. Formal verification has been used widely in digital system designs as a way to effectively ensure functional correctness. There exist two well-studied approaches to formal verification: Model Checking (MC) and Automated Theorem Proving (ATP). Of these approaches, ATP is more powerful – able to handle arbitrarily large and complex systems; however, it is very rare that a theorem prover proves a theorem's validity without human intervention. MC, on the other hand, requires no human intervention but is limited in ability to very simple systems due to the state explosion problem. The state explosion problem is especially dominant in stateful systems such as memory where the very large state space makes MC infeasible.

In our work, we explore the trade-offs between ATP and MC to determine the proper balance in a hybrid approach that maximizes automation. We show that neither can accomplish the verification alone, and propose a novel decomposition method to drastically reduce the computational complexity for verifying a subset of stateful systems. We demonstrate the utility of our approach in formally verifying a random access memory (RAM) implementation and discuss how our technique could be applied to other stateful systems.

Keywords – formal verification; automated theorem proving; model checking; BDD; random access memory

I. INTRODUCTION

In certain high consequence systems, the requirement that safety and liveness properties are upheld is of paramount importance. The most common method for determining whether a system implementation is working to spec is simulation based validation. A large set of test inputs and expected outputs must be created in an attempt to cover all runtime paths that the system may exhibit. The system is then run on the given inputs and checked. Correct operation during such validation is then used to claim that the system works correctly and is ready to be put into production. However, such tests cannot be exhaustive, and the reliance upon simulation based validation for producing high consequence systems is known to be a costly mistake [5][7].

Formal verification aims to eliminate these consequences by offering mathematically and logically sound techniques for determining whether a design implementation is specification adherent. Since its inception, two approaches have taken hold as viable techniques for formal verification: Model Checking (MC) and Automated Theorem Proving (ATP). MC is the exhaustive examination of a system's reachable states that ensures desired properties hold. ATP is the logical derivation of desired properties from a mathematical definition of the system implementation and a collection of axioms. Each technique has its own strengths and weaknesses (as shown in Table I) and neither can really be considered a cure-all for the formal verification problem.

TABLE I. AUTOMATED THEOREM PROVING VS MODEL CHECKING

	Automated Theorem Proving	**Model Checking**
Strengths	• Ability to handle very complex systems • Expressive logic • Generation of machine checkable proof	• Easy generation of model from HDL source • Automatic verification • Generation of counter examples
Weaknesses	• Requires human input • No counter example • Not automated	• Design size limitation • Not feasible for complex data path

A key observation made when comparing ATP and MC is that one's weakness is the other's strength. Where ATP is unable to perform without human intervention, MC requires no human oversight; and where MC cannot handle complex systems, ATP is not limited by the system's complexity. Since these two techniques are complementary, an obvious solution would be to combine the best features of each and create a completely automatable verifier that is not limited to simple systems.

Much work has been done to this end, and while progress has been made to combine the techniques of ATP and MC into a hybrid verification tool, success has been limited to problems that are not easily generalized.

$$S = \{(I,A,T,O,Y) | I \in \mathbb{Z}_{2^{64}}, A \in \mathbb{Z}, T \in \{READ, WRITE\}, O \in \mathbb{Z}_{2^{64}}, Y \in \mathbb{Z}_M^N\}$$
$$S_0 = (0,0,\bot,0,0^N)$$
$$R \subseteq S \times S = \{(S,S') \mid T_S = READ \ \wedge read(S,S') \vee T_S = WRITE \wedge write(S,S')\}$$
$$L: S \rightarrow 2^{AP} = \ L\big((I,A,T,O,Y)\big) = \{input = I, address = A, control = T, output = O\} \cup \{memory_i = Y_i \ | i \in \mathbb{I}_N\}$$

$$read\big((I,A,T,O,Y),(I',A',T',O',Y')\big) = \left(\bigwedge_{i \in \mathbb{I}_N} (Y_i' = Y_i)\right) \wedge (A \in \mathbb{Z}_N \rightarrow O' = Y_A) \wedge (A \notin \mathbb{Z}_N \rightarrow O' = 0)$$
$$write\big((I,A,T,O,Y),(I',A',T',O',Y')\big) = \left(\bigwedge_{i \in \mathbb{I}_N} (i \neq A \rightarrow Y_i' = Y_i) \wedge \big(i = A \rightarrow Y_i' = mask_N(I)\big)\right) \wedge (O' = O)$$
$mask_N(X) = X \ \& \ (2^N - 1)$, Where "&" is the bitwise "AND" operation.

Figure 1 – Formal Definition for RAM Kripke Structure

Very early work in combing ATP and MC attempted to partition a system into properties that are control intensive (to be used with MC) and data intensive (to be used with ATP) [11]. The limitation of such an approach is that most systems, especially high consequence systems, have very complex interactions between these two categories, making partitioning infeasible or very hard. Some alternative approaches emerged to supplement model checking with proof assistants that aim to decompose a complete verification into several model-checkable subtasks [9]. Examples of decomposition rules include temporal splitting, data abstraction, and compositional verification. Among the listed, abstraction is a commonly used technique that can reduce the verification of a complete system to the verification of an abstract system.

Another verification approach aims to loosely integrate MC and ATP under into deductive environment [12]. This environment provides capabilities such as modular debugging and verification through abstraction and MC. The major obstacle for a tight integration of MC and ATP is the successful abstraction across domains and discovery of good abstract representations. Our approach takes advantage of the automation of MC in combinatorial logic from NuSMV and avoids the state explosion problem by decomposing our model into small function preserving partitions. In order to maintain soundness, we employ a theorem prover (ACL2) and are consequentially able to scale up verification results to arbitrarily large models. This approach is general enough that it can be applied to other digital systems.

We choose RAM as a case study because of its wide application, especially in high consequence systems. With state-of-the-art semiconductor process technology, memory design and verification has drawn a lot of attention in both analog and digital aspects [8]. Verification of memory has been an important and challenging problem. Memory is unique because (1) there are normally a very large number of cells; (2) each of these cells has identical functionality and controlled by the same control signals; and 3) there are generally many structural symmetries in RAM architectures.

Verification of memory started with switch-level simulation [3], which works very well for small sized memories. Later on different techniques have been published, such as symbolic trajectory evaluation (STE) of memory arrays [10] and bounded model checking of embedded memories [6]. The STE based verification is essentially a form of symbolic simulation and is able to overcome the infeasible simulation coverage issue by reducing the system model – taking advantage of the structural symmetry of RAM. Bounded model checking (BMC) made the handling of large embedded memory designs feasible through an effective abstract model [6]. In this approach, each memory bit is abstracted and constraints are added at every analysis step. However, because BMC is employed, soundness is not guaranteed for general systems.

In this paper, we present a novel framework for verification of stateful hardware systems and employ its utility in verifying a RAM design, emphasizing the importance of automation and soundness.

In section II, we formally define our RAM model as a Kripke structure and demonstrate the pitfalls of straight forward MC. In section III we go on to present our decomposition approach and prove its soundness. We go on to present our results in section IV and conclude with suggested future research in section V.

II. Random Access Memory

Before we can discuss the formal verification of a RAM system, we must first attempt to formally define our system in a way that is easy to understand.

A. Definitions

We describe a Kripke structure that reflects the semantics of a generic RAM implementation. Figure 1 formally describes the finite state ω-automaton that we used in model checking. We define a state as a 5-tuple (I,A,T,O,Y) where I represents the input value, A the input address, T the Read/Write control bit, O the output value, and Y an ordered N-length list of M-bit values representing the values being stored within the RAM.

We define the transition relation between states by the Boolean relations $read$ and $write$.

Read. *For the* $read$ *relation, two states* S *and* S' *are related if the following three statements hold:*

$$\boldsymbol{AG}\left(\left((mem[a] \,\&\, (1 \ll b)) > 0\right) \land (status \neq WRITE \mid addr \neq a)\right) \rightarrow \boldsymbol{AX}(mem[a] \& (1 \ll b) > 0)$$

$$\boldsymbol{AG}\left(\left((mem[a] \,\&\, (1 \ll b)) = 0\right) \land (status \neq WRITE \mid addr \neq a)\right) \rightarrow \boldsymbol{AX}(mem[a] \& (1 \ll b) = 0)$$

$$\boldsymbol{AG}\,(input \,\&\, (1 \ll b) > 0 \land status = WRITE \,\&\, addr = a) \rightarrow \boldsymbol{AX}\,(mem[a] \,\&\, (1 \ll b) > 0)$$

$$\boldsymbol{AG}\,(input \,\&\, (1 \ll b) = 0 \land status = WRITE \,\&\, addr = a) \rightarrow \boldsymbol{AX}\,(mem[a] \,\&\, (1 \ll b) = 0)$$

$$a \in \mathbb{Z}_N, b \in \mathbb{Z}_M$$

Figure 2 – Optimized liveness specification for memory writes

1. $\forall_{i \in \mathbb{I}_N} Y_i' = Y_i$
2. $A \in \mathbb{Z}_N \rightarrow O' = Y_A$
3. $A \notin \mathbb{Z}_n \rightarrow O' = 0$

Semantically, the $read$ relation ensures that when a state transition is initiated by a read operation, the next state must (1) maintain the integrity of values being stored and should update the output value to either (2) reflect the value being stored in memory if the address is valid or (3) to 0 if the address line is not valid.

1) Write. For the $write$ relation to hold, two states S and S' must satisfy the following expressions:

1. $\forall_{i \in \mathbb{I}_N}(i \neq A \rightarrow Y_i' = Y_i)$
2. $\forall_{i \in \mathbb{I}_N}(i = A \rightarrow Y_i' = mask_N(I))$
3. $O' = O$

These rules ensure that when a state transition is initiated by a write operation, the next state should (1) maintain integrity of values Y_i where $A \neq i$, (2) update the value Y_i where $A = i$ to $mask_N(I)$, and (3) ensure that the output value does not change. Notice that these rules were crafted in order to preserve the safety of the system. That is, the $mask$ function ensures that only values of the proper bit-width are stored in memory, and the update step implicitly ensures that writes to illegal addresses do not corrupt the memory content.

B. Specifications

We next describe the formal specifications that we want the model checker to verify. Here, we simply present the naive specifications expressed in Computation Tree Logic (CTL). Optimizations for reducing complexity will be presented in the next section. We enforce two liveness and one safety properties:

1) Liveness. The first property that we check is whether our implementation correctly implements the read operation. We define the property *Read Liveness* (RL) to be: *If the status bit = $READ$ and the address = A, then in the next state, the output should be $Y[A]$.*

$$\boldsymbol{AG}(addr = a \land status = R) \rightarrow \boldsymbol{AX}\; output = mem[a];$$
$$a \in \mathbb{Z}_N$$

The second liveness property ensures that the write operation is correct. *Write Liveness* (WL) is defined as: *If the status bit = $WRITE$ and the address = A and the masked_input = I, then if A is a valid address, in the next state $Y[A] = I$ will be true. Furthermore, if address ≠ A' then in the next state, $Y[A']$ will equal the current value of $Y[A']$.*

$$\boldsymbol{AG}\,(addr = a \land status = WRITE \land masked\_input = i)$$
$$\rightarrow \boldsymbol{AX}\; mem[a] = j;\, a \in \mathbb{Z}_N, i \in \mathbb{Z}_{2^{64}}$$

$$\boldsymbol{AG}\,(addr \neq a \land status = WRITE \land mem[a] = k)$$
$$\rightarrow \boldsymbol{AX}\; mem[a] = k;\, a \in \mathbb{Z}_N, k \in \mathbb{Z}_{2^{64}}$$

2) Safety - The only safety property we enforce is that at all states, the values stored in memory should be members of a specified range of integers defined by the value M in our definition. In our specifications, we state Safety to be: *For each memory address A, the value stored at Y[A] should be in the range of values 0 to* $2^M - 1$.

$$\boldsymbol{AG}\,(0 \leq mem[a] \land mem[a] \leq\; 2^M - 1);\, a \in \mathbb{Z}_N$$

C. Optimized specifications

In order to obtain viable runtime results for model checking, we needed to rewrite our *Write Liveness* property to avoid $O(N \cdot 2^M)$ specifications. We accomplish this by performing a bit-level comparison across the M bits of data values. The improved specifications are shown in Figure 2. Using this optimization, we are able to cover the same properties in $O(N\; \cdot M)$ specifications.

D. Limitations

Despite our efforts to express our RAM model in a way that would make the model checking problem tractable, the fact of the matter remains that RAM is a stateful system and subject to the state explosion problem. Unable to model check RAM of size larger than 100 bytes, we began looking at other approaches to the problem. Using a theorem prover, we would be able to trivially verify our properties, however, for our solution, we required a completely automatable system, and thus direct theorem proving would not suffice.

Our next thought was to take a hybrid approach. We developed an idea for decomposing RAM into smaller pieces and model checking the pieces individually. While this seems trivial, the implications of being able to reduce an intractable problem into smaller tractable parts were very appealing. Our first step would be to formally prove that such an approach would be work.

(m,k,s)	RAM
$m \in \mathbb{Z}^n_{2^M}, n \le N$	value-list
$k \in \{0,1\}^*$	mask
$s = \|m\| \in \mathbb{Z}^+$	size
$([0,0,\dots,0], \{1\}^M, N)$	Initial RAM

$$\frac{(adr \ge 0) \wedge (adr < s)}{read\big((m,k,s),adr\big) \longrightarrow m_{adr}}\,(Memory\ Read)$$

$$\frac{(adr \ge 0) \wedge (adr < s)}{write\big((m,k,s),adr,val\big) \longrightarrow \big(m_{(0..adr-1)} :: (val\ \&\ k) :: m_{(adr+1)..(s-1)}, k, s\big)}\,(Memory\ Write)$$

$$\frac{(r > 0) \wedge (r < n)}{decompose\big((m,k,s),r\big) \longrightarrow \Big(\big(m_{0..(r-1)},k,r\big),\big(m_{r..(s-1)},k,(n-r)\big)\Big)}\,(Decomposition)$$

$$\frac{}{compose\big((m_1,k,s_1),(m_2,k,s_2)\big) \longrightarrow (m_1 :: m_2, k, s_1+s_2)}\,(Composition)$$

$$\frac{(adr \ge 0) \wedge (adr < s_1+s_2)}{read_{decomp}\big((m_1,k,s_1),(m_2,k,s_2),adr\big) \longrightarrow \Big(adr < s_1 ?\ read\big((m_1,k,s_1),adr\big) :\ read\big((m_2,k,s_2),adr-s_1\big)\Big)}\,(Decomposed\ Read)$$

$$\frac{(adr \ge 0) \wedge (adr < s_1+s_2)}{\begin{array}{c} write_{decomp}\big((m_1,k,s_1),(m_2,k,s_2),adr,val\big) \longrightarrow \\ \Big(adr < s_1 ? \Big(write\big((m_1,k,s_1),adr,val\big),(m_2,k,s_2)\Big) : \Big((m_1,k,s_1), write\big((m_2,k,s_2),adr-s_1,val\big)\Big)\Big)\end{array}}\,(Decomposed\ Write)$$

Figure 3 Syntax and operational semantics for RAM as modeled in ACL2

III. DECOMPOSITION OF RAM

In order to maintain soundness in our RAM verifier while taking advantage of a decomposition property, we had to first ensure that the decomposition step was sound and did not affect the system's validity. We used the ACL2[1] theorem prover to prove that a RAM satisfying the properties described in section IV can be divided into small elements that will also satisfy the properties. Furthermore, we proved that two smaller RAMs that satisfy the properties could be concatenated together and the resulting RAM would also satisfy the properties. Finally, we defined the mapping for $READ$ and $WRITE$ operations from the large RAM onto the decomposed pieces and prove their semantic equivalence.

A. Decomposition Proof

In ACL2, we model memory as a 3-tuple (m,k,s) where m is an ordered list of size s and k is the value mask that is applied upon memory writes.

We define the syntax and operational semantics for our model in Figure 3. In the remainder of this section, we prove property adherence for our model, the equivalence of the decomposed operations with their corresponding simple operations on the original memory, and conclude with a soundness proof for decomposed property verification. In the following theorems, let $R = (m,k,s) \in RAM, p \in \mathbb{Z}_s$.

Theorem 1 (Liveness for Read and Write). If the read operation is invoked with a valid memory address, then the resulting output should be the corresponding value located in memory. Similarly, if the write operation is invoked with a valid memory address, then the resulting memory should be identical to the original with the exception that the value at the designated memory address has been updated to reflect the input value.

Proof. The proof of this theorem is a straight-forward application of the definitions for Read and Write operations.

Theorem 2 (Safety for Read and Write). When a write operation is performed on a RAM with a valid memory address, given that the RAM is initially safe, the resulting RAM will also be safe. Here, safety is defined as in section III, namely that after every read or write operation, every value being stored should be within a specified range.

Proof. For this theorem, we perform an exhaustive proof across operations (namely read and write). For read operations, the proof is trivial since reads have no affects on the values stored in memory, as shown in the operational semantics. Writes performed, however, do affect the memory store. Let $R = (\mu,\kappa,\sigma)$ be a RAM that satisfies safety with respect to the system parameter $k = 2^M - 1$. We aim to prove the safety of $R' = write(R, adr, val)$. Consider that R' is safe iff $(val\ \&\ \kappa)$ is in the range $(0..k)$. By definition, $(val\ \&\ \kappa)$ is in the range $(0..\kappa)$. Since R is given to be safe, it follows that $(0..\kappa) \subseteq (0..k)$ and that safety is preserved.

Theorem 3 (Decomposition and Inverse). When memory is decomposed into two partitions, these partitions can each be classified as a RAM by definition. Furthermore, the composition of two RAMs sharing the same mask value into a single memory yields a RAM. Finally, the ordered composition of partitions resulting from decomposition of a RAM results in a RAM that is semantically equivalent to the original.

$$compose(R_0', R_1') = R \leftrightarrow decompose(R, p) = (R_0', R_1')$$

Theorem 4 (Decomposed Read Liveness). If a decomposed read operation is performed on two partitions of RAM, the resulting output is the same as that of the read operation performed on the parent RAM.

$$Let\ R' = decompose(R, p)$$
$$read_{decomp}(R_0', R_1', addr)$$
$$\equiv read(compose(R_0', R_1'), addr)$$
$$\equiv read(R, addr)$$

Theorem 5 (Decomposed Write Liveness). When a decomposed write operation is performed on two partitions of a parent RAM, the resulting partitions are equivalent to those resulting from the decomposition of the updated RAM.

$$Let\ R' = decompose(R, p)$$
$$write_{decomp}(R_0', R_1', adr, val)$$
$$\equiv decompose(write(R, adr, val), p)$$

Proof. Proofs of theorems 3 – 5 follow from a straightforward application of definitions from Figure 3.

Theorem 6 (Decomposition Soundness). If a RAM is decomposed into partitions, and those partitions satisfy the safety and liveness properties for RAM stated in section III.B, then the original RAM also satisfies these properties.

Proof. For decomposition soundness, we prove each property separately as its own lemma, namely Read Liveness (RL), Write Liveness (WL), and Safety. In the following lemmas, let $decompose(R, p) = (R_0', R_1'), p \in \mathbb{Z}_s$.

Lemma 1. $RL(R_0') \wedge RL(R_1') \rightarrow RL(R)$.

Proof. Let $R_0' = (\mu_0, k, p)$. By definition, $RL(R_0') \equiv \boldsymbol{AG}\ (addr = i \wedge status = R) \rightarrow \boldsymbol{AX}\ output = \mu_0[i]; i \in \mathbb{Z}_p$. Similarly, let $R_1' = (\mu_1, k, s - p)$. Again, by definition, this time taking the offset p into account $RL(R_1') = \boldsymbol{AG}\ (addr = (i - p) \wedge status = R) \rightarrow \boldsymbol{AX}\ output = \mu_2[i - p]; i \in \mathbb{Z}_s/\mathbb{Z}_p$. From here, it follows $RL(R_0') \wedge RL(R_1') \equiv \boldsymbol{AG}\ (addr = i \wedge status = R) \rightarrow \boldsymbol{AX}\ output = mem[i]; i \in \mathbb{Z}_{p+(s-p)} = \mathbb{Z}_s = RL(R)$.

Lemma 2. $WL(R_0') \wedge WL(R_1') \rightarrow WL(R)$.

Lemma 3. $Safety(R_0') \wedge Safety(R_1') \rightarrow Safety(R)$.

Proof. For Lemmas 2 and 3, the proof takes a similar form to the proof given in Lemma 1. The underlying property that allows decomposition soundness to hold is the fact that all semantics of RAM can be described in a piecewise fashion and that each property is enforced over these individual pieces.

From these soundness lemmas, we conclude that decomposition is sound with respect to our properties.

B. Implications

The utility of such a decomposition property is an obvious advantage to model checking as it allows one to convert a problem of size $O(N \cdot 2^M)$ into N problems of size $O(2^M)$. Furthermore, because the transition space of such a graph is sparse, the construction of an efficient Binary Decision Diagram (BDD) is easy, further reducing the problem's complexity into something computable on commodity hardware.

In addition to reduced complexity, the division of one problem into N problems is an obvious candidate for parallel computing, thus yielding further computational benefits.

IV. Results

In this section, we describe the run-time performance for our verification system. We begin with a runtime analysis using a model checker only. We choose NuSMV, an open source symbolic model checker as our base line for measuring performance.

A. NuSMV Model Checking

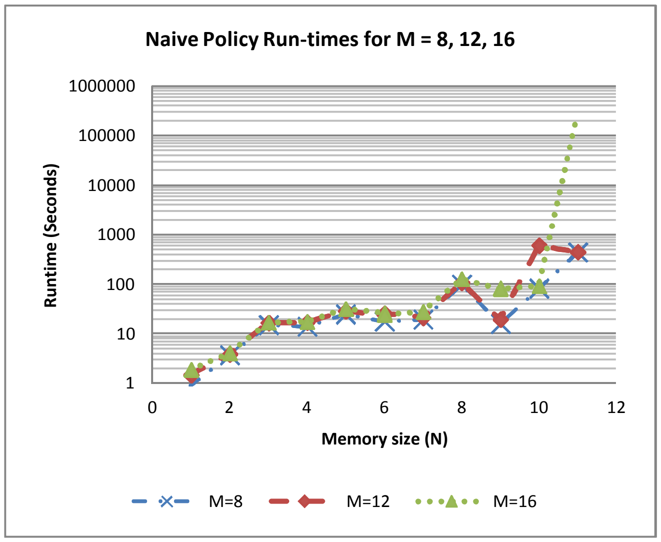
We captured NuSMV's performance for increasingly large memory size. Recall that in our model, M is the word width and N is the number of words being modeled in our RAM. We found that we could achieve the best performance when running with *coi*, *df*, and *dynamic* flags enabled (cone-of-influence, do-not-compute-reachable-set, and dynamic-variable-reordering respectively). We refer the reader to the user manual for a detailed description of how these options improve the efficiency for model checking in NuSMV [2]. All results here were obtained on a Windows 7 64-bit PC with an Intel Core i5-2500 3.3GHz CPU and 8GB of RAM.

We first look at a runtime comparison using our initial set of naïve specifications that included some Linear Temporal Logic (LTL) specifications not mentioned here (Fig. 4-a). Sampling ten runs per data point, we demonstrate the importance of efficient specifications and conclude that beyond a memory size of about 12 words, this approach would not complete in a reasonable amount of time (we terminated execution at 3 days for M=16).

B. Hybrid Verifier

We next compare runtimes across our two verification approaches – first on NuSMV with optimized specifications, and then in our hybrid approach (Fig. 4-b). The performance gain from efficient specifications is obvious, however, we again see state explosion beyond about 600 bits of RAM. We attribute this success of model checking 2^{600} states to NuSMV's efficient BDD representation for our model, but reiterate that most modern digital systems have more than 100 bytes of RAM.

Our hybrid verifier performed the best over-all. In the graph, we compute the total decomposition runtime as the time required to verify our proof in ACL2 + the time required to run N instances of decomposed RAM in NuSMV. We expect the linear growth to continue well beyond the point where simple model checking fails. Furthermore, we speculate that parallelization would yield even better runtimes, and we note that the re-verification of our machine proof is actually a one-time cost but we decided to include it for completeness.



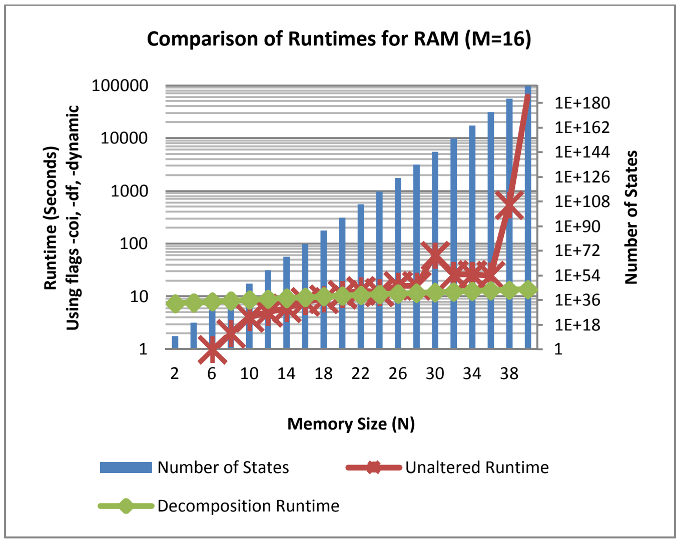


Figure 4a&b Runtime comparisons for RAM verification system.

V. Conclusion

In this paper we present a novel approach to formally verifying the subset of stateful digital systems that exhibits the decomposition property as defined here. When utilized, this property allows a verifier to model check partitions of the system individually, eliminating the unnecessary overhead of checking specifications in states that are inconsequential to the specification's validity and avoiding the state explosion problem.

We emphasize that using this framework, we maximize automation – a key feature in promoting its use in design verification. Furthermore, because we integrate the use of a theorem prover to validate our decomposition property, our verifier is sound.

In the future, we hope to include other digital systems in this class of decomposable designs and possibly build a classifier that is able to automatically determine when a state space can be partitioned without compromising soundness. Such an automated system would prove invaluable in promoting the use of formal verification for creating provably secure systems.

Acknowledgments

This work was supported by Laboratory Directed Research & Development grant from Sandia National Laboratories, US Department of Energy. SAND#XXXXXX

References

[1] ACL2 Automated Theorem Proving Tool (http://www.cs.utexas.edu/~moore/acl2/)

[2] NuSMV Model Checking Tool (http://nusmv.fbk.eu/)

[3] R.E. Bryant, "Formal verification of memory circuits by switch-level simulation", IEEE Transactions on Computer-Aided Design, vol 10, No. 1, January 1991.

[4] E.M. Clarke, O. Grumberg, and D.A. Peled, "Model checking", The MIT Press, 1999.

[5] Intel Corporation, Statistical Analysis of Floating Point Flaw, FDIV Replacement Program, November 1994.

[6] M. Ganai, A. Gupta, and P. Ashar, "Efficient modeling of embedded memories in bounded model checking", Proceedings of CAV'2004, pp 440-452, 2004.

[7] J.L. Lions, Report by the Inquiry Board, Ariane 5 Flight 501 Failure, July 1996.

[8] B. McGaughy, S. Wuensche, and KK Hung, "Advanced simulation technology and its application in memory design and verification", 2005 IEEE International Workshop on Memory Technology, Design, and Testing.

[9] K.L. McMillan, "Verification of infinite state systems by compositional model checking", *Correct Hardware Design and Verification Method,* LNCS 1703, pp 219-233, Springer Verlag 1999.

[10] M. Pandey and R.E. Bryant, "Formal verification of memory arrays using symbolic trajectory evaluation", Proceedings of International Workshop on Memory Technology, Design and Testing, pp 42-49, 1997.

[11] N. Shankar, "Combining Theorem Proving and Model Checking through Symbolic Analysis", CONCUR 2000, LNCS 1877, pp 1-16, Springer Verlag 2000.

[12] T. Uribe, "Combinations of model checking and theorem proving", FroCos 2000, LNAI 1794, pp 151-170, Springer Verlag 2000.