NFSM Generation for Semantics Based Model Abstraction [*]

Yee-Wing Hsieh

Department of Electrical Engineering
University of Pittsburgh

Steven P. Levitan

Department of Electrical Engineering
University of Pittsburgh

Abstract

We present a method for abstracting non-deterministic finite state machine (NFSM) models from behavioral VHDL descriptions for formal verification. The method is based on semantic matching of the results of data-flow analysis of the VHDL source code and the specifications to be verified, with known abstraction templates. Using NFSM models for counters, comparators and registers we have shown our approach can yield many orders of magnitude ($10^2 - 10^{11}$) reductions in state space size and substantial improvements in performance of formal verification runs.

1 Introduction

Validating a design using formal verification methods is inherently computationally intensive, and, as a result, only small designs can be verified. To handle large designs, model abstraction is necessary. Model abstraction takes a model and replaces it with a high-level description of non-deterministic finite automata (NFA) that encapsulates the behavior of the model it replaces. Using this high-level abstract representation, model abstraction reduces the number of states necessary to perform formal verification and thus reduces the state space to be explored by formal verification tools.

The traditional synthesis based technique in formal verification is to take a behavioral VHDL model, synthesize the design using this behavioral model, and then verify the resulting synthesis implementation of that design using various formal verification engines such as COSPAN [1], SMV [2] or Vis [3].

The problem with the traditional synthesis based technique is that we are validating the synthesized implementation instead of the original model of a design. Ideally, design verification should be performed to identify design errors early in the design cycle at a high level design abstraction before any costly synthesis tasks are performed. Furthermore, synthesis tasks may utilize various optimization techniques (e.g., functional unit sharing to minimize area), where registers may be allocated for temporary variables, which aggravates the state space explosion problem. More importantly, it is difficult to observe and trace the state of a variable when multiple registers may be allocated for a single variable or a single register may be allocated for multiple variables. As a result, it is difficult to perform abstraction because the behavioral semantics of the model are lost at the RTL level.

A better approach is to verify designs using an abstract state space. Our approach extracts the semantics of the behavioral model, performs abstraction based on the semantics of the behavioral model, and then verifies the resulting abstract model. In our approach, semantic extraction identifies the abstract state space for model abstraction, instead of exploring the state space of the original, unreduced model. Using this abstract state space, model abstraction analysis identifies signals or parts of the original model for abstraction.

Our Semantics-based Model Abstraction method, called SMA, consists of three model abstraction techniques: key value extraction, model partitioning through min/max data-flow analysis and data abstraction through relational algebra. Experiments have shown SMA can yield many orders of magnitude ($10^2 \rightarrow 10^{11}$) reductions in state space size and substantial improvements in performance of formal verification runs [4].

In this paper we present our method for abstracting NFSM models from behavioral VHDL descriptions for formal verification. The method is based on semantic matching of the results of data-flow analysis of the VHDL source code [5], and the specifications to be verified, with known abstraction templates. By replacing abstraction template matches with the corresponding NFSM models, the resulting abstract model has a state space size much smaller than that of the original model, but it is functionally equivalent with

[*]This research was supported, in part, by the National Science Foundation under Grant MIP-9102721

respect to the specifications to be verified.

The remainder of the paper is organized as follows. First, we briefly review previous work on model abstraction for formal verification. Next, we describe our algorithm for abstraction template matching of counters, comparators, and registers and describe our algorithm for generating corresponding NFSMs for each abstraction. Finally, we present experimental results and conclusions.

2 Related Work

Previous work in automatic model abstraction includes Kurshan's property-dependent *localization reductions* (LR) in COSPAN [6] using ω-automata and Long's process reduction via observation in SMV+ [7] using CTL.

In property-dependent *localization reductions*, the parts of design model which are irrelevant to the property being checked are automatically abstracted away [8]. However, the method works by exploring the state-space of the unreduced model and works best if the user specifies a reduction starting point.

SMV+, on the other hand, uses a state minimization procedure to obtain a reduced process that is equivalent to the original process with respect to observation via its inputs and outputs [7]. However, the minimization techniques are fairly strict in terms of the required relations between the original and reduced processes, and the user must supply an abstraction mapping.

More recently, Somenzi presented an automatic abstraction technique which identifies *reducible* data paths whose outputs are not read by any controllers of the system and reduces each of these reducible data paths to four-state non-deterministic finite state machines [9]. The abstraction technique can be applied to modulo n counters if none of the controllers depend on the outputs of the counter except for the terminal count signal. Like reducible data paths, the abstraction technique reduces each of these reducible modulo n counters to a four-state non-deterministic finite state machine. The restriction can be relaxed for modulo n counters whose intermediate values are read by some controller of the system. In this case the reduced non-deterministic finite state machine contains more than four states. The reduction is a homomorphic [6] transformation, therefore a homomorphic check is not required. However, this abstraction technique can only be applied to non-interacting data paths and non-interacting modulo n counters.

With the application of boolean decision diagrams (BDDs) [10] to formal verification [11, 12, 13], many BDD optimization techniques such as *dynamic BDD variable re-ordering* [14], *reducing BDD size by exploring functional dependencies* [15], *safe BDD minimization using don't cares* [16], *tearing-based structural decomposition* [17] and many others have been used to improve the performance of formal verification runs. However, these BDD-based abstraction techniques perform abstractions at the structural level after the behavioral model has been synthesized into a gate-level net-list.

Our semantics-based model abstraction method, performs abstraction by examining the semantics of the model itself, not the state-space of some implementation of the model. Our abstraction method is similar to the one presented in [9]. However, our abstraction techniques are not limited to non-interacting modulo n counters. Our abstraction techniques handle tightly coupled, interacting counters with bi-directional counting and loading capabilities.

We test the validity of the reduction by performing a homomorphic check of the abstraction against the portion of the original model it replaces with respect to the specified system *safety* and *fairness* properties.

Since our model abstraction approach performs abstraction at the behavioral level, SMA can be applied in conjunction with other model reduction approaches such as COSPAN's property-dependent LR approach.

3 Methodology

The main principle behind our model abstraction method consists of two tasks: (1) extracting portions of the VHDL models whose semantic behavior matches known abstraction templates, and (2) replacing those portions of the model with functionally equivalent non-deterministic finite state machines (NFSM).

Extracting Semantic Attributes

To extract semantic attributes of a model for abstraction template matching, we first perform control/dataflow analysis on the VHDL model to determine signal/variable dependencies of assignments and expressions [5]. With the results from dependency analysis, we then identify semantic attributes from the model that match the semantic attributes of our abstraction templates.

For counters we identify signals/variables in a model that have memory semantics and whose value is incremented by a constant c. The simplest form of this structure is of the type $(x <= x + c)$ and $(y := y - c)$. An example of counter attribute template is shown in Table 1.

For registers, we identify signals/variables in a model that have memory semantics and val-

Signal /	reset		load		direction		bound		wrap around		step size		modulo-N	
Variable	sync	async	sync	async	up	down	upper	lower	upper	lower	1	N	yes	no

Table 1: Counter Template

ues assigned from another variable or signal. We also extract semantic attributes such as synchronous/asynchronous reset or load to match various classes of register templates.

Identifying the abstraction template and the class within the template that a model matches enables us to generate appropriate NFSMs for that particular class of abstraction.

Extracting Abstract Data Types

In order to generate NFSMs, we must first identify the abstract data type set $\alpha(s)$ consisting of key values and/or symbolic constants for each signal/variable s.

To calculate $\alpha(s)$, we analyze all signal/variable definitions in assignments and all signal/variable uses in expressions based on the structure of the model and VHDL semantics. Each $\alpha(s)$ may inherit key values from operations with constants or inherit the abstract data type from operations with other signals/variables. If a signal/variable s does not match known abstraction templates, an error token ε is inserted into $\alpha(s)$ marking it unreducible. The error token may propagate and mark all signals/variables that depended on this signal/variable to be unreducible.

Each $\alpha(s)$ represents the entire operating range for the signal/variable s. Initially, each $\alpha(s)$ has only one symbolic constant k_s, where k_s represents the operating range of s. With each inheritance of a key value, the symbolic constant k_s within $\alpha(s)$ may be sliced into multiple symbolic constants, each representing a smaller range of values not containing the newly inherited key value. No member in an abstract data type set may overlap another, and the entire set combined is continuous. The resulting abstract data type set is a set of operating ranges for a signal/variable where the model has consistent semantic behavior. The consistency in semantic behavior allows the reduction of state space by replacing models with NFSMs.

Due to the recursive nature of the signal/variable dependencies, an iterative technique is required to calculate abstract data type sets. The analysis algorithm that generates the abstract data type sets works as follows:

For simple assignments of a constant c (e.g., a VHDL signal assignment $x <= c;$), $\alpha(x)$ inherits the constant c (i.e., a key value of the signal x). Since the constant c overlaps k_x, which is initially assigned for $\alpha(x)$, we slice k_x into k_{x_1}, c, and k_{x_2}, where k_{x_1} represents values less than c and k_{x_2} represents values greater than c. Thus, $\alpha(x)$ becomes $\{k_{x_1}, c, k_{x_2}\}$.

For simple assignments of a signal or variable (e.g., $x <= y;$), $\alpha(x)$ inherits the abstract data type from $\alpha(y)$ by merging constants and symbolic constants in $\alpha(y)$ with $\alpha(x)$. For example, if $\alpha(x)$ is $\{k_{x_1}, c_1, k_{x_2}\}$, $\alpha(x)$ is $\{k_{y_1}, c_2, k_{y_2}\}$ and c_1 is less than c_2, then $\alpha(x)$ becomes $\{k_{x_1}, c_1, k_{x_3}, c_2, k_{x_4}\}$, where k_{x_3} represent values between c_1 and c_2, and k_{x_4} represent values greater than c_2. If $\alpha(y)$ contains an error token ε (i.e., does not match abstraction template), then $\alpha(x)$ also inherits the error token marking signal x to be unreducible.

For simple assignments of a signal or variable with an operation on a constant (e.g., $x <= x + c;$), $\alpha(x)$ remains the same since this assignment has a counter semantics.

For simple relational expressions between a signal and a constant (e.g., (x rel_op c)), $\alpha(x)$ inherits the constant c. In general, for conditional expressions between two expressions (e.g., (e_1 op e_2)), the two expressions inherit each other's abstract data type as long as the operation is a relational operation. The two expressions do not inherit each other's abstract data type if the operation is a logical operation (i.e., operations returning boolean results). Otherwise, both expressions inherit an error token.

The analysis iteratively updates each abstract data type set based on the structure of the model until there are no changes in any of the abstract data type. Using the calculated abstract data type sets we generate NFSMs for counters, registers, and comparators.

Counter NFSM

With the extracted semantic attributes and the abstract data type for a model, we construct NFSMs for abstraction. For counters, each key value in the abstract data type set is a deterministic state in the NFSM while each symbolic constant is a non-deterministic state in the NFSM covering a range of values. Transitions from deterministic states are deterministic while transitions from non-deterministic states are non-deterministic requiring a non-deterministic variable to resolve conflicts. Transitions are determined by the class of the abstract template matched and hence are determined by the semantic attributes extracted from the model.

For each abstract counter template, there is a corresponding NFSM template. The general NFSM template for an up counter with wrap around is shown in Figure 1a and the general template for a up/down counter without wrap around is shown in Figure 1b. In both cases, each solid circle/arrow is a deterministic state/transition, while each dashed circle/arrow is a non-deterministic state/transition. The lower and upper bounds of the counter are indicated by $L.B.$ and $U.B$, respectively. An abstract counter NFSM traverses from deterministic key value state c_i to non-deterministic symbolic state k_j, where it non-deterministically remains in state k_j while the non-deterministic variable *ND_K* is 2 or traverses to the next deterministic state c_{i+1} when *ND_K* is 1. A value of 2 for *ND_K* represents that the counter is 2 (or more) clock cycles away from the next deterministic state. Note that for wrap around counters, both lower and upper bounds may be merged with adjacent symbolic states, if they are not key value states.

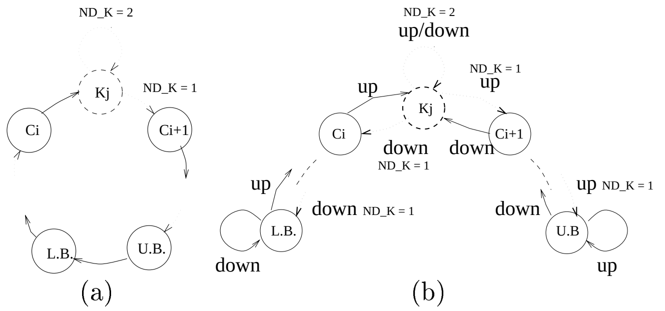

Figure 1: NFSM Template for (a) a Up Counter with Wrap Around (b) a Up/Down Counter without Wrap Around

For tightly coupled counters (e.g., two counters being compared), restrictions need to be placed on each corresponding *ND_K*$_i$ to eliminate the spurious behaviors introduced in the system by non-determinism of the reduced machine. These restrictions are similar to assigning extra edges to an extra trap state in [9] and pruning state space of the abstract model not belonging to the state space of the original model. For example, if two up counters x and y are being compared and both have the symbolic constant K_j. The corresponding non-deterministic variables *ND_K*$_x$ and *ND_K*$_y$ for the two counters can take on a value of {1,2}. If x is greater than y then *ND_K*$_x$ and *ND_K*$_y$ can not both be 1 (i.e., both counters can not reach state C_{i+1} at the same time in one clock cycle). In this case, the restriction placed on *ND_K*$_x$ and *ND_K*$_y$ is that *ND_K*$_x$ non-deterministically takes on a value of {1,2}, while *ND_K*$_y$ must be 2. Similarly, if x is less than y then the restriction placed on *ND_K*$_x$ and *ND_K*$_y$ is that *ND_K*$_x$ must be 2, while *ND_K*$_y$ non-deterministically takes on a value of {1,2}. On the other hand, if x is equal y then the restriction placed on *ND_K*$_x$ and *ND_K*$_y$ is that both *ND_K*$_x$ and *ND_K*$_x$ non-deterministically take on the same value of {1,2}. These sets of restrictions maintain the comparison order between counter x and y. A similar set of restrictions exist for two down counters x and y being compared when both have the symbolic constant K_j. These sets of restrictions are summarized in Figure 2.

Comparator NFSM

Like abstract counter templates, each abstract comparator template has a corresponding NFSM *K_COMP* template which depends on the semantic attributes extracted from the model. For example, the general NFSM *K_COMP* template for a non-wrap around up/down counter being compared to a register is shown in Figure 3a. For non-deterministic states $<$ and $>$ we use a non-deterministic variable *ND_D* which is non-deterministically assigned a value 1 or 2, representing the difference of 1 or 2 (or more) between the counter and the register. The abstract data type for *ND_D* is determined by the maximum change in the difference while the counter is counting. For a counter and a register, the maximum change in value while counting is 1. Therefore, the distance from the current non-deterministic state $<$ or $>$ to the deterministic state $=$ is 1 or 2 (or more).

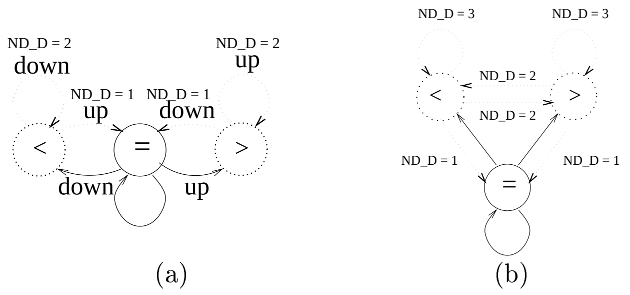

Figure 3: NFSM *K_COMP* Template for (a) a Up/Down Counter Comparing with a Register (b) Two Up/Down Counters Compared

The general NFSM *K_COMP* template for two non-wrap around up/down counters being compared is shown in Figure 3b. In this case, the maximum change in value between two counters is 2. Therefore, *ND_D* is non-deterministically assigned a value 1, 2 or 3, representing the difference of 1, 2, or 3 (or more) between the two up/down counters.

For counters with the semantic attributes of loading, the abstraction template consists of a NFSM *K_COMP* and a non-deterministic variable *ND_COMP*. The reason for this combination is due to the counter behaving like a register when data is

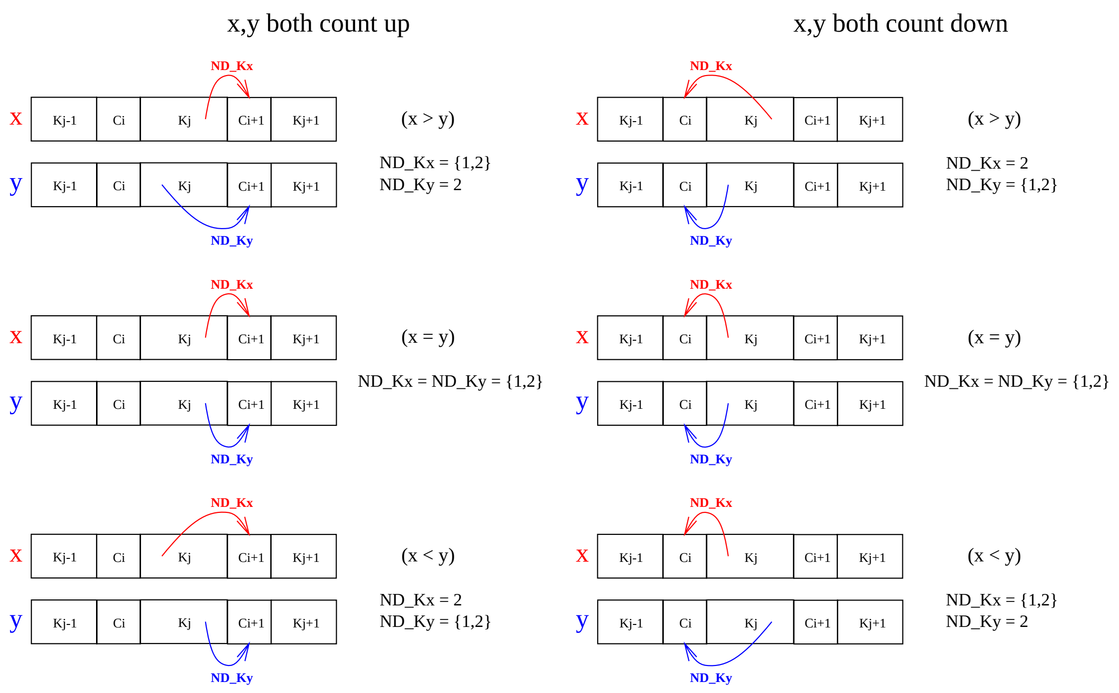

Figure 2: Restrictions on ND_K_x and ND_K_y when Counters x and y have the Same Symbolic Constant K_j

loaded into the counter. Therefore, in order to maintain equivalent test coverage and functionality, both non-deterministic variables are needed.

4 Experimental Results

In this section, we present our model abstraction experiments on a DMA controller, a car seat controller, and two robot grip controllers using the semantics-based model abstraction method presented above.

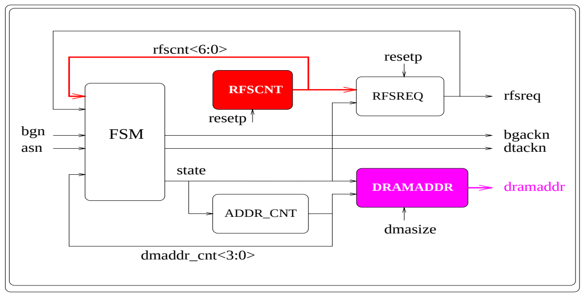

Figure 4: DMA Controller Block Diagram

The block diagram of the DMA controller is shown in Figure 4. Each block represents a VHDL process. The two system properties we want to verify are (1) *every bus request is eventually granted* and (2) *if address strobe is asserted, eventually data acknowledge is asserted*. For the DMA controller, the 7 bit refresh counter was reduced to a 3-state, non-deterministic finite automaton (NFA) shown in Figure 5. The abstraction contains key values: {0,1,k} extracted from the VHDL model, where the non-deterministic state k covers values from 2 to 127.

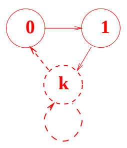

Figure 5: NFA for the Abstract Refresh Counter

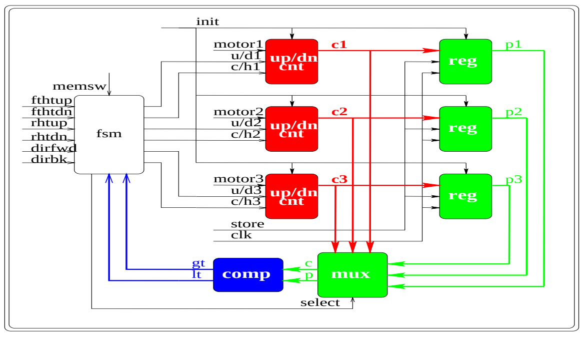

Figure 6: Car Seat Controller Block Diagram

The block diagram of the car seat controller is shown in Figure 6. The system property we want to verify is *the restoration of the car seat from any current position to the last stored position*.

The abstraction for each of the three up/down counters (that keeps track of the current seat position), contains three key values {0,k,255}, where the non-deterministic state k covers values from 1 to 254. The abstraction contains a NFA (shown in Figure 7a) that covers the entire range of seat movement.

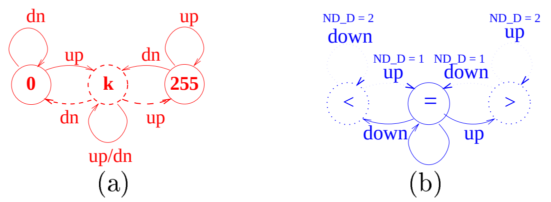

Figure 7: NFA for Car Seat (a) UP/Down Counter (b) Comparator

The abstraction for each of the three registers (that keep track of the last stored seat position) inherits the key values (abstract data type) from the abstract up/down counters. Similarly, the abstraction for the multiplexor (that selects seat movement in one of three axes) also inherits the key values (abstract data type) from the abstract up/down counter.

The abstraction for the comparator is partitioned into three comparison relations: $(c = p)$, $(c < p)$, and $(c > p)$, which cover all possible directions of seat movement. The abstraction contains an interacting NFA (shown in Figure 7b) that determines whether the seat has reached the desired position.

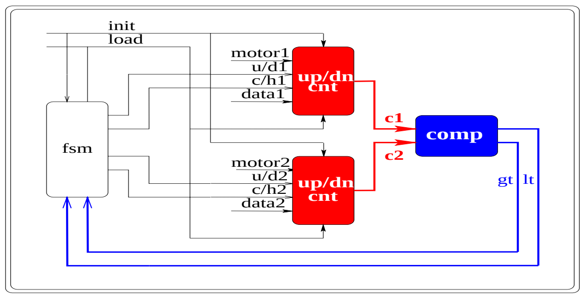

Figure 8: Robot Grip Controller Block Diagram

The block diagram of the robot grip controllers is shown in Figure 8. For the robot grip controllers the system property we want to verify is *grip object from any left grip and right grip positions*.

The two robot grip controllers, A and B, are exactly the same except for the FSMs. Controller A conforms to design specifications and can grip objects placed between any two grip positions. On the other hand, a bug in the FSM of the controller B causes overshoot of the grip movement and fails to grip objects in some grip positions.

Each of the two controllers contains two interacting, synchronous load up/down counters and a comparator. The abstraction for each of the two up/down counters (that keeps track of the current left/right grip position), contains five key values {0,k_1,31,k_2,63}, where the non-deterministic state k_1 covers values from 1 to 30 while the non-deterministic state k_2 covers values from 32 to 62. The abstraction contains a NFA (shown in Figure 9a) that covers the entire range of grip movement. The abstraction for the comparator contains an interacting NFA (shown in Figure 9b), which covers all directions of grip movement.

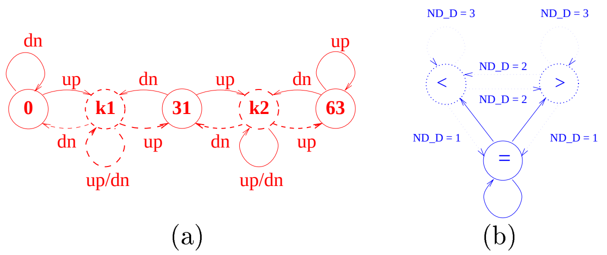

Figure 9: NFA for Robot Grip (a) UP/Down Counter (b) Comparator

The result of the abstraction experiments on the four designs is summarized in Table 2. In the table, columns 1 through 4 show the size and number of registers, counters, and comparators extracted from the original model, which are replaced by corresponding abstract models. Columns 5 through 10 show the characteristic of each corresponding class of the abstract model. Columns 5, 7, and 9 show the number of states in the generated NFSM for each abstract register, counter, and comparator. Columns 6, 8, and 10 show the number of non-deterministic variables needed for the entire design.

Original Model Replaced				Abstractions by SMA					
Bits	Reg	Cnt	Cmp	Reg		Cnt		Cmp	
	♯	♯	♯	ST	ND	ST	ND	ST	ND
DMA Controller									
7		1				3	1		
Car Seat Controller									
8	3	3	3	3		3	3	3	3
Robot Grip Controller A									
6		2	1			5	2	3	2
Robot Grip Controller B									
6		2	1			5	2	3	2

Table 2: Abstractions

To demonstrate the effectiveness of our model abstraction method, we compare the performance of formal verification runs on four designs using the formal verification tool COSPAN. As a basis for comparison, we first verify the system properties of the four designs using the original models. Next, we compare

Specs Verified	State Space Size†			ND-Selection			States Searched			CPU Time‡ (seconds)		
	Orig	LR	SMA	Orig	LR	SMA	Orig	LR	SMA	Orig	LR	SMA
DMA Controller												
Bus Req	5.94e+14	2.42e+10	2.61e+12	16	8	32	33240	33028	1066	242.07	59.68	13.23
Data Ack	3.96e+14	6.44e+10	1.74e+12	16	8	32	124929	91506	4005	680.25	135.18	37.03
Car Seat Controller												
Mem Pos	1.08e+18	1.08e+18	7.56e+07	108	108	3456	(27839126)	(16353651)	89741	(2932316)	(1824030)	177306
Robot Grip Controller A												
Grip Obj	2.10e+07	1.05e+07	9.60e+04	4096	4096	900	44768	43152	334	20822.7	17068.73	67.35
Robot Grip Controller B												
Grip Obj	2.10e+07	1.05e+07	9.60e+04	4096	4096	900	24603	24357	203	10844.8	9891.05	40.97

† State space size includes both the system model and the environment model.
‡ Verification runs performed using 70MHz SUN SPARC5 with 128MB main memory and 732MB swap space.
$^{()}$ Verification run terminated due to insufficient swap space.

Table 3: Model Abstraction Experiments

the performance of formal verification runs on the four designs using COSPAN's property-dependent *localization reduction* (LR) method, and our semantics-based model abstraction method (SMA). Thus, our model abstraction experiments consists of three sets of formal verifications runs: Original, LR, and SMA. The results of the verification runs are shown in Table 3 Note that in all the verification runs, where the LR method was applied, the results stated in the table are the results of the last LR iteration reported by COSPAN. In this experiment, all the verification runs verified the specified system properties except for the following cases: (1) counter examples were found in the verification runs of the robot grip controller B and (2) verification runs of unreduced car seat controller models failed to complete due to large memory requirements.

Table 3 shows the state space size, the number of non-deterministic selections, the number of states searched, and CPU time for the same three sets of verification runs. The state space size includes both the system model and the environment model [1] reported by COSPAN for each verification run. The number of non-deterministic selections indicates the verification overhead induced by non-deterministic variables in both the environment model and the abstract model. These first two set of results show the cost of verifying a particular model. The number of states searched is the size of the state space explored before either the system property is verified or a counter example is found. Along with cpu-time, these last two sets show the performance of each model abstraction method used in verifying the system properties of the four designs.

In the table, column 1 shows the system properties verified for each of the designs. Columns 2, 5, 8, and 11 (labeled Original) show the results for the first set of verification runs verifying the four designs using the original models. Columns 3, 6, 9, and 12 (labeled LR) show the results for the second set of verification runs verifying the four designs using the original models and applying COSPAN's property-dependent *localization reduction*. Columns 4, 7, 10, and 13 (labeled SMA) show the results for the third set of verification runs verifying the abstract models of the four designs reduced by our SMA method.

From Table 3, we can see that by using our SMA method, the abstract models for all the designs reduced the state space size by many orders of magnitude ($10^2 - 10^{11}$) when compared to the original model. Furthermore, in all the examples, the verification performance is consistently improved with 3.6 to 640.7 fold reduction in CPU time or higher using our SMA method than that of COSPAN's LR method. It is important to note that for the DMA controller, the abstract models obtained from our SMA method have larger state space size and non-deterministic selection overhead than that of the abstract models obtained from COSPAN's LR method. However, despite the larger overhead, the verification time is many fold smaller using models reduced by our SMA method compared to models reduced by COSPAN's LR method. More importantly, the verification runs using abstract models from our SMA method all completed, where as, using the original models, some examples terminated prematurely after running out of swap space. As the designs in this experiment showed, our abstraction techniques handle a wide range of designs containing counters including tightly coupled, interacting counters.

5 Conclusions

In this paper we presented a method for abstracting non-deterministic finite state machine (NFSM) models from behavioral VHDL descriptions for formal verification. Our abstraction algorithm replaces counters, comparators and registers in the behavioral VHDL model with the corresponding NFSM models. The result is an abstract model that has a state space size much smaller than that of the original model but functionally equivalent with respect to the specifications to be verified. Our three model abstraction techniques: key value extraction, model partitioning and data abstraction enable us to verify models not possible without abstractions.

The abstraction algorithm can be extended for counters that increment or decrement by values other than one with some caveats for non-modulo N counters. Furthermore, performance optimizations such as reducing non-deterministic overhead by enumerating symbolic constants or merging mutually exclusive non-deterministic variables can also be integrated into the abstraction algorithm.

We are currently implementing an automatic model abstraction tool based on model abstraction algorithm presented in this paper. A set of abstractions will be generated by analyzing the semantics extracted from the VHDL models using our existing semantic extraction tool. Each abstraction will be evaluated based on a performance cost matrix to determine which subset of abstractions should be selected to replace parts of a model to drive the verification process.

The reduction presented in [9] has been formally proven to be a homomorphic transformation, therefore a homomorphic check is not required. Due to the similarity between our abstraction method and theirs, we believe that the reduction presented in this paper is also a homomorphic transformation with respect to the specified system properties. Since formal proof of the reduction is currently not available, we test the validity of the reduction by performing a homomorphic check of the abstraction against the portion of the original model it replaces.

The future work in this research includes providing a formal proof that the our abstraction method is a homomorphic transformation and expanding our semantic extraction method to identify higher level semantics to perform abstraction of high-level models. Also, we plan to incorporate VIS [3] into our formal verification paradigm.

Acknowledgments The authors would like to thank Dr. Robert Kurshan of Lucent Technologies at Bell Laboratories for providing us with the COSPAN formal verification tool.

References

[1] Z. Har'El and R. P. Kurshan, *COSPAN User's Guide.* AT&T Bell Laboratories, February 1993.

[2] K. L. McMillan, *Symbolic Model Checking.* Kluwer Academic Publishers, 1993.

[3] R. K. Brayton, et al., “Vis: A system for verification and synthesis,” in *Proc. of Conference on Computer-Aided Verification*, pp. 428–432, July 1996.

[4] Y.-W. Hsieh and S. P. Levitan, “Model abstraction for formal verification,” in *Proc. of Design, Automation and Test in Europe Conference*, pp. 140–147, March 1998.

[5] Y.-W. Hsieh and S. P. Levitan, “Control/dataflow analysis for vhdl semantic extraction,” *J. of Information Science and Engineering*, vol. 14, pp. 547–565, September 1998.

[6] R. P. Kurshan, *Formal Verification of Coordinating Processes: The Automata-Theoretic Approach*, ch. Reduction of Verification. Princeton University Press, 1994.

[7] D. E. Long, *Model Checking, Abstraction, and Compositional Verification.* PhD thesis, Dept. of Electrical Engineering, Carnegie Mellon University, 1993.

[8] E. M. Clarke and R. P. Kurshan, “Computer aided verification,” *IEEE Spectrum*, pp. 61–67, June 1996.

[9] E. Macii, B. Plessier, and F. Somenzi, “Formal verification of digital systems by automatic reduction of data paths,” *IEEE Trans. on CAD*, vol. 16, pp. 1136–1156, October 1997.

[10] R. E. Bryant, “Graph-based algorithm for boolean function manipulation,” *IEEE Trans. on Computers*, vol. C-35, pp. 677–691, August 1986.

[11] J. R. Burch, et al., “Sequential circuit verification using model symbolic model checking,” in *Proc. of DAC*, pp. 46–51, June 1990.

[12] H. J. Touati, et al., “Implicit state enumeration of finite state machines using bdd's,” in *Proc. of ICCAD*, pp. 130–133, November 1990.

[13] M. Chiodo, et al., “Automatic compositional minimization in ctl model checking,” in *Proc. of ICCAD*, pp. 172–178, November 1992.

[14] R. Rudell, “Dynamic variable ordering for ordered binary decision diagrams,” in *Proc. of DAC*, pp. 42–27, June 1993.

[15] A. J. Hu and D. L. Dill, “Reducing bdd size by exploiting functional dependencies,” in *Proc. of DAC*, pp. 266–271, June 1993.

[16] Y. Hong, et al., “Safe bdd minimization using don't cares,” in *Proc. of DAC*, pp. 208–213, June 1996.

[17] W. Lee, et al., “Tearing based automatic abstraction for ctl model checking,” in *Proc. of ICCAD*,pp. 76–81, November 1996.