Model Abstraction for Formal Verification *

Yee-Wing Hsieh Steven P. Levitan

Department of Electrical Engineering
University of Pittsburgh

Abstract

As the complexity of circuit designs grows, designers look toward formal verification to achieve better test coverage for validating complex designs. However, this approach is inherently computationally intensive, and hence, only small designs can be verified using this method. To achieve better performance, model abstraction is necessary. Model abstraction reduces the number of states necessary to perform formal verification while maintaining the functionality of the original model with respect to the specifications to be verified. As a result, model abstraction enables large designs to be formally verified. In this paper, we describe three methods for model abstraction based on semantics extraction from user models to improve the performance of formal verification tools.

1 Introduction

Traditional methods of validating a design are based on test vector simulation. Although this method of design validation is simple, it has its inherent drawbacks. First, it is tedious to generate test vectors to test a particular design specification, and often the test vectors used only cover a limited state-space. Second, to generate the test vectors, the designer must know implementation details to set the simulator to a specified state to test a specific event. Third, it is difficult to verify asynchronous events such as reset, abort, bus request, etc. Finally, it is difficult to verify safety and liveliness properties of a design through test vector simulation.

Formal verification provides much better test coverage and does not have the inherent drawbacks of test vector simulation. However, 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 a non-deterministic automaton 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 such as COSPAN [1].

In previous papers [2] we focused on semantic extraction of VHDL models for formal verification. We have implemented a semantic extraction tool based on control/data-flow analysis techniques to automatically extract memory semantics in the VHDL models. The focus of this paper is our model abstraction techniques based on the semantics of the VHDL models obtained from semantic extraction. Our model abstraction approach reduces the state-space using three techniques: key value extraction, model partitioning through min/max data-flow analysis and data abstraction through relational algebra.

The remainder of the paper is organized as follows. First, we briefly review previous work on model abstraction for formal verification. Next, we describe the motivation for semantic extraction for model abstraction. Afterwards, we describe our model abstraction techniques made possible by examining VHDL semantics extracted from the control/data-flow analysis of the original VHDL model. Finally, we present experimental results and conclusions.

2 Related Work

Model reduction transforms the verification problem to an equivalent problem in a smaller state-space. Reduction is generally achieved by replacing processes in the model by smaller processes that have similar or identical communication behavior [3].

The two main approaches in model checking are logic-based (e.g., CTL [4]) and automaton-based (e.g.,

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

ω-automata [5]). ω-automaton reduction is achieved through property-dependent *localization reductions*, in which the parts of design model which are irrelevant to the property being checked are automatically abstracted away [6]. In COSPAN, localization reduction is applied dynamically and the model is adjusted by advancing its "free fence" of induced primary inputs, in order to discard spurious counterexamples to the stated query. However, our experiments showed that the automatic reduction algorithm works best if the user specifies a reduction starting point. This may include some components of the model that probably would be necessary for analysis or some components that could be excluded. This implies that the designer needs to know the relationships and dependencies among components in a model. More importantly, the reduction is achieved by exploring the state-space of the unabstracted machine, which could be time consuming for large designs.

CTL based approaches, on the other hand, use 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]. The reductions preserve the truth value of all formulas in a suitable logic. The minimization techniques are fairly strict in terms of the required relations between the original and reduced processes. However, the user must supply an abstraction mapping, which implies that the reduction process is not automatic.

Unlike the automatic ω-automaton reduction approach, our model abstraction approach performs abstraction by examining the semantics of the model itself, not the state-space of some implementation of the model. The abstractions obtained through our model abstraction techniques are homomorphic [5] to the original model with respect to the specified queries. In fact, we believe, our model abstraction approach could be applied in conjunction with either the ω-automaton reduction approach or the CTL based approach.

3 Semantic Extraction

The memory semantics of a model ultimately determine the state-space explored by formal verification tools. Therefore, in order to determine which signals or which parts of the original model to abstract, we must first use semantic extraction to identify the parts of the model that exhibit memory semantics.

One may argue, why not simply synthesize the circuit to obtain the set of signals implemented using registers or flip-flops? The problem with this method is that synthesis tools often utilize optimization techniques to minimize hardware resources which may introduce more registers or merge two or more signals into a single register. Such optimizations make semantic extraction and model abstraction difficult. Furthermore, the end result of the synthesis step may hide some semantics for proper model abstraction.

Therefore, we use semantic extraction to directly obtain a set of abstraction candidates, that is pieces of the original design, which might speed up the verification process if they were mapped into more abstract models. We then must evaluate the gain of each of the abstraction candidates to determine the set of signals or the parts of the original model on which to perform abstraction. Finally, we perform model abstraction to drive the verification process.

We have implemented a VHDL semantic extraction tool that analyzes both concurrent and sequential VHDL models [2]. The semantic extraction tool is based on data-flow analysis techniques in compiler designs for code optimization. The semantic extraction method identifies an abstract state-space that is independent of synthesis optimizations. The remainder of this paper focuses our model abstraction techniques using the semantics extracted from VHDL models.

4 Model Abstraction

Our model abstraction approach derives abstractions based on the semantics of the model from the behavioral VHDL description. The abstraction technique operates on three main principles: key value extraction, model partitioning and data abstraction.

Both the key value extraction and data abstraction techniques reduce the state-space by identifying signals and variables which may be *replaced* by symbolic or abstract data types with much smaller ranges. For each abstraction candidate identified a non-deterministic finite automaton is generated such that every state in the abstraction is mapped to a state in the corresponding part of the model it replaces.

On the other hand, the model partitioning technique reduces the state-space by *removing* parts of a model not related to the specification to be verified. It also reduces the state-space by identifying parts of a model which may be replaced by non-deterministic finite generators to provide stimuli for the rest of the model.

The resulting abstract models can replace the original model for formal verification, provided that each of the abstractions is homomorphic to the corresponding part of the original model that it replaces with respect to the specifications to be verified [5].

4.1 Key Value Extraction

For control dominated systems, the properties to be verified are often related to whether the system is at a specified state under a particular set of conditions. Therefore, it may be sufficient to simply verify the system when the conditions are met and when the conditions are not met. For example, if we want to verify whether the system moved from state σ to state σ' under the condition that variable v is equal to constant c, we need to verify the system when $v = c$ and when $v \neq c$. More precisely, all the possible values for variable v can be partitioned into three categories: $(v = c)$, $(v < c)$ and $(v > c)$ [1] An abstraction for variable v (denoted by $\hat{v}$) is the set of key values: $\{c,k_1,k_2\}$, where k_1 represents all value less than constant c and k_2 represents all value greater than constant c. The state-space for variable v is reduced from every possible value v can take to three key values.

To extract key values for a variable v in a model, we examine every conditional expression where variable v is referenced (either directly or indirectly) and we build a relation for each variable in the conditional expression. For example, if a comparison is made between the variable v and constant c, then $\hat{v}$ has key value c and the following relation $\mathcal{R}$:

$$\hat{v} = \begin{cases} c & \text{if } v = c \\ k_1 & \text{if } v < c \\ k_2 & \text{if } v > c \end{cases}$$

In addition to the conditionals, key values for a variable v may be extracted from assignments where constants are assigned to variables. Often, such constant assignments set initial values for the variables. For example, it is a common practice to reset counters to 0.

The abstraction for variable v is therefore the union of all key values from all the conditional expressions where variable v is referenced plus any intermediate symbolic key values derived from the product of all relations for the variable. For example, if the variable v is referenced in the following two conditional expressions: $(v = c_1)$ and $(v = c_2)$, the result is the following two relations:

$$\hat{v_1} = \begin{cases} c_1 & \text{if } v = c_1 \\ k_{1,1} & \text{if } v < c_1 \\ k_{1,2} & \text{if } v > c_1 \end{cases}$$

$$\hat{v_2} = \begin{cases} c_2 & \text{if } v = c_2 \\ k_{2,1} & \text{if } v < c_2 \\ k_{2,2} & \text{if } v > c_2 \end{cases}$$

Assume $c_1 < c_2$, then the product of the two relations is

$$\hat{v} = \begin{cases} k_1 & \text{if } v < c_1 \\ c_1 & \text{if } v = c_1 \\ k_2 & \text{if } c_1 < v < c_2 \\ c_2 & \text{if } v = c_2 \\ k_3 & \text{if } v > c_2 \end{cases}$$

Here, each constant k_i is a symbolic constant that covers a range of values for v. As a result, $\hat{v}$ has key values $\{c_1,c_2,k_1,k_2,k_3\}$, where $k_1 < c_1 < k_2 < c_2 < k_3$.

4.1.1 Model Characteristics

As mentioned in the previous section, key values may be extracted from variable assignments of the form:

$$v := c;$$

Often, such constant assignments set initial values for the variables. If the variable has an assignment of the form:

$$v := v + c;$$

the variable has the characteristics of a sequencer or counter. A variable v_1 can inherit key values from another variable v_2 if the variable has an assignment of the form:

$$v_1 := v_2;$$

or has an assignment of the form:

$$v_1 := v_2 + c;$$

In the first case, $\hat{v_1}$ (i.e., the abstraction for variable v_1) simply inherits key values from $\hat{v_2}$. In the second case, $\hat{v_1}$ inherits key values from $\hat{v_2}$ with constant c folded into each of the key values from $\hat{v_2}$.

If a variable has the characteristics of a counter or sequencer, we construct a non-deterministic finite automaton (NFA) based on the key values selected for the variable. In order to cover the entire state-space of the variable, the NFA must generate (accept) real constants extracted from the model and symbolic constants derived from the relations for the variable. Specifically, each key value corresponds to a state in the NFA. All symbolic constants k_i covering a range of values are non-deterministic states, while the rest are deterministic states. All transitions from non-deterministic states are non-deterministic transitions, while all transitions from deterministic states are deterministic transitions. Only non-deterministic states

[1] It is important to note that each of the six comparison operations $(=,\neq,<,>,\leq,\geq)$ partitions values for a variable v into the three categories: $(v = c)$, $(v < c)$ and $(v > c)$.

have transitions onto themselves which represent the transition through the range of values symbolically covered.

To illustrate the construction of the NFA, we now examine the refresh counter model shown in Figure 1. The refresh counter generates a pulse for a refresh memory request (*rfsreq*) whenever the DMA controller is in the refresh state *rfs*. In this model, the refresh counter (*rfscnt*) is 7 bits wide with 128 possible states. From the model the signal *rfscnt* has key value 0 due to constant assignment (rfscnt <= "0000000") and has key values 1, k_1 and k_2 due to the conditional expression (rfscnt <= "0000001"). The symbolic constant k_1 represents values less than 1, or in this case, 0. The symbolic constant k_2 represents values greater than 1, or in this case, 2 through 127. The signal *rfscnt* also has counter characteristics due to the assignment (rfscnt <= rfscnt + 1). Therefore, the key values for the signal *rfscnt* is the set {0,1,k} where symbolic constant constant k represents values other than 0 and 1.

```
process
begin
   wait until ((clk'event) and (clk = '1'));
   if (resetp = '1') then
      rfscnt <= "0000000";
   else
      rfscnt <= rfscnt + 1;
   end if;
end process;

process
begin
   wait until ((clk'event) and (clk = '1'));
   if ((resetp = '1') or (state = rfs)) then
      rfsreq <= '0';
   elsif (rfscnt = "0000001") then
      rfsreq <= '1';
   end if;
end process;
```

Figure 1: VHDL Model for a Refresh Counter

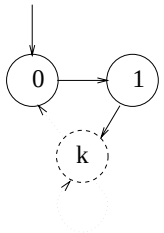

Figure 2: NFA for the Abstract Refresh Counter

The NFA for the signal *rfscnt* is shown in Figure 2. States 0 and 1 are deterministic states while state k is a non-deterministic state. The transitions from state 0 to state 1 and state 1 to state k are deterministic transitions, which are represented by the solid arrows in the figure. The transition from state k back to state k and state k to state 0 are non-deterministic transitions, which are represented by the dashed arrows in the figure. The non-deterministic transition from state k back to state k symbolically covers values in the range from 2 to 127.

To summarize, key value extraction reduces the state-space by replacing signals or variables with symbolic data types with much smaller ranges. Along with the generated NFA, the resulting abstraction can replace its original counterpart for verification.

4.2 Model Partitioning

In addition to key value extraction, model abstraction may be achieved through model partitioning. Model partitioning takes a portion of a model and replaces it with an abstract model. For example, if a portion of a model does not affect (i.e., is independent from) the rest of the model with respect to the properties to be verified, it may be advantageous to abstract that portion of the model away. Partitioning relevant portions of the model can be achieved by analyzing the signal flow graph generated from the model. Here, the nodes of the signal flow graph are operations and the edges are signals or variables. Starting from the properties to be verified (e.g., verify the value of variable v), we traverse the flow graph recursively in reverse order, from outputs to inputs, and label those nodes and edges visited on the way. The labeled branches of the signal flow graph are property dependent, while the unlabeled branches of the signal flow graph are property independent and may be abstracted away.

For some properties, it may not be important how a particular signal or variable is evaluated, but the status or the completeness of the coverage is all that matters. In those cases, it may be possible to abstract away the portion of the model that generates the signal or variable. For example, if a system has an error detection module and the properties being verified depend on the status of the error detector, (e.g., one can't verify properties if an error is detected), it may be sufficient to abstract away the error detection part of the circuit and fix the status of the error detection so that properties can be verified without it. On the other hand, if we want to verify error handling properties of the model, it may be more appropriate to generate errors randomly. In this case, it may be sufficient to exercise the range of error status non-deterministically and verify the properties without the error detection part of the circuit. This type of

abstraction can be performed through min/max partitioning of the signal flow graph.

In this way, we use partitioning to identify signals with a small range but high cost in terms of the size of the state-space from all the branches generating this signal. The non-deterministic finite generators replace the partitioned model by providing stimuli for the rest of the model.

4.3 Data Abstraction

For some systems, abstraction using key value extraction and model partitioning alone may be ineffective. Data abstraction may be necessary to reduce the size of the state-space. Data abstraction is a very difficult problem. However, in some systems the actual values of some variables are not important, but the relations between the variables are all that matters. For example, no key values can be extracted from the conditional expression of the form:

$$(v_1 < v_2)$$

However, if the values of the two variables are not important, an abstraction can be made based on the relation between the two variables. Specifically, the two variables can be abstracted as follows:

$$\hat{v_1} = \begin{cases} k_1 & \text{if } v_1 < v_2 \\ k_2 & \text{if } v_1 = v_2 \\ k_3 & \text{if } v_1 > v_2 \end{cases}$$

$$\hat{v_2} = \begin{cases} k_3 & \text{if } v_1 < v_2 \\ k_2 & \text{if } v_1 = v_2 \\ k_1 & \text{if } v_1 > v_2 \end{cases}$$

Here, k_1, k_2 and k_3 are symbolic constants that represent the set of value pairs (v_1, v_2) such that $(v_1 < v_2)$, $(v_1 = v_2)$ and $(v_1 > v_2)$, respectively.

The number of relations or comparisons between a set of variables depends on the number of equality operators, the number of inequality operators and the grouping of equalities.

As this variable comparison technique suggests, this type of abstraction depends only on the relative comparisons between variables. Different variable sizes would still yield the same abstraction. On the other hand, as the number of variables grows, the number of comparison permutations grows exponentially. In some cases, this type of abstraction may be too costly.

Thus, data abstraction reduces the state-space in a similar way to key value extraction, however, using abstract values rather than constants.

5 Model Abstraction Experiments

In this section, we describe the abstractions made manually on a DMA controller, a HDLC serial communication controller and a car seat controller using the three model abstraction techniques presented above. Each experiment is presented in detail to illustrate the abstraction analyses performed on the semantics extracted automatically from the VHDL models. These abstraction analyses will be automated in our model abstraction tool.

To demonstrate the differences in the state-space size and performance, the original and the abstract models were verified separately using the same environment model and reference model [1]. The results of verification runs for the original and the abstract models are shown in Table 1. The state-space sizes presented in the table are the state-spaces for both the system model and the environment model reported by COSPAN from each respective verification run.

Specification Verified	State Space		CPU Time[1] (seconds)	
	Original	Abstract	Original	Abstract
DMA Controller				
Mem. Ref.	1.05e+15	2.47e+13	496.58	30.20
Data Ack.	7.04e+14	1.65e+13	1353.03	83.08
HDLC Controller				
Under Flow	4.05e+32	2.36e+22	(55208.1)[2]	135.10
RX Valid	3.65e+34	2.21e+24	(58912.6)[2]	10.12
Car Seat Controller				
Mem. Pos.	6.91e+19	1.79e+08	(3053191.0)[2]	24920.2

Table 1: Model Abstraction Experiments

For the DMA controller, an abstraction was made on the refresh counter part of the circuit using the key value extraction method. The model for the refresh counter was shown previously in Figure 1. The refresh counter is a 128-state deterministic automaton. We reduced the state-space for this refresh counter by abstracting the counter into a 3-state, non-deterministic finite automaton (NFA), shown in Figure 2. The resulting abstract DMA controller model yields a 42 fold reduction in state-space, which enables the two system properties: *memory refresh* and *data acknowledge* to be verified with a 16 fold improvement in verification performance.

The HDLC serial communication controller consists of two separate circuits: the transmitter and the receiver. The block diagrams of the transmitter and receiver are shown in Figure 3. The two circuits operate independently except for a serial communication medium. The controller uses cyclic redundant

[1] verification runs performed using SUN SPARC5 with 64M bytes main memory and 732M bytes swap space.

[2] verification run terminated due to insufficient swap space.

check (CRC) to detect errors induced by noise in the communication medium. In the CRC error detection method [8], the transmitter generates an n-bit sequence, known as a frame check sequence (FCS) based on a k-bit frame or message. The resulting frame consists of $k + n$ bits and is divisible by some predetermined polynomial. The receiver then divides the incoming frame by the same polynomial. If the remainder is zero, no error is detected. The HDLC serial communication controller uses the frame check polynomial $(x_{16} + x_{12} + x_5 + 1)$ or 16 bit frame check sequence.

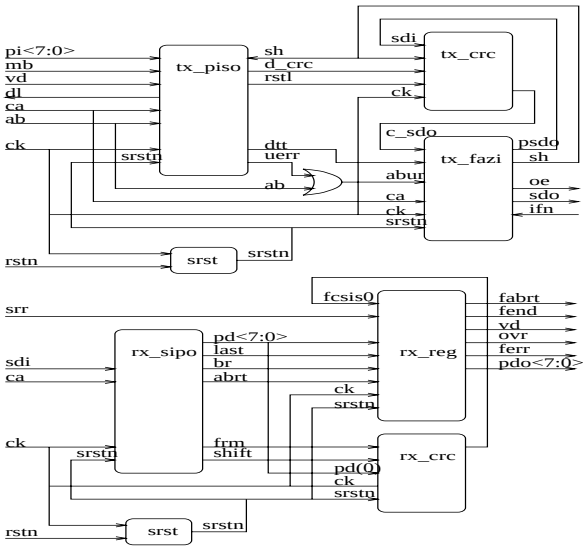

Figure 3: HDLC Controller Block Diagram

From min/max model partitioning, both the transmitter and receiver CRC part of the circuit were determined to be strong candidates for abstraction. In the abstraction we eliminate the CRC part of circuit completely. To check the CRC error handling properties, we non-deterministically generate CRC errors at the receiver's CRC output (*fcsis0*). On the other hand, to check the other properties, we statically force *fcsis0* not to generate any errors. The resulting abstract HDLC controller model yields 10 orders of magnitude in state-space size reduction, which enables system properties: *under flow error* and *receive valid data* to be verified within 135.1 and 10.12 seconds, respectively. More importantly, the verification run on the original controller was terminated prematurely due to insufficient swap space.

The block diagram for the car seat controller is shown in Figure 4. The car seat controller has 3 up/down counters, 3 registers, 1 comparator, 1 multiplexor and a finite state machine controlling all the components. The current seat position is stored in the three 8-bit up/down counters (c_1,c_2,c_3). The memory position is stored in the three 8-bit registers (p_1,p_2,p_3). In the manual mode, the seat position is controlled by the six signals (*fthtup,fthtdn,rhtup,rhtdn,dirfwd,dirbk*). In the memory mode, the multiplexor selects one of the axis's current position c and corresponding memory position p from (c_1,c_2,c_3) and corresponding (p_1,p_2,p_3), respectively. The 8-bit comparator calculates the direction of movement by comparing the two positions c and p. The car seat is moved one axis at a time until the current position matches the memory position in all three axes.

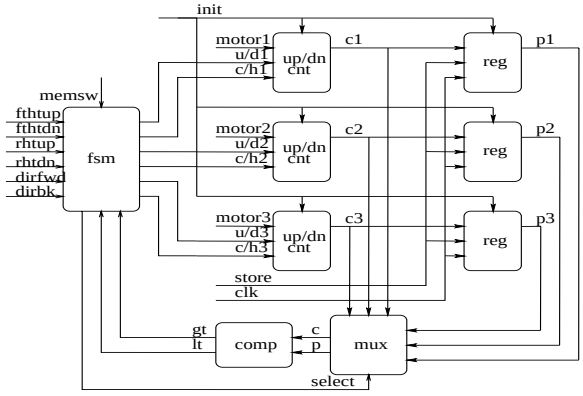

Figure 4: Car Seat Controller Block Diagram

The VHDL model for the 8-bit up/down counter is shown in Figure 5. From the conditional expression ($c \neq 255$) in the model we obtain the key values (k_1, 255, k_2) where the symbolic constant k_1 represents numbers less than 255 and the symbolic constant k_2 represents numbers greater than 255. Similarly from conditional expression ($c \neq 0$) in the model we obtain the key values (k_3, 0, k_4) where the symbolic constant k_3 represents numbers less than 0 and the symbolic constant k_4 represents numbers greater than 0. From the signal assignment ($c <= 0$;) we obtain the key value 0. Since signal c is an 8 bit signal, the lowest and highest numbers an 8 bit binary number could represent are 255 and 0, respectively. Therefore, only key values (0, k_2, k_3, 255) are valid. Merging the two symbolic constants k_2 and k_3, we obtain the three key values (0, k_5, 255) where the symbolic constant k_5 represents numbers 1 to 254.

The current seat position can be stored in memory. This is achieved by storing the contents of the three up/down counters (c_1,c_2,c_3) into three corresponding registers (p_1,p_2,p_3). These are assignments of the form:

$$v_1 := v_2;$$

```
process (clk,pc)
begin
    if (pc = '0') then
        c <= 0;
    elsif ((clk'event) and (clk = '1')) then
        if (count = '1') then
            if (direction = '1') then
                if (c /= 255) then
                    c <= c + 1;
                end if;
            else
                if (p /= 0) then
                    c <= c - 1;
                end if;
            end if;
        end if;
    end if;
end process;
```

Figure 5: VHDL Model for the Up/Down Counter

Therefore, each of the three registers inherits the key value extracted from the three up/down counters. Since the comparison operation is of the form:

$$(v_1 < v_2)$$

we abstract each up/down counter and register comparison pair (i.e., (c_1,p_1), (c_2,p_2) and (c_3,p_3)) into three symbolic comparison relations: $(c < p)$, $(c = p)$ and $(c > p)$. Each of the three counter/register comparison pairs is independent, so the abstraction we made for a counter/register comparison pair applies to all three cases. Expanding the comparison relations with the key values for counters and registers, we obtain the comparison cases shown in Table 2.

$c < p$		$c = p$		$c > p$	
	$0 < 0$	$\surd$	$0 = 0$		$0 > 0$
$\surd$	$0 < k_5$		$0 = k_5$		$0 > k_5$
$\surd$	$0 < 255$		$0 = 255$		$0 > 255$
	$k_5 < 0$		$k_5 = 0$	$\surd$	$k_5 > 0$
$*$	$k_5 < k_5$	$*$	$k_5 = k_5$	$*$	$k_5 > k_5$
$\surd$	$k_5 < 255$		$k_5 = 255$		$k_5 > 255$
	$255 < 0$		$255 = 0$	$\surd$	$255 > 0$
	$255 < k_5$		$255 = k_5$	$\surd$	$255 > k_5$
	$255 < 255$	$\surd$	$255 = 255$		$255 > 255$

Table 2: Comparison Relations Between c and p

Eliminating those comparison relations that are not valid, we are left with those relations marked with a ($\surd$). Since the symbolic constant k_5 represents numbers 1 to 254, the three relations marked with an ($*$) are also valid. From these relations, we construct abstractions for the up/down counter and comparator. The comparison relations and the corresponding abstractions for the counter and comparator are shown in Table 3.

c	p	relation	direction	count
0	0	$=$	*don't care*	0
0	k_5	$<$	up	1
0	255	$<$	up	1
k_5	0	$>$	down	*non-det.*
k_5	k_5	$<=>$	*non-det.*	*non-det.*
k_5	255	$<$	up	*non-det.*
255	0	$>$	down	1
255	k_5	$>$	down	1
255	255	$=$	*don't care*	0

Table 3: Abstractions for the Up/Down Counter and Comparator

For the comparator, the abstraction is simply the comparison relations between the two key values for counter c and the register p. However, if the counter c and the register p are both k_5, we can not determine the comparison relation between the two because the symbolic constant k_5 could be any number 1 to 254. As a result, three possible comparison relations exist when both c and p have key value k_5. A nondeterministic finite automaton shown in Figure 6 is used to solve this uncertainty problem. When both c and p have key value k_5, it either non-deterministically generate a count *up* or count *down* signal. As a result, the abstraction for the comparator covers all possible seat movements from any current position to a specified memory position.

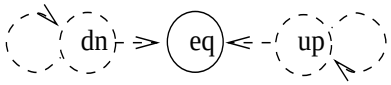

Figure 6: NFA for the Comparator

Previously, we mentioned the abstraction for the up/down counter (signal c) contains the three key values (0, k_5, 255). Semantic analysis also showed that the model has characteristics of a sequencer or counter, as indicated by the signal assignments ($c <= c+1;$) and ($c <= c-1;$). Therefore, the abstraction for the up/down counter also contains a non-deterministic finite automaton as shown in Figure 7. Since the two signal assignments increment and decrement by a constant, the NFA for the counter has transitions between key values in ascending and descending order. Specifically, the counter would count from key value 1 up to k_5 and from 255 down to k_5 deterministically in one

clock cycle. On the other hand, the counter would count from key value k_5 down to 1 and from k_5 up to 255 non-deterministically. The counter would non-deterministically remain at key value k_5 which could cover numbers 1 to 254. This non-deterministic automaton is necessary in order to cover all possible cases for the counter.

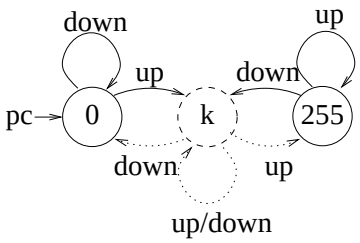

Figure 7: NFA for Up/Down Counter

The result of the abstractions on the up/down counters, registers and comparator yield 11 orders of magnitude in state-space size reduction. The system property *move the car seat from current position to memory position* was verified in under 26219 seconds using the abstract model of the car seat controller. On the other hand, the verification run using original model failed to complete after 3053191 seconds or 848.1 hours due to insufficient swap space.

In summary, these experiments have shown that our model abstraction technique of using key value extraction, model partitioning and data abstraction are effective in reducing state-space and hence improve the performances of verifications tools. More importantly, these abstraction techniques enables us to verify models not possible without abstractions.

6 Conclusions and Future Work

In this paper we have shown that in order to improve the performance of formal verification tools such as COSPAN, model abstraction is necessary. Our model abstraction approach reduces the state-space using key value extraction, model partitioning through min/max data-flow analysis and data abstraction through relational algebra. The model abstraction is made possible by examining VHDL semantics extracted from the control/data-flow analysis of the original VHDL model. Experiments have shown that these abstractions techniques are effective in reducing the state-space and hence improving the performance of verification runs. It is important to note that these abstraction techniques could be applied in complement to the ω-automaton reduction approach.

We are currently implementing an automatic model abstraction tool based on model abstraction techniques 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 future work in this research includes expanding our semantic extraction method to identify higher level semantics to perform abstraction of high-level models. Also, we plan to apply these abstraction techniques to CTL based model checkers such as Vis [9].

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] Y.-W. Hsieh and S. P. Levitan, "Control/data-flow analysis for vhdl semantic extraction," in *Proc. of The 4th Asia-Pacific Conference on Hardware Description Languages*, pp. 68–75, August 1997.

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

[4] E. M. Clarke and E. A. Emerson, "Design and synthesis of synchronization skeletons using branching time temporal logic," in *Proc. of the Workshop on Logics of Programs*, pp. 52–71, May 1981.

[5] R. P. Kurshan, *Formal Verification of Coordinating Processes*. Princeton University Press, 1994.

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

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

[8] W. Stallings, *Data and Computer Communications*. Macmillan Publishing Company, 1988.

[9] R. K. Brayton, et al., Vis: A system for verification and synthesis. In *Proc. of Conference on Formal Methods in Computer-Aided Design*, November 1996.