Accounting for Various Register Allocation Schemes During Post-Synthesis Verification of RTL Designs[*]

Nazanin Mansouri and Ranga Vemuri
{nmansour,ranga}@ececs.uc.edu
Department of Electrical and Computer Engineering and Computer Science
University of Cincinnati
Cincinnati, OH 45221-0030, USA

Abstract

This paper reports a formal methodology for verifying a broad class of synthesized register-transfer-level (RTL) designs by accommodating various register allocation/optimization schemes commonly found in high-level synthesis tools. Performing register optimization as part of synthesis process implies that the mapping between the specification variables and RTL registers is not bijective. We propose a formalization of dynamic variable-register mapping, and techniques based on symbolic analysis and higher-order logic theorem proving for verifying synthesized RTL designs. The proposed verification methodology has been successfully implemented using the PVS theorem prover.

1 Introduction

High-level synthesis systems generate register-transfer level (RTL) designs from algorithmic behavioral specifications. The RTL design consists of a data path and a controller. During the high level synthesis process, the behavioral specification goes through many transformations, until the RTL design is produced. The synthesis system generates a control-data flow graph (CDFG) from the behavioral specification and goes through the steps of scheduling, functional unit allocation, register allocation, interconnect allocation and controller generation.

Formal verification systems in general, can usually verify only a small subset of synthesized designs. This is mainly due to the fact that for RTL designs to be verifiable, they should be generated using certain subsets of the synthesis algorithms or transformations. To expand the class of 'verifiable' designs, the verification systems should cater for a larger range of these algorithms. In particular, most verification methods often consider the synthesized designs which have gone under any degree of register optimization, 'unverifiable'. This is for the reason that in the absence of the register optimization, the relation between the variables of the specification and the corresponding data-path registers is bijective. After register allocation phase, this relation is no longer bijective and the transformed RTL design bears no obvious relation to its original specification. However, most industrial high-level synthesis tools use sophisticated register allocation and optimization algorithms during the synthesis. If the use of such algorithms are prohibited to ease the verification task, the synthesized RTL designs will be unrealistically simple. A detailed study of the algorithms used in register allocation phase persuaded us that even though the relation between the elements of the design (variables, control flow of the operations, etc.) before and after register optimization is not obvious, it can be precisely defined and used during the verification exercise.

Research in formal verification of synthesized designs, can be classified as transformation-based synthesis (formal synthesis) and post-synthesis verification. In formal synthesis, the transformations which may be carried out during the synthesis process, are mathematically proven correct [1–7]. Unfortunately, even though it guarantees designs that are correct by construction, transformation synthesis is largely interactive. In post-synthesis verification the correctness of a synthesized design, with respect to its specification, is mathematically established [8–12]. This approach is attractive since verification process is more susceptible to automation, but it has a few drawbacks: it is severely limited by design size, and often, demand computing resources that increase exponentially with design size.

In this paper, we present a post-synthesis methodol-

[*]This work is sponsored in part by DARPA and monitored by US Army Ft. Huachuca under contract number DABT63-96-C-0051.

ogy based on theorem proving for formally verifying the correctness of the RTL designs generated through high-level synthesis. This methodology is developed by considering three desired characteristics in a verification approach: (1) automation, (2) effectiveness with regard to the size of verifiable designs, and, (3) effectiveness with respect to the complexity of verifiable designs; so, it achieves the advantages of both classes of verification methods, while avoiding their drawbacks. We show that in the verification of automatically synthesized designs, functions and constant symbols can be left uninterpreted. This leads to efficient verification, based mostly on rewriting strategies, and as a consequence, designs of larger size can be verified. Moreover, the necessary rewriting steps can be automatically generated by making use of the variable and state binding information. We proposed a similar approach for verification of register level designs in [13], where verification can be integrated with synthesis systems which perform little or no register optimization. We extend the class of verifiable designs by accommodating various register allocation-optimization schemes. We focus on transformations which may be performed at register allocation phase of synthesis and develop our verification strategies in parallel to them. These strategies can therefore be applied to synthesized designs that have undergone any amount of register optimization. A completely automated Correctness Condition Generator, CCG, based on the methodology outlined above, has been developed. CCG generates the proof of the correctness of the implementation design, which may then be verified by the proof checker of the PVS theorem proving system [14].

2 Formal Models of Specification and Implementation

The nature of abstract specification - which is a description of the behavior - and the structural implementation - which is an architecture realizing that behavior - are inherently different. A common ground for comparison of these very different designs should be found. We made the observation that the flow of the operations described by specification can be captured by a finite state machine. The controller of an RTL design (which controls the flow of the register transfer operations) is usually represented by a finite state machine also. Modeling the specification and implementation as state machines provides the required common ground for comparing them.

A *behavior automaton* models the behavioral specification (Figure 1). This automaton is an extended finite state machine which is represented as a five tuple, $(S_b, S0_b, R_b, \delta_b, \sigma_b)$. In this model S_b is the set

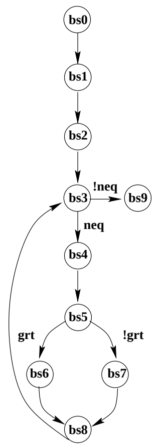
```
variables a, b, neq, grt;

a := read_input();
b := read_input();
neq := (a $\neq$ b);

while neq do
    grt := (a > b);
    if grt then
        a := a - b;
    else
        b := b - a;
    endif;
    neq := (a $\neq$ b);
enddo;

write_output( a );
```

(a) **Algorithmic Specification**



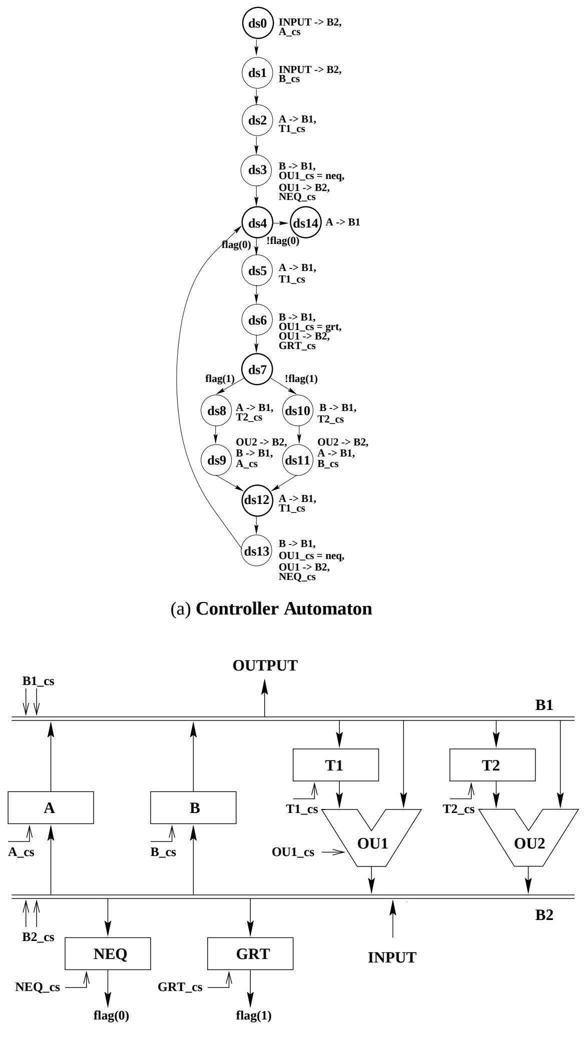
(b) **Behavioral Automaton**

Figure 1. A Behavior Specification

of states and $S0_b$ is the initial state of the automaton. A state may be an *assignment* state or a *conditional* state. An assignment state is annotated with one or more assignment statements and has exactly one outgoing transition to a *next* state. A conditional state is not annotated with any assignment statements and has exactly two outgoing transitions, one of which is labeled with the condition v and the other is labeled with the condition $\neg v$, where v is a specification variable. $R_b = \{r_{b_1}, r_{b_2}, \cdots, r_{b_m}\}$ is the set of behavior registers or specification variables. The domain D_{r_b} defines the set of all possible values that any behavior register can assume. $\sigma_b : S_b \times \overbrace{D_{r_b} \times D_{r_b} \times \cdots \times D_{r_b}}^{m} \mapsto S_b$ is the next state or state transition function, which defines the set of possible state transitions based on the value of behavior registers at the current state. $\delta_b : S_b \times S_b \times R_b \mapsto D_{r_b}$ is a symbolic function which defines the value of behavior registers after each state transition. In the example shown Figure 1, $S_b = \{bs_0, bs_1, bs_2, bs_3, bs_4, bs_5, bs_6, bs_7, bs_8, bs_9\}$, $S0_b = bs_0$, and $R_b = \{a, b, neq, grt\}$.

An RTL implementation consists of a controller and a data-path. The controller is a finite state machine which interacts with the data path through *control signals* and *flags*. The controller extended finite state machine models the implementation. This automaton is represented with a five tuple $(S_d, S0_d, R_d, \delta_d, \sigma_d)$. S_d, $S0_d$, R_d, δ_d and σ_d are defined similar to their counterparts in specification model. The states of the controller also, are of two types: *assignment* and *conditional*. Conditional transitions of the controller are labeled with data-path flags and no registers will be loaded in conditional states. Figure 2 shows the RTL implementation of the behavior specification shown in Figure 1.

In this example $S_d = \{ds_0, ds_1, \cdots, ds_{14}\}$, $S0_d = ds_0$, $R_d = \{A, B, NEQ, GRT\}$.



(a) **Controller Automaton**

(b) **Data Path**

Figure 2. Example of a register level design generated by a high-level synthesis system

3 Outline of the Verification Method

A correct implementation performs the operations that its specification describes. Assuming that equivalent results are generated by equivalent operations, and considering the fact that specification variables hold the results of the specification operations and RTL registers hold the results of the implementation operations, by comparing the values of certain critical specification variables (registers) in certain critical behavior states with those of certain critical RTL registers in certain critical controller states, the correctness of the implementation can be verified.

The critical states are the points of comparison or the check points of the designs. We should identify the states of the behavior automaton which act as border points (where no code motion across these points is allowed), as critical states. These states introduce control flow dependencies into the CDFG. These check-points are selected so that sufficient comparisons are made to decide about the equivalence of a specification and its implementation. The decision on what states should be considered critical is the result of a detailed study of the scheduling algorithms and the transformations which may be performed during the scheduling process, and is not discussed in this paper.

The critical variables are the objects of comparison and the success of our verification approach relies on proper identification of these variables. A thorough study of the register allocation phase, and the transformations which may be performed at this stage (for register allocation or optimization) is necessary to decide which variables are critical for verification. The identification of critical variables is the subject of Section 5.

After defining the sets of critical specification variables and critical behavior states, their counterparts in the implementation model are defined as the sets of critical design registers and critical controller states. Determining the right pairing of variables and registers or behavior or controller states depends on the particular algorithms used in the synthesis process and requires help from the synthesis tool. By maintaining links with the elements of the behavior specification throughout the synthesis process, the high-level synthesis tool can generate detailed binding information, in the form of auxiliary information as a byproduct of the synthesis process. Many authors, Thomas et al. [15] for example, have described detailed methods to maintain the binding information during the synthesis process. In particular, we assume that the high-level synthesis tool can generate the mapping between critical variables in the specification and registers in the RTL data path and the mapping between the critical states in the behavior automaton and states in the RTL controller.

Critical variables, critical states and critical state binding information play a key role in our verification method. In the following section, we will revisit these elements and discuss our verification algorithm in a more formal setting.

4 Formalization of the Verification Method

Let $CR_b \subseteq R_b$ be the set of *critical variables* in the behavioral specification. Let $CS_b \subseteq S_b$ be the set of critical states of the behavioral automaton. We assume that every critical state is reachable from the start state

$S0_b$. Let CP_b be the set of critical paths [1] among these critical states. For any critical path $p \in CP_b$, $F_b(p)$ and $L_b(p)$ denote the originating and terminating states of p. The counterparts of these sets and functions (CR_d, $CS_d, CP_d, \cdots$) in the implementation model are defined similarly.

Following the previous discussion, we postulate that the high-level synthesis tool can produce, as a byproduct of the synthesis process, the following two mappings (called bindings in the synthesis terminology): $B_r : CR_b \times CP_b \rightarrow CR_d$, is the *critical register binding*, and, $B_s : CS_b \rightarrow CS_d$ is the *critical state binding*. The start state of the behavior FSM is always mapped to the start state of the controller FSM, that is, $B_s(S0_b) = S0_d$.

The above definition of B_r is *dynamic* - the register binding may change across the critical paths and the same behavior register may be mapped to different design registers along different paths. In contrast we might have *static register binding* $B_{r_s} : CR_b \rightarrow CR_d$. This function maps each specification variable to the same design register along all the critical paths. Static register binding will be discussed in detail in Section 5. In addition to these two bindings, we define a particular binding B_{r_0} called the *initial register binding*. This function defines the mapping between the behavior and design registers, at the initial states of the two FSMs ($S0_b$ and $S0_d$) and is different from the previously defined functions in that the binding is defined at particular states, rather than along the critical paths. Static register binding can be defined in terms of initial register binding: $\forall r_b \in CR_b : B_{r_s}(r_b) = B_{r_0}(r_b)$.

From B_s we can easily derive another mapping $B_p : CP_b \rightarrow CP_d$ which is the *critical path binding*. A critical path $p_b \in CP_b$ is mapped to critical path $p_d \in CP_d$ if and only if their originating and terminating states are mapped by B_s and the transition conditions, if any, on the outgoing transitions of their originating states match. More formally, $B_p(p_b) = p_d$ if and only if $B_s(F_b(p_b)) = F_d(p_d)$, $B_s(L_b(p_b)) = L_d(p_d)$, and if v ($\neg v$) is the condition variable annotation on the originating transition of p_b then $B_r(v, p_b)$ ($\neg B_r(v, p_b)$) is the condition register annotation on the originating transition of p_d. This ensures that if p_b is traversed in the behavior FSM, p_d will be traversed in the RTL controller.

An *execution path* of the behavior is a finite sequence of critical states $[s_{b_1}, s_{b_2}, \cdots, s_{b_i}, s_{b_{i+1}}, \cdots]$ such that the first state in the sequence is the start state $S0_b$ and any two successive states in the sequence form a critical path, that is, $\forall i > 1, \exists p \in CP_b : b_{s_i} = F_b(p)\ and\ b_{s_{i+1}} = L_b(p)$. Execution path of the RTL controller is similarly defined. Let EP_b denote the set of all possible behavioral execution paths and EP_d denote the set of all possible RTL execution paths. We can construct an execution path binding, $B_e : EP_b \rightarrow EP_d$ using the critical state binding as follows: If $e = \{s_{b_1}, s_{b_2}, \cdots, s_{b_i}, s_{b_{i+1}}, \cdots\}$ is an execution path in the behavior automaton, then $B_e(e) = \{B_s(s_{b_1}), B_s(s_{b_2}), \cdots, B_s(s_{b_i}), B_s(s_{b_{i+1}}), \cdots\}$. The last state in an execution path e is called the *termination state* of e and denoted by $T_b(e)$ for the behavior automaton and by $T_d(e)$ for the RTL controller. The final two states of an execution path e define the terminating critical path or the *tail* of execution path, which is denoted by $Tail_b(e)$ for the behavior automaton and $Tail_d(e)$ for the RTL controller.

For the purposes of defining co-execution equivalence, we postulate an uninterpreted domain of *values*, V. These values can be 'stored' in behavioral variables as well as RTL registers. We postulate two functions for assigning values to critical variables and critical registers: $V_b : EP_b \times CR_b \rightarrow V$ determines the value of a critical variable r_b when the behavioral automaton traversed the execution path e and reached the state $T_b(e)$. $V_d : EP_d \times CR_d \rightarrow V$ is similarly defined. In the next section we show axiomatic definitions of V_b and V_d suitable for symbolic manipulation, automatically generated from behavioral specifications and RTL descriptions respectively.

We are now ready to define various equivalence relationships between the behavior and the RTL design. We say that the initial state $S0_b$ in behavior automaton is *equivalent* to the initial state $S0_d$ in the controller provided $\forall r \in CR_b : V_b([S0_b], r) = V_d([S0_d], B_{r_0}(r))$. We denote initial state equivalence by $S0_b \equiv S0_d$.

We say that a critical state s_b in the behavior and the RTL critical state $B_s(s_b)$ are *equivalent* to each other provided $S0_b \equiv S0_d \Rightarrow \forall e \in EP_b : s_b = T_b(e), \forall r \in CR_b : V_b(e, r) = V_d(B_e(e), B_r(r, Tail_b(e)))$. We denote state equivalence by $s_b \equiv B_s(s_b)$.

We say that a behavioral execution path e is *equivalent* to the RTL execution path $B_e(e)$ provided $S0_b \equiv S0_d \Rightarrow \forall s_b \in e, s_b \equiv B_s(s_b)$. We denote execution path equivalence by $e \equiv B_e(e)$.

We say that the RTL design is *equivalent* to the behavior specification provided $\forall e \in EP_b, e \equiv B_e(e)$.

We say a behavioral critical path p is *equivalent* to the RTL critical path $B_p(p)$ provided $F_b(p) \equiv F_d(B_p(p)) \Rightarrow L_b(p) \equiv L_b(B_p(p))$. We denote critical path equivalence by $p \equiv B_p(p)$.

We claim that critical path equivalence implies execution path equivalence per the following theorem, offered here without proof:

Theorem: If every critical path in the behavior is

[1] We define a *critical path* as a directed path from a critical state to another critical state without traversing through any other critical state.

equivalent to the RTL critical path to which it is bound during the synthesis process, then the RTL design is equivalent to the behavior specification. Formally, $\forall p \in CP_b, p \equiv B_p(p) \Rightarrow \forall e \in EP_b, e \equiv B_e(e)$.

The proof of this statement is straight forward and follows from the fact that, in both behavioral automaton and the RTL controller, the critical states are reachable from the respective start states. The proof is based on induction on the length of the execution paths.

5 Verification with Various Critical Register Binding Schemes

A predominant concern in real high-level synthesis systems is resource sharing. The goal of high-level synthesis is to determine constraint-satisfying sharing of ALUs, registers and interconnections. To facilitate resource-sharing, high-level synthesis tool performs scheduling which permits time sharing of resources whose life times do not overlap across the scheduled time-scale. Operator, register and interconnect allocation algorithms, which follow the scheduling step, are typically based on clique partitioning or graph coloring following life-cycle analysis of the scheduled flow graph.

The goal of register optimization is to share registers whose lifetimes do not overlap across the scheduled time-scales. This is done by life cycle analysis of the design registers.

Two types of register allocation-optimization schemes are commonly found in high-level synthesis tools: *value based* and *carrier based*. When transformations based on these schemes are performed during the synthesis process, the binding relation between the specification variables and RTL registers is no longer bijective. Since the definition of equivalence directly depends on critical register binding information, appropriate register binding functions need to be defined precisely and the verification strategies should be adjusted accordingly. We will show that the register binding information under various register allocation and optimization schemes can be obtained and consequently the verification method is extendable to RTL designs which have undergone any degree of register optimization.

5.1 Register Allocation with No Optimization

In a simplistic synthesis process, with no register optimization, all critical variables are preserved by the HLS tool and manifest in the RTL design in the form of critical registers. In this case, there is a one to one mapping between the specification variables and design registers (Figure 3). As the upper limit, all the specification variables can be marked as critical. The comparisons between all the critical variables and their corresponding RTL registers are 'sufficient' to validate the correctness of the implementation, even though they may not be 'necessary'. As the lower limit all the output variables (variables which directly write to output) can be marked as critical. A static register binding function $B_{r_s} : CR_b \mapsto CR_d$ is defined since the register binding function does not vary along different critical paths of the designs.

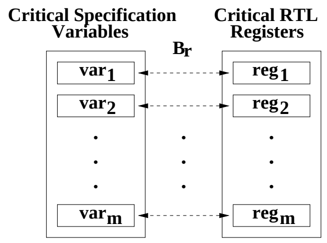


Figure 3. Register Allocation with No Optimization

5.2 Carrier Based Register Allocation

The carrier based register allocation scheme yields a mapping from the variables to the registers. FACET [16], HAL [17], and CHARM [18] use carrier-based techniques for register optimization. Register optimization is only possible when two (or more) variables, have non-overlapping lifetimes, in which case they are bound to the same register. Therefore the mapping from the specification variables to RTL registers is a many to one relation (Figure 4). For example if carrier based register allocation was performed in generating our example design of Figure 2, in the resulting data-path the two variables neq and grt would have bound to the same register $NEQ - GRT$ instead of two registers NEQ and GRT.

Since the register allocation algorithm guarantees that only variables with non-overlapping life-times can be mapped to the same register, at any state at most one variable (the one that is live) may be mapped to its corresponding register. A static register binding function in this scenario is not appropriate, since the binding holds at points where a variable is live, and does not hold at the others.

A variable can be critical only at those states that it is alive. This means that a variable may be critical along certain critical paths and not critical along the others. Therefore, the binding function depends on the critical paths as well as the variables, i.e. $B_r : CR_b \times CP_b \mapsto CR_d$. This binding function is referred to as a *dynamic register binding function*. It uniquely maps each criti-

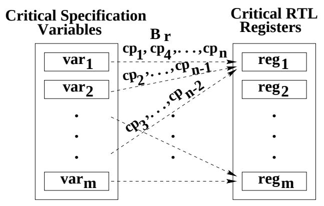


Figure 4. Carrier Based Register Allocation

cal variable to an RTL register along a critical path. The synthesis tool can generate this binding information as a byproduct of synthesis. Since some of the critical registers are not live across all critical states when this register allocation scheme is used as part of the synthesis process, verification is performed by *criticality masking technique*. Assuming that a dynamic register binding function is defined and that criticality masking is performed during the verification process, the formalizations presented in Section 4 will be valid.

5.3 Value Based Register Allocation

In value-based approach, register optimization is modeled as the problem of mapping data values produced and used by operations in a data flow graph representation of the specification into registers. Register optimization is only possible when two (or more) operations use the same data values. Considering the partial specification of Figure 5, a value based register allocation scheme may assign register R_1, R_2, R_3 and R_4 to the values $B+1$, $A+1$, $B+2$ and $A+2$ respectively. So, if we assume that all the variables of this partial specification are critical, then along the critical path $[\cdots, S_3, S_4, \cdots]$, the specification variable A is mapped to the register R_1 and along the critical path $[\cdots, S_7, S_8, \cdots]$ it is mapped to R_3. Many high-level synthesis systems such as REAL [19], EMUCS [20] and EASY [21] are the systems which use value-based register optimization techniques. Register optimization problem can be modeled as a channel routing problem; the life span of each value is modeled as a net interval. The minimum number of tracks corresponds to the number of registers needed.

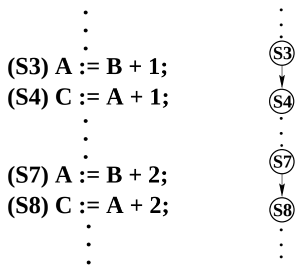


Figure 5. Example of a partial specification

It is obvious that a variable during its lifetime, or even at different states of the same critical path, may assume different values. Also, it is possible that the same value is assigned to different variables. The mapping from specification variables to RTL registers is a 'many to many' relation (Figure 6). In this case also, a static register binding function is not appropriate for specifying this relation. In order to define the critical binding function, we investigated how the elements of verification are affected under this scheme. Unlike the carrier-based register allocation scheme, the critical variables do not uniquely correspond to the RTL registers along each critical path. But we made the observation that each variable at the final state of a critical path, holds the last value assigned to it, and it is this value which will be compared to the value of a corresponding register during the verification process. Therefore, each variable can be uniquely mapped to the register corresponding to the last value assigned to it, along a critical path, and the binding function can be defined according to this rule. Also we noted, the fact that multiple variables may be bound to the same register along a critical path, does not affect the register binding function. A dynamic critical register binding function $B_r : CR_b \times CP_b \mapsto CR_d$ like the case of carrier-based register allocation can specify the binding between the variables and the registers. Considering this dynamic binding function, the formalizations of section 4 are still valid in this case.

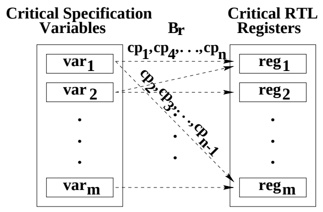


Figure 6. Value Based Register Allocation

6 Correctness Condition Generator

In this section, we discuss how the proof of correctness of the designs is automatically generated by our correctness condition generator, CCG. We assume that the operating environment of the designs ensures that $S0_b \equiv S0_d$. Typically, the environment ensures that all the data and control registers are reset at the start states. The goal of our proof effort is to show that each critical path in the behavior is equivalent to its corresponding critical path in the structure. We determine this by using symbolic term rewriting in a higher-order logic theorem prover.

Our proof effort is carried out in the PVS theorem prover environment [14]. We modified the high-level

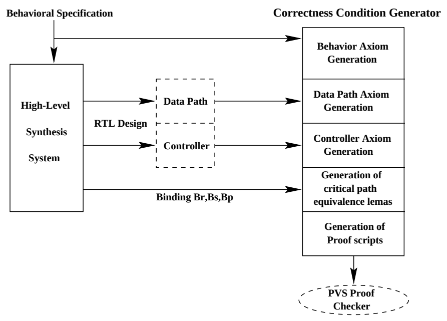

Behavioral Specification
Correctness Condition Generator
High-Level Synthesis System
Data Path
RTL Design
Controller
Binding Br,Bs,Bp
Behavior Axiom Generation
Data Path Axiom Generation
Controller Axiom Generation
Generation of critical path equivalence lemas
Generation of Proof scripts
PVS Proof Checker

Figure 7. Stages of Correctness Condition Generation

synthesis system DSS [22] to generate the three bindings B_r, B_s and B_p. These bindings along with the behavior specification and the RTL design are the inputs to the Correctness Condition Generator (CCG) shown in Figure 7. The CCG has five steps during which the following theories are generated: (1) behavior axioms; (2) data path axioms; (3) controller axioms; (4) critical path correctness lemmas; and, (5) proof scripts for each correctness lemma. Each of these steps will be discussed in this section. Due to lack space, the PVS model could not be included here.

A set of general definitions exist in CCGs axiom library, based on that, CCG generates a set of declarations specific to the design under investigation. This set of declarations and axioms define information about the behavior specification and RTL design. This information includes specification variables and the RTL component declaration, critical path specification for both behavior and RTL automata, and binding functions B_r, B_s and B_p.

Behavior Axiom Generation - This step examines the behavior specification, written in a simple subset of VHDL in our case, and converts it into a series of axioms that collectively specify the value transfers in the behavior. For each state transition in the behavior design one axiom is generated. This axiom specifies the value of each specification variable, at the destination state of the transition, in terms of the values of specification variables prior to transition.

Data Path Axiom Generation - A pre-existing library of axioms defines the behavior of each type of RTL component. The input of a component can be the output of some other component or a primary input. Each library axiom specifies the operation of each component by defining its input-output relation, at each state. The value at the output of a component at a particular state is defined in terms of the data and control inputs of the component at that state, or its output at a previous state in the case of sequential components.

At the second stage of correctness condition generation, the data-path of the synthesized RTL design is modeled as a PVS theory [23]. For each component of the data-path, an axiom is generated, which specifies its type, and its interface with the rest of the components. The data-path axioms together with the axioms in the component library define the behavior and interface of each individual component. Also, the interconnection of the control inputs of the RTL components with the controller, and the interconnection of the flags from the data-path to the controller are specified in this theory.

Controller Axiom Generation - A constructive model of the RTL controller is now generated. The functions σ_d, and $Control\_Signal$ are extracted from the controller description and converted to PVS functions. In function $Control\_Signal$, the value of control signals from the controller to data-path at each state of the controller is defined.

Generation of Critical Path Equivalence Lemmas - For each pair of the behavior-RTL critical paths which are bound by the function B_p, a set of lemmas are generated. A general lemma states that if the initial states of the pair of the critical paths are *equivalent*, their final states should be *equivalent*. A set of sub-lemmas (one sub-lemma for each live specification variable v) state that if the initial states of the pair of critical paths are equivalent, then the values stored in the live specification variable v and its RTL counterpart $B_r(v)$ at the final states of the critical paths are the same. The proof of these sub-lemmas together complete the proof of the main lemma for the critical paths.

Generation of Proof Scripts - Generation of proof scripts is the most elaborate stage in CCG. In this stage all the information about the designs is processed and the rules for proving each lemma are produced. Proof scripts are generated by performing symbolic analysis on the behavior operations and RTL register transfers along critical paths. These proofs are then subjected to verification by the PVS proof checker. These proofs make extensive use of symbolic rewriting, involving instantiation of definitions, axioms and other proven lemmas.

7 Implementation and Results

The method discussed in the previous sections has been implemented in a correctness condition generator module integrated with the DSS [22] high level synthesis system. DSS has been in development for about ten years and is relatively mature. DSS accepts behavioral specifications in VHDL and generates RTL designs also in VHDL. Using parallel synthesis algorithms, DSS searches through vast regions of design space. DSS uses enhancements of force-directed list scheduling and a hierarchical clique partitioning algorithm for register allocation. DSS has been used to generate numerous designs both in the university and industry and has been throughly tested using systematic benchmark development, test generation and simulation.

Figure 7 shows the integration of the correctness condition generator (CCG) with the DSS system as explained in the previous section. The CCG component of DSS is experimental to help us determine how much of the verification effort can be automated and further develop the techniques discussed in this paper. A major limitation of the verification condition generator currently is that it can handle a smaller subset of VHDL than that can be synthesized by DSS. The modified DSS system with this generator produces a PVS file containing declarative specifications of the behavior and data path and constructive specifications of the controller. In addition, it produces all of the critical path equivalence lemmas and proof scripts to prove these lemmas. The PVS theories generated are not necessarily very elegant, but are amenable to completely automated verification. PVS system is used to execute these scripts automatically. No manual interaction is necessary to conduct the proof and inspection is necessary only in the event of a failure.

References

[1] Srinivas Devadas, Hi-Keung Tony Ma, Richard Newton, ”On Verification of Sequential Machines at Differing Levels of Abstraction”, IEEE Transactions on Computer-Aided Design, June 1988.

[2] Michael McFarland, “An Abstract Model of Behavior for Hardware Descriptions”, IEEE Transactions on Computers, July 1983.

[3] Steven Johnson, “Synthesis of Digital Designs from Recursion Equations”, MIT Press, Cambridge, 1984.

[4] Ranga Vemuri, “On the Notion of Normal Form Register-Level Structures and Its Applications in Design-Space Exploration ”, European Design Automation Conference, March 1990.

[5] Sreeranga Rajan, “Correctness Transformations in High Level Synthesis: Formal Verification”, Proceedings of the International Conference on Computer Hardware Description Languages, Japan, August 1995.

[6] N. Narasimhan, R. Vemuri, ”On the Effectiveness of Theorem Proving Guided Discovery of Formal Assertions for a Register Allocator in a High-Level Synthesis System”, to appear in The 11th Conference on Theorem Proving in Higher Order Logics (TPHOL’98), September 1998.

[7] N. Narasimhan et al, ”Theorem Proving Guided Development of Formal Assertions in a Resource-Constrained Scheduler for High-Level Synthesis”, ICCD’98, October 1998.

[8] Luc Claesen, Mark Genoe, Eric Verlind, Frank Proesmans, Hugo De Man, “SFG-Tracing: A Methodology of Design for Verifiability”, Proceedings of Advanced Workshop on Correct Hardware Design Methodologies, North-Holland, 1991.

[9] Francisco Corella, “Automated High-Level Verification Against Clocked Algorithmic Specifications,” Proc. Computer Hardware Description Languages and Their Applications, April 1993.

[10] M. K. Srivas and S. P. Miller, “Formal Verification of the AAMP5 Microprocessor,” Chapter 7 in Industrial Applications of Formal Verification.

[11] E.M. Clarke, E.A. Emerson, A.P. Sistla, “Automatic Verification of Finite-State Concurrent Systems using Temporal Logic Specifications”, ACM Trans. Prog. Lang. Syst., pp. 244-263, 1986.

[12] Kenneth L. McMillan, “Symbolic Model Checking: An Approach to the State Explosion Problem” Carnegie Mellon University, 1992.

[13] Nazanin Mansouri, Ranga Vemuri, “A Methodology for Completely Automated Verification of Synthesized RTL Designs and Its Integration with a High-Level Synthesis Tool”, to appear in International Conference on Formal Methods in Computer-Aided Design, Palo Alto, CA, 1998.

[14] N. Shankar, S. Owre and J. M. Rushby, ”The PVS Proof Checker: A Reference Manual (Beta Release)”, March 1993.

[15] D. E. Thomas, R. L. Blackburn, and J. V. Rajan, “Linking the Behavioral and Structural Domains of Representation for Digital System Design”, IEEE Trans. CAD, vol. CAD-6, pp. 103-110, January 1987.

[16] C.Tseng, D.P. Siewiorek, ”Facet: A Procedure for the Automated Synthesis of Digital Systems”, 20th ACM/IEEE Design Automation Conference, pp. 490-496, 1983.

[17] P.G. Pualin, J.P. Knight, E.F. Girczyc, ”HAL: A Multi-Paradigm Approach to Automatic Data Path Synthesis”, 24th ACM/IEEE Desgin Automation Conference, pp 263-270, June 1986.

[18] Nam-sung Woo, ”A Global, Dymanic Register Allocation and Binding for Data Path Synthesis System”, 27th Design Automation Conference, pp. 505-510, 1990.

[19] F. Kurdahi, A. Parker, ”REAL: A Program for REgister ALlocation”, 24th Design Automation Conference, pp. 210-215, 1987.

[20] D.E. Thomas C. Y. Hitchcock III, T.J. Kowalski, J.V. Rajan, A. Walker, ”Automatic Data Path Synthesis”, IEEE Computers, pp. 59-70, Dec. 1983.

[21] L. Stok, R. Van Den Born, ”EASY: Multiprocessor Architecture Optimiztion”, Proceedings International Workshop on Logic and Architecture Synthesis for Silicon Compilers, pp. 313-328, 1998.

[22] J. Roy, N. Kumar, R. Dutta, R. Vemuri, “DSS: A Distributed High-Level Synthesis System”, IEEE Design and Test of Computers, June 1992.

[23] S. Owre, N. Shankar, J. M. Rushby, “The PVS Specification Language (Beta Release)”, June 1993.