A Symbolic Approach for Checking Functional and Timing Compatibility of Synthesized Designs

Chih-Tung Chen and Alice C. Parker

CENG Technical Report 93-26

Department of Electrical Engineering - Systems
University of Southern California
Los Angeles, California 90089-2562
(213)740-4476

May 1993

A Symbolic Approach for Checking Functional and Timing Compatibility of Synthesized Designs.

Chih-Tung Chen and Alice C. Parker

May 24, 1993

Abstract

In a redesign situation, the compatibility of the replacement chips is critical. This can only be obtained by verifying both their functionality and timing. In this report, we will postulate that the general RTL verification problem is too difficult to be solved. Hence, it is necessary for a RTL verification system to trade off generality for usability. In fact, there is a real need for an automatic tool that can check the functional and timing compatibility of automatically *synthesized* chips quickly and effectively. We found that synthesized designs have several common properties that can be utilized to facilitate this task. By making use of the USC design data structure (DDS) graph models, we developed a hybrid symbolic approach that will not only verify functional compatibility but also take into account the timing and the interaction between the controller and the datapath. Experiments have been performed on two designs, an AR filter and a robot arm controller synthesized by the ADAM system. The results are very encouraging. In fact, these experiments helped to identify problems with the early version of the control signal generator (CSG) software.

1 Introduction

In a redesign situation, the compatibility of the replacement chips is critical. This can only be obtained by verifying both their functionality and timing. In this report, we will postulate that the general RTL verification problem is too difficult to be solved. Hence, it is necessary for a RTL verification system to trade off generality for usability. In fact, there is a real need for an automatic tool that can check the functional and timing compatibility of automatically *synthesized* chips quickly and effectively. We found that synthesized designs have several common properties that can be utilized to facilitate this task. By making use of the USC design data structure (DDS) graph models [KP85], we developed a hybrid symbolic approach that will not only verify functional compatibility but also take into account the timing and the interaction between the controller and the datapath. Experiments have been performed on two designs, an AR filter and a robot arm controller synthesized by the ADAM system. The results are very encouraging. In fact, these experiments helped to identify problems with the early version of the control signal generator (CSG) software [WP92].

This report is organized as follows. We start with a brief discussion of the related work in Section 2. Next, we apply computation theory to show the *undecidability* of the general RTL verification problem in Section 3. Hence, We analyze a high-level synthesis model in Section 4 to identify properties of automatically synthesized designs. In Section 5, we give an overview of our approach for the compatibility checking of synthesized designs. The hybrid symbolic simulation model used in our approach will be described in Section 6. In Section 7 we will show that there is an *isomorphic* property between the behavioral specifications and the extracted behaviors of the corresponding implementations. A behavior-comparison procedure based on this property will then be given. Finally, some experiment results are given in Section 8

2 Related Work

Compatibility checking is essentially a verification problem. There are a variety of approaches for design verification (see [CP88, McFb] for the survey). However, formal methods are still not practical at present and simulation-based ones have their theoretical limitations. We found that the RLEXT system by Knapp and Winslett [KW89] is the only one which can achieve some of our objectives. RLEXT is a rule-based system which relies on a formal model derived from the USC DDS. It allows the user to modify the design structure and then check the consistency of the design with the ability to repair some design-rule violations automatically. This feature can be considered as a form of partial redesign. Their approach, however, does not mention the control logic, nor the interaction between the datapath and the controller. How the clocking scheme and other important physical parameters such as wiring delays are taken into account is not described.

In our research, symbolic simulation is utilized to check the functional and timing compatibility of the *synthesized* chips. The idea behind symbolic simulation is to evaluate circuit behavior over expanded sets of signal values so that a number of operating conditions can be simulated in a single run. In late 1970's, researchers at IBM first applied symbolic simulation to hardware verification at the register-transfer level [Dar79]. The research activities [Cor81] on this problem only lasted till the early 1980's due to the weakness of the algebraic manipulation and the non-determinacy caused by conditional and looping constructs [Bry90]. Although this form of symbolic evaluation is not powerful enough for general functional verification at the RT level, we will show in this proposal that it can be made very useful when applied to the synthesis domain where both the functional behavior and timing of a design are equally important and the implementation is derived from the specification in a well-defined manner.

3 The General RTL Verification Problem

The general RTL verification problem can be described as follows:

Given:

- a behavioral design specification S, and
- an RTL implementation I.

Goal:

show the behaviors B_S and B_I of S and I respectively are *functionally equivalent.*

The specification S specifies what the system should do, and it is usually described functionally in some HDL (Hardware Description Language). On the other hand, the implementation I, typically described by a netlist, models the system structurally as an interconnection of RTL components. The verification task is to show that, for all feasible inputs, B_I is equivalent to B_S required by the specification S [McFa]. The behaviors B_S and B_I are regarded as the way the system or its components interact with their environment, i.e., the mapping from inputs to outputs [MPC88]. However, if the system possesses sequential behavior[1], B_S and B_I describe the mapping from "sequences" of inputs to "sequences" of outputs. Therefore, the definition of equivalence of two behaviors has to be modified accordingly.

Since S and I are given in different languages, it is necessary to translate both of them into a model in which their behavior can be compared. For example, we can simulate both S and I using their respective simulators for all feasible inputs, in theory, to get B_S and B_I and verify their output correspondence. Alternatively, we can describe both S and I in a formalism in which a theorem-prover can be used, hopefully, to prove that B_S and B_I are equivalent. Whichever way we use, a "reasonable" verification model representing the behaviors must be capable of expressing every design in the functional model of S and in the RTL model of I. Figure 1 shows the relationships between the verification model and the associated functional and RTL models.

[1]The output at any point depends on the current state which in turns relies on the past history of inputs [McFb].

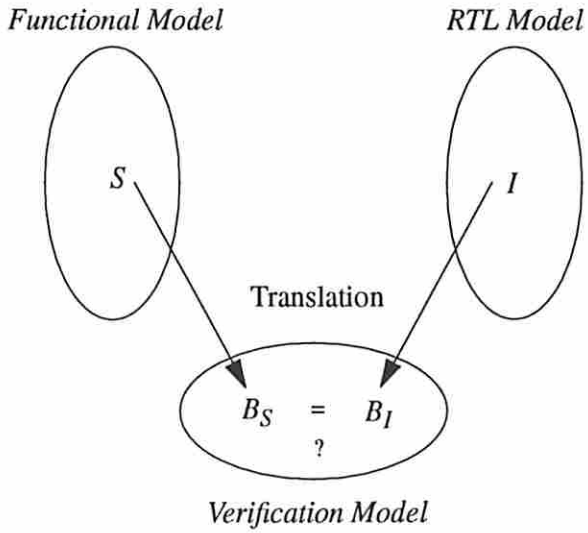


Figure 1: The verification model and its relationships with the functional and RTL models.

Definition 3.1 *A verification model is called* complete *if for each instance ϕ in the functional/RTL model there exists a counterpart ψ in the verification model such that ϕ and ψ are functionally equivalent.*

Definition 3.2 *A* complete *verification model is* feasible *if we can find a translation from the associated functional/RTL model to it.*

In other word, a *complete* verification model is at least as powerful as the functional and RTL models in terms of the expressive capability, and the model is *feasible* (useful) only if we can effectively translate all the designs into it.

If we do not consider the fact that I is derived from S, B_S and B_I simply correspond to two independent points in a *complete* verification model. Let Ω_V be the *complete* verification model. Solving the general RTL verification problem is then equivalent to finding a decision function f such that for all $\psi_i, \psi_j \in \Omega_V$

$$f(i,j) = \begin{cases} 1 & \text{if } \psi_i \equiv \psi_j \\ 0 & \text{otherwise} \end{cases} \tag{1}$$

In what follows, we will show that it is not possible to find such a decision function no matter which complete verification model we use. However, some background computation theory needs to be introduced first.

Informally, a function is a *partial recursive function (prf)* if it is effectively "computable" [MY78]. In other words, given a definition of a partial recursive function we can produce an algorithm, e.g. write a RAM program, to compute it. A *programming system* is a list of programs $\phi_0, \phi_1, \ldots$ which includes all of the *prfs*. A programming system is an *acceptable programming system (APS)* if and only if the following conditions are met.

- For every *prf* f, there exists an index i such that $\phi_i \equiv f$. That is, there is at least a program for each *prf*.
- For all indices i, ϕ_i is a *prf*. In other words, every program is a *prf*.
- There exists a *universal* program ϕ_u such that for all i and x $\phi_u(i, x) = \phi_i(x)$, where x is the argument over N (natural numbers).
- There exists a total recursive function c such that $\phi_{c(i,j)} = \phi_i \circ \phi_j$ for all i and j, where $\circ$ denotes function composition.

In fact, any "reasonable" programming system will satisfy the definition of an APS. Hence, any results applied to APSs should also hold for all "reasonable" programming systems, and certainly for all existing general purpose programming languages [MY78].

Now, we can introduce the notion of undecidable (algorithmically unsolvable) problems concerning APSs. Let N be the set of natural numbers.

Definition 3.3 *For all $S \subseteq N$, the function $C_S : N \to \{0, 1\}$ is called the characteristic function of S if and only if*

$$C_S(i) = \begin{cases} 1 & \text{if } i \in S \\ 0 & \text{otherwise} \end{cases}$$

Definition 3.4 *For all $S \subseteq N$, S is* decidable *if and only if C_S is a* prf.

The following lemma describes an important *undecidable* set which will be used to show the undecidability of the general RTL verification problem.

Lemma 3.1 *For all APS* $\{\phi\}$, *the set* $V = \{< i, j >| \phi_i \equiv \phi_j\}$[2] *is undecidable.*

The proof method can be found in [MY78]. Basically, Lemma 3.1 says that there is no algorithm for deciding whether or not two arbitrary programs in any APS are equivalent.

Finally, we can establish the main result of this section by the following theorem.

Theorem 3.2 *The general RTL verification problem is* algorithmically unsolvable *for all* complete *verification models.*

Outline of Proof: Recalling earlier discussion, we know that solving the general RTL verification problem is equivalent to finding a decision function f as defined in Equation 1. Also, the general RTL verification problem can be represented by the following set:

$$V = \{< i, j >| \psi_i \equiv \psi_j \text{ for all } \psi_i, \psi_j \text{ in } \Omega_V\}$$

Hence, f is actually the characteristic function of V.

From Lemma 3.1, we know that V is undecidable for all APSs. Informally, any *complete* verification model Ω_V should satisfy the definition of an APS[3] since its associated functional model of design specifications is based on some hardware description language such as VHDL or HardwareC which certainly qualifies as a "reasonable" programming system. Hence, f is not a partial recursive function and the general verification problem is algorithmically unsolvable for all *complete* verification models. □

4 Properties of the Synthesized Designs

In high-level synthesis, the synthesis system actually derives the structural design from the behavioral specification in a *well-defined* manner [MPC88]. Consequently, the specification

[2] $<,>$ is any pairing function which can establish an effective one-to-one correspondence between the 2-tuples in $N \times N$ and the numbers in N. For example, $f(x, y) = 2^x(2y + 1) - 1$.

[3] A formal proof will need to show this argument mathematically. However, we feel that this enormous work will lead us out of focus of this research.

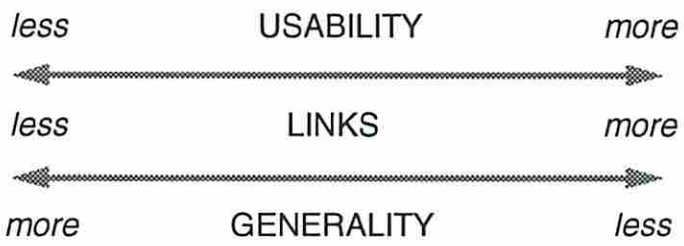


Figure 2: The tradeoff between generality and usability of verification systems

S and the implementation I are not independent. In fact, there are *links* between S and I. These links can be utilized in verifying their correspondence. Depending on the use of these links, a verification system, however, is trading off generality for usability as shown in Figure 2. This is because each such link required by a verification system represents an additional assumption (restriction) on the verification problem to be solved even though it may help to make the problem tractable. While this is inevitable, an obvious goal is to ask for only reasonably available links while achieving the ability to verify all the designs of interest.

In Figure 3, we show a generic model of high-level synthesis. In this model, the synthesis process is divided into two major steps - *transformation* and *mapping*. The transformation step performs a number of behavior-preserving transformations to "optimize" the initial behavioral specification $CDFG_S$. For example, tree-height reduction is a typical transformation for reducing the critical paths of a behavioral specification at the expense of additional computation [NP91]. Next, a structure I is synthesized at the mapping step according to the transformed behavior $CDFG_T$. The mapping step generally consists of three tasks; namely, *scheduling*, *data path allocation* and *controller synthesis* [MPC88]. The order of these tasks, however, is not important in our model.

Based on this model, the verification problem which we are describing can be divided into two subproblems:

1. Show $CDFG_S$ and $CDFG_T$ are functional equivalent.

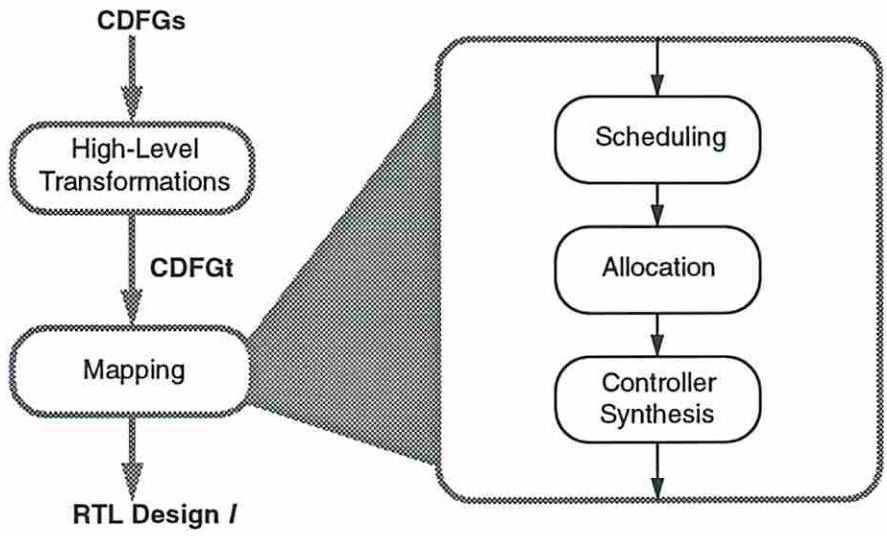


Figure 3: A high-level synthesis model

2. Show the RTL design I possesses the behavior $CDFG_T$.

The first subproblem requires us to show the functional equivalence of two CDFGs. If we do not acknowledge the fact that $CDFG_T$ is the result of a sequence of transformations on $CDFG_S$, this problem is still equivalent to the general RTL verification described in Section 3, which is undecidable. Even the problem which simply shows the equivalence of two finite-precision algebraic expressions is in neither P nor NP if NP $\neq$ co-NP.

Lemma 4.1 *Showing the equivalence of two finite-precision algebraic expressions is a co-NP problem which cannot be done in P nor NP time if NP $\neq$ co-NP.*

Proof: The proof is straightforward. First, we know that the *not tautology* problem[4] is an NP-complete problem and its complement, the *tautology* problem, is a co-NP problem which cannot be solved in P nor NP time if NP $\neq$ co-NP [GJ79].

The *tautology* problem can be easily reduced in polynomial time to the problem of showing the equivalence of two Boolean expressions since each instance of this problem is the same as

[4] A *tautology* is a Boolean expression that has the value 1 for all assignments of values to its variables. The *not tautology* problem is to return 1 if a given Boolean expression is not a tautology and return 0 otherwise.

showing whether or not the Boolean expression in question is equivalent to constant 1. The later problem in turn can be easily reduced to one which determines the equivalence of two finite-precision algebraic expressions since Boolean expressions are a subset of finite-precision algebraic expressions. Therefore, the latter problems are both co-NP problems which cannot be done in P nor NP time if NP $\neq$ co-NP. □

To solve this subproblem effectively, we must rely on the link between $CDFG_S$ and $CDFG_T$, which is a sequence of transformations $T_1, T_2, \ldots, T_k$. Obviously, $CDFG_S$ and $CDFG_T$ are functionally equivalent if $T_1, T_2, \ldots, T_k$ are all behavior-preserving. To achieve this, we could, in theory, prove the transformation algorithms used by the optimizer so that the transformations it made always preserved behavior. The advantage of this is the proofs would only have to be done once, rather than for every design. In practice, even if the abstract transformation algorithms could be verified, this would not guarantee that their implementations were correct. At present, we do not have the technology to verify them at the code level [McFb]. Alternatively, if we can prove each type of transformation abstractly once and in advance, we only have to check the conformity of the transformations $T_1, \ldots, T_k$ which actually have been done during the transformation step with proven transformation models. This approach still only requires us to do the proofs once, but the errors resulting from the incorrect implementations of transformation algorithms can be found. However, the checking has to be done for every design. Unfortunately, the verification of high-level transformations is beyond the scope of this research. Currently, we assume that this task has been carried out manually by applying some formal verification techniques[CP88].

In the second subproblem, we are asked to show that a structural design I possesses the behavior $CDFG_T$. The problem is that $CDFG_T$ specifies the dynamic relationship between sequences of inputs and outputs, while the implementation I is a static structure. Since they are built on different concepts, it becomes very difficult to prove their equivalence if they are regarded as two independent descriptions.

Fortunately, the design I is the result of a mapping from $CDFG_T$. The major tasks of this mapping involve assigning the operations to control steps (scheduling), assigning the

operations and values to hardware (data path allocation), and generating a controller to deliver the required control signals (control synthesis) [MPC88]. Consequently, the design I, if mapped correctly, will have the following properties:

Property 4.1 *For each operation op in $CDFG_T$, there exists a functional unit u in I such that u can be configured to perform op and op is achieved by directing all the input values of op to the corresponding input ports of u.*

Property 4.2 *For each data dependence (val, op) in $CDFG_T$, there exists an interconnect path in I between the source of val and the corresponding input port of the functional unit u which is designated to perform op. The source of val can be an output port of a functional unit or a storage element. The interconnect path is set up by using buses and/or switching devices.*

Property 4.3 *$CDFG_T$ defines the required computations which have to be done in I for every execution instance.*

We will show in Sections 6 and 7 how these properties can be effectively utilized to verify synthesized designs.

In addition, there exist links between $CDFG_T$ and I, and the information regarding these links is generally available after the mapping step. For example, this information is represented explicitly in DDS by means of *bindings*. We found that this information is particularly useful in diagnosing the design errors because these bindings represent the design decisions made during the mapping process and the events that should occur while the design is operating. If errors are found, the bindings which did not occur during the simulation of the implementation can be traced to determine the cause.

5 Approach Overview

In order to produce an automatic tool for checking the compatibility of synthesized designs quickly and effectively, we developed an approach which combines symbolic simulation at

the RT level with a behavior-comparison procedure based on the properties described in Section 4.

The motivations to apply symbolic simulation for our approach are twofold. First, it provides formal results because the simulator operates over a *symbolic* domain, and at the same time we are able to take into account design timing. Second, the symbolic simulation results are ready for behavior comparison since the design specifications for high-level synthesis are usually represented in a similar form such as a CDFG.

Figure 4 shows a flow chart which briefly illustrates our approach. First, the design data is read from the DDS. It includes the behavioral specification, the structural implementation, the physical information if available, and the module library. Next, we set up the simulation parameters such as delays and clocking, and the input/output protocol. We estimate the wiring delays if the floorplan is provided. The control flow of the design is analyzed to produce a list of all possible execution paths. For each execution path, the associated *path condition* will be used to drive the simulation.

The hybrid symbolic simulation performed next is event-driven. It is a hybrid model because the data path is evaluated symbolically but the controller is simulated numerically so that all the control signals will be either 1, 0 or *unknown* throughout the simulation. A transport delay model is used in the simulation. During the evaluation of active elements, an additional timing analysis procedure is performed to detect potential timing violations. The simulator will also constantly monitor the occurrences of value collision on the wires. If any timing or collision error is found during the simulation, a diagnosis procedure is called to determine the cause. The result of the simulation is represented by a dataflow graph which describes the actual data operations and data transfers done by the design.

Finally, the graphical simulation result is compared with the graphical specification. If the comparison procedure finds any difference between these two graphs, a diagnosis procedure is called to find the possible design errors. The whole process will be repeated until no more execution paths are left.

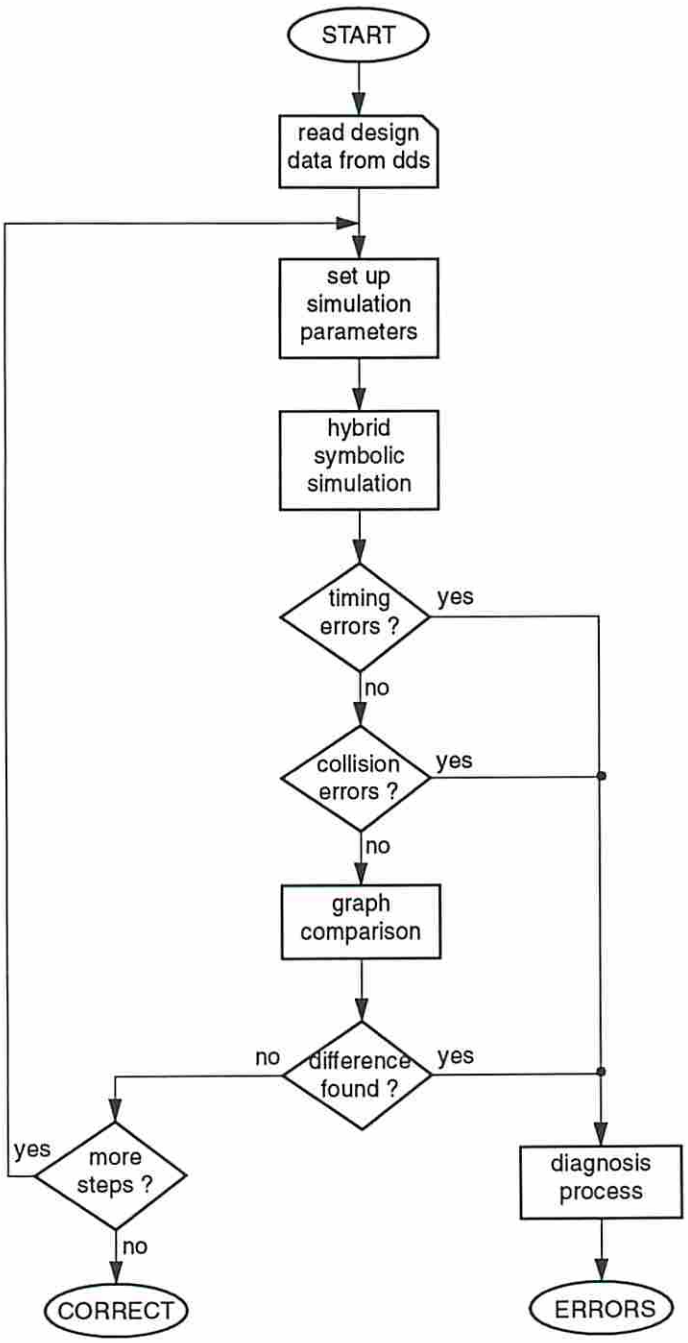


Figure 4: A flow chart of our approach for compatibility checking.

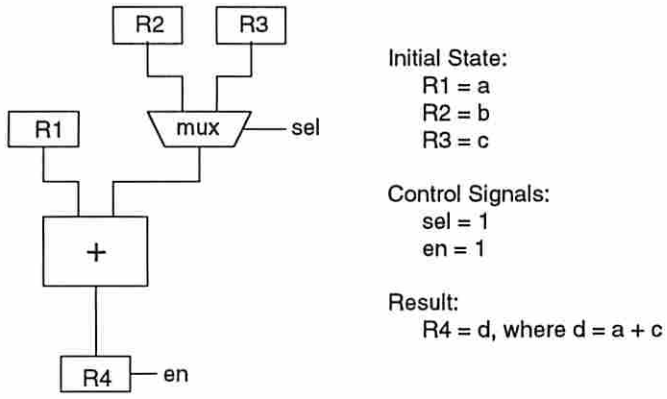


Figure 5: An example of symbolic simulation

6 Hybrid Symbolic Simulation

As we have discussed earlier, what is important for a synthesized design is whether or not it performs the required data operations and the correct sequencing of data transfers for each execution instance. Also, many design errors are related to the control and timing of the design. Hence, we need to be able to extract the circuit behavior in terms of the symbolic data operations and data transfers that occur in the datapath and at the same time to emphasize exercising the control and modeling the timing. The hybrid symbolic simulation to be described here performs exactly this task.

The idea behind symbolic simulation is similar to extending arithmetic over numbers to symbolic algebraic operations over symbols and numbers. For example, Figure 5 shows an RTL circuit with an adder, a two-to-one multiplexor and four registers. Let the symbolic values a, b and c represent the initial register values stored in R1, R2 and R3 respectively. Suppose both the control signals *sel* and *en* are 1. After simulating the circuit symbolically, a new symbolic value d is produced by the adder and stored in R4, and d is equivalent to $a + c$. In this way, we have the response to all possible values of R1, R2 and R3, given a particular control sequence, in one simulation run.

The simulation model we developed is event-driven. Figure 6 shows a typical flow of

event-driven simulation [ABF90]. The main difference between our model and traditional ones is that the evaluation of activated elements and the representation of data values are symbolic in our model. In addition, a transport delay model is used to reflect the circuit operation more accurately. We also incorporate a timing analysis procedure in the element evaluation to detect timing violations. The delay modeling and timing analysis, however, will not be elaborated in this report.

In our simulation model, the simulation of a design proceeds from one execution path to another.

Definition 6.1 *An execution path is a direct path from an* initial *state to an* end *state in the state-transition graph of a FSM controller.*

For example, the state-transition graph shown in Figure 7 contains three execution paths, each of which is associated with a *path condition* that represents the assumptions made along the path during the control flow analysis.

Definition 6.2 *A path condition is an assignment for a set of Boolean symbolic values which together determine alternative paths through the state-transition graph.*

6.1 Element Evaluation

The evaluation of an element is done to compute its output values according to its current input values.

Data Path

The evaluation of a datapath module depends on its behavior model. In DDS, this information is available from the behavioral model of the component used to implement the module. We represent this information internally in the form of function tables. The function table of a datapath module defines the manipulation of symbolic data for each possible condition on the control lines. For example, a simple four-function ALU is shown in Figure 8. The following types of elements are evaluated in the data path:

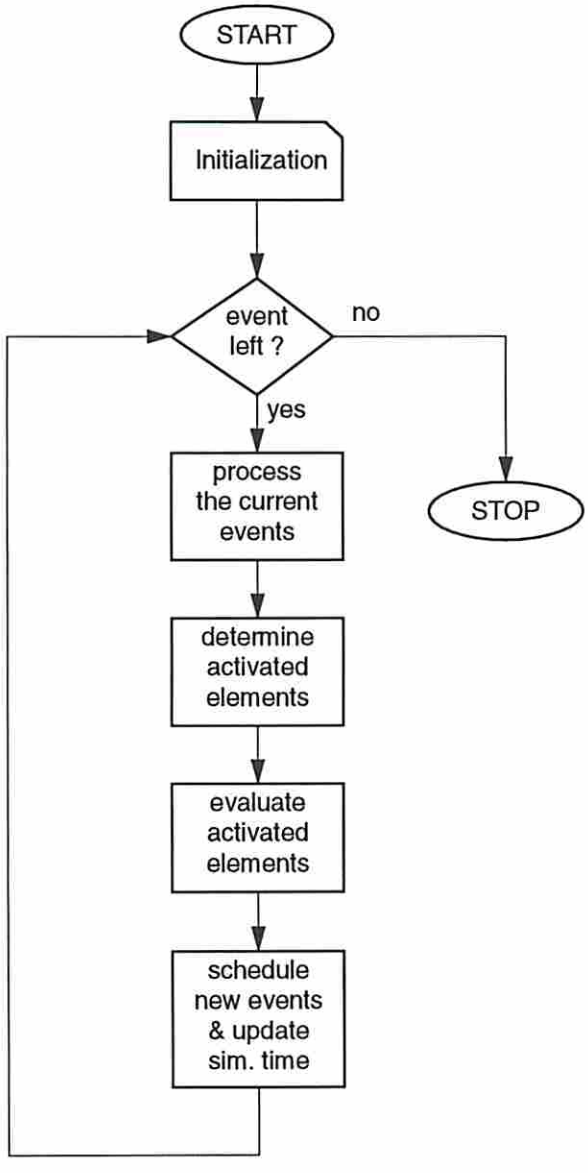
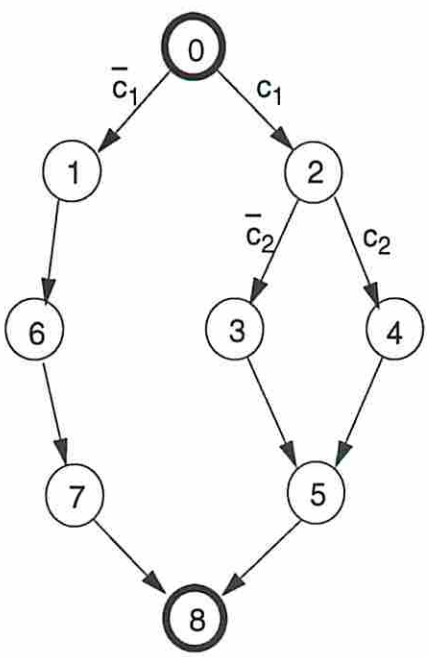
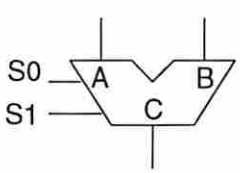


Figure 6: Typical flow of event-driven simulation



Executaion Paths	Path Condition
0-1-6-7-8	c_1=0
0-2-3-5-8	c_1=1, c_2=0
0-2-4-5-8	c_1=1, c_2=1

Figure 7: Execution paths of a state-transition graph



S0	S1	C
0	0	A + B
0	1	A - B
1	0	A & B
1	1	A | B

propagation delay = 10 ns

Figure 8: A four-function ALU

- **Functional Units.** These modules are used to perform the operations given in a specification (CDFG). When a functional unit is evaluated, a new symbolic operation is performed. In general, its control inputs, if any, are used for function selection. Then, one or more new symbolic values are produced at its output ports after a specified delay time. The input/output symbolic values are related by the symbolic operation being performed.

- **Switching Devices.** The result of evaluating a switching device is that symbolic values are transferred from its input ports to its output ports. The current values of its control lines determine the paths on which the transfers take place. Similarly, the output change is separated from the input change by a propagation delay. No new symbolic value is produced in the evaluation of a switching device.

- **Storage Elements.** Symbolic values can be written into or read from storage elements via their data input/output ports. Both registers and on-chip memory are allowed in our simulation model. A storage element is evaluated whenever its clock or enable signals change. The memory addresses are regarded as control signals; therefore, they are numeric.

If the input condition of a module being evaluated is invalid, its outputs and data storage, if any, are set to *unknown.*

The datapath carriers (nets) are used for propagating the symbolic values. A carrier connecting more than one output port requires those outputs to be tristate. Normally, at most one tristate output is enabled at any time. A *value collision* occurs if two or more output ports drive a carrier at the same time [KW89]. This type of design error can be easily detected by the simulator.

Controller

The controller is evaluated when there is a change on its clock signal. If the clock change results in a state transition, the outputs are computed and the controller moves to the next

state in the current execution path. The path condition of the current execution path is updated, if necessary, at each state transition. For example, if the state transition requires the inputs i_1 and i_2 to be 1 and 0 and if the symbolic values currently appear at i_1 and i_2 are a and b respectively, then $a = 1$ and $b = 0$ will be added to the path condition. If there is a conflict between the assumptions made in the state transition and the path condition to be updated, the current execution path is a *false path* which will never occur. The simulation will proceed to the next execution path immediately if a false path is found. On the other hand, if a required input for the state transition contains an *unknown* value, the simulator aborts the current execution path and reports a data-dependency violation.

6.2 Representation of Symbolic Data

The symbolic values and operations which occur during the simulation are the actual events exhibited by the RTL design. These values and operations constitute the actual datapath behavior of the structural implementation to be compared with the design specification. Hence, it is very important to represent these symbolic data in a way that is suitable for comparison with the specification.

In our simulation model, the symbolic values and operations produced during the simulation are used to build a bipartite data flow graph, which is the same representation used for the design specification. A vertex is created in the data flow graph whenever a new symbolic value or operation is produced during the simulation. If a symbolic value is the result of an operation performed by a functional unit, the operation becomes its direct predecessor. Similarly, the symbolic values which appear at the input ports of a functional unit become the direct predecessors of the operation being performed.

Our model is different from the early works [Dar79, Cor81] on symbolic simulation at the RT level in the following ways:

1. The overhead to propagate the algebraic expressions is eliminated since we focus on collecting the actual data operations and data transfers that occur in the data path.

2. A powerful algebraic manipulator is not required since we do not try to simplify the expressions during the simulation. Instead, the data flow graph representing the simulation result is compared with the specification using the graph-isomorphism property.

This difference is also the reason that symbolic simulation can be effectively applied to solve our problem.

Besides the data flow graph which represents the computations performed by the design, the simulator can also check the bindings between the specification and the implementation. This is because the occurrence of a symbolic value or operation, the structural component in which it take place, and the current simulation time constitute a binding which represents what is actually happening during the simulation. By collecting the actual operation and value bindings from the simulation and comparing them with those specified in the DDS, we are able to determine which design decisions are causing the problem if the design fails.

7 Graph-Based Verification

In our approach, the behavior comparison is based on the behavioral model of DDS. Since the design specification is already represented in this model, only the behavior of the structural implementation has to be translated into this model. The hybrid symbolic simulation described in Section 6 performs the translation task for us. Therefore, the verification of the RTL implementation I becomes the problem of comparing two CDFGs, $CDFG_T$ derived from the input specification and $CDFG_I$ derived during the simulation. Because the design I is the result of a mapping from $CDFG_T$, there exists a strong relationship between $CDFG_T$ and $CDFG_I$ (see Section 4). In fact, we will show that there is an isomorphic property between the two. Consequently, a graph-matching procedure based on this property is developed to compare them efficiently.

7.1 The Isomorphic Property

From Section 4, we know that the RTL implementation I, if mapped correctly, will have several properties. In summary, I will perform the required computations specified by $CDFG_T$ for every execution instance. The computations are done by making sure the input values of each required operation are available at the corresponding input ports of the designated functional unit which is configured properly.

Let DFG_I be the result of simulating I for one execution instance under the path condition pc. If the specification $CDFG_T$ is interpreted symbolically under the same path condition pc, the result is a data flow graph DFG_T such that the predicate of each operation in DFG_T is evaluated to *true* under pc.

Before we show the isomorphic property between DFG_T and DFG_I, we will first establish the correspondence for all their primary input/output values. This correspondence is important because it provides the starting point to compare these two graphs.

Lemma 7.1 *There exists an one-to-one correspondence between DFG_T and DFG_I for the primary input/output values.*

Proof: The primary input values are applied to I according to the input protocol. For each primary input value in_T of DFG_T, there exists an input port *iport* of I and some period of time $[t_s, t_e]$ such that a symbolic value in_I, which corresponds to in_T, is created and applied to *iport* externally from t_s to t_e during the simulation. Therefore, in_I is in DFG_I as a primary input value and it is assumed to correspond to in_T by the input protocol.

Similarly, for each primary output value out_T of DFG_T, there exists an output port *oport* of I and some time t such that a symbolic value out_I is read from *oport* at time t. Hence, the symbolic value out_I is in DFG_I and is assumed by the output protocol to correspond to out_T. □

Intuitively, the isomorphic property between DFG_T and DFG_I exists because I is synthesized in such a way that each required operation in DFG_T will be performed by a designated functional unit at some time and every data dependency will be preserved by establishing a

proper interconnection. Hence, if I is synthesized correctly, each corresponding primary outputs of DFG_T and DFG_I should have similar geometric properties which will be explained by the following theorem.

Theorem 7.2 *For each pair of the corresponding primary output values* (out_T, out_I) *of* DFG_T *and* DFG_I, *the cones*[5] *of* out_T *and* out_I *are* isomorphic.

Proof: From Property 4.1 and 4.2, we know that for each operation opr_T in DFG_T, there exists a functional unit u of the design I and some time t such that u is configured to perform the operation type of opr_T; otherwise, a synthesis error occurs.

Hence, if opr_T is a required computation, a list of symbolic values in DFG_I which corresponds to the input values of opr_T in DFG_T, must appear at the respective input ports of u at time t so that an operation opr_I which is equivalent to opr_T is performed. Consequently, a list of new symbolic values which corresponds to the output values of opr_T in DFG_T will be produced in DFG_I.

In other words, there exists a one-to-one mapping[6] from DFG_T to DFG_I for all the operations and values in DFG_T.

Let C_T and C_I be the cones of out_T and out_I respectively. We claim that there exists a one-to-one and *onto* relation between C_T and C_I. This relation is one-to-one as we have discussed earlier. Assuming this relation is not *onto*, there must exist either a vertex or a edge in C_I which does not have a counterpart in C_T.

Case I. Let v_I be the vertex that has no correspondence.

Since v_I is a predecessor of out_I, there exists a path in C_I from v_I to out_I. In this path, there must be an edge (v_I^s, v_I^d) such that v_I^d corresponds to v_T^d in C_T but

[5] A cone of a vertex v in a graph $G = (V, E)$ is a subgraph $C = (V', E')$ such that

- $V' = \{ v \} \cup predecessors(v)$
- for all v_1, v_2 in V', if edge (v_1, v_2) in E then (v_1, v_2) is also in E'.

[6] It is not necessary an *onto* mapping.

v_I^s does not have a counterpart in C_T. Hence, (v_I^s, v_I^d) is an incident edge of v_I^d which does not correspond to any of v_T^d. However, the correspondence between v_I^d and v_T^d implies that there is a one-to-one correspondence for all their incident edges[7]. Therefore, we have a contradiction.

Case II. Let (v_I^s, v_I^d) be the edge that does not have a counterpart in C_T.

From Case 1, we know that every vertex in C_I must have a counterpart in C_T. Let v_T^d be the vertex in DFG_T that corresponds to v_I^d. Then, v_I^d has an incident edge (v_I^s, v_I^d) which does not correspond to any of v_T^d. Therefore, this edge do not exist.

Thus, there exists a one-to-one and onto relation between C_T and C_I for all the vertices and edges; i.e., C_T and C_I are isomorphic. □

The isomorphic property between C_T and C_I not only implies that there is a one-to-one correspondence between their vertices and edges such that the incidence relationship is preserved, but also requires that each pair of corresponding vertices are compatible. In other words, if the corresponding vertices are operations, they must be of the same type. If they are values, they have same bitwidths.

7.2 A Graph Matching Procedure

Knowing that there is an isomorphic property between $CDFG_T$ and $CDFG_I$, it becomes straightforward to develop a method for behavior comparison. In fact, all we need to do is to check whether or not the cones of their corresponding output values are isomorphic for all the execution paths.

Unlike the general isomorphism problem in graph theory, which is still an important unsolved problem, it is much easier to check the isomorphic property between the cones of the corresponding output values because the correspondences of their primary input and

[7]Otherwise, if v_I^d is an operation, it will have one more input than v_T^d does, which contradicts the fact that their operation types are the same. On the other hand, if v_I^d is an value, it will be driven by two operations in C_I, which violates the single-assignment rule.

output values are known in advance. Furthermore, the correspondences of two operations can be established as soon as they are of the same type and all their input values are equivalent whereas the vertices in the former problem are typeless.

In what follows, we will present a polynomial-time procedure for checking the isomorphic property between the cones of two corresponding output values. Let out_T and out_I of DFG_T and DFG_I be a pair of corresponding output values and let C_T and C_I be their respective cones to be checked. The following procedure will return true if C_T and C_I are equivalent; otherwise, *false* is returned.

equiv_check(C_T, C_I)

1. Create an attribute for each vertex in C_T and C_I and initialize it to nil.
2. For each pair of corresponding primary input values of C_T and C_I, give their attributes a unique identifier.
3. Find all the operations in C_T and C_I such that
   - they are of the same type;
   - their attributes are still nil; and
   - all of their corresponding input values have the same attributes.

   If found, then
   (a) Give the attributes of these operations an unique identifier.
   (b) For each set of corresponding output values of these operations, give their attributes an unique identifier as well.
4. Repeat step 3 until no change can be made.
5. If there exists a vertex in C_T or C_I whose attribute is nil, return $false$. Otherwise, return $true$.

8 Experiments

In order to show the effectiveness of our approach, we performed a number of experiments with the designs synthesized from the USC ADAM system. In fact, our preliminary experiments immediately identified that the controllers generated by CSG for these designs were incorrect. CSG was then revised using the FINESSE package from the Cascade Design Automation. Finally, two successful experiments were performed.

We first experimented with a non-pipelined AR filter. MAHA was used for scheduling and Mabal for datapath allocation and binding. This design is characterized as follows:

- It has 4 time steps.
- Both the input and output values are not latched.
- A two-phase non-overlapping clocking scheme is used.

The analysis of the controller generated by CSG resulted in only one execution path with four states. The experiment was carried out by holding the input values (symbolic) at the input ports during the execution and obtaining the output values from the output ports at the end of the 4th clock cycle. The cones of these output values were then extracted from the data flow graph which was built during the simulation and compared correctly with the ones specified in the original data flow graph.

The second experiment dealt with a robot arm controller whose control flow is much more complex than the previous one. The design was synthesized in a similar way except we required the inputs values to be latched. The RTL implementation has 12 time steps and 16 possible execution paths. The controller was generated by CSG using status registers. We were able to verify this RTL implementation with the following conclusions:

- All the constant values were required to be supplied externally, which results in inefficient use of input ports.
- Conditional values were unnecessarily routed to the output ports.

- Some of the input values were not latched as specified.

From these experiments, we feel that our approach is indeed capable of verifying synthesized designs effectively and efficiently.

9 Conclusions

In this report, we have demonstrated the difficulty of the general RTL verification problem and have identified the properties of automatically synthesized designs to facilitate the verification task. We also presented a hybrid symbolic approach for checking both the functional and timing compatibility of synthesized designs. Several experiments have been conducted using this approach, and the results indicate that our approach is indeed effective and efficient.

Further experiments with designs whose timing is critical should be performed to demonstrate our ability and the advantage to take into account design timing during verification. In addition, data-dependent delays and loops need to be worked on in the future.

References

[ABF90] Miron Abramovici, Melvin A. Breuer, and Arthur D. Friedman. *Digital Systems Testing and Testable Design.* W. H. Freeman and Company, New York, 1990.

[Bry90] R.E. Bryant. Symbolic Simulation - Techniques and Applications. In *27th ACM/IEEE Design Automation Conference*, pages 517–521, 1990.

[Cor81] W.E. Cory. Symbolic Simulation for Functional Verification with ADLIB and SDL. In *18th ACM/IEEE Design Automation Conference*, pages 82–89, 1981.

[CP88] Paolo Camurati and Paolo Prinetto. Formal Verification of Hardware Correctness: Introduction and Survey of Current Research. *Computer*, 21(7):8–19, July 1988. the Computer Society, IEEE.

[Dar79] J.A. Darringer. The Application of Program Verification Techniques to Hardware Verification. In *16th ACM/IEEE Design Automation Conference*, pages 375–381, 1979.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, New York, 1979.

[GP93] P. Gupta and A. C. Parker. PLA Delay Estimation. Technical Report CEng 93–16, University of Southern California, 1993.

[KP85] D.W. Knapp and A.C. Parker. A Unified Representation for Design Information. In *CHDL-85*, Elsevier, 1985.

[KW89] D.W. Knapp and M. Winslett. A Formalization of Correctness for Linked Representations of Datapath Hardware. In *IFIP Workshop on Applied Formal Methods for Correct VLSI Design*, November 1989.

[McFa] M.C. McFarland. Formal Verification of Digital Hardware. Boston College, Chestnut Hill, MA 02167.

[McFb] M.C. McFarland. Practical Lessons in Verification and HIgh-Level Synthesis. AT&T Bell Laboratories, Murray Hill, NJ 07974-2070.

[MPC88] M.C. MacFarland, A.C. Parker, and R. Camposano. Tutorial on High-Level Synthesis. In *25th ACM/IEEE Design Automation Conference*, pages 330–336, 1988.

[MY78] M. Machtey and P. Young. *An Introduction to the General Theory of Algorithms.* the Computer Science Library. Elsevier North Holland, Inc., 1978.

[NP91] Alexandru Nicolau and Roni Potasman. Incremental Tree Height Reduction For High Level Synthesis. In *28th ACM/IEEE Design Automation Conference*, pages 770–774, 1991.

[Sak83] T. Sakurai. Approximation of Wiring Delay in MOSFET LSI. *IEEE Journal of Solid-State Circuits*, 18(4):418–426, August 1983.

[WP91] Jen-Pin Weng and Alice C. Parker. 3D Scheduling: High-Level Synthesis with Floorplanning. In *28th ACM/IEEE Design Automation Conference*, pages 668–673, 1991.

[WP92] Jen-Pin Weng and Alice C. Parker. CSG: Control Path Synthesis in the ADAM System. Technical Report CEng 92–03, Department of EE-Systems, University of Southern California, April 1992.