Multi-Level Formal Verification of High Level Synthesis: a Reality!*

Luc Claesen

IMEC & K.U.Leuven
Kapeldreef 75, B-3001 Leuven (Belgium)

Mark Genoe

Bell-Alcatel
Francis Wellisplein 1, B-2000 Antwerp

This paper describes the application of a practical multi-level formal verification approach that is used for the independent verification of the results of a high-level synthesis system by means of *SFG-Tracing*. *SFG-Tracing* is a formal verification methodology that allows to verify observational behavioral equivalence between two finite state systems: one can be seen as a specification and the other as an implementation. The two representations can be on a different level of abstraction in space and in time. The methodology makes use of the concept of reference signals and three value logic symbolic state space evaluation in order to be able to partition the problem at hand in manageable pieces. A practical implementation of *SFG-Tracing* has been realized for the verification of the results of a high level synthesis system from algorithmic specifications down to the gate or transistor netlists. The verification is fully automatic and has been demonstrated by the verification of systems of 230,000 transistors. Design bugs have been uncovered that were previously not found using traditional simulation.

I. Introduction

A. Why verification?

The possibilities offered by the steadily increasing complexities enabled by the VLSI technology have resulted in the fact that more and more complex systems can be build on integrated circuits.

Complex arithmetic algorithms as required by digital signal processing applications can now be implemented in real time. This has resulted in applications such as digital audio, speech processing, telecommunication, mobile telephony (GSM), wireless communication systems, image recognition systems, robot control, automotive control, efficient satellite communication and radar applications, digital television, ISDN, ATM, micromechanic and mechatronic applications [17]. Most of these applications have been enabled by the current complex integrated circuit technologies.

*research results sponsored in part by the ESPRIT 6018 CHARME Basic Research working group.

The realization of complex systems has become *design limited* instead of technology limited. The challenge is indeed to design electronic systems *first time right*. This is required to avoid costly redesigns, and delays in market introduction of new products. These economic reasons are the drive behind research efforts to check the correctness of designs with respect to their specifications. A well-known illustration of the possible negative effects of design that are not discovered in time is given by the well known "Pentium" bug.

To achieve correctness of designs with respect to specifications, two aspects have to be envisioned. First the *design specifications* have to be formalized. Second the *design implementation* has to realized in accordance to the specifications.

In the case automatic synthesis tools are being used, it is also good practice to cross check the results of automatic synthesis tools with independent verification algorithms. This is motivated by the high costs, and time involved in processing iterations for integrated circuits. This strengthens the confidence in the design correctness. This concept of using verification is also accepted for a long time in the "correct by construction approach" of standard cell or gate array layout from schematics specifications. After the layout design, a circuit extraction and a netlist comparison is performed in order to raise the confidence in the design correctness. This will further avoid the overlooking of software bugs, or erroneous manual interventions in designs.

Traditionally simulation (at multiple levels of design abstraction) is being used, and is standard industrial practice, to verify the correctness of electronic designs before they are produced. It is however very well known that for even moderately sized circuits and systems it is not feasible to try out all possible input excitations in these simulations, due to the combinatorial explosion problem in the number of patterns.

B. Approaches in Formal Correctness Verification.

If one would use simulation for verification of a (controlable and observable) design with m inputs and n registers, a number of 2^{n+m} simulations with specific input signal values should have to be performed, which is im-

possible for every realistically sized design.

Instead of using specific values for performing simulation, *symbolic values* can be used, to allow performing simulations [10] for a more general number of signal values. Symbolic simulation on its own however does not solve the problem of proving correctness of an implementation with respect to its higher level specification.

By a method based on Floyd's inductive assertions [16], Darringer [14] has proposed a method, where *assertions* are added to the specifications, so that symbolic simulation can be used to verify a number of execution paths in a hardware implementation. This however requires the introduction and definition of the appropriate assertions in the design descriptions, in order to allow the appropriate verifications.

For pipelined hardware, verification based on symbolic logic simulation has been proposed by Bose and Fisher [1].

Verification of Boolean functionality. Good results (especially at the lower levels of design abstractions) have been reported and are commercially available. For the verification of Boolean functionalities of gate level implementations to structural RT level specifications, the methods based on OBDD's (Ordered Binary Decision Diagrams) [4, 27] have become a de facto standard. Extremely code optimized and efficient OBDD packages have become available [27, 3, 23].

For the case where the implementation and the specification are known to have the same encodings for the state variables in the registers, the verification problem of such sequential systems can be reduced to the verification of a number of combinatorial logic functions by means of OBDD's [27, 2].

Verification of sequential systems. By using symbolic state space representation methods, it has become possible to verify sequential systems and also check adherence to temporal logic formulae [13, 9]

Behavioral verification by means of *SFG-Tracing* . In this section we present the method that is underlies the automatic verification from the behavioral signal flow graph specification down to lower implementation levels. These can go down to the gate- or transistor switch level if a suitable symbolic simulator is used. In line with the automatic verification algorithms, as much as possible the structure available in the problem at hand is being exploited. The first application target has been the automatic verification of high level synthesis results as obtained by the CATHEDRAL silicon compilers [15]. The methodology is however generally applicable.

The algorithms are intended to operate with as little interaction from the user as possible. The underlying assumption is that the flow graph specification is synthesized while keeping track of *mapping relationships* of a set of well chosen *reference signals* of the specifying flow graph and of the implementation. The global verification problem is reduced to a manageable size by partitioning the information in the global signal flow graph into acyclic subgraphs and providing correspondence (mapping) functions between the interface values (reference signals) in the partitioned graph and the signal values at specific cycle and clock phase times in the implementation. The correctness of each individual subgraph is proven by making use of a (switch-level) symbolic simulator that acts on the actual switch level models of transistor circuits.

To give an indication of the information explosion from high level behavioral (SFG) specifications down to the implementation, consider the modem pulse shaper and equalizer chip [29]. This system implements the filter flow graph that can be formally specified in the SILAGE language in 70 lines of text. The chip implementation as synthesized by CATHEDRAL-II [29] results in a micro coded architecture with 3 ALU's of 14 bits and consists of more than 31000 transistors. Together with the explosion in terms of elements and space, there is also an expansion of the time concept: from sample periods at SFG level over micro-code instruction cycles, clock phases down to clock wave forms at the switch level. Notice that all the signals that appear in the SFG specification occur in some form during specific times at specific places in the transistor implementation of the chip. Operations in the SFG can however occur on the same hardware blocks such as ALU's at different instances of time. This relationship between algorithmic SFG signals and signals in space and time of the implementation forms the basis for the *SFG Tracing* verification methodology.

In this paper we present the application of the *Signal Flow Graph Tracing (SFG-Tracing)* methodology for the autmatic verification of high level synthesis results in CATHEDRAL. In line with the automatic verification algorithms, as much as possible the structure available in the problem at hand is being exploited. The first application target in the research in IMEC is in the verification of high level synthesis results as obtained by the CATHEDRAL silicon compilers [15].

II. *SFG-Tracing* Methodology.

The goal of the verification process is to verify the behavioral input-output correctness of the lower level implementation with respect to the high level signal flow graph specification. Of course it would be the most interesting to perform the verification from a level as high as possible to an implementation as detailed as possible. In this paper, we consider the SILAGE SFG level as the specification, and the transistor switch level as the representation. Higher levels of the implementation could also be considered (such as gate level or sRT or bRT level). The same techniques as indicated below would apply in each of these cases. The switch level implementation is however pre-

ferred, because it reflects the best the circuit implementation. Appropriate symbolic analysis techniques based on Bryant's method [5, 6] for the switch level have been developed and are supported in CAD tools [7, 21, 26].

A. Flow Graph Specification.

In what follows, we will describe the *SFG-Tracing* verification methodology. This methodology starts the verification from the specifying signal flow graph topology (in fact by "tracing" the specifying SFG). The methodology can be used to verify lower level implementations. These can be either at the bRT-, the sRT- or the switch level. The lower the verification methodology is used in terms of levels of design abstraction, the more practical useful the methodology becomes. For the explanation of the method it is irrelevant to which level we are considering the verification. As we believe it will be a feasible approach, the more challenging verification up to the MOS switch level is described here. This is also motivated by the existence of excellent MOS symbolic analysis CAD tools [5, 6].

The symbolic simulation of a lower level implementation is always done over a number of cycles and clock phases within the global SFG sample period.

For the SFG-tracing, two aspects have to be considered. The first consists of the verification of the *initialization sequence*, and the second aspect consists of the verification of the *steady state behavior*. The initialization sequence is used to bring the implemented system in a known state. Starting from that known state, cycles and clock phases can be defined, which correspond to the SFG level sample periods. The initialization sequence consists of the sequence following for example the reset pulse. The symbolic simulator will have to be started from the initialization sequence in order to be able to bring the implemented system in a known state. The SFG specification also contains initialization information (initial values at SFG level registers). The verification will consist of two phases: the initialization and the steady state. Although similar techniques can be used for both phases, this paper will concentrate further on the verification of the steady state behavior.

As mentioned before, behavioral system level specifications at the SFG level only specify the algorithmic dependencies of variables, but not how the algorithm is implemented in actual hardware.

In this paper, we consider the SILAGE language [22]. In order to use the *SFG-Tracing* algorithms, only the basic signal flow graph semantics of SILAGE are important. SILAGE, as an applicative language, allows for function abstraction, the use of manifest functions [22] and variables. The commands included for the C preprocessor can be removed by simple substitution, the manifest variables can be computed at compile time. The user defined functions can be expanded in terms of primitive functions (built-in operators such as shift, add, subtract, multiply etc.).

For the conditionals in the SILAGE description, one could either build *path conditions* or expand the conditions in the flow graph representation.

Path conditions as introduced in [14] indicate the logical conditions under which specific constructs (nodes in the flow graph) could be reached. These path conditions are Boolean expressions describing the condition of the signals to be in that part of the signal flow graph or code.

The conditionals in the SILAGE description can also be expanded to all of the nodes that are affected by it. This results in extra conditional nodes in the expanded signal flow graph. It has the disadvantage of loosing the insight in the structure of the conditionals on SILAGE descriptions, but it has the advantage that the specification can be represented completely as a flow graph.

We assume that all of these higher order features of SILAGE have been removed at compile time, and that the system is described in the basic SILAGE signal flow graph semantics.

B. Basic SILAGE Signal Flow Graph Semantics.

The basic SILAGE signal flow graph semantics are modeled by a graph $\mathcal{G}(V, E)$.

The set of vertices V of this signal flow graph $\mathcal{G}$ are defined by vertices $v_i \in V$ corresponding to the primitive operations in SILAGE. Examples are: arithmetic operations (addition, subtraction, multiplication...), shift, logical operations and conditionals.

The set of edges is E is defined by edges $e_j \in E$, where each e_j corresponds to a signal in the SILAGE flow graph. In SILAGE signals are defined as one-sided infinite streams, characterized by a specific sampling rate.

Two functions

$Inputs : V \rightarrow E^*$ and

$Outputs : V \rightarrow E^*$

can be defined:

$Inputs(v_i) = \{e_k, e_{k+1}, ...e_m\}$ and

$Outputs(v_j) = \{e_l, e_{l+1}, ...e_n\}$

which describe the inputs and outputs of operators in SILAGE. In SILAGE only one output is used per operator.

To each edge e_j corresponds a SILAGE *signal*, that is modeled as a stream. However at specific moments in the algorithm time t_{sfg}, individual element values of the stream can be considered $e_j(t_{sfg})$. The signals can be words representing numeric binary values of a specific word length w_{e_j}. The signal consisting of a binary word can be represented as $e_j[1..w_{e_j}]$. It is assumed that individual bits in signals representing binary values are ordered from most significant bit (MSB) (index 1) to the least significant bit (LSB) (index w_{e_j}). The k'th individual bit of the signal e_j is represented as $e_j[k]$.

B1 *Reference signals* **and** *Mapping functions.*

In *SFG-Tracing* we make the following assumptions:

1. There exist a number n_{ref} of *reference signals* $e_r \in RefSignals(\mathcal{G}(V,E))$ corresponding to edges in the SFG algorithm specification and signals at specific (cycle and clock) times in the implementation. The specification SFG is implemented in hardware maintaining the same behavioral relationships for these reference signals.

   For all reference signals $e_r \in RefSignals(\mathcal{G}(V,E))$ the signals ${e_r}^s$ in the specification and ${e_r}^i$ in the implementation can be defined:

   - The reference signals in the SFG specification ${e_r}^s(t_s)$ have the following semantics in terms of Boolean bit words:

     $$ {e_r}^s[k_s](t_s) \in \mathcal{B} \qquad (1) $$

     for all bits $k_s \in \{1..w_s\}$ in the SFG signal word and for a specific sample time t_s. B is the set of Booleans. Often at the SFG level, the individual bits in signal words are not considered.
   - The reference signals in the implementation are characterized by:

     $$ {e_r}^i[k_i](t_{ik_i}) \in \mathcal{B} \qquad (2) $$

     for individual bits with index $k_i \in \{1..w_i\}$ at specific implementation times t_{ik_i}. The index k_i of t_{ik_i} indicates that each bit of a reference signal has to be considered at a specific cycle and clock phase individually. This is for example already necessary in bit-serial implementations of SFG specifications.

2. There exist a set of *mapping functions* $\mathcal{F}$ that describe the behavioral correspondence in space and time of *reference signals* in the SFG algorithm specification with respect to the lower level implementation at the specific implementation times.

   $$ \mathcal{F} : SFG\_sig\_semant \rightarrow Switch\_sig\_semant \qquad (3) $$

   or:

   $$ \mathcal{F} : \mathcal{B}^{w_i} \rightarrow \mathcal{B}^{w_s} \qquad (4) $$

   where $\mathcal{B}$ is the set of Boolean values.

   The function $\mathcal{F}$ is defined as:

   $$ {e_r}^s(t_s) = \mathcal{F}({e_r}^i[1](t_{i1})...{e_r}^i[w_i](t_{iw_i})) \qquad (5) $$

   This is a vector assignment over the individual bits of the reference signal in the SFG.

3. All edges and vertices in $\mathcal{G}(V,E)$ are reachable via directed paths starting at the edges corresponding to reference signals.

4. The reference signal partitions the graph $\mathcal{G}(V,E)$ such that the subgraphs are acyclic.

The most essential form of *reference signals* would be the input and the output to the algorithm to be implemented in hardware. The verification effort and complexity can be reduced if more reference signals are available.

For the reference signals it is required that mapping relations are available, which state the relationship between reference signals in the specification and in the implementation. This could be in the form of a certain word at a specific sample time in the SFG level begin implemented in terms of bits in specific registers (at specific time phases) at the lower level implementation. Most of the relationships will be simple correspondences of the logic values in specification and implementation. Other relationships could include a specific logic function to convert the logic representation in the specification into the logic representation in the implementation or vice versa. The simplest form of this are signals in the specification that are identical or inverted in the implementation. However, more complex relationships can be envisioned: e.g. an integer word at the SFG level represented in the implementation in carry save technique.

The third condition is required so that the *SFG Tracing* algorithm can use a directed graph traversal algorithm to reach all of the parts in the specification SFG in order to do the comparison.

The requirement that the global SFG is partitioned by providing reference signals and mapping functions is not that hard a requirement in order to be able to perform the verification in a reasonable way:

1. The SFG level representation of a design contains much less detail than actual implementations. To illustrate this, compare the high level SILAGE description in [11] of one page, to the ultimate hardware implementation, consisting of 31614 transistors.

2. When using this methodology to verify results from high level synthesis (as is the case in our application), the required information of reference signals (and mapping functions) is generated by the synthesis tools as well. The correspondence between signals at the SILAGE level and lower levels of implementation is already maintained for most signals in a number of cross reference lists and naming conventions used in the CATHEDRAL-II silicon compiler [29]. If errors would be introduced in the translation of the reference signals and/or mapping functions, this would also manifest itself in errors in the SFG tracing verification.

3. For manual designs, accurate SFG level descriptions will have to be made and appropriate reference sig-

nals will have to be identified in order to be able to perform the SFG Tracing verification. The information that has to be supplied by a designer is in any case still much less than what is required in theorem prover based methods [24, 25].

C. SFG-Tracing strategy: From SFG to lower design abstraction levels.

A straightforward way to compare the SFG specification to an implementation would be the expansion of the specification and implementation into predicate logic formulas at the lowest level of representation of design abstraction, which would then have to be verified by tautology checking. Except for very trivial design examples, such an approach will suffer major complexity problems and will be unfeasible.

In the *SFG-Tracing* strategy, the goal is to fully exploit the structure of the SFG level specification in order to guide the lower level verification process. In *SFG-Tracing* the specifying signal flow graph (SFG) is traversed over all of its vertices and all of its nodes.

Depending on the design abstraction level of the implementation (switch, sRT or bRT), more detailed information with respect to reference signals is required when comparing to lower level implementations. In the most general case, with the comparison to the switch level, it is assumed that at least one signal reference edge in the SFG is provided per loop in the SFG. This will correspond normally to the algorithmic registers at the SFG level. Algorithms to determine the strongly connected components in a directed graph, and consequently determining the directed loops have been published in [28]. For the application of the modem filter [29], this would mean that the signals in the z^{-1} delay blocks have to be provided as reference signals and that the mapping relations would indicate how these reference signals are implemented at the lower (switch) level in terms of the specific bits at specific cycles and clock phases.

When breaking the directed loops in the z^{-1} blocks in [29] we end up with 12 reference signals consisting of the 2 input signals and 10 signals corresponding to DSP registers. Due to the fact that canonical signed digit representations are used for the representation of the coefficients in the filter, no full multiplications are needed. They are implemented by a few add/subtract functions. In this case, the verification can probably often be done with tautology checking on the output reference signals for each SFG subgraph. For the application in [29] further reference signals (such as in between biquadratic sections and after multipliers) would ease the verification process. Novel representations such as binary moment diagrams [8] can be used here.

Due to the choice of reference signal edges, such that they can break all directed loops in the specifying SFG, the SFG can be reformulated as a directed acyclic graph, when the edges corresponding to the loop edges are not taken into account. The operations corresponding to the vertices v that have reference signal edges $e_j \in RefSignals$ as inputs, it holds that: $Inputs(v) \subset Refsignals$. This means that the operator corresponding to the vertex in the SFG can be evaluated symbolically and matched against the result in the implementation. This matching allows to proceed in the further SFG-Tracing algorithm, by matching edges furtheron in the SFG.

D. Signal Flow Graph partitioning.

The choice of appropriate reference signals and mapping functions allows that SFG graph $\mathcal{G}(V, E)$ is partitioned into a signal flow graph PSFG (Partitioned Signal Flow Graph) consisting of a set of disjoint and acyclic subgraphs $\mathcal{G}_p(V_p, E_p)$. Each subgraph $\mathcal{G}_p(V_p, E_p)$ consists of a cut set of vertices of $\mathcal{G}(V, E)$ where the edges between vertices in the cut set and vertices out of the cutset correspond to the reference signals, related to that subgraph.

E. Description of the SFG-Tracing method.

The reference signals allow a subdivision of the global SFG in a number of subgraphs in the PSFG. For each subgraph in the PSFG a verification of the implication of the specification by the implementation is verified by performing a symbolic simulation of the implementation.

```
SFG_Tracing()
{
    read_ref_signals_and_mapping_funct();
    init_symbolic_simulation();
    PSFG = Partition_SFG();
    for each subgraph in the PSFG
    {
       for impl_time = start_time to end_time;
       {
          symb_initialize_impl_signal(impl_time);
          symbolic_simulate_step(impl_time);
       }
       symb_compare_signals();
    }
}
```

In `read_ref_signals_and_mapping_funct();` the reference signals and the mapping functions are read. Making use of this information the partitioning of the signal flow graph is performed in `Partition_SFG`. Hereafter for each subgraph the verification is done by a symbolic simulation . Since reference signals in the implementation can occur in different cycles and clock phases, (within a global SFG clock period of the system) the values of implementation signals have to be initialized in the symbolic simulation at the appropriate implementation times. Therefore the symbolic simulation has to be done from `start_time`

to `end_time`, such that all the signals that are input to the PSFG subgraph can be initialized and that after that, all signals at the output of the PSFG subgraph can be evaluated in the appropriate cycle time and clock phases.

In the symbolic simulation, the reference signals and the signals dependent on them are evaluated symbolically. External signals that are always recurring during each global SFG time period will have specific values. This is the case for external clock signals, that are used for the specific values in the respective phases. Other signals like reset signals and signals to put the circuit in specific constant values. Doing such a symbolic simulation will result in specific (Boolean 1,0) signals for the control circuits, and symbolic signals for the other circuitry. Most of the time 'x' signals are used in the symbolic simulation. Only for those signals implementing the operations of the subgraph of the PSFG at hand, symbolic values will be computed.

The symbolic simulation as described here does an unfolding of the specific hardware implementation at hand, to end up with a (maximally parallel) representation that corresponds to the specifying subgraph in the SFG. In several implementations such as in micro-program controlled data paths as in CATHEDRAL-II, several of the operations in the SFG specification are mapped onto the same hardware operators, such as for example the same ALU. The controller takes care of the sequencing in time of the hardware operations that have to be performed on the same hardware operator (e.g. the same ALU). By doing symbolic simulation, the effect of the sequencing by the controller is removed, and the hardware operators can be seen as unfolded for the specific operations that they have to perform.

By this symbolic simulation, the micro-code controller will normally operate with instantiated signal values ('1', '0', 'x') instead of symbolic values in the execution of cycles and clock phases. These instantiated signal values can directly be used (and reduced) in the symbolic simulations. By this fact of unfolding (or unrolling) the algorithm again to its maximal parallel representation the effect of the controller, and its specific encodings can be 'simulated away'.

After the symbolic simulation, symbolic expressions are obtained for the output signals corresponding to the subgraph under consideration. Notice that these symbolic output signals have to be taken at the appropriate cycle and clock phase times as defined by the reference signals. As already explained these output signals correspond to the maximally parallel representation as in the SFG specification, and the correctness has to be verified by comparison.

¿From the semantic definitions of the primitive operations in the specifying SFG, the mapping functions for the reference signals (that form the interface for the subgraph at hand), and the results of the symbolic simulation a comparison is done in `symb_compare_signals`.

¿From the semantics of the primitive operators in the subgraph of the PSFG under consideration, the input output behavior at the SFG level for the subgraph can be derived. This is characterized by the function:

$$\mathcal{S}_{sfg} : \mathcal{B}^* \rightarrow \mathcal{B}^* \tag{6}$$

This function provides the behavioral relationship as extracted from the SFG semantics between reference signals at the input ${e_{r_{in}}}^s$ and at the output ${e_{r_{out}}}^s$ of the subgraph under consideration:

$${e_{r_{out}}}^s = \mathcal{S}_{sfg}({e_{r_{in}}}^s) \tag{7}$$

In the same way the input-output behavior function as derived by the symbolic simulation of the implementation can be defined:

$$\mathcal{S}_{impl} : \mathcal{B}^* \rightarrow \mathcal{B}^* \tag{8}$$

This function provides the relationship as obtained by the symbolic simulation between reference signals at the input ${e_{r_{in}}}^i$ and at the output ${e_{r_{out}}}^i$ of the subgraph under consideration:

$${e_{r_{out}}}^i = \mathcal{S}_{impl}({e_{r_{in}}}^i) \tag{9}$$

The mapping functions for the reference signals at the inputs and outputs of the subgraph under consideration provide the following relationships:

$${e_{r_{out}}}^s = \mathcal{F}_{r_{out}}({e_{r_{out}}}^i) \tag{10}$$

and:

$${e_{r_{in}}}^s = \mathcal{F}_{r_{in}}({e_{r_{in}}}^i) \tag{11}$$

¿From the above relationships, the subgraph behavioral functions and the mapping functions, the following condition for the correct behavioral verification of the subgraph under consideration can be derived:

$$\mathcal{S}_{sfg}(\mathcal{F}_{r_{in}}({e_{r_{in}}}^i)) = \mathcal{F}_{r_{out}}(\mathcal{S}_{impl}({e_{r_{in}}}^i)) \tag{12}$$

The verification will normally be done by tautology checking, based on efficient methods such as OBBD's [4, 27]. In this comparison, one can however also make use of the information available from the signal flow graph, such as the fact that at the SFG level signals are representing bit-words. Optimized verification algorithms and vector-based reduction rules such as presented by Eveking [2] can be used to improve the cpu-time efficiency of the verification.

A step by step example of the *SFG-Tracing* methodology is given in [12].

III. Verification of high level synthesis.

The methodology of SFG-Tracing has been included in the CATHEDRAL synthesis environment [15] as indicated in [11, 18, 19, 20].

Starting from a SILAGE description the basic SFG is derived. This is partitioned into the PSFG in such a way that it results in manageable pieces for further verification. The interface signals *(reference signals)* for the subgraphs in the PSFG are provided to the synthesis environment, to make sure that the corresponding signals in the layout for the switch level and the *mapping functions* can be generated. The synthesis environment provides the cycles in the microcode that correspond to the global SFG time period, because this is needed to perform the appropriate symbolic simulation sequences.

Starting from the PSFG, the reference signals and the correspondence functions, the SFG-Tracing is performed by the "Symbolic Simulation Manager" called `SfgTracer`, that prepares the simulation commands for the symbolic simulator at hand. After individual symbolic simulations on subgraphs, the results are verified for correctness. For the symbolic simulation the COSMOS program [7] is used. This works on the transistor as it is obtained from the layout circuit extraction. Also gate netlists can be provided via the `lgc` format [7]. In case of inconsistencies for specific subgraphs of the PSFG, the Symbolic Simulation Manager generates the appropriate error messages, to indicate where the error occurs. It could also occur that the subgraph under consideration is too large to be able to perform the verification. In this case the subgraph has to be partitioned further. This can be achieved by the user giving hints on SFG nodes, where the SFG has to be further partitioned in order to give rise to smaller subgraphs.

The specific treatment of loop constructs in the specifications and the use of induction in the formal verification is presented in [19], while the posibilities for parallel execution are discussed in [20].

IV. Practical Results.

The *SFG-Tracing* verification methodology has been applied successfully [18, 19, 20] to the verification from the layout extracted transistor netlist in comparison with the high level specifications for a number of designs as synthesized by CATHEDRAL-II as summarized in table I.

design	count	rec3	echo	voco
# MOS tr.	7,108	31,614	45,000	230,000
# st.cells	580	1,859	3,246	12,067
# flipflops	230	852	1,394	6,526
# uniq.subn.	32	36	33	38
# mach.cyc.	503	19	1,310	114,686
# sim. cyc.	20	29	79	1,400
# cpu time	22 s	567 s	2,142 s	8,000 m
# scr.line	640	4,782	48,375	680,000
# partit.	14	68	256	2,200
# speclines	15	92	30	1460

Table I. Results of verification of transistor implementation with respect to high level specification for a number of designs synthesized by CATHEDRAL-II on a DEC 3100.

`count-s` is an application that has `while`-loops and conditional statements in its specification. `rec3-s` is the modem chip [29] implemented with 3 ALU's in the datapaths. `echo` is an echo canceler application and `voco` is an 800 bit/sec vocoder application. The cpu times (DEC 3100) include symbolic simulation and the checking by means of OBDD's [3] of the proof obligations per member of the partition.

V. Conclusions.

SFG-Tracing is a general formal verification methodology that allows the symbolic verification between behavioral specifications and lower levels of implementation (gate and SPICE transistor level). This allows to implicitly partition behaviors (but not structure!) at the low level. The methodology partitions the dataflow graph and systematically traces its behavior symbolically. To cope with complexities on realistic design applications a three value logic symbolic evaluation is used [7]. The control flow in the specification is dealt with by an enumeration of its state transitions. As such *SFG-Tracing* expoits the duality of control flow and data flow as it occurs in many systems in the same way as this is done in behavioral synthesis.

In this paper we presented the application of the *SFG-Tracing* methodology for the verification of high level synthesis results as it has been implemented arround the Cathedral synthesis program. The practicality has been demonstrated by large design examples (230,000 transistors).

SFG-Tracing has however much larger potential for applicability to be used also for manual designs. Novel symbolic representation and manipulation methods such as binary moment diagrams (BMD's) will extend the practical usefulness of the method.

References

[1] S.Bose, A.L. Fisher, *Verifying Pipelined Hardware Using Symbolic Logic Simulation*, "Proc. of the IEEE International Conference on Computers and Design", ICCD-89, pp. 217-221.

[2] A. Bratch, H. Eveking, H.-J. Faerber, J. Pinder, U. Schellin, *LOVERT - A Logic Verifier of Register Transfer Level Descriptions*, "in "Formal VLSI Correctness Verification"", Ed. L.Claesen, North-Holland, 1990, pp. 247-256.

[3] K.S. Brace, R.L. Rudell, R.E. Bryant, *Efficient Implementation of a BDD Package*, "ACM-SIGDA/IEEE 27th Design Automation Conference Proc." pp. 40-45.

[4] R.E. Bryant, *Graph Based Algorithms for Boolean Function Manipulation*, "IEEE Transactions on Computers", Vol. C-35 No. 8, August 1986, pp. 667-691.

[5] R.E. Bryant, *Algorithmic Aspects of Symbolic Switch Network Analysis*, "IEEE Trans. on Computer-Aided Design", Vol. CAD-6, No. 4, July 1987, pp. 618-633.

[6] R.E. Bryant, *Boolean Analysis of MOS Circuits*, "IEEE Transactions on Computer-Aided Design", Vol. CAD-6, No. 4, July 1987, pp. 634-649.

[7] R.E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffer, *COSMOS: A Compiled Simulator for MOS Circuits*, 24th DAC, pp. 9-16, 1987.

[8] R. E. Bryant, Yirng-An Chen, *Verification of Arithmetic Circuits with Binary Moment Diagrams*, "Prof. 32nd ACM/SIGDA DAC", 1995, pp. 535-541.

[9] J.R. Burch, E.M. Clarke, K.L. McMillan, D. Dill, *Sequential Circuit Verification Using Symbolic Model Checking*, "Proc. 27th DAC", 1990, pp. 46-51.

[10] W.C. Carter, W.H. Joyner Jr., D. Brand, *Symbolic Simulation for Correct Machine Design*, "Proc. 16th Design Automation Conference", pp. 280-286, 1979.

[11] L. Claesen, F. Proesmans, E. Verlind, H. De Man, *" SFG-Tracing": a Methodology for the Automatic Verification of MOS Transistor Level Implementations from High Level Behavioral Specifications*, "Proc. International Workshop on Formal Methods in VLSI Design", ed. P.A. Subrahmanyam, Miami, January 9-11, 1991.

[12] L. Claesen, M. Genoe, E. Verlind, F. Proesmans, H. De Man, *Application Example of Multi-Level Digital Design Verification by the* SFG-Tracing *Methodology*, "Proc. IEEE EUROASIC-91", pp. 379-384.

[13] O. Coudert, C. Berthet, J.-C. Madre, *Verification of Sequential Machines Using Functional Vectors*, "in "Formal VLSI Correctness Verification"", Ed. L.Claesen, North-Holland, 1990, pp. 267-286.

[14] J. Darringer, *The Application of Program Verification Techniques to Hardware Verification*, "Proc. 16th Design Automation Conference", pp. 375-381, 1979.

[15] H. De Man, J. Rabaey, P. Six, L. Claesen, *Cathedral-II: A Silicon Compiler for Digital Signal Processing*, "IEEE Design & Test of Computers", December 1986, Vol. 3, No. 6, pp.73-85.

[16] R.W. Floyd, *Assigning meanings to programs*, in " Proceedings Symp. in Applied Mathematics, 19 - Mathematical Aspects of Computer Science", Schwartz, J.T. ed. AMS, 1967, pp. 19-32.

[17] M. Genoe, L. Claesen, H. De Man, C. Tricarico, R. Delpretti, D. Dauw, *An ASIC for Die-Sinking Spark Erosion Simulations*, "Proceedings EUROASIC-90", Paris, 29-31 May 1990.

[18] M. Genoe, L. Claesen, E. Verlind, F. Proesmans, H. De Man, *Illustration of the* SFG-Tracing *Multi-Level Behavioral Verification Methodology, by the Correctness Proof of a High to Low Level Synthesis Application in CATHEDRAL-II*, ICCD-91, Cambridge MA, October 14-16, 1991.

[19] M. Genoe, L. Claesen, E. Verlind, H. De Man, "Formal Verification of High Level Synthesis by means of *SFG-Tracing*", *Proc. Sixth ACM/SIGDA & IEEE Worksh. on High Level Synthesis*, Dana Point, California, Nov. 4-6, 1992, pp. 336-343.

[20] M. Genoe, L. Claesen, H. De Man, "A Parallel Method for Functional Verification of Medium and High Throughput Digital Signal Processing Synthesis", ICCD'94, Oct. 10-12, 1994, pp. 460-463.

[21] P. Herrebout, *BOTRYS: A Program For the Symbolic Analysis of MOS Circuits at the Switch Level*, Thesis IMEC - Kath. Univ. Leuven Belgium, July 1988.

[22] P. Hilfinger, *Silage, a High-Level Language and Silicon Compiler for Digital Signal Processing*, "Proc. IEEE CICC-85", Portland, May 1985, pp.213-216.

[23] S. Höreth, H. Eveking, *Improving the Performance of a BDD-based Tautology - Checker*, "Advanced Research Workshop on Correct Hardware Design Methodologies", Turin (Italy), 12-14 June, 1991, pp. 387-397.

[24] W. Hunt, *FM8501: A Verified Microprocessor*, " Technical Report 47, The University of Texas at Austin", February 1986.

[25] J.Joyce, *Formal Verification and Implementation of a Microprocessor*, "in "VLSI Specification, Verification and Synthesis"", editors: G. Birtwistle and P.A. Subrahmanyam, Kluwer 1987, pp. 129-157.

[26] W. Lempens, *Symbolic Analysis of Digital MOS Circuits at the Switch Level*, Thesis IMEC - Katholieke Universiteit Leuven Belgium, July 1989.

[27] J.C. Madre, J.P. Billon, *Proving Circuit Correctness using Formal Comparison Between Expected and Extracted Behavior*, " Proc. of the 25th DAC", 1988.

[28] R.E. Tarjan, *Depth First Search and Linear Graph Algorithms*, SIAM J. Computing 1:2, pp. 146-160.

[29] J.Vanhoof, I.Bolsens, S.De Troch, E.Blokken, H.De Man, *Evaluation of High-Level Design Decisions using the Cathedral-II Silicon Compiler to Prototype a DSP ASIC*, "Proc., IFIP Workshop on High Level and Logic Synthesis", ed. G. Saucier, Paris, 30 May-1 June 1990.