In Proc. "Advanced Research Workshop on Correct Hardware Design Methodologies", ed. P. Prinetto, P. Camurati, Turin, June 12-14, 1991. North-Holland.

# *SFG-Tracing:* a methodology of "Design for Verifiability" *

Luc Claesen ¦ Mark Genoe, Eric Verlind, Frank Proesmans, Hugo De Man†

IMEC, Kapeldreef 75, B-3001 Leuven Belgium
phone: +32-16-281203, email: claesen@imec.be

## Abstract.

In this paper a novel and practical methodology of "design for verifiability" for the formal correctness verification of digital (VLSI) designs is presented. This methodology aims at bridging the gap from transistor switch level circuits, as obtained from circuit extraction, up to high level specifications. The *SFG-Tracing* verification methodology inherits its power from the exploitation of the inherent algorithmic information the high level (signal flow graph level) specifications. Given the fact that the circuit designer provides the appropriate reference signals and mapping functions, the methodology has already successfully been experimented on the full formal verification of VLSI circuits of more than 32.000 transistors as extracted from the layout.

## 1 Introduction.

The possibilities offered by the steadily increasing complexities offered by the VLSI technology has resulted in the fact that more and more complex systems can be build on integrated circuits. The realization of complex systems has become design limited instead of technology limited. The chalenge 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 a lot of efforts to check the correctness of designs with respect to their specifications.

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 it is not possible to try out all possible input excitations in these simulations, due to the combinatorial explosion problem in the number of possible patterns. Therefore designers have to chose an appropriate subset of input stimuli for verification by simulations. This method however leaves open the possibilities for undiscovered design errors. This motivates the need for analytic verification techniques that are input pattern independent. The technique of *static timing verification* is an analytic technique that has currently gained industrial acceptance for the verification of the speed performance of circuits.

The analytic verification of the *behavioral correctness* of digital designs with respect to their specifications is however still in its infancy. It is mainly hindered by the problems of combinatorial exploision in handling the mathematical formulas describing the systems at hand.

Formal correctness verification techniques have been investigated already for a few decades in theoretical computer science. Although better insights have been gained in the mathematical

*This research has been sponsored as part of the CHARME and CHEOPS ESPRIT Basic Research Actions.
†professors at Kath. Univ. Leuven

modeling of computer programs, no full correctness proofs of practical computer programs can be done in a realistic way. Formal verification techniques derived from these developed in theoretical computer science have been applied in hardware designs and illustrated by the correctness proofs of small microprocessors using mechanical theorem proving methods [12, 13, 14]. Even for these small sized applications, the correctness proofs require several months of (mechanical theorem proving) expert interaction for conducting the correctness proof. It is also not obvious how design specific theorems and proof strategies can be automatically generated from specifications or how they can be reused in new designs. The promissing approach for the use of theorem provers is in formal design derivation [15] and the proof of generic synthesis primitives.

For the representation and manipulation of Boolean formulas, the ordered binary decision diagrams (OBDD's) [24] is currenlty the best known technique. It is currently used in the verification of combinatorial logic and in logic synthesis. Several additional techniques are still being proposed that improve the efficiencies that can be obtained. Analytic methods [18, 19, 22] have been developed that allow to extract symbolic equations from MOS switch level circuits, that accurately model bidirectional information flow, multiple strengths of nodes and transistors and 'X' behavior. For the verification of finite state machines (modeling the controlers in digital systems) promissing techniques have been worked out [2].

In [28] an alternative method for verification of high level synthesis has been presented, which is based on backannotating the specification with `clock` statements according to the schedule. This methodology still needs to be extended by the verification of the correctness of the schedule.

In [29] a method for the verification of high level synthesis results is presented that is based converting the flowgraphs of the specification and the implementation into a normal form. This can up to now however only be applied for a restricted class of synthesis transformations as are described in [29], and further research is ongoing to make this method more general applicable.

The main breaktroughs in formal verification methods for *behavioral correctness* have been achieved by methods that take advantage by exploiting the circuit structure in the verification algorithms. This is the only way to avoid the problem of combinatorial explosion that results when trying to formulate the correctness problem in a general way (e.g. Boolean formulas) and have a general decision procedure trying to figure out the correctness.

Further along these lines of correctness verification we propose a method called *SFG-Tracing* that exploits the information available in the signal flow graph level specification that describes the algorithms to be implemented.

In this paper we present a new method for the automatic verification from the behavioral signal flow graph specification down to lower implementation levels. These can go down to the 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 is in the verification of high level synthesis results as obtained by the CATHEDRAL silicon compilers [9, 10], but the methodology is 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 (SFG = Signal Flow Graph) specifications down to the implementation, consider the modem pulse shaper and equalizer chip indicated in figure 1 and as designed by Vanhoof e.a. [11]. This system implements the filter flow

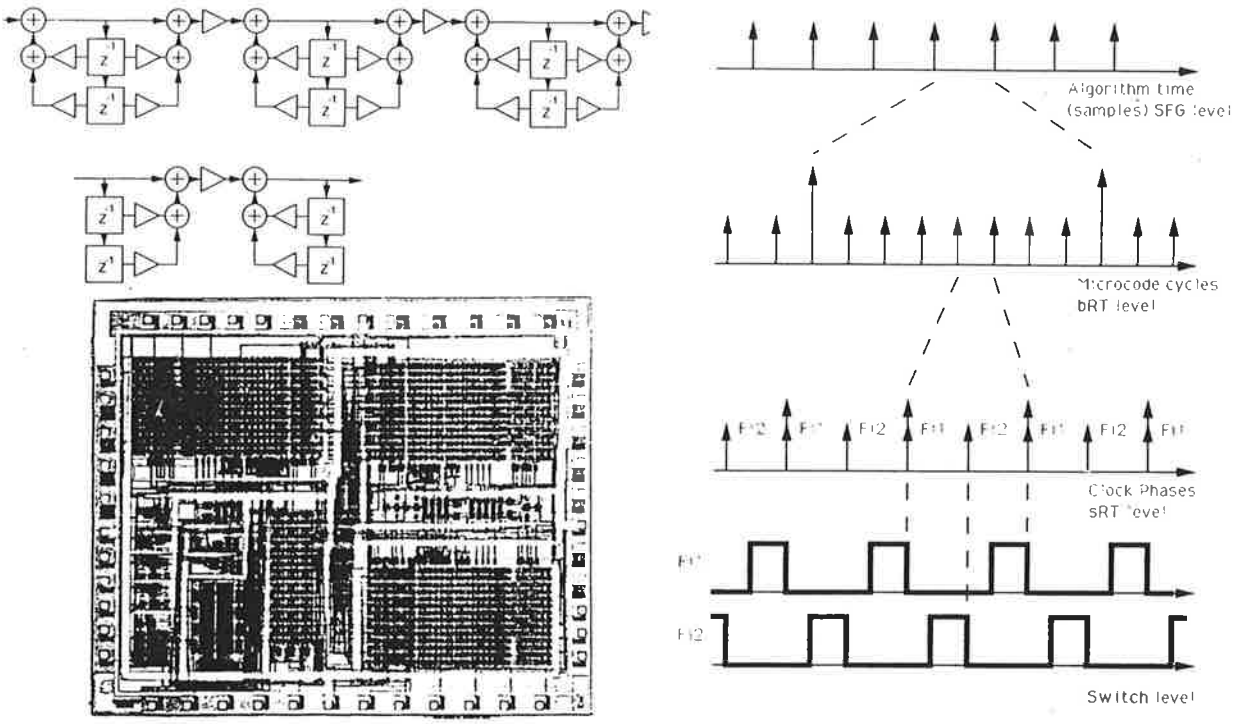



Figure 1: Design & time abstraction levels from SFG (signal flow graph) downto transistor layout, for a receiver pulse shaper and equalizer, containing 3 ALU's of 14 bits as synthesized by CATHEDRAL-II

graph indicated in the top of the figure and can by formaly specified in teh SILAGE language in 70 lines of text. The chip implementation as synthesized by CATHEDRAL-II [11] results in a microcoded architecture with 3 ALU's of 14 bits and consists of more than 12000 transistors. Near the figure is shown the time abstraction from sample periods at SFG level over micro-code instruction cycles, clock phases downto clock waveforms 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 give a short overview of the theoretical background of the *SFG-Tracing* methodology. For a more elaborate explanation of the methodology and the relationship with existing formal verification methods we refer to [5]. The step-by-step application of the methodology has been described by means of the full verification of the switch level implementation with respect to the high level specifications of a small design example (BCD-recognizer) as has been published in [6].

In the next section, we give an overview of the *SFG-Tracing* methodology. The concepts are illustrated in section 4 by the full verification from the layout-extracted transistor netlist up to the high level specification of a small signal processor as has been synthesized by the CATHEDRAL-II system [10].

## 2 *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 preferred, because it reflects the best the circuit implementation. Appropriate symbolic analysis techniques based on Bryant's method [18, 19] for the switch level have been developed and are supported in CAD tools [22, 20, 21].

### 2.1 Flow Graph Specification.

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.

### 2.2 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 $e_j[k]$.

### 2.3 *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:

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

     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:

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

     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.

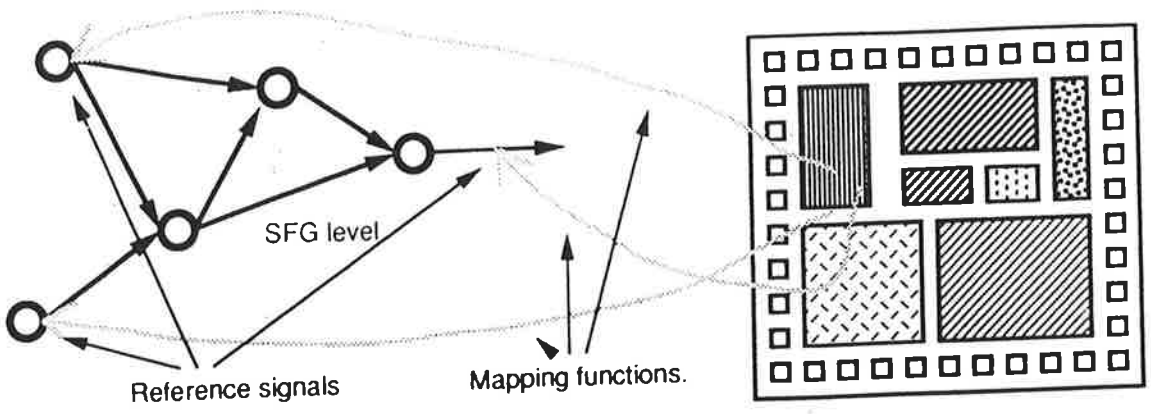



Figure 2: Illustration of the concepts of *reference signals* and *mapping functions* that relate signals in the SFG specification to signals in lower level implementations. (Here down to the switch level).

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.

$$\mathcal{F} : Switch\_signal\_semantics \rightarrow SFG\_signal\_semantics \tag{3}$$

or:

$$\mathcal{F} : \mathcal{B}^{w_i} \rightarrow \mathcal{B}^{w_s} \tag{4}$$

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

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

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

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.

The concept of *reference signals* and *mapping functions* is illustrated in figure 2.

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.

## 2.4 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 cutset of vertices of $\mathcal{G}(V, E)$ where the edges between vertices in the cutset and vertices out of the cutset correspond to the reference signals, related to that subgraph.

## 2.5 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_reference_signals_and_mapping_functions();
     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_reference_signals_and_mapping_functions();` 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 will be 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 will be used for

the specific values in the respective phases. Other signals like reset signals and signals to put the circuit in test mode, will be set to the 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 will be 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 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:

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

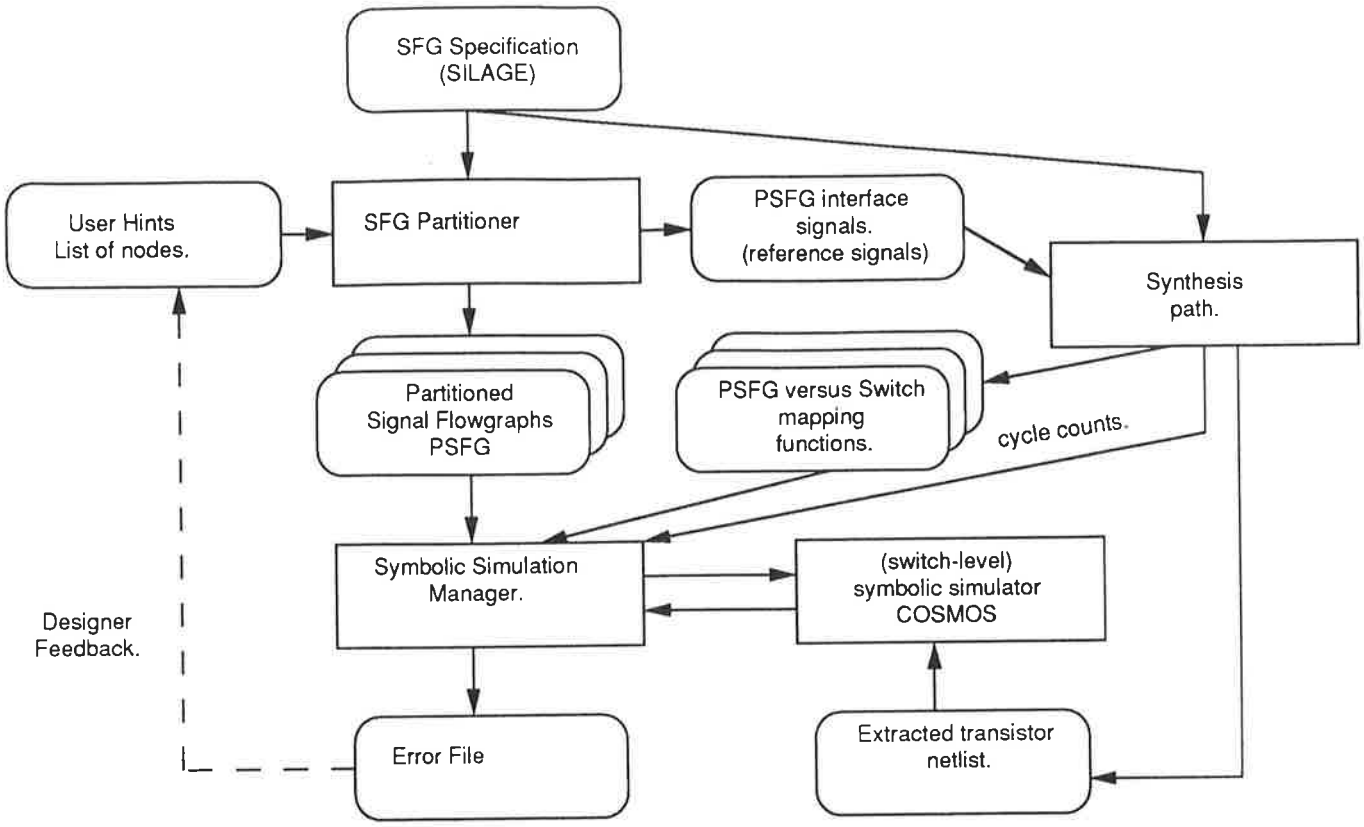



Figure 3: CAD environment for verification by SFG-Tracing

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 [24]. 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 [27] and Simonis [26] can be used to improve the cpu-time efficiency of the verification.

# 3 Integration in a CAD environment.

The methodology of SFG-Tracing is included in the CATHEDRAL CAD environment as indicated in figure 3.

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 have to be 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 has to provide the number of cycles 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", that prepares the simulation commands for the symbolic simulator at hand. After individual symbolic simulations on subgraphs,

the results will have to be verified for correctness. For the symbolic simulation the COSMOS program [22] is used. This will work on the transistor netlist as it is obtained from the layout circuit extraction. In case of inconsistencies for specific subgraphs of the PSFG, the Symbolic Simulation Manager can generate 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.

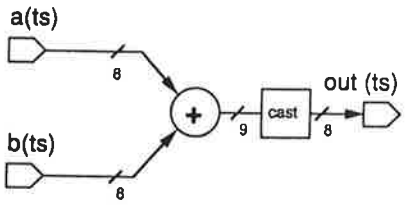
# 4 Design example: A small CATHEDRAL-II processor.

A step-by-step illustration of the *SFG-Tracing* methodology has been described in [6] by means of a BCD-recognizer [3]. In this section the *SFG-Tracing* methodology is illustrated by the verification of a small signal processor [7] that is synthesized by the CATHEDRAL-II system. This verification is done from its transistor netlist up to the SILAGE specification. This application, although simple, includes a datapath, register files, multi-branch microcoded controller, and additional circuitry as necessary for Design for Testability measures. This application illustrates the *SFG-Tracing* verification methodology as applied to one part of a partioned SFG behavioral specification.

The application example **aplusb** adds two streams of incomming 8 bit numbers and is specified in the SILAGE language [8] as follows:

```
#define WORD  fix<8,0>

func main( a : WORD; b : WORD ) out : WORD  =
begin
 out = WORD (a + b);
end;
```




Also indicated is the corresponding SFG graphical representation. The implementation of this specification can be done in various ways. For example bit-parallel, bit-serial [9], with microprogrammed architectures [10] etc... At the SFG level only the relationships between behavioral signal definitions is given. The corresponding flowgraph for this small example is so small that it need not be partioned any further in subgraphs (as is a requirement for most other applications.).

This small application has been synthesized for illustrative purposes into a chip layout by the CATHEDRAL-II system [10, 11] into a dedicated microcoded architecture. (It is only an illustrative example and it is clear that such a small application should not be synthesized as a microcoded architecture). The datapath and the controller are shown in figure 4.

The datapath includes i/o pads, a small ALU, mux's and register files. The controller implements an initialization sequence of 7 cycles as well as a steady state operation sequence of 3 cycles per sample in the high level specification. The controller consists of a program conter, instruction rom, and multi-branch controller and instruction register. After module generation and floorplanning the layout consisting of 1935 transistors is generated 5. This includes all the circuitry for testability such as scanpaths etc. It is from the layout extraced transistor netlist that the verification is done.

Notice that in the specific implementation of this **aplusb** problem as a microcoded architecture several additional state registers are included (such as i/o pads, register files, program counter, instruction register,...) that are not seen in the high level specification in SILAGE. The verification by means of SFG-Tracing, avoids the need to know the exact encodings of the

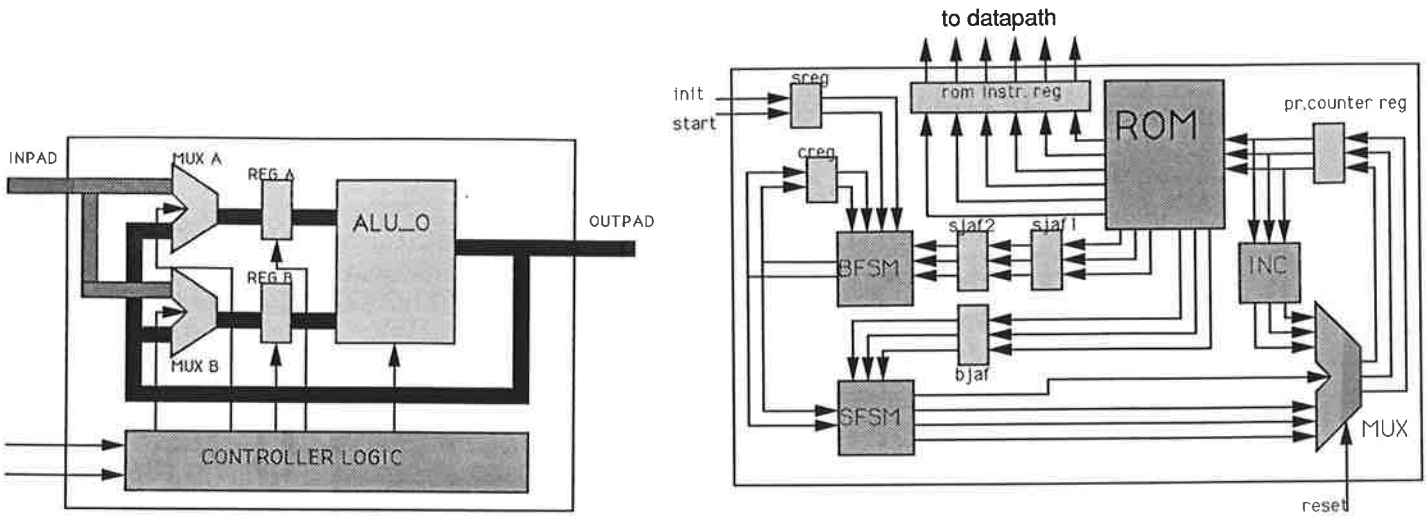



Figure 4: Datapath and controller for **aplusb** as synthesized by CATHEDRAL-II.

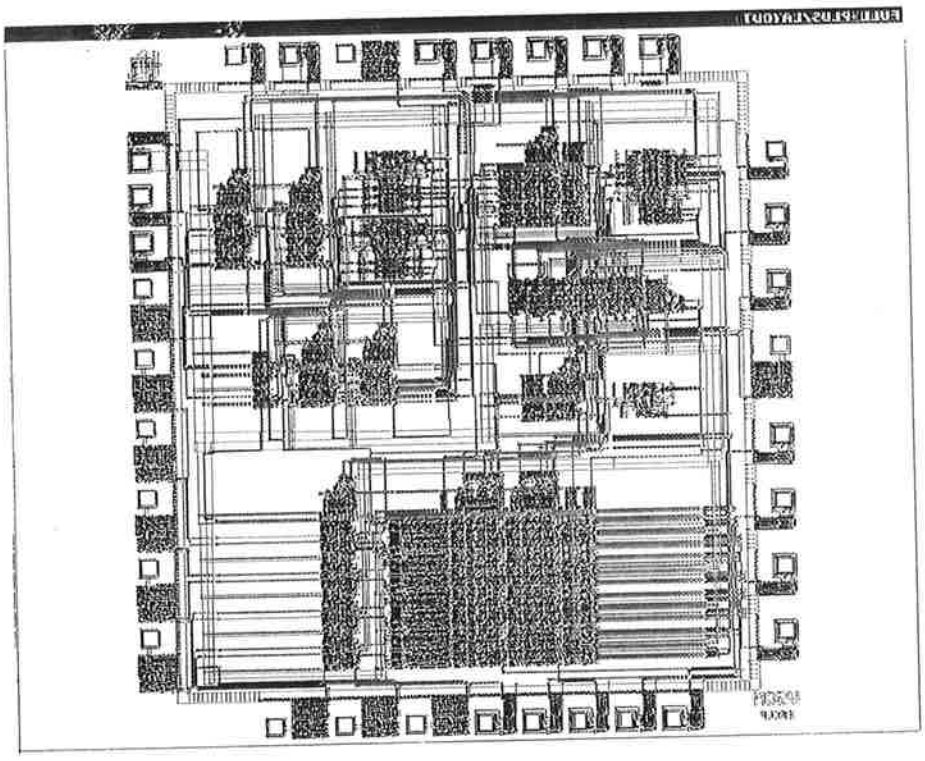


Figure 5: Layout of the **aplusb** application example as automatically synthesized by CATHEDRAL-II.

states in most of these registers, because only the reference signals that are also available in the SILAGE specification are of importance (`a, b, out`). By providing the mapping functions of these signals in the high level specification with their implementation in specific cycles in the implementation, the effect of the controller can be eliminated by the symbolic evaluation at the switch level in COSMOS. The formal verification is performed by considering the *intialization* sequence, the *steady state* operation and the *test* sequence. The full verification of this `aplusb` example [7] takes 11.42 cpu seconds on a DEC-3100.

A more elaborated description of the verification of this design example is available in [7].

## 5 Conclusions and Future Work.

In this paper a methodology of "Design for Verifiability" has been presented that can be used for the full formal verification of hand made as well as automatically synthesized designs. In contrast to constraining the design style, no special requirements are posed (other than are required in addition to what is expeced in "Design for Testability" methodologies [16]. The underlying assumptions of the methodology are that both a formal high level specification is available and that reference signals and mapping functions assist in subdividing a huge and complex problem in manageble pieces.

The *SFG-Tracing* methodology is currently being worked out for proving the correctness of the synthesis results in CATHEDRAL-I [9] and CATHEDRAL-II. Design applications as synthesized by both compilers have already successfully been verified by the above mentioned techniques. The largest *SFG-Tracing* application up to now has been a modem chip of more than 32.000 transistors. The COSMOS [22] compiled-code switch-level simulator is used as a symbolic simulator in the algorithm.

## Acknowledgements.

The authors hereby wish to thank Randy Bryant and the whole COSMOS team for making the COSMOS system available to perform the research mentioned in this paper. The authors also thank the partners in the ESPRIT CHARME and CHEOPS Basic Research Actions for the fruitful cooperation and interesting discussions on the subject of formal hardware verification methods.

## References

[1] C.F. Kurth, keynote speech, ECCTD'83, Stuttgart 5-9 Sept. 1983.

[2] 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.

[3] P.Camurati, T. Margaria, P. Prinetto, "Application selection: Finite State Machines", report ESPRIT CHARME-PDT-1.A-01, July 1, 1990, pp. 6-11.

[4] P. Camurati, P. Prinetto, "Formal verification of hardware correctness: introduction and survey of current research", *IEEE Computer*, July 1989, pp. 8-19.

[5] 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.

[6] 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. EUROASIC-91 Conference*, 27-31 May 1991, Paris.

[7] 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", Internal report IMEC, 1 February 1991.

[8] P. Hilfinger, "Silage, a high-level language and silicon compiler for digital signal processing", *Proc. IEEE CICC-85*, Portland, May 1985, pp.213-216.

[9] R. Jain, F. Catthoor, J. Vanhoof, B. Deloore, G.Goosens, N. Goncalves, L. Claesen, J. Van Ginderdeuren, J. Vandewalle, H. De Man, "Custom design of a VLSI PCM-FDM transmultiplexer from system specifications to circuit layout using a computer aided design system", *IEEE Transactions on Circuits and Systems*, Vol. CAS-33, No.2, pp. 183-195, February 1986.

[10] 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.

[11] 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", *Proceedings, IFIP Workshop on High Level and Logic Synthesis*, ed. G. Saucier, Paris, 30 May-1 June 1990.

[12] W. Hunt, "FM8501: A verified microprocessor", *Technical Report 47, The University of Texas at Austin*, February 1986.

[13] 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.

[14] G. Birtwistle, B. Graham, "Verifying SECD in HOL", *in "Formal Methods for VLSI Design"*, editor: J. Staunstrup, Elsevier Science Publishers B.V. (North Holland), pp. 129-177.

[15] M.P. Fourman, "Formal System Desing", *in "Formal Methods for VLSI Design"*, editor: J. Staunstrup, Elsevier Science Publishers B.V. (North Holland), pp. 191-236.

[16] H. Fujiwara, "Logic Testing and Design for Testability", *Computer System Series*, The MIT Press, ISBN 0-262-06096-5, 1985.

[17] R.E. Bryant, "A Switch-Level Model and Simulator for MOS Digital Systems", *IEEE Transactions on Computers*, Vol. C-33, No.2, February 1984, pp. 160-177.

[18] R.E. Bryant, "Algorithmic aspects of symbolic switch network analysis", *IEEE Transactions on Computer-Aided Design*, Vol. CAD-6, No. 4, July 1987, pp. 618-633.

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

[20] P. Herrebout, "BOTRYS: A program for the symbolic analysis of MOS circuits at the switch level", Thesis IMEC - Katholieke Universiteit Leuven Belgium, July 1988.

[21] W. Lempens, "Symbolic analysis of digital MOS circuits at the switch level", Thesis IMEC - Katholieke Universiteit Leuven Belgium, July 1989.

[22] R.E. Bryant, D. Beatty, K. Brace, K. Cho, T. Sheffer, "COSMOS: A Compiled Simulator for MOS Circuits", 24th Design Automation Conference, pp. 9-16, 1987.

[23] 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.

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

[25] R.E. Bryant, "On the Complexity of VLSI Implementations and Graph Representations of Boolean Functions with Application to Integer Multiplication", *report Carnegie Mellon University*, September 27, 1988.

[26] H. Simonis, "Formal Verification of Multipliers", *in "Formal VLSI Correctness Verification"*, Ed. L.Claesen, North-Holland, 1990, pp. 267-286.

[27] 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.

[28] F. Corella, R. Camposano, R. Bergamaschi, M. Payer, "Verification of Synchronous Sequential Circuits Obtained from Algorithmic Specifications", *Proc. International Workshop on Formal Methods in VLSI Design*, ed. P.A. Subrahmanyam, Miami, January 9-11, 1991.

[29] F. Feldbusch, R. Kumar, "Verification of Synthesized Circuits at Register Transfer Level with Flow Graphs", *proceedings IEEE EDAC Conference*, 25-28 February 1991, Amsterdam, pp.22-26.