A VHDL-AMS Compiler and Architecture Generator for Behavioral Synthesis of Analog Systems*

Alex Doboli, Ranga Vemuri
Digital Design Environments Laboratory, Department of ECECS
University of Cincinnati, Cincinnati, OH 45221-0030
Email: {adoboli, ranga}@ececs.uc.edu

Abstract

This paper presents a complete method for automatically translating VHDL-AMS behavioral-specifications of analog systems into op amp level net-lists of library components. We discuss the three fundamental aspects, that pertain to any behavioral synthesis environment: the specification language, the rules for compiling language constructs into a technology-independent, intermediate representation, and the synthesis (mapping) of representations to net-lists (topologies) of library components, so that performance constraints are satisfied. We motivate the effectiveness of the method by presenting our synthesis results for 5 examples.

1. Introduction

Any automated synthesis tool addresses two distinct, but related, facets of a design: functionality (behavior) and performance constraints. Recent research on *circuit synthesis* [13] [19] [10] exclusively concentrates on using performance attributes for guiding the synthesis task. They assume a known circuit-topology, and search for the transistor dimensions, that optimize performance attributes. This approach is reasonable because usually circuit functionality is overwhelmingly simple, as it corresponds to the complexity of a single operation, i.e. addition, integration, etc. As opposed to circuit synthesis, *behavioral system-synthesis* also considers functionality, while optimizing performance criteria [5]. System functionality is described with a specification language as equation sets, transfer functions, or algorithmic representations. For conducting the traditional synthesis activities, a behavioral specification must be first translated into a structural net-list of analog circuits. This step is not a trivial task, as it is not a one-to-one mapping of the specification constructs onto circuit net-lists.

*This work was sponsored by the USAF, Air Force Research Laboratories, Wright Patterson Air Force Base under contract number F33615-96-C-1911

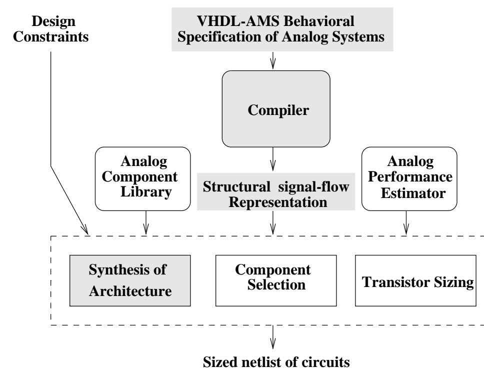

Figure 1. Design flow of the VASE synthesis environment

This paper proposes a method for translating behavioral, system-level specifications with VHDL-AMS (Very High Speed Integrated Circuits Description Language for Analog Mixed Systems) [1] into a structural net-list of components, that is used for the subsequent synthesis steps. We focus on the three fundamental problems, that are specific to any behavioral synthesis environment: the *specification language*, the *translation* of language constructs into their intermediate representations, and the *mapping* of representations onto structural net-lists of electronic components. Our synthesis-oriented VHDL-AMS subset is minimal, however, it includes all language constructs, so that the description power of VHDL-AMS is preserved. Besides, for synthesizing more effective hardware structures, we indicate some meaningful language restrictions/annotations. We suggest translation (compiling) rules for all VHDL-AMS constructs present in the subset. They convert a specification into a set of interconnected signal-flow graphs, and a control part for configuring the flow of signals through the net. Finally, we discuss our mapping algorithm, that synthesizes the signal-flow representation to an op amp level net-list of library circuits [7]. We designed a *branch-and-bound* [9] based mapping algorithm, that looks for the net-list, that satisfies all imposed performance constraints, and minimizes the

overall ASIC area. The algorithm uses Analog Performance Estimation Tools [17] [4] for ranking the visited solutions with respect to their performance attributes. As our experience showed, having a technology-independent (compiling) and a technology-dependent (mapping) step exposes the entire design to more optimization opportunities.

Figure 1 outlines the overall design-flow of the VASE (VHDL-AMS Synthesis Environment) behavioral synthesis tool. The design aspects addressed by this paper, are depicted as shadowed boxes. VASE accepts VHDL-AMS input specifications, that are first compiled into a structural signal-flow representation. The representation is used for exploring alternative implementation solutions by performing three tasks: *synthesis of architecture* maps the representation into a structure of components at the op amp level, *component selection* picks from a component library [7] real circuit topologies for the op amps, and finally, *transistor sizing* decides the physical dimensions for all transistors of a design. The three synthesis tasks are guided by Analog Performance Estimation Tools [17] [4], that compute approximate values for performance measures of a system. By combining our existing performance-estimation tools with the design-space exploration algorithms discussed in this paper, a complete environment for automated, behavioral-synthesis of systems was obtained.

The organization of the paper is as follows. Section 2 provides some related work on analog synthesis. Next, we discuss our VHDL-AMS subset for synthesis, and Section 4 describes the translation of the main subset constructs into their signal-flow representations. Section 5 presents the algorithm for architectural synthesis. Section 6 shows the effectiveness of our behavioral synthesis flow for some real-life examples. Finally, we provide our concluding remarks.

2. Related Work

Most of the recent analog-circuit synthesis tools are based on *optimization* methodologies [15]. They assume a known circuit-topology, and search for the physical design parameters (transistor dimensions), that optimize the performance attributes. These tools start by building the circuit performance model, that relates performance attributes to design parameters. Various methods are suggested for this task. Simplified symbolic equations are calculated for performance parameters in [13]. [10] includes, as terms, the Kirchhoff's laws into the cost function for the optimization algorithm, and [16] relies on the square-law equations of the CMOS op amps. Next, the performance model is used by an optimization algorithm, i.e. simulated annealing, geometric programming, etc., to find values for the physical design parameters. Finally, the quality of a solution point is evaluated through different simulation methods, i.e. symbolic simulation [13], general-purpose simulators (i.e.*SPICE*) [19], or approximate simulation [10].

As opposed to circuit synthesis, *behavioral system-synthesis* also considers functionality as a design dimension for the synthesis task. [5] translates system-level specifications into input representations for the ARCHGEN synthesis tool. However, their method targets only filters, described using a specialized C++ library. In [6], mixed-signal systems are represented with a limited subset of VHDL-AMS. Specifications are mapped to KIR, their representation for synthesis. Nevertheless, the authors do not address the issue of translating VHDL-AMS programs into KIR-graphs.

In this paper, we present a well-defined VHDL-AMS subset for specification of analog systems. Besides, we introduce a two-step methodology for converting such specifications into a net-list of hardware components. First, a compiler translates a specification into a technology-independent structural representation. Then, the representation is mapped onto a net-list of components at the level of op amps. Having the two translation steps increases the effectiveness of the synthesized structure, as now, the design is exposed to more optimization opportunities.

3. VASS: a VHDL-AMS Subset for Behavioral Synthesis of Analog Systems

Based on our experience with a large set of real-life examples [3], we suggest that an analog system can be modeled behaviorally as two interacting sub-components: one with *continuous-time* functionality, and the second with *event-driven* behavior. The first performs continuous-time processing of analog input signals. It can have multiple modes of behavior, that are described as sets of differential and algebraic equations (DAE), transfer functions or algorithmic descriptions. Depending on specific functioning conditions (events), the event-driven part generates control signals for selecting among the distinct modes of continuous-time functioning. Events originate in the continuous-time part, or the external environment. The main advantage for synthesis is that this model differentiates the essentially distinct parts of a system (with respect to their functionality and performance characteristics), that go through distinct design steps, in our synthesis environment. At this point, its worthwhile to stress that, in the context of our work, by *event-driven part we also mean analog sub-components having this kind of functionality*. The behavior of analog circuits [12] i.e. voltage comparators, zero-cross detectors, Schmitt triggers, is a typical event-driven one. Although ultimately, an event-driven behavior can also be expressed in continuous-time, we prefer the first style, as it reduces simulation time and simplifies synthesis.

The rest of this section describes VASS, which is an abbreviation for VHDL-AMS Subset for Synthesis. When defining the subset, we targeted two distinct objectives: *description power* and *synthesizability*. First, the subset must include all required language constructs for expressing the

```
ENTITY telephone IS
 PORT (
    QUANTITY line: IN real; -- IS voltage
    QUANTITY local: IN real; -- IS voltage
    QUANTITY earph: OUT real;
                  -- IS voltage
                  -- limited
 )                -- drives 270 O at 285 mV peak
END ENTITY;
ARCHITECTURE behavioral OF telephone IS
   QUANTITY rvar: real;
   SIGNAL c1: bit;
BEGIN
(1) earph == (Aline * line + Alocal * local) * rvar;
(2) IF (c1='1') USE
      rvar == r1c;
    ELSE
      rvar == r1c + r2c;
    END USE;
    PROCESS (line'ABOVE(Vth)) IS
    BEGIN
      IF (line'ABOVE(Vth) = TRUE)
      THEN  c1 <= '1';
      ELSE  c1 <= '0';
      END IF;
    END PROCESS;
END ARCHITECTURE;
```

Figure 2. Behavioral specification of the receiver module

functional aspects of an analog system. According to our behavioral model, VASS can describe continuous-time and event-driven behavior, and their interactions. Second, all constructs in VASS should be realizable in hardware. Because it is oriented towards simulation, VHDL-AMS must be partially "adjusted", so that it can be effectively synthesized. The adaption involves both restricting some of the language constructs, and augmenting the language with missing synthesis-oriented constructs. A *for*-loop inside a *procedural* statement [1] exemplifies the need for restrictions. Its semantics is defined in terms of a discrete counter, but which is difficult to be realized in a continuous signal-flow structure. Instead, we impose that the number of iterations is statically known for each *for*-loop, so that its body can be unrolled. Second, it is common, that a system terminal must have a low output impedance because of its external connections. This requirement can be achieved by synthesizing a specific output stage, that can not be inferred, unless it is accordingly indicated in the specification (i.e. through annotations).

The overall structure of a VASS program consists of entity declarations, architecture bodies, package declarations, and package bodies, with the following remarks:

- VASS accepts *signal* [1], *quantity*, and *terminal* ports. *Terminal* ports describe the structural interconnection of a system with its external environment. Still, we impose that, for each terminal port, only one of its *through* (current) or *across* (voltage) quantities is used in a specification. This accommodates the rest of a behavioral description, in which only one of the facets (current/voltage) of an analog signal is utilized.
- Quantities define signals with a continuous-time behavior, and *signals* those with an event-driven behavior. VASS admits only quantities of *nature type* (floating-point or a composite type with elements of nature type), as they naturally represent analog signals. *Signals* are of nature or bit-vector types.
- Continuous-time behavior can be *implicitly* formulated by describing DAEs as *simple simultaneous* and *simultaneous if/case* statements.
- *Procedural* statements *explicitly* describe continuous-time behavior as a sequence of assignments, branches, loops, and function calls. The simulation semantics of *while*-loops can be preserved after synthesis, only if constraints are defined, so that the loop denotes a sampling functionality. We impose that signals referred inside a *while*-loop are constant while the loop body executes, and it is sufficient if signals assigned inside the loop are produced at time intervals equal to the loop delay. Constraints are required as the simulation cycle assumes that a loop is always executed in zero time steps, while in reality, a non-zero delay can occur.
- VASS includes *process* statements for specifying event-driven behavior. However, the peculiarities of our design problem can be exploited for restricting process definitions, so that they result in more effective structural descriptions. Synchronization and inter-process communications are common for discrete-time systems, but can hardly be accommodated with a continuous-time behavior. Thus, it is realistic to consider a simplified model of process interactions, so that the simulation cycle has not to be explicitly implemented in hardware. We assume that *processes react to events, and after resuming, they execute their entire body (calculate control signals), and then suspend*. Process definitions do not include *wait* statements. Events originate either in the continuous-time part (events on *signal 'above'*), or the external environment (events on ports).

[1]To distinguish VHDL-AMS signals from physical signals, we will indicate the first in italics.

We successfully specified in VASS a set of 11 real-life examples [3]. Figure 2 depicts a simplified version of our specification for the receiver module of a telephone set [14]. The main functionality of the receiver is to provide an audible signal to the earphone of the telephone set. It amplifies, with different gains, incoming signals transmitted from the calling part, and those produced locally by its own microphone amplifier and transmitter module. Besides, it automatically compensates losses introduced by different telephone line lengths. The output has a signal limiting capability, and is capable of driving a 270 Ω load at 285 mV peak amplitude. The VASS specification of this example is depicted in Figure 2. The output voltage *earph* is a weighted sum of the input voltages *line* and *local*. The resultant value is multiplied with a variable value $rvar$, that models the variable compensation resistance. The compensation algorithm is represented by a $process$ statement, which, depending on the comparison between quantity *line* and a threshold voltage V_{th}, selects the corresponding compensation value.

As opposed to VHDL-AMS, our subset includes a declarative mechanism for describing properties of quantities and ports. Such a mechanism is required for synthesis, as the behavior is strongly heterogeneous with respect to signal types and characteristics. In Figure 2 all port quantities are annotated to indicate their kind (voltage), and output $earph$

is also augmented with information about its limiting and driving characteristics. Besides, a declarative description style can be complementary to traditional description methods. Typically, the behavior of filters is expressed as transfer functions [12]. Nevertheless, if the transfer function is provided, then also the filter type, and its structure are decided. Instead, we could describe signal properties along the signal path, i.e. frequency ranges, and let the synthesis tool infer an appropriate filter type. Currently, VASS accepts annotations, that describe signal properties such as kind (voltage, current), value and frequency ranges, and impedances at terminal ports.

We have implemented a compiler, that translates a specification in VASS, into a structural signal-flow representation for the continuous-time part, and a Finite State Machine (FSM) for the event-driven part. The translation rules embedded in our compiler are discussed next.

4. Compiling VASS specifications for synthesis

VHIF (VASE Hierarchical Intermediate Format) [2] is a representation for structural description of analog systems. It can express both continuous-time and event-driven behavior, and their interactions. Besides, at the time we designed VHIF, we assured that all blocks in the representation are implementable with electronic circuits from a library [7]. Continuous-time behavior is denoted as signal-flow graphs, that include exact knowledge about flows and processing (operations) of signals. Event-driven behavior (including *event-driven analog functionality*) is traditionally represented by a Finite State Machine (FSM). Each state denotes a set of concurrent operations, and states are interconnected through arcs for showing the execution flow. Also, arcs can be controlled by conditions (i.e. the arc between *state 3* and *4*), so that conditional behavior is achieved. Operations pertaining to a state are described as a data-path structure. Figure 3a depicts an example in VASS, and Figure 3b presents its corresponding VHIF representation. Because it explicitly captures the flow of signals among operational blocks, we use VHIF as the internal representation for our synthesis environment. The remaining part of this section discusses the main translation rules for converting VASS programs into VHIF.

Except for cases where input and output signals are explicitly known or can be inferred, *simple simultaneous* statements can not be mapped into a unique signal-flow structure. Each structure represents a distinct "solver" for the DAE set. Our synthesis tool considers all VHIF topologies that "solve" a DAE set, while searching for the best implementation.

Procedural statements are translated into a pure functional block. The block computes analog outputs depending on its input signals (analog or digital), and without relying on any state (memory) information. This interpretation accommodates the rule that no information is saved between

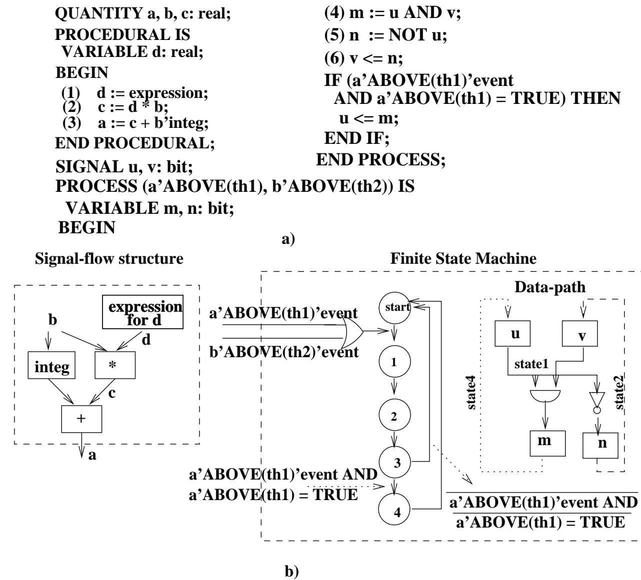

Figure 3. Structural representation of a systems

consecutive invoices of a procedural [1]. The two most interesting language aspects pertaining to procedurals correspond to instruction sequencing, and *while*-loops.

A designer describes the computational flow of a system by indicating the *sequence* (order) in which instructions are performed. In a signal-flow representation (thus, without any memory cells), this order is preserved, *if and only if* the output of the structural block for an instruction is an input of the block for a following instruction. Moreover, the block interconnection for an instruction sequence is inferred from the manner in which variables/quantities are assigned and referred (their data dependencies): the output of the block for a variable/quantity assignment is connected as input to the block for an instruction, that refers the same variable/quantity. The correct ordering of *instruction 1* and *instruction 2*, in Figure 3a, is achieved by interconnecting their blocks as in Figure 3b.

While-loops are correctly translated into VHIF, if the constraints, presented in Section 3, for input and output signals are satisfied. These constraints allow to compile a $while$-loop into a block structure, that has a sampling functionality. Compiling a $while$ statement is also difficult, because of the "behavior" of its conditional expression. Before entering the loop, a conditional uses signal values calculated outside the loop, but after the loop is entered, the conditional refers to values computed by the loop body itself. If the conditional expression were implemented as a single block-structure, then its inputs have to be multiplexed. To avoid the laborious synthesis of multiplexing signals, we decided to transform a $while$ loop into a semantically equivalent construction, but having two distinct blocks for evaluating the conditional. Figure 4a shows the transformed construction, and its corresponding block-structure is depicted in Figure 4b. The filled block evaluates the conditional, that

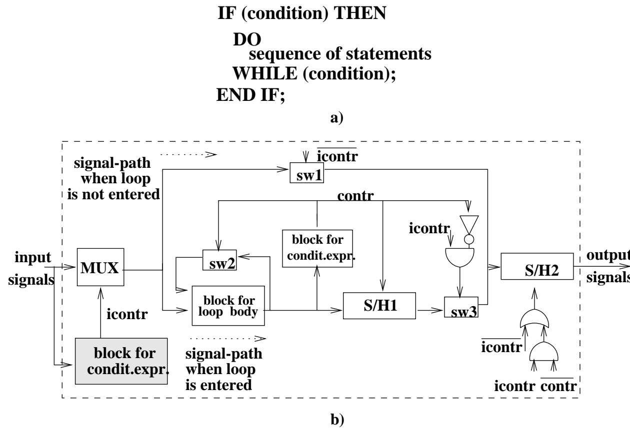

Figure 4. Translation of a $while$ **statement**

decides if the $while$ loop is entered or not. If signal *icontr* is true, inputs are routed to the block, that implements the loop body. As long as the conditional is true, signal *contr* keeps the switch *sw2* open, and the sample-and-hold circuit [12] *S/H1* trails the output of the loop body. The closed switch *sw3* avoids that *S/H2* gets erroneous inputs during the execution of the loop body. If the conditional is false, switch *sw3* opens, and the output of *S/H1* is saved by the sample-and-hold circuit *S/H2*. *S/H2* preserves the outputs constant, while the loop body executes.

The effectiveness of the synthezised structure for a *process* statement can be increased, if translation rules contemplate the following two aspects. First, the structure should be fast enough to accommodate the continuous-time behavior. Accordingly, we decided to translate a process into a structure with a high concurrency. Second, memory modules are expensive hardware resources (area, manufacturing), hence, whenever possible, our method reduces their use in the structural representation. In general, VHDL-AMS *signals* require memory cells for their drivers, apart from those for their current value. We restrict *signals*, so that each is realized as one memory block. We impose that a *signal* can not be referenced, after being assigned a value. It is worth being mentioned, that reducing the number of memory modules is also harmonious with the simplified model for process interactions.

We refer to Figure 3 for outlining our translation rules for *process* statements. Figure 3a shows the VASS code for a process, that is resumed by events on *signals a'ABOVE(th1)* and *b'ABOVE(th2)* (thus, it has the two signals in its sensitivity list). Figure 3b depicts the VHIF representation for the process. It has a $start$ state to indicate the *suspended* status of the process. Resuming the process by an event is denoted by the transition from $start$ to *state 1*. The transition is controlled by a logical *OR* between events on the two signals in the sensitivity list. As we assumed that only one event occurs at a time, no special arbitration of events is required.

VHIF states denote a set of operations that can execute concurrently. As we want to produce a process structure with a high concurrency, we group two or more successive instructions into the same state, if they can run concurrently, hence, there are no data-dependencies between them. If the current statement has a data-dependency with one of the statements of the current state, then a new state is created and the statement is associated to it. In Figure 3a, *assignments 4* and *5* are associated to *state 1*. *Assignment 6* is data-dependent on *assignment 5*, due to the variable *n*, and is linked to *state 2*.

The next section presents our algorithm for architecture synthesis. It automatically maps a VHIF representation into a net-list of electronic components, so that the ASIC area is minimized and the rest of the constraints are met.

5. Architecture Generator for Analog Systems

This section presents our algorithm for architecture synthesis. Its goal is to map a set of signal-flow graphs, and the FSM of the VHIF representation for a system into a net-list of components, so that all performance constraints are satisfied, and the total ASIC area is minimized. We discuss only the algorithm for mapping signal-flow graphs, as this is a more challenging problem. For analog systems, the FSM has very often a simple structure, that can be entirely mapped to analog circuits [12], i.e. Schmitt triggers, zero-cross detectors, sample-and-hold circuits, etc. More complex structures, including any extensions for mixed-signal synthesis, can be obtained using well known digital-synthesis methods [8].

A general algorithm for architecture synthesis finds a solution by exploring a vast solution-space. Hence, the search can be unpractically long, unless this space is organized and explored in a *hierarchical* manner [11] [17]. In the overall synthesis environment, our work concentrates on the topmost level of the hierarchy, as it considers systems at the level of op amps. Besides, having an op amp level exploration step is also beneficial for *design-space pruning*, as unrealistically expensive solutions are discarded very early in the exploration process. Solutions that prove to be attractive are further analyzed by using our performance estimation tools at lower levels of abstraction [17] [4].

Our algorithm contemplates different op amp-level mappings for a VHIF representation, while trying to minimize the overall ASIC area. The goal of area minimization is addressed by analyzing two possibilities of *hardware sharing*: between blocks in different signal paths, and between blocks of the same signal-flow path. Blocks in distinct signal paths can share the same component, if they have identical inputs, and perform similar operations. Blocks of the same signal-flow path can share a component, if the component implements the overall functionality of the blocks. Any optimal algorithm must analyze all possible mappings, as the two sharing options can conflict each other. Although the problem of architecture synthesis is NP-hard [9], we decided

```
procedure mapping (signal-flow, cur block, opamp nr) is
◇forall sub-graph ∈ signal-flow, that have cur block as
  output block and are mappable to one library-component;
  in decreasing order of the number of blocks in sub-graph do
    if sharing is possible and library component for sub-graph
      exists in net-list then
        make interconnections for sub-graph in net-list;
        if signal-flow was completely mapped then
        • call analog performance estimation Tools, and
          save mapping solution, if it is best so far;
       else
          signal = select an input signal of sub-graph;
          mapping (signal-flow, block ∈ signal-flow with output
           signal, opamp nr);
        end if
    end if
  □if (opamp nr + nr of opamps in comp) * MinArea <
      current best then
        allocate circuit for sub-graph, and add it to net-list;
        if signal-flow was completely mapped then
        • call analog performance estimation tools, and
          save mapping solution, if it is best so far;
        else
          signal = select an input signal of sub-graph;
          mapping (signal-flow, block ∈ signal-flow with output
            signal, opamp nr + nr of opamps in sub-graph);
         end if
     end if
  end for
end procedure
```

Figure 5. **Algorithm for architecture synthesis**

to solve it optimally by using a *branch-and-bound* algorithm [9]. VHIF representations for real-life examples tend to be of small or medium size, so that it is practical to map them optimally. Besides, an optimal algorithm can be used as a reference for designing future mapping heuristics.

The pseudo-code of our algorithm for architecture synthesis is depicted in Figure 5. It maps the signal-flow graph denoted by variable *signal-flow* into the minimum area net-list indicated by variable *net-list*. Variable *opamp nr* represents the number of op amps in a partial mapping. As *branch-and-bound* is a popular algorithm [9], we show only the three elements, that are specific to our problem:

- *Branching rule*: marked with ◇ in Figure 5, describes how distinct mapping solutions are produced for a partial solution-point. It distinguishes all VHIF block-structures, pointed by variable *sub-graph*, with *cur block* as their output block, and which are directly mappable to library components [7]. Besides, the branching rule contemplates two kinds of transformations, in a signal-flow graph. *Functional transformations* replace a particular block structure with a distinct, but semantically equivalent structure, i.e. for improving bandwidth, an op amp is replaced by a chain of two op amps with lower gains [12], or two non-inverting amplifiers are substituted for/by two inverting amplifiers [12], etc. Transformations pertaining to circuit *interfacing* introduce additional circuits, i.e. follower circuits [12], or various input/output stages [12], etc., for diminishing loading/coupling effects among interconnected components.
- *Bounding rule*: marked with □ in Figure 5, eliminates a partial solution, if it finds that the minimum area, that can result, is greater than the area of the best solution found so far (variable *current best*). The minimum area for a partial mapping is estimated using value *MinArea*, the minimum area of an op amp (with transistors sized to the minimum dimensions).
- *Sequencing rule*: is a heuristic rule, that decides the order in which branching alternatives are traversed. A good sequencing rule can dramatically improve the speed of the overall algorithm, as the bounding rule becomes very efficient, if a high-quality solution is found early. Our sequencing rule approximates the ASIC area with the number of op amps in the design. Thus, branching alternatives, that map a higher number of blocks to one library component, are visited first in the attempt to find early a mapping with reduced number of op amps. Besides, the algorithm first analyzes the case, where blocks in *sub-graph* share existing components in the net-list, and then maps *sub-graph* to its dedicated hardware component.

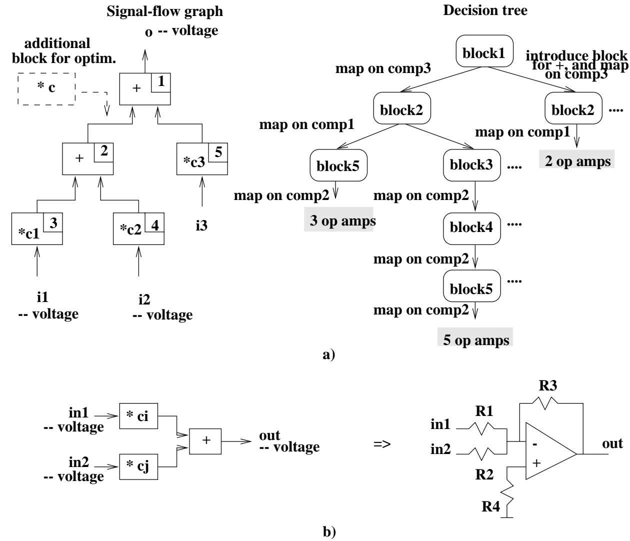

Figure 6. **Example for architecture synthesis with branch-and-bound**

The algorithm calls (marked with • in Figure 5) two analog performance estimation tools [17] [4], that calculate approximate performance attributes (UGF, slew rate, power) and hardware area by instantiating op amps with precise circuit topologies, and sizing their transistors.

A signal-flow graph and a fragment of the *decision tree* for its mapping are shown in Figure 6a. Each node in the

Application	VASS Specification				VHIF Representation			Synthesis Results
	continuous-time	quantities	event-driven	signals	nr.blocks	nr.states	data-path	
Receiver Module	4	4	4	2	6	4	1	2 amplif., 1 zero-cross det.
Power Meter	8	6	3	3	6	2	2	2 zero-cross det., 2 S/H, 2 ADC
Missile Solver	4	9	-	-	13	-	-	2 integ., 1 anti-log.amplif. 4 amplif., 1 log.amplif. (reduced)
Iter.Equat. Solver	1	1	4	2	6	2	2	3 integ., 1 S/H 1 diff. amplif.
Function Generator	2	2	4	3	4	2	1	1 integ., 1 MUX, 1 Schmitt trigger

Table 1. Behavioral synthesis results for 5 real-life applications

tree corresponds to a partial-solution point (a specific value for variable *cur block*), and arcs are annotated by the corresponding mapping decisions. The number of used op amps are indicated for each complete mapping of the signal-flow graph. The algorithm uses a library of patterns, that relate VHIF block-structures to electronic circuits in the component library. A block-structure (referred as *comp1*) and its corresponding electronic circuit are exemplified in Figure 6b. The example uses similar patterns for a block, denoted as *comp2*, that multiplies an input voltage by a constant, and a block, named as *comp3*, that adds two input voltages. Its worth mentioning that for *block1* the branching rule introduces an additional block *comp2* (depicted as a dashed box), when finding the mapping with only 2 op amps.

6. Examples

This section presents our experience on applying the behavioral synthesis flow, discussed in the paper, for a set of 5 real-life examples. We described in VASS each of the examples, next, we compiled the specifications into VHIF representations, and finally, we mapped the representations to net-lists of components. We observed the correctness of our compiling rules by checking the produced VHIF representations. Two of the net-lists (*Receiver Module* and *Power Meter*) were exposed to selection of circuit topologies and transistor sizing, the design steps which are next to behavioral synthesis. The produced circuits were simulated, and their output signals were observed. The remainder of this section presents the essence of our behavioral synthesis results, and describes, in more detail, our experiment with the receiver module.

Our set includes 5 real-life applications for analog-signal processing. Four of them have a continuous-time part, as well as an event-driven part. *Receiver Module* was already discussed in Section 3. The acquisition part of the *Power Meter* [18] samples two input signals from sensors, and converts them into digital data. *Iterative Solver* and *Missile Solver* are two distinct equation solvers [2]. *Function Generator* describes a ramp-signal generator [6].

Table 1 presents the main results of the synthesis experiments. All examples were synthesized so that the global area was minimized. Columns 2 to 5 introduce the characteristics of the VASS specifications (number of lines for continuous-time part, number of quantities, number of lines for event-driven part, number of *signals*). The next 3 columns outline the main attributes of the VHIF representations: number of blocks in the signal-flow graphs, number of states in the FSMs, and number of elements in the data-paths. The last column shows the components in the synthesized net-lists. All examples were correctly compiled and synthesized to net-lists of electronic components.

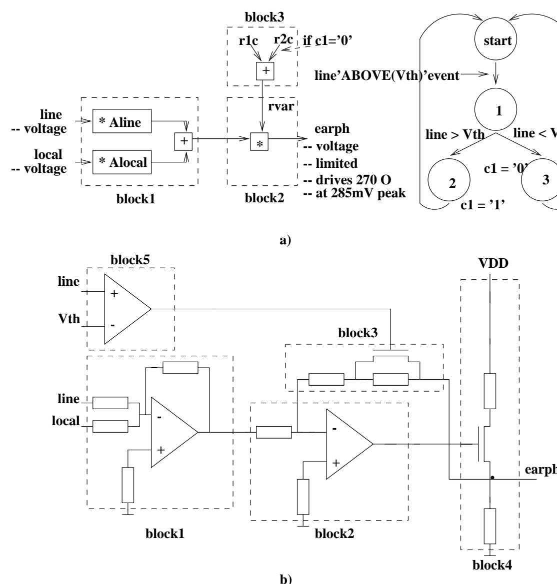

Figure 7. Synthesis of the receiver module

Next, we detail our synthesis experiment for the receiver module (already described in Section 3). Our compiler translated the VASS specification in Figure 2 into the signal-flow graph in Figure 7a. Instruction 1 in the VASS program is translated into the sequence of blocks 1 and 2, Instruction 2 regulates the value of $rvar$ and is converted into block 3. The process statement is compiled into the structure of a Finite State Machine. For this example, the mapping was quite straightforward, and the resulting circuit structure is depicted in Figure 7b. Corresponding blocks in the VHIF representation, and in the circuit representation are annotated with similar names. Although the control part appears to be quite "sophisticated", its behavior can be realized by a simple zero-cross detector [12], with a small hysteresis margin, so that repeated switchings between states are avoided.

Its worthwhile mentioning, that *block 4* was inferred from attributes specified for the terminal port, and not from

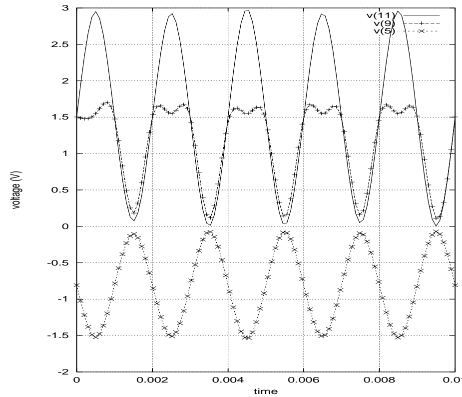

Figure 8. **Simulation of the receiver module**

VHDL-AMS code. As opposed to the other blocks, *block 4* does not process signals, but it adapts the system output to the loading requirements of the external environment.

For all op amps in the design, we selected 2-stage operational amplifiers, in the MOSIS SCN-2.0um technology. The resulting design was described in SPICE, and then, simulated. Simulation plots are presented in Figure 8, and they show that the design functions correctly. *v(11)* is input signal for the op amp of *block1*, *v(5)* is its output signal, and *v(9)* represents signal *earph*. We deliberately considered an input signal with a high amplitude, so that we could observe the signal limiting capability of the output stage. Signal *v(9)* was clipped at 1.5V.

7. Conclusions and Future Work

This paper presents a complete behavioral-synthesis method for analog systems. The method uses system-level specifications described with a VHDL-AMS subset for synthesis. We indicate compiling rules for converting a VHDL-AMS program into a set of interconnected signal-flow graphs, and a FSM for their configuring. Finally, with a *branch-and-bound* based algorithm, the signal-flow representation is mapped to a net-list of library components, so that ASIC area is minimized and the rest of performance constraints are met. Our experiments with 5 real-life examples motivate that our method can successfully synthesize behavioral system-level specifications into structural net-lists of electronic circuits.

Currently, we pursue the presented work towards two main objectives. While conducting this research, we learned that the quality of synthesis can be improved, if meaningful annotations/restrictions are provided for a specification. Ongoing work will explore a more systematic way of describing synthesis-oriented annotations of VHDL-AMS programs. Second, we are aware that because of its time-complexity, the proposed *branch-and-bound* algorithm might fail for larger designs. This shortcoming can be partially alleviated if more effective bounding rules are found. However, ongoing work attempts to replace the *branch-and-bound* method by a more time-affective exploration heuristic.

References

[1] *IEEE Standard VHDL Language Reference Manual (Integrated with VHDL-AMS changes)*. IEEE Std.1076.1.

[2] A. Doboli, A. Nunez-Aldana, N. Dhanwada, R. Vemuri. VHIF - A Hierarchical Representation for Behavioral Synthesis of Analog Systems from VHDL-AMS. *Technical Report, DDEL, University of Cincinnati*, 1998.

[3] A. Doboli, R. Vemuri. The Definition of a VHDL-AMS Subset for Behavioral Synthesis of Analog Systems. *IEEE/VIUF Workshop on Behavioral Modeling and Simulation*, 1998.

[4] A. Nunez, R. Vemuri. An Analog Performance Estimator for Improving the Effectiveness of CMOS Analog Systems Synthesis. *Proc. of DATE*, 1999.

[5] B. Antao, A. Brodersen. ARCHGEN: Automated Synthesis of Analog Systems. *IEEE Transactions on VLSI*, 3(2):231–244, June 1995.

[6] C. Grimm, K. Waldschmidt. Repartitioning and Technology Mapping of Electronic Hybrid Systems. *Proc. of DATE'98*, pages 52–58, 1998.

[7] P. Campisi. *A CMOS Analog Cell Library for Analog Synthesis Systems*. Master of Science Thesis, University of Cincinnati, 1998.

[8] D. Gajski, N. Dutt, A. Wu S. Lin. *High-Level Synthesis*. Kluwer Academic Publishers, 1992.

[9] E. Horowitz, S. Sahni. *Fundamentals of Computer Algorithms*. Computer Science Press, 1985.

[10] E. Ochotta, R. Rutenbar, R. Carley. ASTRX/OBLX: Tools for Rapid Synthesis of High-Performance Analog Circuits. *Proc. of the 31st ACM/IEEE DAC*, pages 24–30, 1994.

[11] F. Leyn, W. Daems, G. Gielen, W. Sansen. A Behavioral Signal Path Modeling Methodology for Qualitative Insight in and Efficient Sizing of CMOS Opamps. *Proc. of ICCAD*, pages 374–381, 1997.

[12] S. Franco. *Design with Operational Amplifiers and Analog Integrated Circuits*. McGraw Hill, 1998.

[13] G. Gielen, H. Walscharts, W. Sansen. Analog Circuit Design Optimization Based on Symbolic Simulation and Simulated Annealing. *IEEE Transaction on Solid-State Circuits*, 25(3):707–713, June 1990.

[14] J. Trontely, L. Trontelj, G. Shenton. *Analog Digital ASIC Design*. McGraw-Hill Book Company, 1989.

[15] L.R. Carley, G. Gielen, R. Rutenbar, W. Sansen. Synthesis Tools for Mixed-Signal ICs: Progress on Frontend and Backend Strategies. *Proc. of the 33rd DAC*, pages 298–303, 1996.

[16] M.del Mar Hershenson, S.Boyd, T.Lee. CMOS Operational Amplifier Design and Optimization via Geometric Programming. *Proc. First Intern.Workshop on Design of Mixed-Mode Integrated Circuits and Applications*, pages 15–18, 1997.

[17] N.R. Dhanwada, A. Nunez, R. Vemuri. Hierarchical Constraint Transformation using Directed Interval Search for Analog System Synthesis. *Proc. of DATE*, 1999.

[18] S. Garverick, D. McGrath, R. Baertsch. A Programmable Mixed Signal ASIC for Power Metering. *IEEE Journal of Solid State Circuits*, 26(12), 1991.

[19] W. Nye, D. Riley, A. Sangiovanni-Vincentelli, A. Tits. DELIGHT.SPICE: an optimization-based system for the design of integrated circuits. *IEEE Transaction on CAD*, 7(4):501–519, April 1988.