Digital Receiver Design Using VHDL Generation From Data Flow Graphs

Peter Zepter, Thorsten Grötker, Heinrich Meyr
Integrated Systems for Signal Processing, Aachen University of Technology
Templergraben 55, D-52056 Aachen, Germany

Abstract— This paper describes a design methodology, a library of reusable VHDL descriptions and a VHDL generation tool used in the application area of digital signal processing, particularly digital receivers for communication links. The tool and the library interact with commercial system simulation and logic synthesis tools. The support of joint optimization of algorithm and architecture as well as the concept for design reuse are explained. The algorithms for generating VHDL code according to different user specifications are described. An application example is used to show the benefits and current limitations of the proposed methodology.

I. Introduction

System design for digital receivers and other digital signal processing applications is usually performed on two different levels of abstraction: Algorithm design and hardware architecture development. Different specifications, simulation tools and libraries are used on each level to obtain the desired results in the shortest possible time. During algorithm development the use of data flow specification and simulation allows to obtain measures of the algorithmic performance (e.g. bit error rate) quickly, because there is no need to determine the timing of the operations and their mapping to computational resources to obtain these results. Clock, reset and other implementation specific signals are not required here. The simulation efficiency is higher than that of discrete event simulators [1]. In contrast, the timing must be specified with respect to one or more clocks to implement the system as synchronous application specific integrated circuit. Hardware description languages are used which offer access to logic synthesis and discrete event simulation. Implementation performance criteria such as throughput, chip area and power consumption can only be obtained on this level.

Algorithm and architecture must be optimized jointly [2] to obtain an efficient ASIC implementation. Thus the design process usually involves multiple iterations between algorithm development and architecture design. The different description styles make this transition difficult. To overcome these difficulties we have developed a design methodology consisting of a VHDL generation program (ADEN) and an extensible library of reusable components (ComBox). The ComBox library transfers the successful reuse of simulation models in the data flow domain to the architecture development process. This approach is supported by the observation that there is a direct correspondence between the complex data flow blocks (such as a filter or Viterbi decoder) in the system simulation and the hardware components in the realization.

The proposed methodology is complementary to high level synthesis. Architectural information and synthesis options can be stored with the library components. They may have an internal algorithmic state and complex behavior as opposed to RT level components. The resulting overall architecture needs only little central control. Any VHDL coding style and architecture type can be encapsulated in a library component. Thus the user can access a broad range of architectures for communication system functions. Due to the direct correspondence between a part of the algorithm and a hardware component, the effects of algorithmic tradeoffs on the implementation cost can be evaluated easily.

Similar existing systems are either based on a time-driven system simulator [3], [4] and thus require the specification of the timing already on the algorithm level, or they do currently neither support multirate, nor dynamic data flow nor different processing times of the block implementations or multiple clocks [5].

We begin with an overview of the proposed design flow. Next the organization and interface description in the library is presented. This is followed by a discussion of the general architecture of every realized system and the user's options to influence this. The novel translation from data flow to VHDL and subsequent optimizations are described. Finally, a digital receiver design example is presented.

II. Design Flow Overview

We distinguish two main abstraction levels in our design flow: The **algorithm level** comprises all system data that may influence the algorithm performance of the system, but no other data. In the case of digital receivers the algorithm performance is measured by e.g. the bit error rate (BER) at a given signal to noise ratio (SNR). These properties are independent of the physical implementation technology and the achieved throughput. The **implementation level** comprises all design data which does not influence the algorithm performance. Here all timing dependent knowledge such as the number of pipeline stages of a certain component is located. Fig. 1 gives an overview of the design flow.

The algorithm design is done using the commercial data flow signal processing simulation tool COSSAP [5]. The user specifies the system including data sources, transmitter (analog and digital parts), channel model, receiver (analog and digital parts) and evaluation blocks (bit error rates, variances) as a hierarchical block diagram. The behavior of each block is modified by setting algorithm parameters. In our approach the system level model, which

This work was supported by the DFG (AZ Me 651/12-3).

32nd ACM/IEEE Design Automation Conference ®

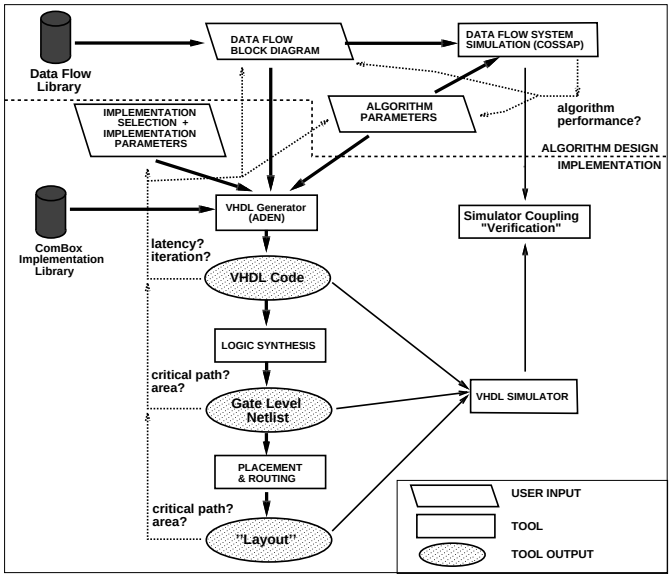

Fig. 1. Design flow with COSSAP, ADEN and ComBox.

is used as a starting and reference point for the implementation, is always arithmetic-true, i.e. all effects influencing the algorithmic performance are represented by modeling the finite-precision arithmetic.

An implementation is selected from the library for every data flow block to be implemented. The implementation parameter selection can also be deferred up to this point because of the new and unique separation of algorithm and implementation data base. The combined algorithm and implementation data together with the ComBox library is used to generate a VHDL description of the system. This VHDL code is used as input to further implementation steps. After each step analysis of the results can lead to a modification of the implementation or algorithm parameters in order to achieve better throughput or less cost.

III. The Library of Reusable Components

The ComBox library is organized in three levels as shown in fig. 2. The example of a controlled decimator was taken which is used to select an item from fixed number of data items in a data stream.

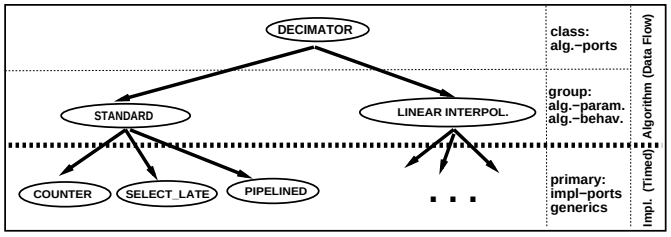

Fig. 2. The three abstraction levels of the ComBox library.

The so called class level of each library entry defines the external data flow ports (connections of blocks / entities and signals) of a function as well as some of its parameters. It is also stated whether the block has an algorithmic state, i.e. whether an internal state must be saved between activations. The class level is intended to comprise several different functional behaviors of the same basic algorithm, which offer access to implementations with different cost and throughput. This supports the joint optimization of algorithm and architecture. The algorithmic tradeoffs are represented by the different groups (second level) belonging to one class. In our example there are a group for the standard implementation and a group which allows to perform interpolation between two samples, if the control input does not select exactly one input. The second alternative can offer better performance at increased hardware cost.

Each group is characterized by its data flow behavior including port data rates $r(e, v)$ for each port P connecting edge e and vertex v (see fig. 3) [6]. A data rate specifies the number of tokens produced respectively consumed at a port of the block during one activation and is represented by the expressions in the circles constituting the ports. These rates can be either static (the same for each activation) or dynamic (data dependent for each activation, indicated by the variable p). In our approach we support a limited subset of the dynamic data flow similar to boolean data flow [7] but with control tokens of arbitrary values. Each dynamic port at a data flow block has a corresponding static port at the same model (the control port) with data rate one. The value of the data item at the control port is used to decide whether the dynamic variable p takes the value 0 or 1, which indicates whether data items are produced (respectively consumed) at the dynamic port during the current activation of the data flow block.

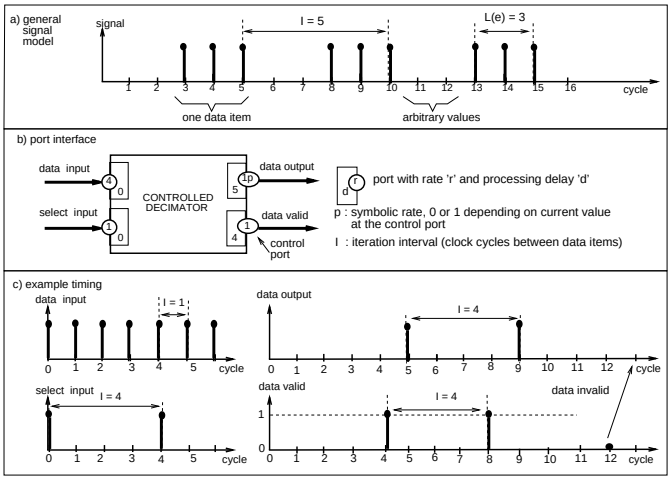

Fig. 3. Signal model, data flow and timing interface.

The third (primary) level provides at least one implementation for each group. Here all information is located which does not influence the algorithmic behavior.

The signal model applied here is shown in fig. 3. Only the timing relative to clock cycles is described. Technology dependent combinatorial delays are not taken into account. The data items at each signal occur in fixed time intervals called the iteration interval $I(e)$ of the signal e. Each data item may take $L(e)$ $(I(e) \geq L(e) \geq 1)$ clock cycles to transmit its value depending on the data type. During the remaining $I(e) - L(e)$ clock cycles arbitrary values may be transmitted.

The timing interface, which may depend on algorithm and implementation parameters, consists of two main parts (fig. 3): The processing delays $d(e, v)$ at a port P connecting edge e and vertex v and the intrinsic iteration interval $I_i(v)$ (the time between two successive activations

of the implementation). The iteration interval at a port is inversely proportional to the static part of its data rate. The data flow simulation requires that at least $r(P)$ data values are available in a FIFO buffer at each input port P of a data flow block. To avoid the unnecessary implementation of FIFO buffers in the hardware, every hardware implementation can start execution, when the first data item arrives at the input port with the smallest processing delay ($d = 0$).

Dynamic data flow allows to declare the values of an iteration as invalid (fig. 3). This allows to model data dependent processing times. The primary level specifies also the implementation ports, which are explained in the next section.

IV. Output Architecture and User Options

Besides block implementation selection and implementation parameter setting, the following options can be used to influence the generated VHDL output:

- For each system input port the input time (arrival of the first data item) can be specified. Default is zero (corresponding to the first clock edge after the external reset).
- Different parts of the design can have clocks with different periods and different phase offsets. The specification is performed hierarchically, i.e. for each instantiated component v and each signal e one can specify its clock relative to that of its parent vertex. The relative clock period $T_{c,r}(v) \geq 1$ and the relative clock phase $0 \leq \phi_{c,r}(v) < T_c(v)$ can be multiples of that of the parent clock.
- To break too long combinatorial paths, a minimum number of shimming delays $d_{s,min}(e)$ may be specified on each signal e. A shimming delay is a register inserted on a signal.
- All registers in the implementation corresponding to algorithmic states need a reset in the implementation to ensure equivalent behavior of data flow graph and synchronous clocked VHDL implementation. However, sometimes it may not be necessary to perform these resets upon initialization for all states, because the system will operate correctly after a finite time regardless of the initial state. Therefore the reset can be suppressed.

Fig. 4 shows the supported target architecture.

For each data flow block there is a corresponding hardware component with a VHDL port for each data flow port. In addition, there are so called implementation ports. The **clock** ports are connected automatically to the correct clock signal (there may be multiple clocks in the system, due to multiple clock specifications and dynamic data flow). **Reset** ports are connected to a reset signal, which is active up to the reference time $t_r(v)$ of the block v. The reference time is the index of the clock edge where the first valid data item arrives at the block. The first clock edge after the external reset has index zero. This reset procedure guarantees an initial behavior equivalent to that of the data flow system. **Enable** ports allow suppression of the state update of a hardware unit. This is the usual way for the realization of dynamic data flow and delayed reset. As an alternative we can use gated

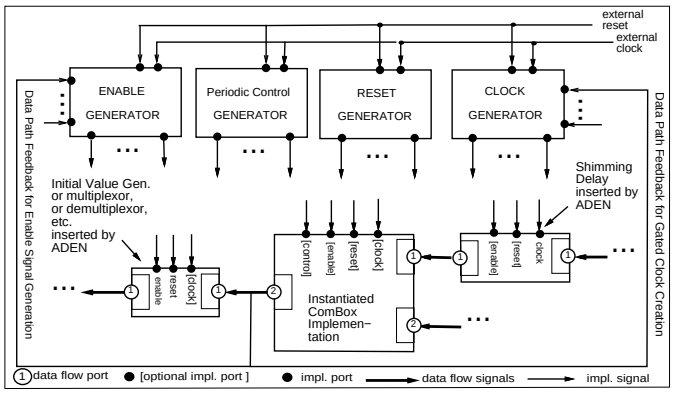

Fig. 4. Basic architecture of the VHDL generator output.

clocks respectively multiple reset signals. **Control** signals allow to centralize control functions in the data path. Especially in the case of many blocks operating at iteration intervals greater than one there may be several controllers performing the same function. A centralized control unit providing central control signals may be more efficient than many (almost) identical local control units. Especially input data independent periodic control functions can profit from this feature. Fig. 5 shows an example. The control signal is periodic and is always zero except for one clock cycle, where it takes the value one.

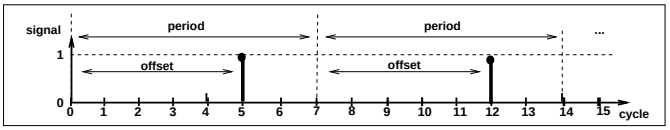

Fig. 5. Simple form of control signal.

The implementation ports are automatically connected to implementation signals provided by central controllers/generators during VHDL code generation. For the creation of enable signals or gated clocks, which implement the data dependent dynamic data flow, the corresponding data flow signal values are provided to the generators as indicated by the feedback signals in fig. 4.

Besides taking care of the creation and interconnection of implementation signals, ADEN must instantiate and connect additional entities in the data path. There are two reasons for this:

- To ensure that data items arrive in time at the ports, it may be necessary to insert shimming delays (registers) on some signals.
- Some signals have initial data items in the data flow block diagram. The number $s(e)$ of initial data items on a signal e is represented by diamonds in the data flow diagrams (fig. 7). If $s(e) > 0$ but the implementation of e contains no registers and initialization is important, the proper initial values must be provided through a multiplexor during the initial cycles.

V. System Analysis and VHDL Generation

Fig. 6 shows the basic steps performed by ADEN during VHDL generation. The reset requirements and clock parameters are inherited from the parents if not specified for a block. After elaboration, the clock period $T_c(v)$ and phase $\phi_c(v)$ relative to the base clock are known for each block.

A flat (non hierarchical) data flow graph is constructed from the block diagram.

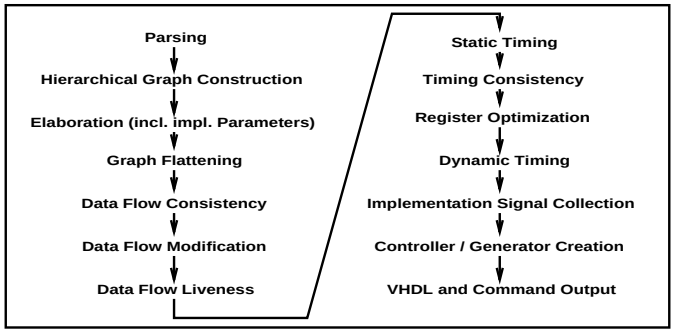

Fig. 6. Basic Steps for VHDL Generation

The data flow analysis is necessary to compute the consistency (limitedness of the the memory needed) and liveness (there must be always one executable vertex) [7], [8]. If the data flow graph has these properties it is possible to execute the system for infinite input data streams. For a static data flow graph a periodic schedule can be found. In this schedule each primitive block v is activated $q(v)$ times. In case of dynamic data flow $q(v)$ is the product of a constant positive integer and zero or more symbolic dynamic port rate variables p_i. Blocks are not activated, if one of the symbolic variables of $q(v)$ evaluates to zero during a period. The number of activations per schedule is computed as the smallest non-zero integer solution of the balance equations [9]:

$$q(v_1) \cdot r(v_1, e) - q(v_2) \cdot r(e, v_2) = 0$$

for all signals $v_1 \xrightarrow{e} v_2$ in the data flow graph (fig. 7). The existence of a solution ensures limitedness. Additionally we can compute the number $n(e) = q(v) \cdot r(v, e)$ of data items transmitted per period on a signal e. Due to our signal model the data items are equidistant in time. The minimum possible iteration interval (called relative iteration interval)

$$I_r(e_i) = \frac{\mathrm{lcm}_{all\ e_j \in E} n(e_j)}{n(e_i)}$$

on each signal can be computed. It specifies the relative length of the iteration intervals on the edges. The absolute value of the iteration interval is obtained later on by multiplication with the system iteration interval I_s common to all signals ($I(e) = I_r(e) \cdot I_s$). Once consistency is shown liveness is checked by executing a schedule with all p_i set to 1 for a complete period.

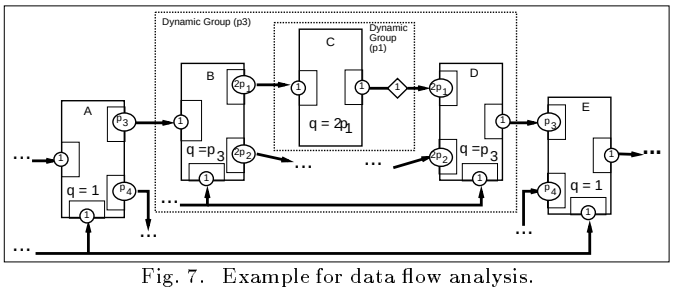

Fig. 7. Example for data flow analysis.

To prepare for the implementation of dynamic data flow, all connected subgraphs of the data flow graph where the $q(v)$ and $n(e)$ values of the edges depend on the same symbolic variable p_i are formed as shown in fig. 7. These subgraphs are called dynamic groups. Only those consistent dynamic data flow graphs can be implemented without FIFO buffering of data items, where a hierarchical sorting of dynamic groups is possible. A dynamic group is a descendant of another dynamic group if its control signal's number of tokens per period n depends on the ancestor's symbolic variable.

A fixed number of clock cycles for the execution of each period of the data flow graph is then computed. Several periods can be executed in overlapping time intervals due to pipelining. The reference times of the vertices have to fulfill the following conditions:

- To allow proper reset the reference time must be non-negative with respect to the external reset: $t_r(v) \geq 0$.
- For causality [10] and fulfillment of the minimum shimming delay requirements, the following equation must hold for each signal e ($v_i \xrightarrow{e} v_j$):

  $$t_r(v_i) + d(v_i, e) - s(e) I(e) - d(e, v_j) - t_r(v_j) \geq d_{s,min}(e)$$

- The reference times are measured in multiples of the base clock period. If a vertex v has a clock period $T_c(v)$ and phase $\phi_c(v)$ the reference time must correspond to a positive clock edge:

  $$t_r(v) = T_c(v) \cdot k + \phi_c(v),\ k \geq 0$$

We have extended the computation of the $t_r(v)$ from [10] to handle the additional constraints. The complexity of the resulting shortest path problem is still proportional to the square of the number of vertices. The resulting time schedule has the minimum latency possible for a given system iteration interval. Another important task is the minimization of the cost of registers (shimming delays inserted) on the edges. This problem adds additional constraints and requirements to the register minimization problem discussed by Leiserson [11]:

- On signal e ($v_i \xrightarrow{e} v_j$) a delay of

  $$d_s(e) = t_r(v_i) + d(v_i, e) - s(e) I_r(e) I_s - d(e, v_j) - t_r(v_j)$$

  must be inserted. However, if the iteration interval $I(e)$ is larger than 1, the cost $c_r(e)$ of necessary registers is less than the shimming delay times the word length $b(e)$ of the signal, when using 'sparse' delays:

  $$c_r(e) = b(e)\left(\left\lfloor \frac{d_s(e)}{I(e)} \right\rfloor L(e) + \min(d_s(e) \bmod I(e), L(e))\right)$$

- The retiming $\rho(v)$ of a vertex v must be proportional to the clock period $T_c(v)$.
- The number of shimming delays on an edge may not be less than the user specified minimum $d_{s,min}(e)$.

Controller cost for the implementation of the delays at edges with iteration interval larger than the clock period is neglected. We have developed a method to transform the optimization problem to a minimum cost flow problem as proposed for the single clock, single rate system in [11]. If the first requirement is slightly modified by replacing the minimum expression by $d_s(e) \bmod I(e)$, the following three main steps are necessary in order to achieve this:

1. Each edge e_k ($v_i \xrightarrow{e_k} v_j$) where the iteration interval is larger than the clock period ($I(e_k) > T_c(e_k)$) is split

into two edges $e_k{}'$ and $e_k{}''$ by an additional intermediate vertex $v_k{}'$. The new sparse edge $v_k{}' \overset{e_k{}''}{\rightarrow} v_j$ has register costs reduced by a factor of $I(e)/T_c(e_k)$ and the difference of the retimings of the vertices adjacent to it must be a multiple of the iteration interval ($\rho(v_j) - \rho(v_k{}') = c(k) \cdot I(e_k)$). The other new edge $e_k{}'$ has register cost equal to the product of its shimming delay and wordlength. The shimming delay is limited ($d_s(e_k{}') < I(e_k)$). The required minimum shimming delay must be distributed among $e_k{}'$ and $e_k{}''$ according to the retiming step sizes.

2. Each edge e_n ($v_l \overset{e_n}{\rightarrow} v_m$), where the clock periods at the end vertices do not match ($T_c(v_l) \neq T_c(v_m)$), is split into two new edges $v_l \overset{e_n{}'}{\rightarrow} v_n{}'$ and $v_n{}' \overset{e_n{}''}{\rightarrow} v_m$. One of the end-vertex clocks is identical with the clock of the original edge ($T_c(e_n) = T_c(v_m)$). Registers on $e_n{}'$ operate at the base clock period, while $e_n{}''$ has registers operating with the larger period $T_c(e_n)$. $e_n{}'$ has the higher cost per delay. $v_n{}'$ can be retimed with the base clock period ($T_c(v_n{}') = 1$). The delay on the base clock edge $e_n{}'$ must be less than the clock period of the original edge ($d_s(e_n{}') < T_c(e_n)$).

3. The retiming $\rho(v)$ of each vertex v is measured relative to its clock and appears with the factor $T_c(v)$ in the constraint inequalities for the linear optimization problem. Thus we cannot immediately transform to a minimum cost flow problem. However, by applying a transformation similar to the equivalent single rate graph transformation of [9], an equivalent formulation with a single clock can be obtained. The resulting equation system has an increased number of variables (corresponding to vertices) and constraints compared to a single rate single clock system, but the register optimization can still be executed in polynomial time.

The next step is the determination of the registers representing the algorithmic state to implement dynamic data flow. Previous work has only considered static systems. Fig. 8 shows the data flow graph from the example in fig. 7 after static timing. The numbers at the ports represent the times $t(e, v)$ at which the first data item appears. These times can be seen as potentials of the ports.

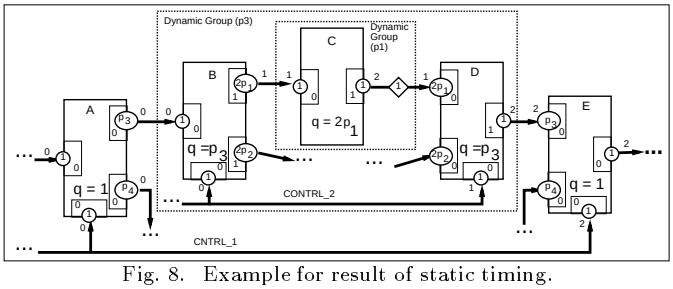

Fig. 8. Example for result of static timing.

The algorithmic state represented by the initial data item on the edge between block C and block D is only updated in the data flow simulation, if both the conditions for p_1 and p_3 are fulfilled. However, the resulting times show that the registers for storing this algorithmic state in the implementation are located in block C. The loading of invalid data to these registers must be prevented. The correct enable signals for the implementation of the dynamic data flow must be connected to this block. The reading and writing to registers which do not represent algorithmic states need not be controlled by a central controller. To find registers representing algorithmic states we developed the following algorithm: For each edge and each vertex with an algorithmic state, all enclosing subgraphs within its dynamic group are created. For each of these subgraphs the timing potentials at its external ports are computed. From these the earliest times for the first and second data item arriving ($t_{1,in}, t_{2,in}$) at the subgraph or leaving ($t_{1,out}, t_{2,out}$) the subgraph are searched. If the following equations are fulfilled, there is no overlapping between iterations at the subgraph, which allows to disable the update of its registers for one iteration.

$$t_{2,in} \geq t_{1,out} \quad t_{2,in} > t_{1,in} \tag{1}$$

$$t_{2,out} > t_{1,out} \quad t_{2,out} > t_{1,in} \tag{2}$$

All subgraphs for all algorithmic states within one dynamic group fulfilling the conditions (called potential groups) are stored. Thereafter, potential groups are selected which cover all algorithmic states in the dynamic group but do not overlap. If the number of ports at a vertex is limited, this algorithm can be executed in polynomial time.

The results can be used to create the enable signal generators. Fig. 9 shows an example enable signal generator which is used when implementing dynamic data flow.

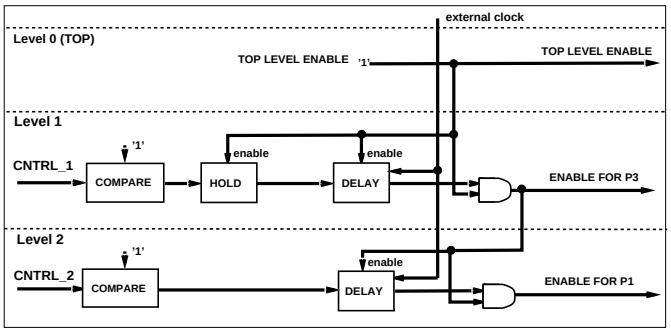

Fig. 9. Example enable generator.

The hierarchy of dynamic groups is found in the enable generator again. For each level the signal coming from the data path is compared to the value necessary for enabling the dynamic group. Holding the comparison result for one iteration and delaying to adapt to the time difference between the comparison and the arrival of the invalid data items at the potential group may also be necessary. Algorithmic states can only be written, if all dynamic groups on higher levels are enabled. Therefore the enable signals of higher levels are used to control lower levels in the dynamic hierarchy.

VHDL code is generated including all controller, implementation signals, shimming delays and multiplexors while retaining the hierarchy. Thus the designer can easily identify the corresponding data flow blocks when analyzing logic synthesis results.

VI. Example Design

Fig. 10 shows the top level block diagram of a minimum shift keying MSK transceiver implementing algorithms from [12]. It is intended for a mobile communication network with packet transmission and equal priority of all participants. Fast synchronization is important while the effort for filtering and coding/decoding could be

kept relatively small. Control functions consume a relatively large amount of silicon area compared to other signal processing applications.

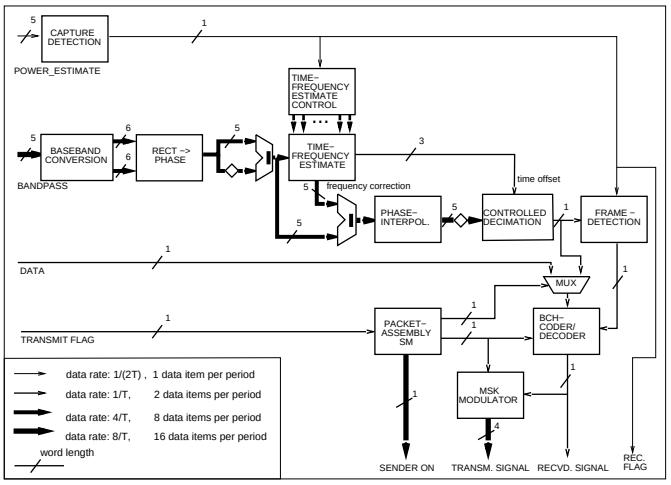

Fig. 10. Top level block diagram of the example digital transceiver.

The application requires a gross data rate of 2.5 Mbit/s. Due to the eightfold oversampling of the incoming symbols the clock period at the bandpass input is 50 ns. Considering the algorithm complexity of several hundred operations per sample no implementation that makes use of programmable devices is possible. To obtain a cost-efficient prototype for field tests and to increase flexibility the prototype will be a field programmable gate array (FPGA).

The data flow description and simulation of the transceiver was performed by a system designer using his familiar data-flow simulation environment. Initially the hierarchical block diagram consisted of about 700 fine granular data flow blocks of lower complexity. During the design several more complex reusable parts were identified and inserted as data flow models and implementations into the ComBox library. Some state machines (e.g. the time frequency estimation controller) may be too specialized to offer much potential for reuse. From the first data flow block diagram with sufficient algorithmic performance a VHDL implementation was generated (approximately 12000 lines of code) and synthesized for area constraints. From this the costly components where identified (quadrature component to phase conversion, capture, time and frequency estimation controller etc.). The data path elements where improved by redesigning the algorithm for smaller word lengths and by exploiting don't care inputs. The implementations of state machines were optimized by more efficient VHDL coding. In a second attempt the synthesis was performed with timing constraints. After some iterations it was possible to achieve the desired throughput, partly by using pipelined components (quadrature component to phase conversion) and partly by selecting options that forced ADEN to insert shimming delays on some signals in the generated VHDL to break critical paths. Table VI shows the final area obtained for TI/Actel FPGA's [13]. To obtain the optimal result four different clocks had to be used. Dynamic data flow was very useful to separate the transmitter from the receiver part while reusing the BCH-Encoder for both parts. The register optimization helped to reduce the block count of the final implementation by about ten percent.

TABLE I

FINAL SYNTHESIS RESULTS (IN LOGIC BLOCKS (LB)). 2104 LB CORRESPOND TO APPROX. 13600 EQUIV. GATES.

component	combinatorial	sequential	sum
capture detection	137	138	275
baseband conversion	107	65	172
rect. $\rightarrow$ phase	146	36	182
time-frequency cntrl.	67	64	131
time-frequency est.	179	137	316
phase interpol.	13	9	22
frame detection	54	26	80
packet assembly FSM	78	26	104
BCH (de)coder	94	75	169
MSK-modulator	49	14	63
Aden-Controller	80	34	114
rest	145	331	476
sum	1149	955	2104

VII. CONCLUSIONS

We have presented a design methodology for the application area of digital communication receivers consisting of a VHDL generation tool and a library of reusable generic implementations. The VHDL generation is based on a standard commercial digital signal processing simulation tool. The library defines standard interfaces to support interface compatibility and comfortable reuse. The VHDL generation takes care of the implementation details. Timing consistency, which is a great problem during manual design, is achieved automatically. It is now important to encourage the users to identify functions appearing often in designs and to extend the library with reusable, generic, optimized implementations. Future versions of the software should also allow to specify resource sharing manually.

REFERENCES

[1] G. Jennings, "A case against event driven simulation of digital system design," in *The 24th Annual Simulation Symposium*, pp. 170–176, IEEE Computer Society Press, April 1991.

[2] O. J. Joeressen, G. Schneider, and H. Meyr, "Systematic Design Optimization of a Competitive Soft-Concatenated Decoding System," in *VLSI Signal Processing 6* (L. Eggermont et al., ed.), pp. 105–113, IEEE, 1993.

[3] Cadence Design Systems, 919 E. Hillsdale Blvd., Foster City, CA 94404, USA, *SPW User's Manual*.

[4] P. B. Tjahjadi, P. T. Yang, B. C. Wong, B.-Y. Chung, E. G. Cohen, and R. Jain, "Vanda - a CAD system for communication signal processing circuits design," in *VLSI Signal Processing IV*, ch. 5, IEEE Press, 1990.

[5] Synopsys, Inc., 700 E. Middlefield Rd., Mountain View, CA 94043, USA, *COSSAP User's Manual*.

[6] E. A. Lee, "Consistency in dataflow graphs," *IEEE Trans. on Parallel and Distr. Systems*, vol. 2, pp. 223–235, Apr. 1991.

[7] J. T. Buck and E. A. Lee, "Scheduling dynamic dataflow graphs with bounded memory using the token flow model," in *Proc. ICASSP'93*, pp. I–429–I–432, IEEE, 1993.

[8] T. Murata, "Petri nets: Properties, analysis and applications," *Proc. of the IEEE*, vol. 77, pp. 541–580, April 1989.

[9] E. A. Lee and D. G. Messerschmitt, "Static scheduling of synchronous data flow programs for digital signal processing," *IEEE Trans. on Computers*, vol. 36, pp. 24–35, Jan. 1987.

[10] H. V. Jagadisch and T. Kailath, "Obtaining schedules for digital systems," *IEEE Trans. on Signal Processing*, vol. 39, pp. 2296–2316, Oct. 1991.

[11] C. E. Leiserson, F. Rose, and J. Saxe, "Optimizing synchronous circuitry for retiming," in *Proc. of the 3rd Caltech Conf. on VLSI*, (Pasadena), pp. 87–116, March 1983.

[12] U. Lambrette and H. Meyr, "Two timing recovery algorithms for MSK," in *Intl. Conf. on Comm.*, 1994.

[13] Texas Instruments, *FPGA Data Manual*.