Synthesis of Multi-rate and Variable Rate Circuits for High Speed Telecommunications Applications

Patrick Schaumont Serge Vernalde Luc Rijnders Marc Engels* Ivo Bolsens

Abstract

A design methodology for the synthesis of digital circuits used in high throughput digital modems is presented. The methodology spans digital modem design from the link level to the gate level. The methodology uses a C++-based untimed dataflow system description, which is gradually refined to an optimized, bit-true and clock cycle true C++-description. Through this refinement, a bridge from link level design semantics to architectural VHDL semantics is made within one and the same environment.

1 Introduction

Currently there is a high interest in digital communication equipment for public access networks. Examples are modems for ADSL, VDSL, and up- and downstream HFC communication. Besides having high complexity and throughput requirements, these systems also need short development cycles. This calls for a design methodology that starts at high level and that provides for design automation as much as possible.

Our contribution to existing design systems for telecommunications is *a gradual refinement within one and the same C++ environment.* The lowest level is semantically equivalent to a RT-level VHDL description. This way, combined semantic and syntactic translations in the design flow are avoided. Otherwise, these translations make design verification a cumbersome task.

The paper is organized as follows. First, the overall design flow we use for the development of a telecommunications system is indicated. Then, the architecture synthesis phase of the design flow will be elaborated, and related work in this area is indicated. At the end, the conclusions are drawn.

*Mark Engels is a senior research assistant of the Belgium National Fund for Scientific Research.

2 OCAPI Design Flow

In the design of a telecommunication system (fig. 1), we distinguish four phases: link design, algorithm design, architecture design and circuit design. These phases are used to define and model the three key components of a communication system: a transmitter, a channel model, and a receiver.

1. The **link design** is the requirement capture phase. Based on telecommunication properties such as transmission bandwidth, power, and data throughput, the system design space is explored using small subsystem simulations. The design space includes all algorithms which can be used by a transmitter/receiver pair to meet the link requirements. Out of receiver and transmitter algorithms contending for an identical functionality, those with minimal complexity are preferred. Besides this exploration, any expected transmission impairment must also be modeled into a software channel model.
2. The **algorithm design** phase selects and interconnects the algorithms identified in the link design phase. The output is a software algorithmic description in C++ of digital transmitter and receiver parts in terms of floating point operations. To express parallelism in the transmitter and receiver algorithms, a data-flow data model is used. Also, the transmission imperfections introduced by analog parts such as the RF front-ends are annotated to the channel model.
3. The **architecture design** refines the data model of the transmitter or receiver. The target architectural style is optimized for high speed execution, uses distributed control semantics and pipeline mechanisms. The resulting description is a fixed point, cycle true C++ description of the algorithms in terms of execution on bit-parallel operators. The architecture design is finished with a translation of this description to synthesizable VHDL.
4. Finally, **circuit design** refines the bit-parallel implementation to circuit level, including technology binding, the introduction of test hardware, and design rule checks.

In the following, we focus on the architecture design phase of this methodology.

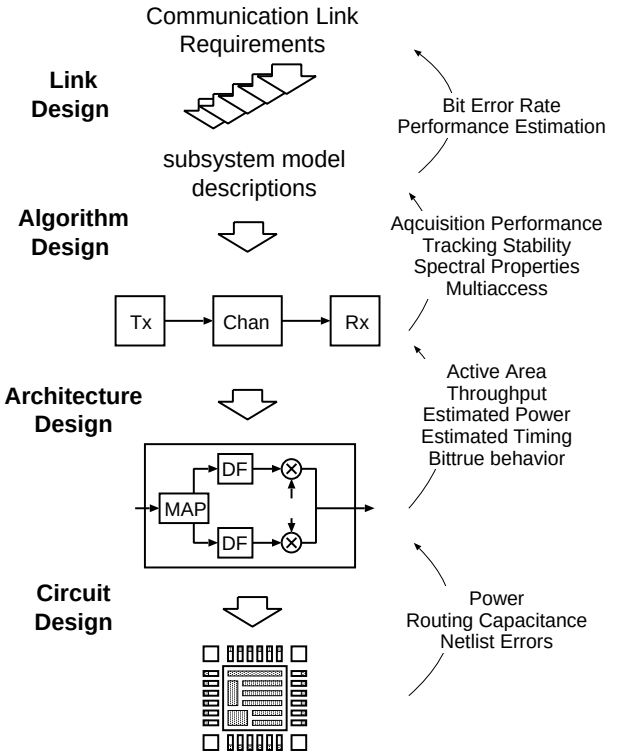

Figure 1: Overall Development Methodology

3 Architecture Design

The architecture design phase starts with a C++ floating point, and untimed data-flow description, and refines this to a fixed point cycle true model. We will explain this refinement in three steps.

- A description of the target architecture.
- A description of the input data model used for architecture design.
- A description of the steps used convert the data-flow input model to the target architecture.

3.1 Target Architecture

The target architecture, shown in figure 2, consists of a network of interconnected application specific processors. Each processor is made up of bit-parallel datapaths. When hardware sharing is applied, also a local control component is needed to perform instruction sequencing. The processors are obtained by behavioral synthesis tools [9] or RT level synthesis tools. In either case, circuits with a low amount of hardware sharing are targeted.

The network is steered by one or multiple clocks. Each clock signal defines a *clock region*. Inside a clock region the phase relations between all register clocks are manifest. Clock division circuits are used to derive the appropriate clock for each processor.

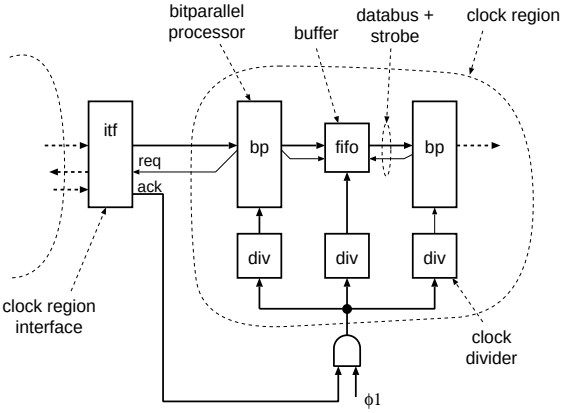

Figure 2: Target Architecture

In between each processor, a hardware queue is present to transport data signals. They increase parallelism inside a clock region and maintain consistency between different streams of data arriving at one processor.

Across clock region boundaries, synchronization interfaces are used, such as [5]. These interfaces detect the presence of data at the clock region boundary and gate clock signals for the clock region that they feed. This way, non-manifest and variable data rates in between clock regions are supported.

The ensemble of clock dividers and handshake circuits forms a parallel scheduler in hardware, synchronizing the processes running on the bit-parallel processor.

3.2 Input Data Model

The input data model is the specification through which the target architecture can be reached. We use an extension to the synchronous data flow (SDF) semantics presented in [11].

SDF is a token flow model that allows to express algorithmic parallelism at high level easily. It describes the system in terms of a graph, with nodes representing the system *actors* and the edges the streams of data in between them. Annotated to each edge is the number of tokens produced/consumed per *firing* or iteration of the actor. The firing itself is solely dependent on the presence of tokens, thereby introducing the parallelism.

Since the SDF semantics do not allow to specify runtime dependent production and consumption in the data streams, a production or consumption is allowed to be of the V type, as shown in figure 3. A V is either 0 or 1, the actual value being decided by the actor responsible for this production or consumption. This extension makes our data model of the dynamic

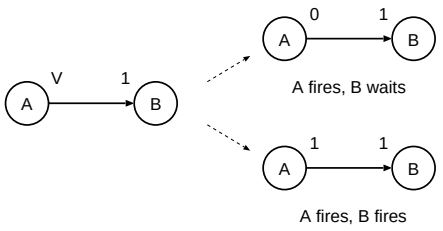

Figure 3: Data Model

dataflow (DDF) type.

The SDF data model allows derivation of a compile-time schedule. For the OCAPI architecture design, a parallel actor schedule is required. The synchronous property also allows thorough compile-time analysis. For example, minimum edge buffering solutions are presented in [1] and [6] for deadlock-free and rate-optimal schedules respectively.

For a runtime dependent V production or consumption, a compile time schedule and minimal edge buffering solution cannot be found. In that case, edge buffering is reduced to zero, and the communicating actor schedules are synchronized instead.

For implementation, we require the data flowgraphs to be live. In addition, hardware realization is possible only if the graph has finite storage requirements and processes an input within finite time [4]. For a synchronous data-flow graph, this corresponds to the consistency property [11]. For dynamic data-flow graphs however, no complete analysis method exists. As a consequence, the user has to verify this condition by means of simulation.

The task now is to refine this data model to the target architecture presented above.

3.3 Refinement Steps

During refinement of the data model to a bit-true and clock cycle true level, several algorithms such as scheduling and pipelining, need to be applied. In this paper, no new algorithms or optimizations will be presented. Rather, we wish to extract the requirements for such algorithms in our approach, and indicate which of the existing ones can be readily applied.

The example in figure 4 shows these transformations for the case of an pulse shaping filter. The top of the figure shows the data-flow model. The circuit conditionally reads the input, performs the shaping by means of an up-sampling finite impulse response (FIR) filter, and synchronously writes samples to the output.

In the refinement, three levels are distinguished between the initial data-flow model and the final bit-true, clock cycle true description: the untimed floating point description, the untimed fixed point description, and

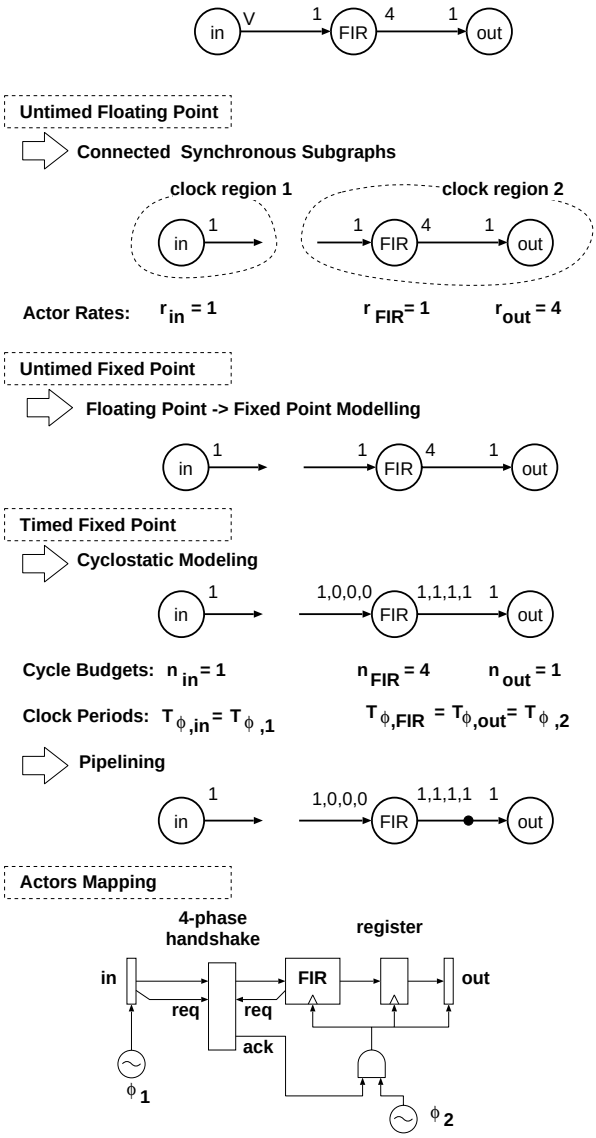

Figure 4: Refinement flow

the timed fixed point description.

3.3.1 Untimed Floating Point Simulation

The initial description describes the circuit at the highest abstraction level. Each actor is simulated in a separate C++ class, while the edges are implemented by means of a generic queue class. Actor execution is decided by means of a *firing rule*, that detects presence of tokens in the input queues.

Instead of repeated testing of all firing rules in the simulation, a more efficient solution is obtained by first grouping all actors that can be statically scheduled. To do this, all connected synchronous subgraphs in the input data model are identified. For each subgraph, a

static schedule can be found by means of a topology matrix [11]. This yields the relative execution rate r_i of each actor in the synchronous schedule.

Each subgraph defines one clock region. In the C++ simulation, a scheduler class is defined for each clock region, which fires the actors according to the rates of the static schedule. The execution of a scheduler object itself can depend on the values of V tokens, which are decided internally in the actors. Therefore, when a scheduler object is allowed to fire, it first queries the actor V inputs and outputs on which it depends. The system schedule finally iteratively tries to fire each scheduler object.

3.3.2 Untimed Fixed Point Description

The second level of refinement converts each actor from a floating point to a fixed point description. Because of the bit-parallel target architecture, the substitution of floating point operations by fixed point operations is sufficient to model the target architecture. The transformation requires definition of all word-lengths and operation overflow behavior.

In C++, this is simulated by using of a hybrid token type that emulates floating point as well as fixed point representation, and use operator overloading in the actor description for both types of simulation. Fixed point modeling in C++ has been reported in literature before [8] and will not be elaborated further here.

3.3.3 Timed Fixed Point Description

In the third level of refinement, conversion from untimed to cycle true behavior is performed.

Inside each synchronous subgraph, the clock dividers of the target architecture implement a maximal parallel schedule for all actors attached to the same clock. To find this maximal parallel schedule, the following two steps are taken.

1. Choose the cycle budget n_i for each actor. The cycle budget corresponds to the number of execution phases of each actor. This is modeled in the multirate graph by converting it to a cyclo-static single-rate description [2], with each execution phase corresponding to one clock cycle.
2. Increase the parallelism in the cyclo-static description by applying pipelining until a maximal parallel schedule is achieved.

Given the actor cycle budget n_i , and the relative execution rate r_i within the clock region, the actor clock period $T_{\phi i}$ can be found. This follows from the observation that each actor should take the same execution time T_i during one iteration of the synchronous schedule.

$$T_i = n_i.T_{\phi i}.r_i \qquad (1)$$

Given this relation for all actors in a synchronous subgraph, we can equate them to find the relation between $T_{\phi i}$ and other $T_{\phi j}$. The smallest common multiple frequency defines the common clock frequency T_ϕ^{-1} for the clock region.

To increase the parallelism in the graph, pipelining is used. Pipelining is visible in the input data model by the introduction of initial tokens. Pipelining for a maximal parallel schedule is not always possible. Feedback loops in graphs for example are typical cases in which parallelism is restricted. To overcome this problem, one must either recur to algorithmic transformations such as loop unrolling and lookahead techniques [12], or else modify the algorithm implementation to include a communication synchronization primitive in each actor.

In C++, the simulation is modified in the following ways.

- The cyclo-static form of an actor is modeled by refining a single-phase actor behavioral description to sequence of phases, each corresponding to one clock cycle. The actor phases are described in terms of operations on a signal type `sig`, which is an encapsulation of the fixed/floating point token type. In addition to the bit-parallel operations, a `sig` also defines the lifetime of the token to be single-phase, needed for intermediate values, or multiple-phase, needed for algorithmic delay.
- The scheduler objects itself does call individual actor phases instead of single actor executions. The schedulers objects are simulated at the highest clock rate.
- The graph pipelining is visible as initial tokens which are inserted in the C++ edge queues.

It is seen that this refinement is independent of the fixed point refinement. Thus, the system timing can be explored without bothering about finite word-length effects. Also, system timing can be introduced gradually because of the simulation queues: An eventual mismatch can be signalled by queue underflow or threshold overflow.

3.3.4 Actor Mapping

Finally, the clock-cycle true and bit-true description of the system is obtained. The bottom of figure 4 shows how this description relates to hardware. The synchronous subgraphs are each mapped to one clock region. The initial tokens on the edge queues translate to registers. The V edges are implemented storageless with a synchronization interface that gates the clock of the dependent clock region.

The multiphase actor description must now be converted into synthesizable VHDL format to map it into gates. An solution in C++ is to encapsulate the token

types into a custom signal type `sig`, for which all desired bit-parallel operators are overloaded to do one of the following.

- When executing the C++ simulation, the execution of the `sig` expressions build up a data structure that corresponds to a signal flowgraph (SFG) syntax tree of the actor. This allows to perform semantical checks such as dead code and dangling input detection.
- The actor can now be simulated by running through the SFG, replacing signal names with actual values read from input queues, and evaluating expressions with the token operators.
- On the other hand, by processing the signal names, behavioral VHDL code generation is possible.

We thus benefit from the C++ parser to do the bulk of the work in code generation. Also, the same data structure for clock-cycle true simulation and code generation is used. Translation errors will also show up in the C++ simulation.

3.4 Related Work

Data-flow is commonly used as data model for DSP system simulations, and many tools are based on it. Only few support the transformation of a high level data-flow model to a hardware architecture. Grape-II [10] performs mapping to an architecture of concurrent DSP processors and FPGA's and is used for prototyping applications. The Aden/Combox [7] environment performs mapping to an application specific architecture.

Runtime dependent execution is supported in different ways by these tools. Grape-II uses the CSDF model [2] in which the run-time dependency is cyclic in time. Aden/Combox uses the BDF [3] model, in which tokens of dual nature are distinguished: control tokens and data tokens. The V annotation to an edge in our model is however of the more general DDF level, which makes the user responsible for the consistency of the input specification.

4 Conclusions

In this paper, we presented a software approach towards architecture synthesis for telecommunications applications. By choosing a target architecture, a high level data flow model can be gradually refined to a bit-true, cycle true model of the architecture. The data flow model allows easy expression of both multi-rate and variable rate networks. The gradual refinement avoids combined semantic and syntactic transitions in the design flow. The designer thus is allowed to catch on one problem at a time, instead of having them all to solve at once. The design environment has minimal requirements: a C++ compiler and appropriate class libraries are sufficient. The OCAPI methodology is currently being applied in the design of an upstream cable modem and a radio link base station receiver.

References

[1] M. Ade. *Data Memory Minimization for Synchronous Data Flow Graphs emulated on DSP-FPGA targets.* PhD thesis, ESAT, Katholieke Universiteit Leuven, October 1996.

[2] G. Bilsen, M. Engels, R. Lauwereins, and J. Perperstraete. Cyclo-static dataflow. *IEEE Transactions on Signal Processing*, 44(2):397 – 408, February 1996.

[3] J. T. Buck. A dynamic dataflow model suitable for efficient mixed hardware and software implementations of dsp applications. In *Proceedings of the Third International Workshop on Hardware/Software Codesign.* Grenable, France, September 1994.

[4] J. T. Buck and E. A. Lee. The token flow model. In *Proceedings of the Data Flow Workshop.* Hamilton Island, Australia, May 1992.

[5] B. F. Furber and P. Day. Four-phase micropipeline latch control circuits. *IEEE Transactions on Computers*, C-36(1):24 –35, January 1996.

[6] R. Govindarajan, G. Gao, and P. Desai. Minimizing memory requirements for rate-optimal schedules. In *Proceedings of the Intl. Conf. on Application Specif. Array Processors*, pages 75 –86, 1994.

[7] T. Grotker, P. Zepter, and H. Meyr. Aden: An environment for digital receiver asic design. In *Proc. ICASSP'95.* Detroit, MI, 1995.

[8] S. Kim, K. Kum, and W. Sung. Fixed point optimization utility for c and c++ based digital signal processing programs. In T. Nishitani and K. K. Parhi, editors, *VLSI Signal Processing, VIII*, pages 197 – 206. IEEE Press, New York, NY, October 1995.

[9] D. Knapp, T. Ly, D. MacMillen, and R. Miller. Behavioral synthesis methodology for hdl-based specification and validation. In *Proc. DAC '95*, 1995.

[10] R. Lauwereins, M. Engels, M. Ade, and J.A. Perperstraete. Grape-ii: A tool for the rapid prototyping of multi-rate asynchronous dsp applications on heterogeneous multiprocessors. *IEEE Comput.*, 28(2):35 – 43, February 1995.

[11] E. A. Lee and D. G. Messerschmidt. Static scheduling of synchronous data flow programs for digital signal processing. *Transactions on Computers*, C-36(1):24 – 35, January 1987.

[12] K. K. Parhi and D. G. Messerschmitt. Pipeline interleaving and parallelism in recursive digital filters – part i: Pipelining using scattered look-ahead and decomposition. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 37(7):1099 – 1116, July 1989.