Development of a Specification Model of the EFR Vocoder

Martin von Weymarn

July 02, 2001

Technical Report ICS-01-35

Department of Information and Computer Science
University of California, Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

mweymarn@ics.uci.edu

Abstract

This report describes the development of a High-Level Specification Model of the Enhanced Full Rate (EFR) Vocoder. The EFR Vocoder is a complex speech compression algorithm which is widely used in cellular telephone systems. The Vocoder Specification Model is based on the bit-exact simulation code provided by the ETSI and was written in the SpecC language. A detailed analysis of the simulation code was performed, followed by the development of the Vocoder behaviors in the specification language. An IP Distribution Model of the Vocoder was also developed. The report concludes with an analysis of the suitability of the SpecC language for this project.

Contents

List of Figures

Listings

1 Introduction

1.1 Overview

The majority of digital systems today are embedded systems, designed specifically for monitoring and controlling a larger system. This larger system might be an automobile, a cellular telephone or a television—to name but a few. Clearly, systems of this class have been designed, manufactured and sold for years. However, as the level of capability of microcomputers and ASIC hardware has increased, the complexity of designing embedded systems has also risen.

Typically, the design of embedded systems was divided among separate engineering groups to handle the hardware and software parts of the design. For small systems, this was appropriate. But with the increase in complexity of current-day designs, a more coordinated approach to designing mixed hardware/software systems has been developed. The approach, called hardware/-software codesign, is an attempt to integrate hardware and software design techniques with the goal of incorporating more of the system design process into a single design methodology. Several systems have been developed with this goal in mind, greatly simplifying the design process and enhancing designer productivity.

However, the recent predictions and roadmaps on silicon semiconductor technology all agree that the complexity of chips will keep growing exponentially according to Moore's Law. In the near future systems with 10ths of millions of transistors on one chip will be standard technology. System-On-Chip (SOC) designs will integrate processor cores, memories and special purpose custom logic into a complete system residing on a single die. The main problem that arises is that the industry is facing a Productivity Gap. The chip complexity that can be handled by current design teams falls short of the possibilities offered by technology advances. Together with growing time-to-market pressures, this drives the need for innovative measures to bring design productivity up to speed with current advances in chip technology.

In the past, the embedded systems design methodology was traditionally a process starting with the functional and architectural specification of the system, followed by hardware and software design, with numerous iterations between each stage. Fixing problems in such a process is very cumbersome and costly due to the serial nature of the methodology and the separate hardware and software design teams. Architectural changes are virtually impossible once the specification phase has been completed.

The solutions for overcoming these deficiencies and achieving a leap in design productivity lie in a shift of the focus from the design process to higher levels of abstraction on the one hand, and in the extensive re-use of predesigned, complex system components (Intellectual Property) on the other hand. In order to be successful, both concepts eventually require the adoption of new methodologies for system design, backed up by the availability of a corresponding set of system-level design automation tools.

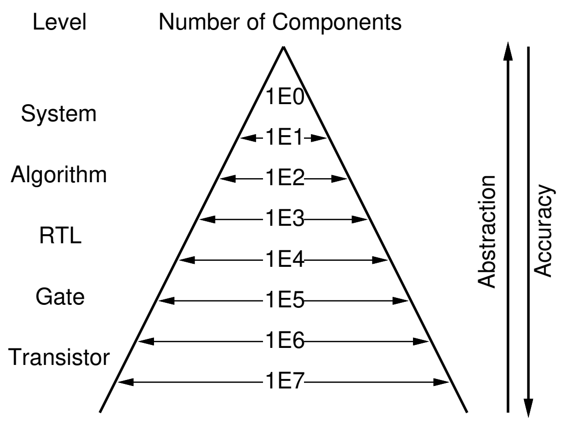

Figure 1: Abstraction vs. Complexity.

Figure 1 illustrates how moving to higher levels of abstraction effectively reduces the number of components to be handled in each task of the design process. A system, which at the transistor level is composed of 10ths of millions of transistors, typically reduces to only thousands of components at the registertransfer level (RTL). Furthermore, RTL components are grouped together at the algorithm (or behavioral) level. Finally, at the system level, the one system is composed of only few components, including processing elements (PEs), memories and busses. A topdown design methodology starts with a specification at the highest level of abstraction, the so-called system level, and stepwise moves down to lower levels refining the model. With each step, the design becomes a more and more accurate model of the final implementation.

The integration of Intellectual Property (IP) is the second key to reducing the workload on the design team and speeding up the design process. Parts of the system's functionality do not have to be developed from scratch but instead can be plugged in in the form of predesigned components. Since the process of developing the system is decoupled from the development of its components, these tasks can be performed by different companies. While system houses focus on the problem of system specification, integration and implementation, IP vendors develop and provide the required IP components. With this approach, the system house benefits from a large library of optimized, well-tested and well-documented components, while the IP providers can take advantage of their expertise in specialized areas.

In contrast to traditional ad hoc design methodologies, a design methodology that is capable of handling the complexity of current and future systems is comprised of a set of well-defined design steps. It starts from a formal specification which defines the functionality as well as the constraints of the intended design. During the design process the designer will go through a series of steps which eventually map the functionality of the specification to the target architecture. These design steps include allocation, partitioning, scheduling and communication synthesis, which form the synthesis flow of the methodology.

The specification model of the system forms the foundation of this methodology. It is a complete formal system consisting of objects and composition rules and describes the system's characteristics. Creating the specification model in an executable language as opposed to a natural language specification is highly desirable for a number of reasons. First, it allows the designer to verify correctness of the system functionality by directly simulating the specification. Second, the specification can serve as an input to design tools, eliminating the need for time-consuming and error-prone rewrites of the model in different languages. Finally, such a specification can serve as comprehensive documentation, providing an unambiguous description of the system's intended functionality.

This paper describes the development of a functional specification model of the Enhanced Full Rate Vocoder. The EFR Vocoder is a speech compression algorithm used in the GSM mobile phone system. Its considerable size and complexity along with rigorous design constraints make the application of new design methodologies imperative.

As discussed above, the specification is the basis for all the steps in a modern Embedded Systems design methodology. A high-quality specification is essential for obtaining good results during the design process. The specification model for the EFR Vocoder was developed as an executable specification in the SpecC language and then simulated and verified for correct behavior. An introduction into the SpecC language and methodology is given below.

1.2 Introduction into SpecC

1.2.1 The Methodology

SpecC is an IP-centric system-level design methodology built around a new system specification language called SpecC (Figure 2 [3, 5]).

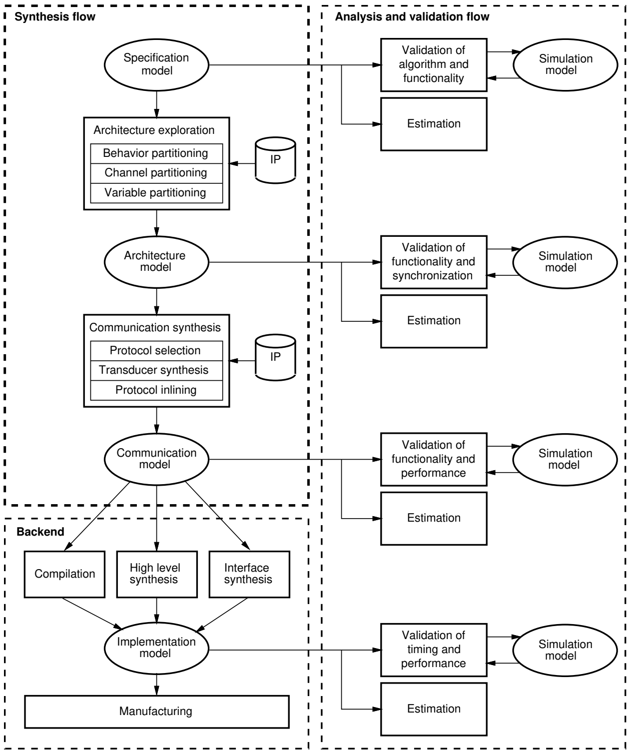

Figure 2: The SpecC Methodology [3].

Based on the specify-explore-refine paradigm, it is targeted towards the codesign of embedded systems and SOCs (systems-on-a-chip) [3]. The SpecC methodology leads designers from an executable specification to an RTL implementation on the target architecture through a well-defined sequence of steps. Easy insertion and re-use of intellectual property (IP) is supported throughout the design process.

A design using the SpecC methodology starts with an executable specification. It describes the functionality of the system as well as constraints such as timing/performance, cost and power. The designer can re-use existing code segments, functions or procedures by instantiating them out of an algorithm library. The source code can be executed with the help of a simulator to facilitate verification of the algorithms and functionality of the system. It is desirable to set the level of abstraction as high as possible since it is easier and more efficient to verify the correctness of the algorithms at a high abstration level than at a lower level which includes implementation details as well.

The next step in the synthesis flow of the SpecC methodology is architecture exploration. It refines the specification into an architecture model and includes the design steps of allocation, partitioning of behaviors, channes and variables, and scheduling. Allocation determines the number and types of system components and includes the re-use of Intellectual Property (IP). Behavior partitioning distributes the behaviors or processes amongst the allocated processing elements, while variable partitioning assigns variables to memories, and channel partitioning assigns communication channels to busses. Scheduling determines the order of execution of the behaviors assigned to the processing elements. The final result is an architecture model which represents a refinement of the specification model.

Communication synthesis then refines the abstract communication between behaviors in the architecture model into an implementation. The tasks involved are selection and insertion of communication protocols, synthesis of interfaces and transducers, and inlining of protocols into synthesizable components. The result is a communication model in which communication is described in terms of actual wires, and timing relationships are described by bus protocols.

The result of the synthesis flow is handed off to backend tools. Processor specific C compilers are used for the software parts of the hand-off model while high-level synthesis tools generate the custom hardware. The generated implementation model represents a clock-cycle accurate description of the whole system and serves as a basis for its manufacturing.

It is impossible to obtain good synthesis results if the initial specification is of low quality. Following a set of well-defined modelling guidelines ensures that the system functionality is captured in a natural way and in a clear and concise manner.

- First of all, it is necessary to clearly separate communication from computation. In addition, inputs and outputs of a computation have to be explicitly specified in order to show data dependencies and data flow.
- Secondly, parallelism inherent to the system needs to be specified and exposed instead of artificially serializing behaviors in expectancy of a serial implementation. Only by making all parallelism available to the exploration tools can the whole optimization space be utilized.
- Using hierarchy is an effective way to group related functionality and make complex systems manageable. Hiding local communication and local data dependencies when the

design is analyzed at a higher level makes refinement tools simpler and the generated results more comprehensible.

- Leaf behaviors form the smallest indivisible units for exploration. Therefore their size should be chosen in such a way that opimization possibilities and design complexity are balanced when searching the design space.

- Finally state transisitons should be used to explicitly model the steps of the computation in terms of basic algorithms or abstracted hierarchical blocks.

1.2.2 The Language

The SpecC language [1, 2, 4] was designed with these modelling guidelines in mind and supports and encourages the designer in adhering to them. In addition to that, it can be used for all models at all stages of the exploration and refinement process. This elminiates simulator interfacing problems and makes translation between languages with different semantics unnecessary. System behavior can be simulated and verified for correctness regardless of the level of abstraction the design has been refined to. Only one set of tools is needed for all models and synthesis tasks are merely transformations from one program into a more detailed one in the same language.

The fact that SpecC is based on ANSI-C [6] allows easy re-use of a huge base of existing code since C is widely used in industry. Virtually every designer knows how to program in C so it is not necessary to start from scratch and learn a completely new language. However the C language on its own is not suitable for system-level design since it does not offer any means to describe hardware, namely concurrency and timing. Two approaches have been proposed to overcome this problem: one is to add the additional functionality in form of class libraries that enable modelling of hardware. Cynlib and SystemC [17, 16] are examples for this solution. The other approach is to actually extend the C language with a set of semantically well-defined constructs to support system-level design and IP integration. This route was chosen for the SpecC language. The major advantage is that the synthesis tools have a much easier task of analyzing and understanding the specification than is the case in the library-based approach. Providing exactly one construct for commonly-used concepts in system design forces the designer to produce a clean and unambiguous description.

SpecC extends the ANSI-C language in the following areas:

1.2.2.1 Structural Hierarchy Semantically, the functionality of a system is captured as a hierarchical network of behaviors interconnected by hierarchical channels. Syntactically, a SpecC program consists of a set of *behavior*, *channel* and *interface* declarations.

A *behavior* is a class consisting of a set of ports, a set of component instantiations, a set of private variables and functions, and a public `main` function. Through its ports, a behavior can be connected to other behaviors or channels in order to communicate. A behavior is called a composite behavior if it contains instantiations of child behaviors. Otherwise it is called a leaf behavior. The functionality of a behavior is specified by its functions starting with the `main` function.

A *channel* is a class that encapsulates communication. It consists of a set of variables and functions, called methods, which define a communication protocol. A channel can be hierarchical, for example subchannels can be used to specify lower level communication.

An *interface* represents a flexible link between behaviors and channels. It consists of declarations of communication methods which will be defined in a channel.

1.2.2.2 Behavioral Hierarchy The composition of child behaviors in time is called behavioral hierarchy. Child behaviors can either be executed sequentially or concurrently. Sequential execution can be specified by standard imperative statements or as a finite state machine with explicit state transitions. Concurrent execution is either parallel or pipelined.

Behavioral hierarchy is specified in the `main` function of a composite behavior. For example, with `a`, `b`, and `c` being instantiated child behaviors, the sequence of calls

```
a.main(); b.main(); c.main();
```

simply specifies sequential execution of the behaviors `a`, `b` and `c`. The `par` and `pipe` statements specify concurrent execution. For example,

```
par { a.main(); b.main(); c.main(); }
```

executes `a`, `b`, `c` in parallel, whereas

```
pipe { a.main(); b.main(); c.main(); }
```

specifies execution in a pipelined fashion (`a` in the first iteration, `a` and `b` in the second, ...). The `par` statement completes when its last statement finishes, the `pipe` statement implicitly specifies an endless loop.

SpecC also supports explicit specification of state transitions. For example

```
fsm { a: { if (x > 0)  break;
           if (x <= 0) goto b; }
      b: { if (y > 0)  goto a;
           if (y == 0) goto b; }
      c: { break; }
    }
```

specifies the state transitions of a finite state machine model with three behaviors `a`, `b`, `c`. Implicitly the first label in the `fsm` statement specifies the initial state (`a`). The FSM exits when a `break` statement is encountered.

In summary, behavioral hierarchy is captured by the tree of function calls to the behavior `main` methods.

1.2.2.3 Synchronization Concurrent behaviors usually must be synchronized in order to be cooperative. In SpecC, a built-in type *event* serves as the basic unit of synchronization. Events can only be used as arguments to `wait` and `notify` statements (or with exceptions). A `wait` statement suspends the current behavior from execution until one of the specified events is notified by another behavior. The `notify` statement triggers all specified events so that all behaviors waiting on one of these events can resume their execution.

1.2.2.4 Exception Handling

SpecC provides support for two types of exceptions, namely *abortion* (or trap) and *interrupt*. The `try-trap` construct in the example below aborts behavior `x` immediately when one of the events `e1`, `e2` occurs. The execution of behavior `x` (and all its child behaviors) is terminated without completing its computation and control is transferred to behavior `y` in case of `e1`, to behavior `z` in case of `e2`. This type of exception usually is used to model the reset of a system.

```
void main(void)
  {
    try { x.main(); }
    trap (e1) { y.main(); }
    trap (e2) { z.main(); }
  }
```

On the other hand, the `try-interrupt` construct in the second example can be used to model interrupts. Here again, execution of behavior `x` is stopped immediately for events `e1` and `e2`, and behavior `y` or `z`, respectively, is started to service the interrupt. After completion of interrupt handlers `y` and `z` control is transferred back to behavior `x` and execution is resumed right at the point where it was stopped.

```
void main(void)
  {
    try { x.main(); }
    interrupt (e1) { y.main(); }
    interrupt (e2) { z.main(); }
  }
```

For both types of exceptions, in case two or more events happen at the same time, priority is given to the first listed event.

1.2.2.5 Timing

In the design of embedded systems the notion of real time is an important issue. However, in traditional imperative languages such as C, only the ordering among statements is specified, the exact information on when these statements are executed, is irrelevant. While these languages are suitable for specifying functionality, they are insufficient in modeling embedded systems because of the lack of timing information. Hardware description languages such as VHDL overcome this problem by introducing the notion of time: statements are executed at discrete points in time and their execution delay is zero. While VHDL gives an exact definition of timing for each statement, such a treatment often leads to *over-specification.*

SpecC, in contrast, differentiates between two types of timing information, *exact timing* and *timing ranges.* Exact timing is used when the timing is known, for example the execution delay of an already synthesized component. This is specified with a `waitfor` statement which suspends the execution of the current behavior for a specified time. The time is measured in real time units such as nanoseconds. Simulation time is only increased by `waitfor` statements, other statements are always executed in zero time.

Timing ranges are used to specify timing constraints at the specification level. SpecC supports timing information in terms of *timing di-*

agrams with minimum and maximum time constraints. Timing ranges are specified as 4-tuples $T = \langle l1, l2, min, max \rangle$ with the `range` statement. For example,

range(l1; l2 ; 10; 20);

specifies at least 10 but not more than 20 time units spent between labels `l1` and `l2`.

1.2.2.6 Additional features In addition to the concepts explained in the last sections, the SpecC language supports further constructs that are necessary for system-level design. First, SpecC provides explicit support for *Boolean* (`bool`) and *bitvector* (`bit[:]`) types, in addition to all types provided by ANSI-C.

Also, constructs for binary import of precompiled SpecC code and support of persistent annotation for objects in the language are provided. Since these constructs are beyond the scope of this paper, please refer to [6] for further details.

It has to be emphasized, that the advantage of SpecC lies in its *orthogonal constructs* which implement *orthogonal concepts*. All SpecC constructs are independent of each other, unlike for example signals in VHDL, which are used for synchronization, communication and timing. The SpecC language covers the complete set of system concepts with a minimal set of constructs. Therefore it is easy to learn and easy to understand.

1.3 Structure of the Paper

The remainder of the paper is organized as follows: Chapter 2 gives an overview of the Enhanced Full Rate Vocoder. The theory behind the employed speech compression algorithm is presented, followed by a description of each of the main functional blocks of the Vocoder.

The C-simulation code of the EFR Vocoder which forms the basis for the Vocoder specification model is described in Chapter 3. This is followed by an in-depth analysis of the simulation code in terms of functionality, structure, and data flow and storage.

Chapter 4 contains a detailed description of the development of the Vocoder specification model. First, the development process itself and related issues are discussed. Then the actual development of the specification model is described, with special attention given to specific problems encountered during development.

The paper concludes with Chapter 5. Here the characteristics of the specification model are discussed, along with the suitability of the SpecC language for the modelling of complex Embedded Systems. Furthermore, an outlook on further steps in the design process of the Vocoder system is given, along with suggestions on the extension of the SpecC language.

2 The Enhanced Full Rate Vocoder

The Enhanced Full Rate (EFR) Vocoder is an industrial size example of an embedded system. It was chosen because its characteristics make it an interesting case to study the effectiveness of new embedded systems design methodologies. The size and computational complexity of the algorithm combined with hard timing constraints allow the evaluation of architectural exploration and communication synthesis tools. Furthermore, its structure and modularity make it an ideal candidate for analyzing a decomposition into separate IP components, as well as for publishing the Vocoder itself as a speech compression IP component.

The EFR Vocoder is a speech coding and compression algorithm that is used in the European cellular telephone network system GSM (Global System for Mobile Communication). Due to the limited bandwidth available in wireless communication systems, the stream of digitized speech samples has to be compressed before it is transmitted over the channel (Figure 3).

2.1 System Overview

The Enhanced Full Rate Vocoder used in this project is part of the European standard GSM for cellular telephone network systems. The codec scheme was originally developed by Nokia and the University of Sherbrooke [11]. The so-called *Enhanced Full Rate (EFR) speech transcoding* is now standardized by the European Telecommunication Standards Institute (ETSI) as GSM 06.60 [9]. In addition, the same codec has been adopted as a standard for the American PCS 1900 system by the Telecommunications Industry Association (TIA) [12]. In general, CELP (Code Excited Linear Prediction) based codecs such as the EFR Vocoder represent the state of the art in speech coding and compression and are used in numerous applications.

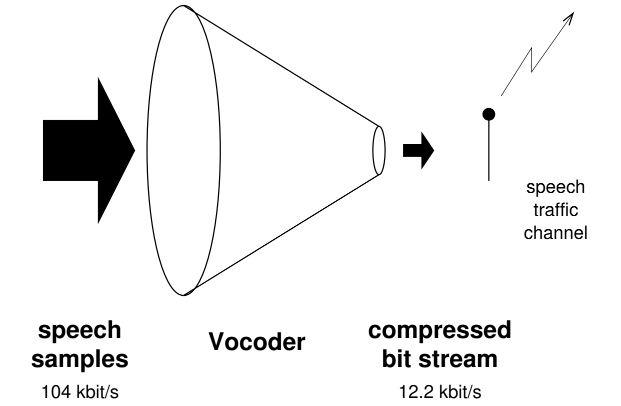

Figure 3: Datarate Reduction for Transmitting Speech Over the Wireless Link.

Cellular phone systems as well as basically all wireless services with a large customer base face the same problem: the frequency band they have been allotted is limited and they need to stay within this band to avoid interference with other services. In order to offer a high subscriber capacity the bandwidth taken up by each user has to be kept as small as possible. In the case of a cellular telephone system, the data payload is the speech data. This means that a trade-off has to be made between the number of users that can utilize the system at the same time and the quality of the audio channel. Obviously there is a lower bound for the speech quality below which voice communication is impossible and the whole system becomes useless. The design goal therefore was to develop a way of transmitting digitized speech data at as low a bitrate as possible while retaining an audio quality almost as good as customers are used to from landline communication.

Traditionally, digital telephone networks have been and are still using Pulse Code Modulation (PCM) codecs for the transmission of spoken communication. The analog speech signal is typically sampled 8000 times per second, and then each speech sample must be quantized. If linear quantization is used, then about 12 bits per sample are needed, giving a bit rate of about 96 kbits/s. However this can be easily reduced by using non-linear quantization. For coding speech it was found that with non-linear quantization, 8 bits per sample was sufficient for speech quality which is almost indistinguishable from the original. This gives a bit rate of 64 kbits/s, and two such non-linear PCM codecs were standardized in the 1960s which only differ in their characteristic quantization curve.

However 64 kbit/s is far too high a bit rate for transmission over a wireless channel so a more efficient coding scheme is called for. High compression rates are achieved by lossy methods which sacrifice the ability to reproduce the original signal bit by bit. The most effective algorithms can be designed by analyzing how speech is generated by the human vocal tract and then exploiting its specific characteristics.

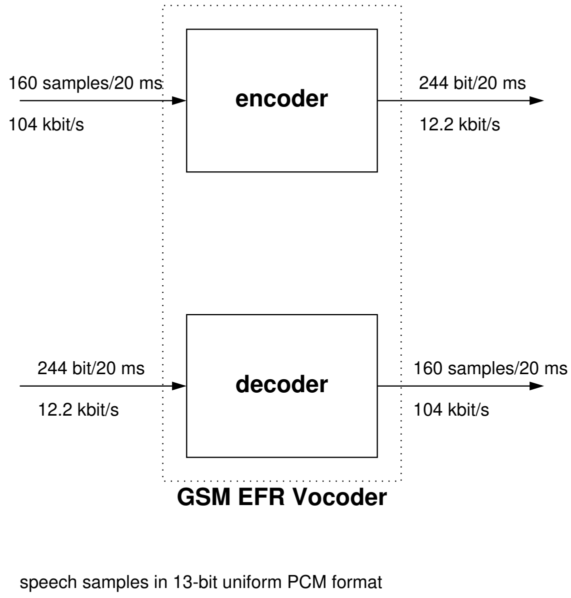

Figure 4: GSM EFR Vocoder Overview.

Human speech is produced when air is forced from the lungs through the vocal cords and along the vocal tract. The vocal tract extends from the opening in the vocal cords (called the glottis) to the mouth, and in an average man is about 17 cm long. It introduces short-term correlations (of the order of 1 ms) into the speech signal, and can be thought of as a filter with broad resonances called formants. The frequencies of these formants are controlled by varying the shape of the tract, for example by moving the position of the tongue. An important part of many speech codecs is the modeling of the vocal tract as a short term filter. As the shape of the vocal tract varies relatively slowly, the transfer function of its modeling filter needs to be updated only relatively infrequently (typically every 20 ms).

2.2 The EFR Speech Compression Algorithm

The EFR Vocoder is a hybrid codec that uses an advanced Code Excited Linear Prediction (CELP) speech synthesis model. It uses the Linear Prediction (LP) approach to model the vocal tract and provide high compression rates. The LP model's disadvantage of not being able to reproduce natural sounding speech is overcome by using a sequence of pulses (the code) to excite the linear prediction filter.

The speech synthesis model used by the Vocoder is shown in Figure 5. In this model, the excitation signal at the input of the short-term LP synthesis filter is constructed by adding two excitation vectors from the long-term pitch filter and the fixed codebook. Together, these two vectors model the vibration produced by the glottis. The speech is synthesized by feeding the two excitation vectors through the short-term synthesis filter.

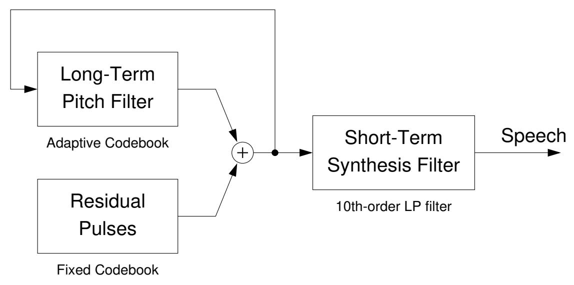

Figure 5: Simplified Block Diagram of the CELP Synthesis Model [9].

The output of the long-term pitch filter is a previous excitation sequence, modified by scaling it with a changing gain factor. The amount by which excitations are delayed in the pitch filter is a parameter of the speech synthesis model and can vary over time. The long-term pitch filter is also referred to as an adaptive codebook since the history of all past excitations can be interpreted as a codebook with varying contents out of which one past excitation sequence, the so-called adaptive code vector, is chosen.

The sequence of so-called residual pulses is built by assembling predefined pulse waveforms taken out of a given fixed codebook. The codebook contains a selection of so-called fixed code vectors which are fixed pulse sequences with varying frequency. As with the adaptive code vector the pulse intensities are scaled by applying a variable gain factor.

Finally, the excitation generated by summing up the adaptive and fixed codebook vectors is passed though the short-term speech synthesis filter which simulates the human vocal tract. Technically the synthesis filter is a tenth-order linear prediction filter which means that its output is a linear combination of ten previous inputs. The ten filter coefficients model the reflections and resonances of the human vocal tract.

The decoder is straightforward and follows the speech synthesis model. The received parameters are decoded, generating the fixed and adaptive code vectors and scaling them appropriately. The sum of both vectors forms the short-term excitation. The linear prediction filter coefficients are decoded and the speech is synthesized by passing the excitation through the parameterized short-term filter. Since the decoder is relatively simple both in terms of theoretical background and computational complexity, it is not further analyzed, and no high level specification model of the decoder was developed.

The EFR Vocoding scheme is very successful in producing a high compression ratio while maintaining a high, almost landline voice quality. All redundant information is removed and the remaining relevant data are encoded in a very efficient way. The Vocoder has also been optimized to be tolerant towards (non-speech) background noise as generally encountered in mobile communication. On the downside the algorithm, in particular the encoding part, is computationally quite intensive and also requires a lot of program space for an embedded application. As technology advances and systems become both faster and larger in capacity, this is less and less an issue.

2.3 Description of the Main Functional Blocks of the EFR Vocoder

To actually encode and compress the incoming speech data, the coder gathers the samples into frames of 20 ms corresponding to 160 samples at the sampling frequency of 8000 samples/s. For each set of 160 speech samples, the speech signal is analyzed to estimate the coefficients of the linear prediction filter. The effects of the linear prediction are then removed by inverse filtering of the incoming speech, generating an intermediate signal called the residual. It is used to extract the pitch filter parameters. Finally, the pitch filter contribution is removed in order to find the closest matching residual pulse sequence in the fixed codebook. The final output of the encoder is a set of 57 speech synthesis parameters with a total of 244 bits. This results in a output bit rate of 12.2 kbit/s.

Figure 6 shows a simplified block diagram of the EFR Vocoder algorithm. It shows the five main functional blocks of the Vocoder which will be described in greater detail below:

- Linear Prediction Coefficient (LPC) Analysis
- Open-Loop pitch search
- Adaptive codebook search
- Innovative (Fixed) codebook search
- Error computation and filter memory update

The block diagram omits the Pre-processing block which performs three functions: framing of the speech samples, initial down-scaling and high-pass filtering. Framing is done by gathering 160 digitized samples which form one speech frame. Each frame corresponds to 20 ms of speech. Down-scaling consists of simply dividing the speech signal by a factor of 2 to reduce the possibility of overflows in the fixed-point implementation. The high-pass filter has a cut-off frequency of 80 Hz and serves as a precaution against undesired low-frequency or DC components.

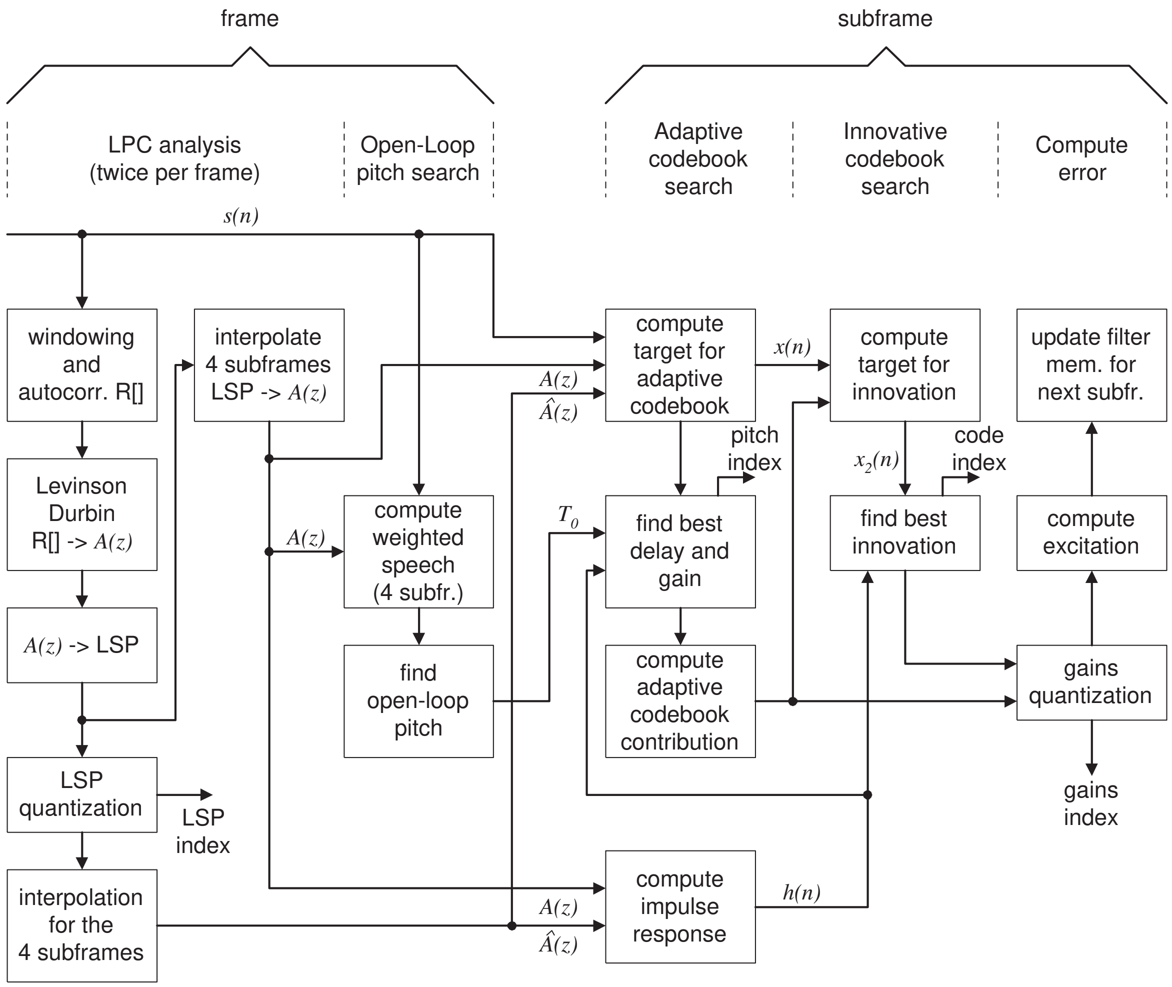

Figure 6: Simplified Block Diagram of the EFR Vocoder [9].

2.3.1 Linear Prediction Analysis

The purpose of the Linear Prediction Analysis block is the extraction of the parameters of the 10th-order linear prediction (LP) filter. This is done twice per frame of 20 ms, taking into account the current speech frame plus one half of the previous frame. The two sets of LP parameters are then converted into so-called line spectrum pairs (LSPs) which reduce the amount of redundant information. The LP Analysis produces a block of 5 parameters containing the two sets of LSPs.

First, two fixed windows $w(n)$ are applied to 80 samples from the past speech frame and 160 samples from the current frame. For each of these windows the autocorrelations $r(k)$ are computed, normalized and checked for overflow. Lag windowing is applied to the autocorrelations to produce a bandwidth expansion which reduces the possibility of ill conditioning in the Levinson algorithm, especially for an intended fixed-point implementation. The Levinson-Durbin algorithm then recursively computes the LP filter coefficients $a(k)$, which are converted to Line Spectral Pairs (LSPs) in the next step. Quantization of the second set of LSPs generates the first 5 parameters that are transmitted to the decoding side to set up the LP synthesis filter. Both the quantized and the unquantized LSPs are directly used for the second and the forth subframe. The LSPs for the first and third subframes are linearly interpolated from the adjacent two subframes respectively. For the unquantized LSPs of the second and forth subframe the LP filter coefficients are already available, for all other cases the LSPs are converted back to LP coefficients. After the LP filter coefficients have been extracted, they are converted into LSPs using Chebyshev polynomial evaluation. If less than ten roots of the polynom are found, LSPs from the past frame are used.

The last task of the linear prediction analysis is the quantization of the Line Spectral Pairs and interpolation of the quantized and the unquantized LSPs. First, the LSPs of both subframes are quantized and the five parameters (prm) that are transmitted over the wireless link generated. Then, interpolation is used to obtain the LP coefficients and the LSPs of the remaining two subframes, instead of calculating them from the speech signal.

The excitation of the LP filter is obtained by using an analysis-by-synthesis approach. The parameters are selected in such a way as to minimize the error between the original speech and the speech that will be synthesized on the decoding side. The error signal is perceptually weighted so that the chosen parameters produce the speech that sounds best. The weighting filter shapes the spectrum of the error signal so that the noise will occur in frequency regions where the speech has high energy. Therefore the error signal will be at least partially masked by the speech, and its subjective significance will be reduced. A straightforward analysis-by-synthesis technique would be to simply pass every possible waveform through the LP filter in a closed loop to see which excitation gives the minimum weighted error between the original and the reconstructed speech. While this would lead to a high-quality speech signal, the numerical complexity involved is huge. Instead, a method of reducing this complexity without compromising the performance of the codec too badly must be found.

The approach taken for the EFR Vocoder reduces the computational complexity by dividing the search for the optimal excitation into an open-loop and a closed-loop search. The open-loop pitch analysis is performed twice per frame in order to find two estimates of the pitch lag in each frame. This greatly simplifies the pitch analysis and confines the closed-loop pitch search to a small number of lags around the open-loop estimated lags.

2.3.2 Open-Loop Pitch Analysis

Open-loop pitch analysis determines delay/lag estimates for the closed-loop calculation of the pitch filter parameters in order to narrow down the actual adaptive codebook search. First, the weighted filter coefficients for the implementation of the perceptual weighting filter are calculated. One set of coefficients is calculated for the inverse LP filter and one for the LP synthesis filter. The perceptual weighting filter is applied by first filtering the speech signal through the inverse LP filter $A(z)$ to generate the LP residual. The signal is then filtered through the LP synthesis filter $1/A(z)$ to obtain the weighted speech.

The weighted speech signal is used to estimate the pitch lag for the first and the second half of the 20 ms speech frame. This is done by finding maximal correlations in the weighted speech which are used to determine the closed-loop search ranges T_0min and T_0max for each respective subframe.

2.3.3 Closed-Loop Pitch Analysis

Once the Open-Loop Analysis has been completed, the closed-loop pitch analysis calculates the two parameters that are transmitted to the decoder in order for it to be able to generate the adaptive codebook contribution to the LP synthesis excitation (see section 2.1). The first parameter is the pitch lag which is an index into the history of past excitations. The second parameter is the gain of the adaptive codebook signal.

The first step in the extraction of the pitch lag and gain parameters is the calculation of the target signal and the impulse response of the weighted synthesis filter. The impulse response $h(n)$ is computed by filtering the vector of weighted synthesis filter $A(z/\gamma_1)$ extended by zeros through the two filters $\hat{A}(z)$ and $A(z/\gamma_2)$. It is needed in the codebook searches to model the effects of a certain excitation vector on the error at the output of the LP synthesis filter. The target signal $x(n)$ for the adaptive codebook search is calculated by filtering the LP residual signal through the combination of the synthesis filter $1/\hat{A}(z)$ and the weighting filter $\frac{A(z/\gamma_1)}{A(z/\gamma_2)}$.

The pitch lag calculation is done by searching for the best pitch delay so that the mean-square weighted error between the target signal $x(n)$ and the past LP filtered excitation is minimized. The search range is limited around the open-loop pitch that was calculated earlier. The pitch lags are then encoded using 9 bits for the first and third subframes and 6 bits for the remaining two subframes where the lags are coded differentially.

2.3.4 Adaptive Codebook Search

To find the adaptive codebook gain, the adaptive codebook vector $v(n)$ is computed in a first step. The past excitation at the pitch lag is interpolated using two FIR filters. The adaptive codebook vector is then filtered through the weighted synthesis filter by convolving it with the impulse response $h(n)$. The codebook gain is computed by matching the filtered adaptive codebook vector with the target excitation signal $x(n)$. Before the gain is transmitted, scalar quantization is performed to encode the pitch gain using 4 bits of the parameter block.

2.3.5 Fixed Codebook Search

Once the closed-loop lag search has yielded the exact filter delay and gain values, the long-term filter contribution is subtracted from the excitation, and the remaining residual becomes the input to the following fixed codebook search. For each subframe an extensive search of the codebook for the closest code vector is performed. All possible vectors are searched so that the mean square error between the code vector and the residual is minimized.

An algebraic codebook with 35 bits is used as the fixed codebook in which the codebook vectors are determined from the transmitted index using simple algebra rather than look-up tables. This is advantageous in terms of storage, search complexity and robustness.

Each excitation vector contains 10 non-zero pulses, with amplitudes +1 or −1. The 40 possible positions in a subframe are divided into 5 tracks, where each track contains two pulses, as shown in table 1.

Track	Pulses	Positions
1	p0, p1	0, 5, 10, 15, 20, 25, 30, 35
2	p2, p3	1, 6, 11, 16, 21, 26, 31, 36
3	p4, p5	2, 7, 12, 17, 22, 27, 32, 37
4	p6, p7	3, 8, 13, 18, 23, 28, 33, 38
5	p8, p9	4, 9, 14, 19, 24, 29, 34, 39

Table 1: Potential Positions of Individual Pulses in the Algebraic Codebook.

The search of the algebraic codebook for the best pulse positions is performed by minimizing the mean square error between the weighted input speech and the weighted synthesis speech. The target signal and the long-term prediction residual used in the closed-loop pitch search are updated by subtracting the adaptive codebook contribution. The impulse response of the synthesis filter is also updated by applying a prefilter. It is used later for filtering the selected codevector.

An exhaustive examination of all possible pulse positions by systematically calculating the excitation vector for all combinations and comparing it to the target vector would be computationally extremely expensive and make a real-world implementation of the Vocoder impossible. In order to speed up the codebook search, a non-exhaustive approach is used, based on a heuristic depth-first tree search technique.

First, the correlation between the target signal $x_2(n)$ and the impulse response $h_E(n)$ is computed. This yields a vector $d(n)$ which corresponds to the backward filtered target signal. The position with the maximum correlation in each of the five tracks is calculated which gives the starting positions of each pulse. The first pulse p0 is always set in the position corresponding to the global maximum value. For each of the five tracks, the pulse positions with maximum absolute values of the sum of the backward filtered target and the prediction residual are searched. From these the global maximum value for all the pulse positions is selected.

In the following five iterations the position of pulse p1 is set to one of the five track maxima. The remaining pulses searched in pairs by sequentially adding each of the pulse pairs (p2, p3), (p4, p5), (p6, p7) and (p8, p9) in nested loops. All nine initial pulse positions are cyclically shifted so that the pulse pairs are changed and the pulse p1 is placed at a local maximum of a different track. In the search, at least one pulse position is located corresponding to the global maximum, and one pulse is located in a position corresponding to one of the five local maxima. Once the 10 pulses have been determined the fixed codebook vector $c(n)$ is established and encoded for transmission.

The selected codevector is filtered through an adaptive prefilter by convolving it with the impulse response $h_E(n)$. This serves to improve speech quality by enhancing spectral components.

3 The EFR Vocoder Simulation Code

3.1 Overview

The European Telecommunications Standards Institute distributes the EFR Vocoder in the form of a bit-exact simulation model [10]. The code represents a complete implementation of the Vocoder algorithms, including voice activity detection, comfort noise generation and bad frame handling. It is written in ANSI-C and uses 16-bit fixed-point arithmetic, intended for implementation on a Digital Signal Processor (DSP). This simulation model is the basis for the high-level specification of the Vocoder that was developed as part of this work. The fact that the C model is already an implementation for a specific type of architecture (DSP) means that it is not the ideal base for extracting the abstract behavior of the Vocoder. However, the knowledge that a specific stream of input samples will always compute to the same output is essential for debugging during the development of the specification model. Furthermore, if an actual implementation of the Vocoder was to be type-approved for use in a GSM system, a test sequence would need to match bit-by-bit with the sequence generated by the simulation model.

The bit-exact Vocoder simulation model consists of three separate programs: the speech encoder, the speech decoder and an encoder/decoder interface. The encoder takes a binary speech file as input, runs the Vocoder encoding algorithm on each frame and produces a binary encoded parameter file. The input to the decoder is a binary parameter file and the output is a binary synthesized speech file. Since the format of the input parameter file required for decoding is not the same as the format of the parameter file generated by encoding, the input file has to be translated. This is the purpose of the interface program. Apart from the source code, the distribution contains a speech sample file with a short sentence that has been recorded and digitized. Known good output files are also provided and can be compared to the files generated during the simulation runs. Correct behavior can thus be verified.

3.2 Analysis of the C Simulation Code

The first step in developing the specification model was to analyze and understand the given

C-simulation code of the EFR Vocoder. This is comprised of several tasks: First it was necessary to obtain an overview of the general structure of the code base. Next, data flow and data storage were analyzed. Finally, the code was investigated for inherent parallelism and opportunities to apply pipelining.

3.2.1 Structure of the Code

Since the EFR simulation code is written in the C programming language, the very first step in the analysis of the code was to locate the `main()` function. This is the function that is called by the operating system, and through which in turn all the other functions are called. In other words, execution of the program always starts and ends with the `main()` function. In the case of the EFR Vocoder, the `main()` function is contained in the file coder.c.

3.2.1.1 The `main()` Function Analysis was continued by examining the structure of the `main()` function itself. A typical program that processes a stream of data such as the Vocoder follows a well defined scheme: First, data storage is allocated and initialized, followed by the reading of the input data. Then, the input data is processed, finally followed by a function that outputs the processed data. The Vocoder simulation code follows this scheme. In the first lines of the `main()` function, file pointers for the input and output files are allocated. These are followed by definitions of variables which act as buffers for the input data as well as the processed output

```
main function {

  initialize coder
  open speech file
  open output file

  while more samples available {
    read one frame into buffer
    filter and downscale the frame
    find the EFR speech parameters
    convert parameters to bitstream
    write bitstream to file
  }

  close files

}
```

Listing 1: Pseudo-Code of the `main()` function.

data. After the definition of further flag and index variables, the function `reset_enc()` is called. It is contained in the file `e_homing.c` and brings the encoder to its initial state.

Execution continues with the opening of both the input file and the output file, the names of which are passed on the command line. The input file contains a stream of digitized speech samples. A while loop in the `main()` function reads samples from the file until one frame of 160 samples has been read. Then the function `pre_process()` is called, which applies a low-pass filter in addition to scaling the input data with a factor of 0.5. Pre_processing is done in the file `pre_proc.c`.

The actual encoding algorithm follows with the call of the function `Coder_12k2()`. It is located in the file `cod_12k2.c` and will be described further below. Output of the encoding function is a vector of 16 bit integer variables which contain the calculated parameters of the input speech samples. The function `Prm2bits_12k2()` converts this vector to a compressed bit stream by applying a table of accuracies needed for each of these parameters. This function can be found in the file `prm2bits.c`. The calculated bit stream is then written to the output file, after which processing continues with the next frame of speech samples read from the input file.

Apart from the routines described above, the `main()` function performs a number of smaller tasks. These include parsing of the command line, error handling, in-band reset of the encoder, and the DTX flag. The provision of an in-band reset allows the re-initialization of the encoder by placing a specific homing sequence in the input data. Upon detection of this sequence the reset function is called and processing continues. DTX stands for Discontinuous Transmission and describes an operating mode of the encoder. When DTX is enabled, the encoder performs a Voice Activity Detection (VAD) on the input data. If the VAD routines determine that the frame does not contain speech data, the encoder generates a special silence frame instead of the parameter vector. This conserves both bandwidth on the air channel and power consumption.

3.2.1.2 The `Coder_12k2()` Function The process of speech encoding takes place in the function `Coder_12k2()`. It takes the input of 160 speech samples and produces the set of encoded parameters which include the LSP, adaptive codebook, and fixed codebook quantization indices. This function forms the bulk of the simulation code and spans over a total of more than 40 source files. Therefore it was important to first of all determine which parts of the encoding functions implement which block of the EFR algorithm. The boundaries between these blocks had to be identified, since in some places they were obscured by code optimizations and additional housekeeping functions. Because the simulation code is distributed among a number of separate source files, it also had to be determined which of these files belong to which blocks of the algorithm.

As can be seen in Figure 6, the main five blocks of the EFR algorithm are Linear Prediction (LP) Analysis, Open Loop Pitch Search, Closed Loop Pitch Search, Algebraic Codebook Search, and Filter Memory Update. Analysis of the function `Coder_12k2()` revealed that these blocks are not implemented as separate functions. Instead, `Coder_12k2()` contains a sequence of all of the lower level function calls needed to perform the encoding part of the EFR algorithm.

3.2.1.3 Linear Prediction Analysis functions LP Analysis is the first step in the speech encoding process, during which the parameters of the linear prediction filter are extracted. It is a three-step process which is performed twice per frame of 20ms. It starts in line 239 of the file `cod_12k2.c` by computing the autocorrelation of the current speech frame and the sec-

```
Coder_12k2 function {
  /* LP Analysis */
  calculate filter coefficients
  convert them to LSPs
  quantize LSPs
  interpolate LSPs

  /* Open-Loop Pitch Search */
  loop over subframes {
    apply speech weighting filter
    calculate residual signal
    synthesize weighted speech
  }
  calculate Open-Loop pitch delays

  /* Closed-Loop Pitch Search */
  loop over subframes {
    find weighted LP coefficients
    compute filter impulse response
    calculate search target vector
    calculate pitch lag
    encode pitch lag
    calculate excitation gain

    /* Codebook Search */
    search codebook
    apply pitch contribution
    qunatize codebook gain

    /* Filter-Memory Update */
    synthesize speech
    update filter memories
  }

  /* Shift signal buffers */
  shift buffers for next frame
}
```

Listing 2: Pseudo-Code of the `Coder_12k2()` function.

ond half of the previous frame. The function `Autocorr()` from the `file autocorr.c` calculates the autocorrelation values. Next, the function `Lag_window()` from the file `lag_wind.c` applies a window to the autocorrelation. It is followed by the application of the Levinson-Durbin algorithm which computes the filter parameters. The corresponding function is `Levinson()` called from the file `levinson.c`. Finally, the parameters are converted to Line Spectral Pairs (LSPs) by the function `Az_lsp()` from the file `az_lsp.c`. The same sequence is executed a second time, the only difference being the application of a different window function on the speech samples. Also included as part of the LP Analysis block are the voice activity detection, along with handling of the DTX flag and preliminary steps for the generation of pseudo-noise in the case of silence having been detected.

The second part of the LP Analysis deals with the quantization of the LSPs and the interpolation of both the quantized and the unquantized filter parameters. All of the LSPs are quantized by the function `Q_plsf_5()` which resides in the file `q_plsf_5.c`. In line 322 the function `Int_lpc2()` is called from the file `int_lpc.c`. It performs the interpolation of the unquantized LSPs. The same source file also contains the function `Int_lpc()` which works on the quantized LSPs. This function also generates a set of LP coefficients from the quantized LSPs. However `Int_lpc()` is only executed if speech has been detected, otherwise the unquantized parameters are used instead of the quantized ones.

3.2.1.4 Open-Loop Pitch Search Functions

The Open-Loop Pitch search of the EFR Vocoder estimates delay and lag values that narrow down the later codebook search. In the simulation code, this block starts in line 371 of `coder_12k2.c`. The first part of the Open-Loop Pitch search consists of a for-loop over the four subframes that form one frame of speech samples. The function `Weight_Ai()` from the file `weight_a.c` implements a perceptual weighting filter. Within the loop it is applied twice to the unquantized filter coefficients of the currently indexed subframe, with different expansion factors respectively. The first set of weighted filter coefficients is then used by the inverse LP filter implemented by the function `Residu()`. It is located in the source file `residu.c` and yields the LP residual signal which is then filtered through the synthesis filter using the second set of weighted filter coefficients. The synthesis filter is implemented by the function `Syn_filt()` contained in the file `syn_filt.c` and generates the weighted speech signal.

In the remainder of the Open-Loop pitch search block the pitch delays are calculated using the weighted speech signal as an input. The calculation is done by the function `Pitch_ol()` contained in the file `pitch_ol.c`. It is called first for the first two subframes, followed by some code to limit the delay values within a certain range. A second call of `Pitch_ol()` calculates the Open-Loop delays for the second two subframes of the speech frame. A DTX mode related function follows which prepares the voice activity detection for the next frame.

3.2.1.5 Closed-Loop Pitch Search Functions

The Closed-Loop Pitch search is the largest block of the EFR Vocoder. Its purpose is to calculate the parameters that describe the excitation signal of the LP filter in order to recreate the speech signal. Similar to the Open-Loop block, the Closed-Loop Pitch search loops over the four subframes of each speech frame. This for-loop starts in line 472 and ends in line 726 of the file `cod_12k2.c`. The loop starts with two calls of the `Weight_Ai()` function which apply weighting filters to the interpolated LP filter parameters. Two calls of the synthesis filter `Syn_filt()` yield its impulse response. The following sequence of calls to the `Residu()` function and the `Syn_filt()` function calculate the target signal for the remaining steps of the EFR algorithm.

The function `Pitch_fr6()` that is located in the file `pitch_f6.c` is then called. It calculates the optimal pitch lag which is then encoded by the function `Enc_lag6()` from the file `enc_lag6.c`. The next four functions compute the gain of the LP excitation signal. The `Pred_lt_6()` function can be found in the file `pred_lt6.c`, the function `Convolve()` is held by the file `convolve.c`. The gain is computed by calling the function `G_pitch()` from the file `g_pitch.c` and the quantized by a call to the function `q_gain_pitch()` which can be found in the file `q_gains.c`. After the gain has been computed, it is applied to the target signal, the residual signal and the impulse response of the synthesis filter.

In line 641 the function `code_10i40_35bits()` is called. Because of its complexity it was decided to handle this function as a separate block of the EFR algorithm and will be described further below. If silence was detected earlier in the code, comfort noise is generated instead of incorporating the gain and calling `code_10i40_35bits()`. Next, the pitch contribution is added to the code followed by the calculation and quantization of the codebook gain. The gain is obtained by calling the function `G_code()` from the file `g_code.c`. It is then quantized by the function `q_gain_code()` from the file `q_gains.c`.

The remaining lines of the Closed-Loop Pitch search synthesize the speech signal and update the filter memories. The total excitation is calculated and the synthesis filter function `Syn_filt()` is called. Finally, the speech and the error filter memories are updated or set to zero in the case of no speech activity in the current frame.

3.2.1.6 Algebraic Codebook Search Functions

The function `code_10i40_35bits()` implements the Algebraic Codebook Search block of the EFR algorithm. It can be found in the file `c1035pf.c` along with all the sub-functions that are called during its execution. First, the function `cor_h_x()` calculates the correlation between the target signal and the impulse response of the synthesis filter. A call of the `set_sign()` function builds the sign vector of the excitation pulses. The function `cor_h()` then computes the autocorrelations of the impulse response which are needed by the actual code search performed by the function `search_10i40()`. Once the code positions have been found, the code vector is built by the function `build_code()`. A loop over all ten pulses uses the function `q_p()` to encode the pulse positions and concludes the Codebook Search block. Execution continues with the remainder of the Closed Loop Pitch search.

```
codebook search function {

  compute correlation
  build sign vector
  compute correlations
  search best codevector
  build codeword

  loop over all pulses {
    set pulse position
  }

}
```

Listing 3: Pseudo-Code of the codebook search function.

At this point, the current speech frame has been completely encoded and the speech parameters are ready for transmission over the wireless link. Because the EFR algorithm uses both the current and the previous frames of speech samples it is necessary to update the corresponding memories before proceeding to the next frame. The update is done at the very end of the `Coder_12k2()` function for the unprocessed speech samples, the weighted speech samples, and the excitation signal. The encoding function then returns and can be called for the next frame of input data.

3.2.2 Implementation Details

The EFR simulation model is a 16-bit fixed point implementation of the EFR algorithm. In order to be able to run the simulation on a number of different systems, the implementation has to be architecture- and compiler-independent. This is ensured by using typedefs for all the variables used in the code and by wrapping all basic operations in an additional wrapper layer.

The typedefs are necessary because in the C language the number of bits used to represent an intrinsic type such as Integer is compiler-dependent. Code that relies on a type having a certain number of bits could malfunction if compiled and executed on a different platform. By using typedefs the variables can be mapped to the according type that uses the desired number of bits for the specific platform. In the case of the simulation code these typedefs can be found in the header file `typedef.h`. Three types are used in the code: `Word16`, `Word32`, and `Flag`. `#ifdef` statements map them to the according types depending on the target platform. For a typical 32-bit architecture these are `short`, `integer`, and `integer` since C does not offer a type that is only one bit long.

Wrapping the basic operations such as additions and multiplications ensures that they perform as expected regardless of the bit width of the target platform. Specific behavior such as overflow and saturation can be emulated if not available, as well as instructions such as MACs

```
Word16 mult (Word16 var1, Word16 var2)
{
  Word32 L_product;

  L_product = (Word32)var1 * (Word32)var2;

  L_product =
    (L_product & (Word32)0xffff8000L) >> 15;

  if (L_product & (Word32)0x00010000L)
    L_product = L_product | (Word32)0xffff0000L;

  return (saturate (L_product));
}
```

Listing 4: 16-bit Fixed Point Multiplication Function.

which might not be offered on the intended platform. The file `basicop2.c` contains a set of functions that emulate the instruction set of a typical 16-bit fixed point Digital Signal Processor (DSP) and thereby makes it possible to run DSP-style code on a general purpose processor. Listing 4 shows the function for a 16-bit multiplication. It takes two 16-bit signed integer values as input and returns a scaled and saturated 16-bit result.

3.2.3 Data Flow and Storage

The next step in the analysis of the EFR simulation code was the investigation of data flow and storage. Being a stream processing type of algorithm, data buffers play a central role in the EFR Vocoder. Since its basic function is to encode frames of 160 speech samples into a parameter block of 244 bits, at least an input and an output buffer are required. Additionally, a number of further important buffers and variables are defined in the simulation code. All of these were examined and are presented below.

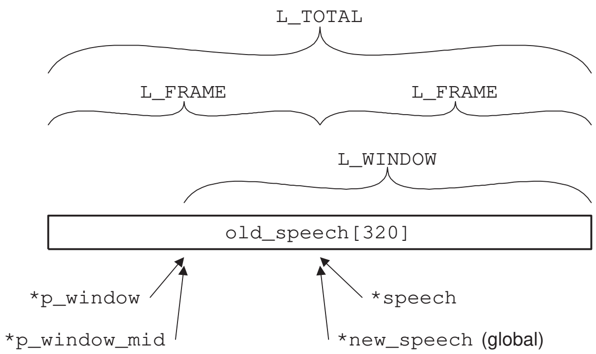

Figure 7: Input Speech Buffer and Pointers (at initialization).

The input buffer of the EFR simulation code is an array that holds the speech samples necessary to compute one parameter block. Its layout is shown in Figure 7. The buffer is defined as a static array `old_speech[]` in the file `cod_12k2.c`. The static declaration means that it can only be accessed directly from within this file. The size of the input buffer is 320 variables of the type Word16. The number of variables is the equivalent of two speech frames, which is necessary because the EFR Algorithm uses the past speech frame in addition to the current frame. The first half of the array always holds the old frame whereas the second half is filled with the new samples. Since the samples are 13 bits wide, the type Word16 was chosen for the input buffer. Apart from the buffer itself several pointers to it are defined which are used throughout the code to access different portions of the speech buffer. Three of these pointers are static, whereas the fourth, `*new_speech`, is a global variable. It is used to fill the buffer with one frame of new speech samples before encoding begins.

Unlike the input buffer, the output buffer `serial[]` is defined in the file `coder.c`. The output of the EFR Vocoder is a block of 244 bits per speech frame. For reasons of simplicity, the output buffer was defined as an array of 244 variables of the type Word16, because the C language does not support single bit types nor bit fields. Therefore the variables of the output buffer only hold values of zero or one.

Apart from the input and output buffers, additional buffers are needed by the simulation code. All of these are defined as static arrays within the file `cod_12k2.c`; the type is in all cases `Word16`. The speech samples that were filtered through the perceptual weighting filter are buffered in the array `old_wsp[]`. It holds one frame of samples along with a history that goes back the maximum pitch lag into the previous frame. The pointer `*wsp` is defined, which is used to access the weighted speech buffer. Similarly the buffer array `old_exc[]` stores the excitation vector, with a history of the maximum pitch lag extended by the length of the interpolation filter. The pointer `*exc` provides access to the excitation buffer. The impulse response of the synthesis filter is buffered in the array `hvec[]` which has a length of two subframes. It is later accessed by the pointer `*h1`. Buffers for the unquantized and the quantized Line Spectral Pairs are defined, along with a total of four buffers that are used as filter memories.

By using pointers to access the buffers indirectly instead of using the names of the arrays directly, a characteristic of the C language is exploited. In C, the name of an array is actually a pointer to its first element. Accessing a specific element is done by simply adding an offset to the address of the first element. In the simulation code, the buffer pointers are initialized to point to the first element of the data that belongs to the current frame, i.e. to an address that lies within the memory block that holds the buffer. The pointer can now be used in place of the array name, and an access of element zero will return the first element of the current data. In order to access the data of the previous frame, negative indices are used which move the address backwards into the history portion of the buffer.

While this technique may lead to faster and more efficient code, it reduces legibility and comprehensibility. In the simulation code, the situation is made worse by the definition of several pointers that point to the same buffer array. For example, a total of four different pointers are defined through which the input speech buffer is accessed. The different names imply different data, which is not the case. Further analysis revealed that two of these pointers are initialized to the same element of the buffer and never changed during execution, making the second pointer redundant.

Data flow between the functions of the simulation code is centered on the buffers described above. Unfortunately the programming style of the simulation code makes it hard to follow the flow of data. This is in part due to the fact that the C language only offers call-by-value function parameters. In the case of arrays only the address of the first element is passed, which means that it basically becomes a call-by-reference call. The consequence of this shortcoming is the increased use of pointers which often obscure the meaning of the code. It is not immediately clear if a single value is passed or a whole array, nor can it be determined if the data is only read or if elements are modified.

To summarize, the simulation code of the EFR Vocoder uses a buffer approach that is typical of data stream processing algorithms. However it is implemented in a way that makes it hard to understand where data is stored and how it flows between the functions. This is in part due to constraints that the choice of the C language imposes, but the placement of the buffers and the use of numerous pointers make this problem worse. For example, there is no logical reason why the input buffer `old_speech[]` is defined in the file `cod_12k2.c` whereas the output buffer `serial[]` is defined in the file `coder.c`. These facts along with the structure of the code make it clear that the simulation code was not meant to be the basis for an actual implementation of the EFR Vocoder as a part of a system. Unfortunately it is the only implementation published by the ETSI, which means that no alternatives exist.

3.2.4 Analysis of Parallelism

3.2.4.1 The Role of Parallelism in a Specification Model After data flow and storage had been investigated, the simulation code was analyzed for potential speedups. A number of techniques exist to increase the execution speed of code, for example constant expression elimination, loop unrolling and chaining. They work by trying to avoid unnecessary CPU cycles through rearranging and modification of instructions. However, most of the classic techniques are geared towards code that is intended to be executed on a single processor. They are therefore not generally suited for embedded systems which often use a number of processors operating in parallel and/or employ custom hardware to execute parts of the functionality. In fact, thus optimized code might actually run slower on such a system than the unoptimitzed version because the design tools and compilers could be capable of applying better scheduling and optimization for a specific target architecture. Another disadvantage is that the optimized code is generally less readable and understandable by a human designer. The optimization techniques mentioned above should therefore not be applied to code that is used to specify the behavior of an embedded system.

In order to fully utilize all the parallel units available in the target architecture, the available parallelism in the code needs to be extracted and specified. While this could possibly be performed by automatic tools, it is a complex task and therefore might not yield good results. It is also likely that the output of such tools will be less readable by a human designer. Identifying parts that could be executed in parallel requires a good understanding of the code, specifically of the data flow.

The execution of two or more blocks of code in parallel requires that they are independent of each other. This means that while each of the blocks is executing, its input data cannot be modified and its output data cannot be accessed. The blocks that are run in parallel should not be very different in terms of complexity either. It would not make sense to run a few simple instructions in parallel with a long loop because of the overhead that the parallel execution introduces.

3.2.4.2 Parallelism in the Vocoder The structure of the EFR Vocoder suggests that a lot of parallelism should be available. A considerable amount of its functions work on a subframe basis, which should make their execution in parallel possible. The stream processing nature of the Vocoder also makes it a candidate for pipelining. Closer analysis of the data flow however revealed that in many cases the potential blocks are not completely independent. This is due to some characteristics of the EFR algorithm. For example, in some cases results are interpolated between the subframes, thereby requiring access to the previous results. However, it was possible to identify a number of blocks that allow execution in parallel.

The process of extracting potential parallelism will be shown by using the LP Analysis block as an example. The LP Analysis calculates the filter coefficients (LSPs) for all four subframes of the current speech frame. It does this by first applying two windows to the speech samples, centered at the second and the fourth subframe respectively. Using these two sets of data, the LSPs of the second and fourth subframes are computed. The LSPs of the remaining first and third subframes are obtained through interpolation, once for the unquantized LSPs and once after the LSPs have been quantized.

The structure of the LP Analysis block described above suggests that the calculation of the LSPs for the second and fourth subframe could be executed in parallel. The interpolation can only be started once both sets of LSPs are available, but the computation of the unquantized and the quantized LSPs could again be performed in parallel. Analysis of the code, however, revealed that not all of the functions in the envisioned parallel branches are truly independent. Calculating the LSPs is a four-step process in which the fourth step is the conversion of the LP filter coefficients into LSPs. The conversion uses Chebyshev polynomial evaluation which does not always find enough roots, in which case the algorithm reverts to the previous values. In the case of the second subframe, these are the LSPs of the fourth subframe of the previous speech frame. However, for the fourth subframe, these are the LSPs of the second subframe. This means that the LSP conversion cannot be performed in parallel because otherwise the LSPs of the second subframe would not be ready in time for use by the conversion of the fourth frame's filter coefficients. The first three steps of the LP coefficient calculation are either independent or—in the case of the autocorrelation function—can be made independent by minor code changes. The `Autocorr()` function returns a scaling factor that is needed by the Voice Activity Detection. In the simulation code, the scaling factor of the second subframe is overwritten by the factor of the fourth subframe. This would cause a hazard if both functions were to be executed in parallel. Since the scaling factor of the second subframe is not used, it can be replaced by a dummy variable, thereby eliminating the hazard.

The interpolation of the remaining LSPs can also be calculated in parallel. One branch contains the interpolation of the unquantized LSPs while the other contains the sequence of the quantization, followed by the interpolation of the remaining quantized LSPs. To summarize, most of the LP Analysis block could be executed in parallel fashion with only minimal code changes necessary to avoid hazards. However, the available parallelism is limited to two functions executing at the same time.

The remaining blocks of the EFR Vocoder offer some limited parallelism. In the Open-Loop Pitch Analysis, two sets of weighted speech samples are calculated using different spectral expansion factors. The weighting is independent and can be executed in parallel. The pitch estimation is performed for both halves of the speech frame and can also be performed in parallel. The Closed-Loop Pitch Analysis is mostly a sequential process. The exception is the calculation of the impulse response and the target vector, the two of which can execute at the same time. A case in which three functions can run in parallel is found in the Fixed-Codebook Search. Here the pitch contribution for both the target vector and the long-term prediction residual are independent along with the prefiltering of the impulse response. All three can therefore execute in parallel. After the code has been found, its prefiltering can be run in parallel with the calculation of its gain.

In conclusion, the EFR Vocoder does offer a limited amount of parallelism, some of which can be exposed by minor modification or rearrangement of the code. It is however in almost all cases limited to two functions running in parallel. The limiting factor are some specific characteristics of the algorithm, specifically the use of history buffers in cases where functions cannot calculate a result. Being a stream processing algorithm, a large number of blocks also immediately take the results of the previous block as input data, which further limits available parallelism.

4 Development of the Specification Model

The choice of SpecC as the modeling language had a significant impact on the development process itself. If a typical language such as VHDL had been used, it would have been necessary to completely rewrite the code of the EFR Vocoder. Considering its size of approximately 13,000 lines of code, this would have been a lengthy process, most likely introducing bugs that would have taken more time to find and correct. It would also have involved adding functions to the simulation code in order to capture intermediate results. For each of the rewritten blocks a testbench would have had to be developed, using the intermediate results to verify correctness. A possible alternative would have been to write a specification model in the C language. While this option would have allowed the re-use of code from the simulation model, it would merely have delayed the rewriting of the model. This is due to the fact that the C language is not synthesizable and therefore not suitable for designing hardware.

The SpecC language combines the advantages of both approaches. Since it is based on ANSI C it allows incorporation of existing C code segments, thereby greatly reducing development and testing time. It is also synthesizable, which means that the language is suitable for designing both software code and specialized hardware units.

The use of the SpecC language allowed the use of a top-down approach for the development of the High-Level Model. Since leaf behaviors can contain arbitrary C code, it was possible to place the complete EFR simulation code into one behavior and execute it using only SpecC tools. From this point, the code was gradually broken down into smaller behaviors and the model refined step by step, starting at the highest level in the structure of the code. Throughout the development process the model always remained executable and thereby simulateable. At no point was it necessary to extend the testbench or write separate local testbenches for testing particular behaviors.

4.1 Graphical Representation of the SpecC Behaviors

In the following sections, a graphical representation of the main Vocoder behaviors is used in order to illustrate structural and behavioral hierarchy and data flow. Figure 8 shows a small constructed design which serves to explain the meaning of the graphical objects. The corresponding SpecC code is presented in Listing 5.

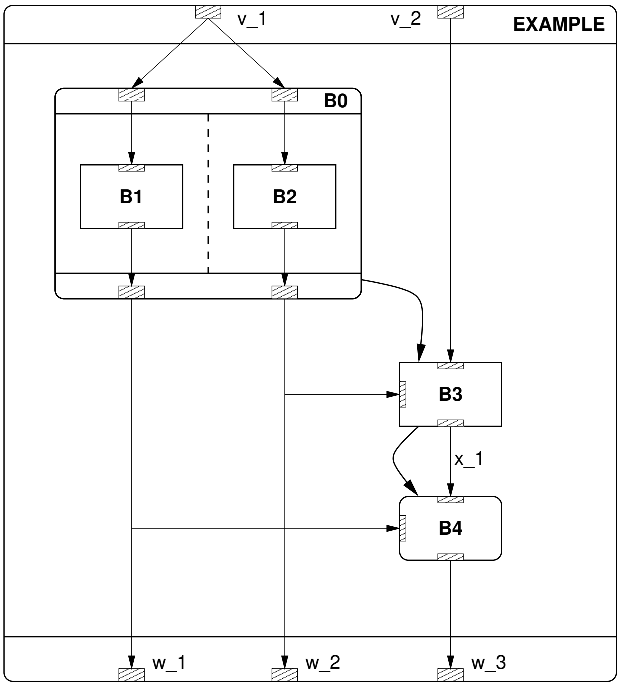

Figure 8: Example of the Graphical Representation of Hierarchy and Data Flow.

```
behavior B1(in int v_1,
            out int w_1 )
{
   void main(void)
      { /* B1 computation */ }
};


behavior B2(in int v_1,
            out int w_2 )
{
   void main(void)
      { /* B2 computation */ }
};


behavior B0(in int v_1,
            out int w_1,
            out int w_2 )
{
   B1 B1_instance;
   B2 B2_instance;

   void main(void)
      { par {B1_instance.main();
             B2_instance.main(); }
      }
};


behavior B3(in int v_2[6],
            in  int w_2,
            out int x_1[6] )
{
   void main(void)
      { /* B3 computation */ }
};
```

```
behavior B4(in int x_1[6],
            in  int w_1,
            out int w_3[6] )
{
   /* further subbehavior instantiations */

   void main(void)
      { /* B4 computation */ }
};


behavior EXAMPLE(in int v_1,
                 in int v_2[6] )
{
   B0 B0_instance;
   B3 B3_instance;
   B4 B4_instance;

   void main(void)
      {
         B0_instance.main();
         B3_instance.main();
         B4_instance.main();
      }
};
```

Listing 5: SpecC Example Design.

The top-level behavior `example` is a composite behavior, denoted by the rounded box. A composite behavior contains instantiations of child behaviors, in this case behaviors `B0`, `B3` and `B4`. `B0` is itself a composite behavior, containing behaviors `B1` and `B2`. The angular rectangles indicate that both are leaf behaviors. A leaf behavior is the smallest indivisible unit for exploration and contains algorithmic C code. The dashed line between `B1` and `B2` means that they are executed in parallel. Upon completion of both `B1` and `B2` execution continues with leaf behavior `B3` and composite behavior `B4`. The sequence of execution is shown by the fat, curved arrows between the rectangles.

Ports and their connection is shown by thin straight arrows. Variables `v_1` and `v_2` enter the example behavior though input ports and are processed by the subbehaviors `B1`, `B2`, `B3` and `B4`. `w_1`, `w_2` and `w_3` are the results of the computation and leave `example` through output ports.

This graphical representation is ideally suited to visualize the structural composition and data flow of SpecC behaviors. Structural hierarchy is a very important feature when dealing with large and complex designs such as the vocoder, and its benefit will become very obvious when the main vocoder blocks are being described in the following. Being able to visualize the design at a certain level of abstraction while hiding all the low level imlementation details and algorithms greatly eases the understanding and clarity of a complex system and lets the designer focus on his task without being swamped by unnecessary information.

4.2 The Development Process

During the refinement of the High-Level model, a number of tasks were performed which will be described in the following.

The first step in the development of the model was to determine which parts of the simulation code belong to the testbench and which parts are the actual Vocoder implementation. This is important because only the Vocoder itself is to be modeled. The simulation code applies some processing to the input and the output data which is not specified by the ETSI and was therefore moved into the testbench. It was also necessary to specify a well-defined interface through which the Vocoder can be accessed and integrated into a system.

During refinement of the model, its structural hierarchy was laid out in a way that reflects the structure of the EFR algorithm. A well-defined structure serves two purposes: first, it greatly enhances legibility and comprehensibility for a human designer. The model can be viewed at any level of the hierarchy and parts that are not of interest can be collapsed to reduce complexity. Secondly, a tight structure also helps the design tools produce better results because it enables them to choose the depth of the hierarchy at which they perform their job.

The granularity of the model was decided upon and applied to the code. It specifies the smallest blocks of code which are not divided any further. In the SpecC methodology these blocks become leaf behaviors. The decision on assigning a specific piece of code to a leaf behavior was based on code size, complexity and already present structures such as functions or loops. In general, the goal was to balance size and complexity while not breaking up any existing natural boundaries.

Throughout the code, the SpecC specific data types were introduced where applicable to replace awkward constructions due to limitations of the C language. In particular, SpecC extends the C language with bit and bitvector types. These are not only more effective in terms of speed and storage but also model the behavior of the code in a more precise way.

All data access outside of leaf behaviors was modified so that it goes through well-defined ports. Both the type and the direction of the ports were specified. This makes it immediately clear which parts of the model access which data and identifies inputs and outputs. No hidden global variable accesses are possible.

The parallelism that had been discovered during analysis of the code was applied to the model. This was done by identifying parallel behaviors through the use of special keywords defined in the SpecC language. Special behaviors were added that instantiate the two or more behaviors and run them in parallel.

The conditional execution of blocks of code was modeled in the form of Finite State Machines using the appropriate SpecC keywords. This also includes loops which were executed manually by introducing separate behaviors for initialization and loop condition.

The definitions and initialization functions of buffers and memories were grouped together with the behaviors in which they are needed. This keeps storage more local and minimizes communication between behaviors, which is particularly important for a target architecture with several functional units running in parallel.

A special reset functionality was developed, taking into account that resets can be triggered externally as well as internally through the detection of a homing frame. The reset functions were placed together with the behaviors in which the storage to-be-initialized is defined.

Finally, changes were applied to the code which are due to language differences between ANSI C and SpecC. Specifically, SpecC's type-safety disallows certain constructs that are frequently used in C programs for e.g. pointer manipulation.

As mentioned above, the development of the specification model was done by incrementally converting the simulation code into SpecC behaviors. This scheme takes advantage of the fact that SpecC is based upon the C language. However, a number of differences exist between ANSI C and SpecC, which had a significant impact on the conversion of the code. The C and the SpecC compilers also use different compilation strategies which made it necessary to use a special technique during development of the model.

4.3 Compiler and Language Related Issues

A typical C compiler translates a program by compiling the source files in translation units. A translation unit is a single source file together with possibly included header files. The compiler is run separately on each translation unit, and when it is running it is aware of only that unit. For each translation unit, the compiler creates an object file. Once all the translation units have been compiled, the linker is called which links all the object files and generates the executable. Since each translation unit is compiled separately, special keywords are provided for handling variable access between the units. The static keyword declares a variable that only exists in the current translation unit (internal linkage), whereas the extern keyword allows access to a keyword that is defined in a different unit from the one that is currently being compiled (external linkage).

The SpecC compiler uses a different approach. For a design consisting of several source files, each file is first parsed and an Internal Representation file (.sir) generated. Before compiling the code, all of the .sir files are merged together to form a complete Internal Representation of the design. Only then is the actual compilation started which generates a single object file. This means that there is no external linkage, the extern keyword is not allowed inside of behaviors. It also has a number of side effects, which became apparent during development of the Specification Model.

Since a top-down strategy was chosen for the development of the model, the first function to be converted to SpecC was the `main()` function from the file `coder.c`. The function was copied into the file `testbench.sc` and placed inside a Main behavior. The Main behavior is the equivalent to the C `main()` function in that it is called when the model is executed. This however was not sufficient to generate a valid SpecC source file.

As was discussed in the code analysis section, the simulation code makes extensive use of global arrays as buffers and uses pointers to provide data access to them. The testbench uses such a pointer to read in new speech samples. Since the pointer is defined in a different file it is declared inside the function with external linkage. This is not allowed in SpecC. Eventually as development progressed, the data flow was cleaned up and this pointer was replaced. However, as a temporary fix during development, its declaration was simply moved outside of the behavior. This works without complications because no name conflicts exist in conjunction with the pointer. The same technique was applied in later stages in order to obtain source files that compile without error and enable a step-by-step development of the model.

4.4 The Incremental Development Approach

At this point, the model consisted of a single leaf behavior containing the complete simulation code. In order to compile and simulate the model in its current state, a special technique was used. Its purpose was to avoid having to convert the complete code base before the first compile was possible. Such a conversion would have been necessary if the SpecC compiler had been used for the compilation of all of the files. Instead, only the already converted source files were compiled with the SpecC compiler, while the remaining unconverted C files were compiled with the standard C compiler. The resulting object files of both the SpecC compiler and the C compiler were then linked together to produce the simulation executable. This technique made the incremental development of the model possible without having to apply major changes to the simulation code just to make it compile successfully.

Due to the size of the simulation code and the number of source files, a Makefile was used to automate the compilation of the model. The Makefile can be obtained from the "Download Examples" section of the SpecC web page [19]. It contains a list of object files from the original simulation code which are compiled using the standard C compiler. The SpecC source file is first converted to the SpecC Internal Representation (.sir) and then compiled into an object file. The target testbench then links all of the object files together and generates the executable. As development of the model progressed and further C files were converted, the C object files were removed from the list and the corresponding SpecC targets added to the Makefile. Once the conversion was completed, the model was compiled using only the SpecC compiler. Note that the dependencies of the original C source files were omitted because it was assumed that these files would not be changed during development.

4.5 Development of the `coder` Behavior

The next step in the development of the specification model was the definition of the interface for the actual encoding behavior (Listing 6). First, the data inputs and outputs were specified. The Vocoder takes speech samples as input which are placed in the array `speech_sample`. This array uses the SpecC bit field type with a width of `SAMPLE_WIDTH-1` which holds one speech sample of 13 bits. For every 160 speech samples a block of 244 bits is generated. This block is placed in the bit field serial. The field is declared as unsigned because the 244 bits represent

```
behavior coder (
 in bit[SAMPLE_WIDTH−1:0] speech_sample,
 out unsigned bit[BITS_PER_FRAME−1:0] serial,
 in Flag dtx_mode,
 out DTXctrl txdtx_ctrl,
 in event new_sample,
 out event serialbits_ready)
implements coder_reset;
```

Listing 6: `coder` Behavior Interface Declaration.

the 57 speech parameters and cannot be interpreted as a single value. The following two parameters are flags related to the DTX function of the Vocoder. `dtx_mode` is a switch that enables or disables DTX, the Flag type is a typedef to `bool`. `txdtx_ctrl` holds 6 flags that represent the state of the DTX functions during encoding of the current frame and are output for debugging purposes. Its type DTXctrl is a typedef to an unsigned bit field with a width of 6 bits. Synchronization is achieved through the use of two events. The `new_sample` event is triggered when a new speech sample has been placed into the `speech_sample` parameter. The Vocoder triggers the `serialbits_ready` event once a complete frame has been encoded and the result can be read from the serial parameter.

The use of a synchronization scheme with the two ready events is a more general way of specifying the behavior of the system. Whereas function calls are software-specific, the events can be implemented both in hardware and in software.

A reset capability is provided by the function `Ireset`. It implements an asynchronous reset of the Vocoder which is described later in this chapter.

The encoder function call was then replaced with a call of the `Cod_12k2` behavior. Because `Cod_12k2` is contained in a separate file, it had to be imported using the `import` keyword. This led to a problem which is the result of the different compilation strategies mentioned above. Both the testbench and the encoder source files import C header files which contain the definition of structs. Because the SpecC compiler merges both files, this leads to a double definition of the structs. The solution is also to import the information from the header files, in which case the compiler automatically resolves the multiple definition. However, in the case of the Vocoder these structs were part of a profiling mechanism that is hard-coded in the simulation code. Since profiling functions are not part of a system specification they were removed completely. Should the designer wish to profile the model, this capability is provided by the SpecC compiler without the code having to be modified.

Before work progressed to the actual EFR encoding behavior, the testbench was cleaned up. The goal was to provide an intuitive and easily understandable testbench. More importantly however, it should simulate a general environment independent of the target architecture of the Vocoder itself. This was accomplished by separating the testbench into a stimulus and a monitor behavior and running them in parallel with the encoder behavior. The stimulus behavior reads the data file that contains the speech samples and writes them into the new_speech buffer. After each sample, it triggers the new_sample event. In contrast, the monitor behavior waits for the serialbits_ready event to be triggered, reads the serial buffer and writes the result into a capture file. This basically reduces the testbench to the instantiation and parallel execution of the stimulus, the monitor and the coder behaviors.

After the testbench had been completed, the development of the encoder behavior was started. This corresponds with a move one step further down in the structural hierarchy of the Vocoder, and the first decision that had to be made was to determine which sub-behaviors should be exposed at this level. Analysis of the EFR Algorithm and the simulation code showed that the speech samples are processed before they are encoded, and that the result of the encoding function is a set of 57 parameters that are then converted to a serial bit stream. Therefore it was determined that the encoder behavior should contain three sub-behaviors, pre_processing, coder_12k2, and post_process. Figure 9 shows the structure of the encoder.

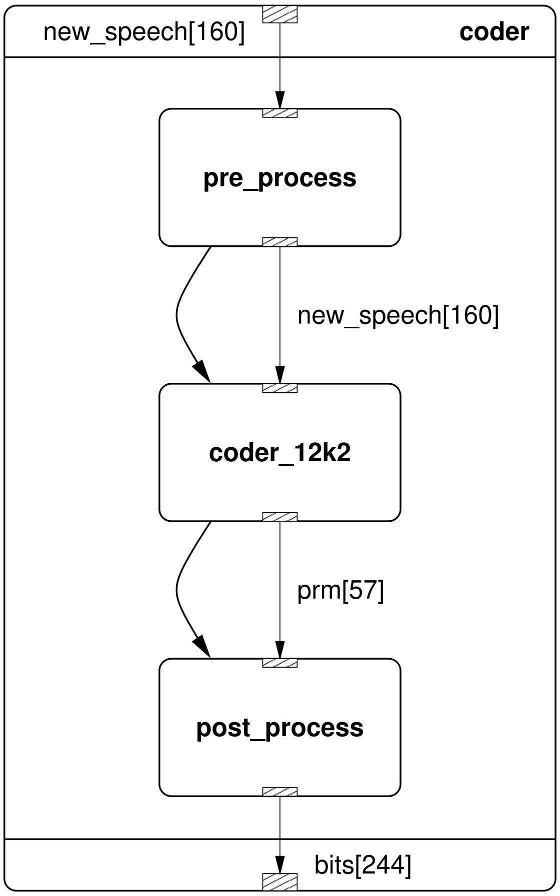

Figure 9: Top-Level Structure of the Encoding Part of the EFR Vocoder.

4.5.1 Pre-Processing

The pre_processing block is executed first and performs two functions on the speech signal: inital down-scaling and high-pass filtering. Down-scaling consists of simply dividing the speech signal by a factor of 2 to reduce the possibility of overflows in the fixed-point implementation. The high-pass filter has a cut-off frequency of 80 Hz and serves as a precaution against undesired low-frequency or DC components. The pre_processing behavior also gathers the 160 samples that form a speech frame and performs the homing frame test. Therefore its interface consists of the speech sample input, the speech frame output, two reset flags, and the new sample event.

coder_12k2 is the main encoding routine which extracts a set of 57 speech-synthesis parameters per speech frame. The interface contains the new speech frame, the encoded parameter block, and the buffer syn which holds a frame of synthesized speech for debugging purposes. Additional flags control the DTX functions of the encoder and allow a reset.

4.5.2 Post-Processing

The function of the `post_process` behavior is to convert and encode the parameter set into a block of 244 bits per frame. The coder behavior passes the speech parameters as 16 bit integer variables which are reduced in their bit width and then serialized. The interface consists of the parameter input, the serial output, DTX flags and reset flag, and the ready event that indicates that a new block of encoded speech is available.

4.5.3 Coder_12k2

As was discussed in Chapter 2, the encoding routine `coder_12k2` consists of three main blocks, the Linear Prediction Analysis, the Open-Loop Pitch Search, and the Closed-Loop Pitch Search. This structure was reflected in the definition of the behaviors, with one added behavior, `shift_signals`, that updates the filter memories for each frame. Figure 10 shows a graphical representation of the encoding behavior. Because of the size and complexity of the EFR Algorithm, not all of its behaviors will be discussed in detail. Instead, the discussion is kept more general, with special attention given to noteworthy details and techniques.

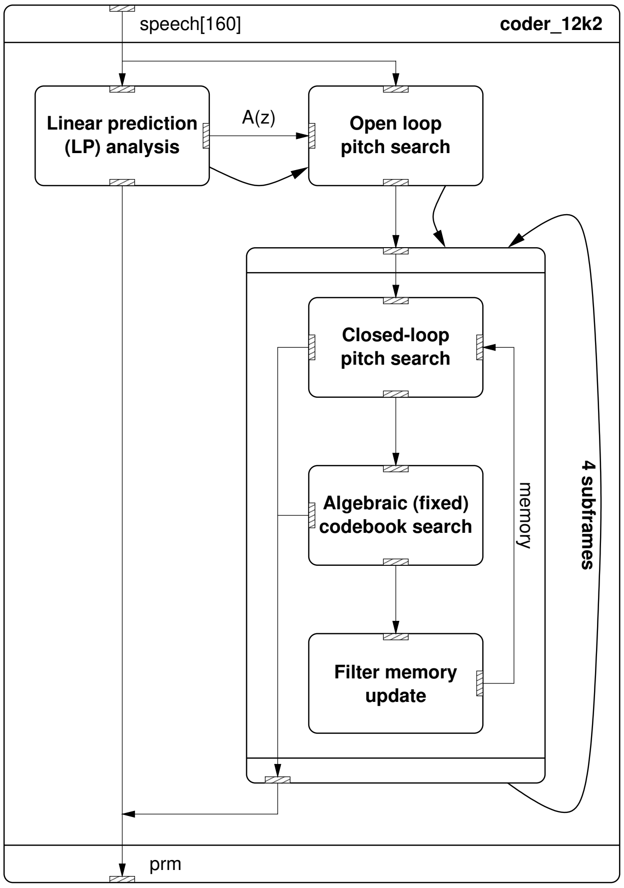

Figure 10: Structure of the Encoding Behavior `coder_12k2`.

4.5.4 Linear Prediction Analysis

The Linear Prediction (LP) Analysis block extracts the five parameters of the LP synthesis filter and is shown in Figure 11. As was discovered during analysis of the simulation model, the LP coefficients of the second sub-frame can be calculated in parallel with these of the fourth subframe. To facilitate this in the specification model a behavior `find_az` was defined which executes the `autocorr`, the `lag_window` and the `levinson` behaviors. Two of these new behaviors are instantiated inside the `lp_analysis` behavior and their ports connected to the speech input, the respective arrays containing the windowing constants, and the computed LP filter coefficients for the second and the fourth sub-frame. The code listing also shows ports for the autocorrelations, a scaling factor for the autocorrelations and the first four reflection coefficients from the Levinson Durbin algorithm. These values are used for the VAD block which makes the voice activity decision. The `old_A[MP1]` array contained in the port serves as a memory for the LP coefficients of the previous frame. They are used as a fall back in case the calculated coefficients for the current frame would lead to an unstable filter. The `old_A` memory is instantiated in the `lp_analysis` behavior and set to zero if a reset occurs.

After the LP filter coefficients have been extracted, they are converted into LSPs by the `Az_lsp` behavior which uses Chebyshev polynomial evaluation. If less than ten roots of the polynom are found, LSPs from the past frame are used. This means that the two `Az_lsp` behaviors cannot be run in parallel since the first behavior works on the LSPs that the second could be using as input data which would result in a conflict and result in an incorrect calculation. So they are run sequentially, first `Az_lsp_1` to compute the LSPs for the second subframe, and `Az_lsp_2` for those of the forth subframe.

The last task of the linear prediction analysis is the quantization of the Line Spectral Pairs and interpolation of the quantized and the unquantized

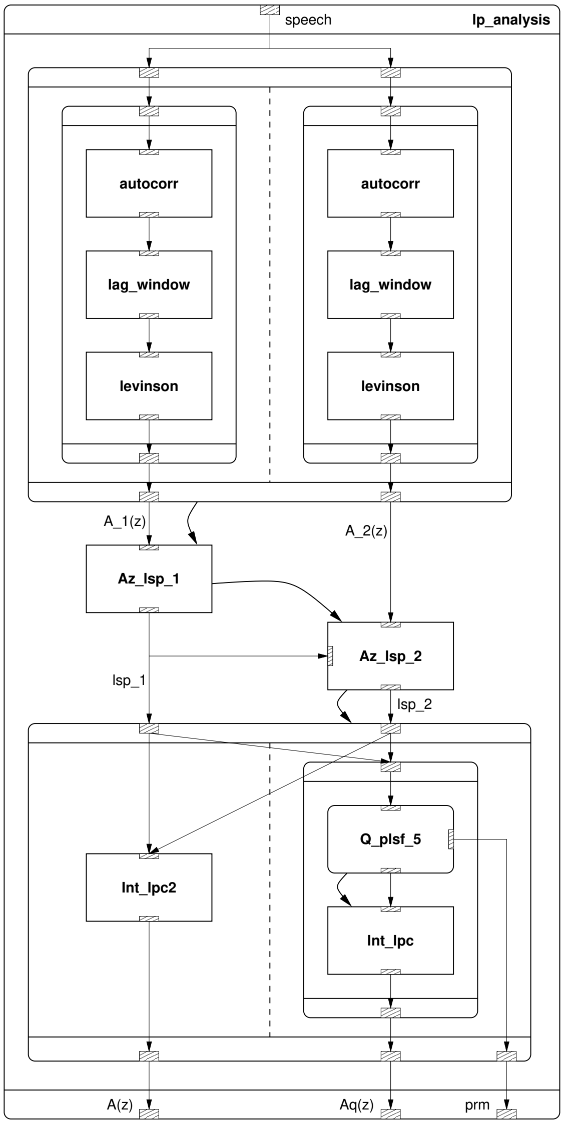

Figure 11: Linear Prediction Analysis.

LSPs. `q_plsf_5` quantizes the LSPs of both subframes and generates the five parameters (`prm`) that are transmitted over the wireless link. Interpolation is used to obtain the LP coefficients and the LSPs of the remaining two subframes. `Int_lpc` does this for the quantized LSPs and also converts them back to LP coefficients. `Int_lpc2` is identical to `Int_lpc` except that it works on the unquantized LSPs. It also omits the back conversion since the unquantized LP coefficients are already available.

Again parallelism is available since interpolation of the unquantized LSPs can be performed at the same time as the LSPs are quantized and these then interpolated. `q_plsf_5` and `Int_lpc` were put into the behavior `q_plsf_and_int_lpc` which is run in parallel with `Int_lpc2`.

The behaviors `q_plsf_and_int_lpc` are executed conditionally. If voice activity detection has determined that speech is not present the interpolation of the quantized LSPs is skipped since no further speech analysis takes place. This is realized by use of a small finite state machine (FSM). Listing 7 shows the main function of behavior `q_plsf_and_int_lpc`. Quantization of the LSPs is always executed in the first state. If discontinuous transmission (DTX) for the encoder is activated in the control word and the speech flag indicates that no speech is present, the fsm statement is exited. Otherwise, the default path is taken which is to execute the next state, the interpolation of the quantized LSPs.

```
void main(void)
 {
  fsm{
       q_plsf_5_exec : {
         if (( txdtx_ctrl & TX_SP_FLAG) == 0)
            break;
         /* goto int_lpc_exec .main */
       }

       int_lpc_exec : {
         /* goto update_lsps_exec */
       }

     } /* msf */
 }
```

Listing 7: `Main()` Function of the `q_plsf_and_int_lpc` Behavior.

4.5.5 Open-Loop Pitch Analysis

Open-Loop pitch analysis determines delay/lag estimates for the closed loop calculation of the pitch filter parameters in order to narrow down the actual adaptive codebook search.

As can be seen in Figure 12, the weighted speech calculation is performed in a loop over the four subframes of 5 ms each. In the original C simulation model this was done by means of a for-loop. Since a for-loop that executes subbehaviors is unclean in the SpecC methodology, a different way of modeling the loop had to be found. The solution is, once again, a finite state machine which emulates the behavior of the for-loop in C syntax.

The general form of a for-loop in C is

```
for ( initialize ; test ; execute after loop)
   {   ....;
   }
```

The initialize statement is carried out once only, at the start of the very first time that the loop is entered. The test is carried out before each execution of the body of the loop, including a test before the very first execution of the loop. The first

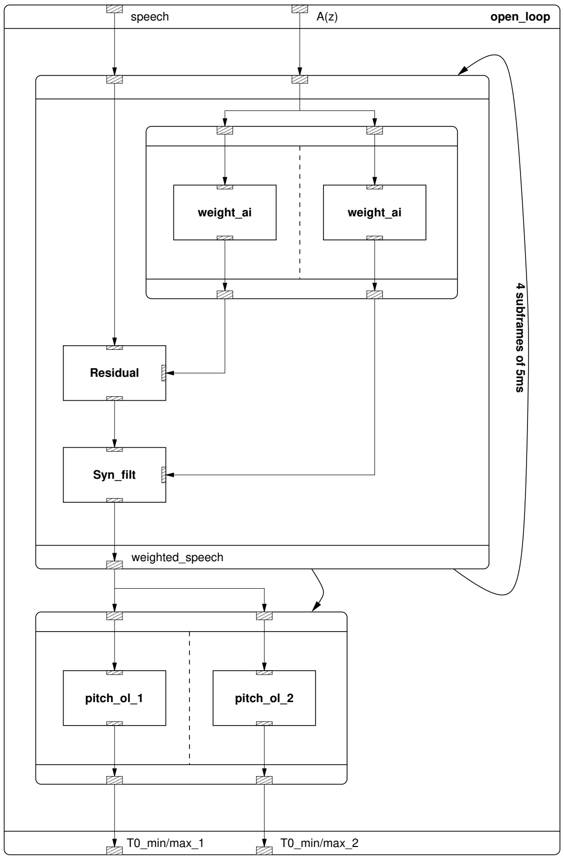

Figure 12: Open-Loop Pitch Search.

time will be immediately after the initialization, and hence there will perhaps be no executions of the loop body if the test fails at this stage. The third expression is a statement executed after every execution of the loop body, before the next test. It typically increments a counter.

For a clean FSM representation of the for-loop, the initialization code and the end of loop code had to be placed in separate sub-behaviors. The test code is used for the conditional state transition to the beginning of the loop. The FSM for the calculation of the weighted speech in the

```
fsm {
   for_init : {
     if (i<=3) goto for_body1;
     goto ol_lag_estimate;
   }
   for_body1:
   for_body2:
   residual :
   syn_filter :
   for_end : {
     if (i<=3) goto for_body1;
     goto ol_lag_estimate;
   }
   ol_lag_estimate :
}
```

Listing 8: FSM for Weighted Speech Calculation.

open_loop block of the vocoder is shown in Listing 8.

The `for_init` behavior contains the loop initialization, in this case simply setting the index variable to zero. As per definition, the loop condition is checked before the first execution of the loop. If it is true, the first iteration is started with the execution of `for_body1` which sets up the pointers for the first speech sub-frame. `for_body2` contains the parallel calculation of the weighted filter coefficients, followed by `residual` and `syn_filter` which perform the perceptual weighting of the speech signal. This completes the first iteration of the loop. The index is incremented in `for_end`, after which the condition test determines whether the loop is executed again or the for-loop is exited. In the latter case, the speech weighting is complete and the open-loop pitch delay estimation is performed by the behavior `ol_lag_estimate`.

4.5.6 Closed-Loop Pitch Analysis

The behavior `subframes` contains the code that calculates the remaining speech parameters. Inside a loop over all four subframes the behaviors `closed_loop`, `codebook_cn` and `update` are called. The closed-loop behavior calculates the pitch lag and the gain of the adaptive codebook signal, whereas the codebook behavior yields the fixed codebook address and the gain of the codebook contribution.

The closed-loop behavior is complex as can be seen from Figure 13 and contains numerous conditionally executed behaviors. It is therefore implemented as an FSM. The basic functionality is as follows: First, the remaining speech signal is again filtered through two weighting filters which are run in parallel. Then, the impulse response of the synthesis filter along with the target vector for the codebook search are calculated in parallel fashion. The remaining behaviors calculate and encode the pitch lag and the gain of the adaptive codebook signal. However, if the voice activity detection has detected silence, none of the behaviors are executed. Instead, the gain variable is set to zero and control is returned to the parent behavior.

4.5.7 Fixed Codebook Search

The codebook search behavior as shown in Figure 14 calculates the remaining contribution to

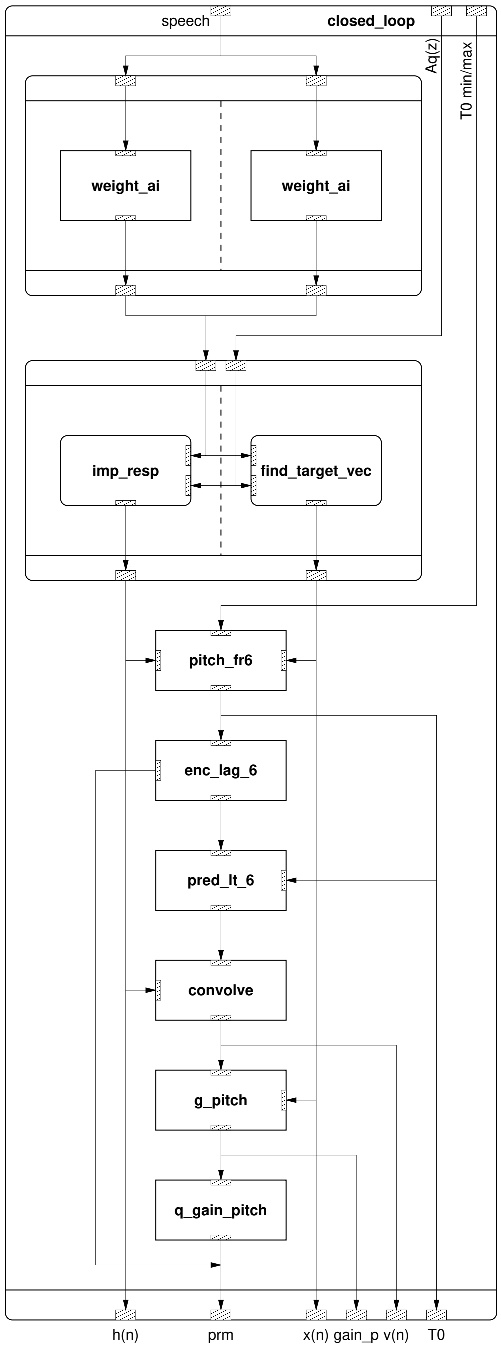

Figure 13: Closed-Loop Pitch Analysis.

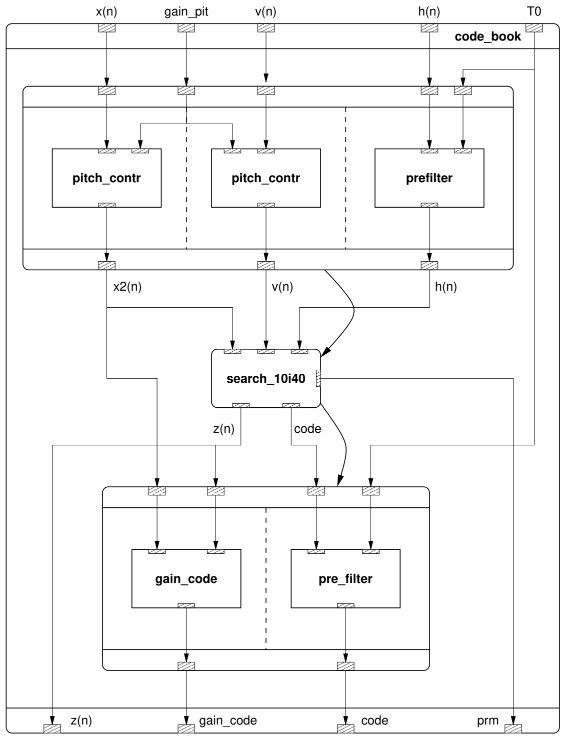

Figure 14: Fixed (Algebraic) Codebook Search.

the LP synthesis filter. Then a codebook is searched for the code that reproduces this contribution. The address to this code is encoded and transmitted to the receiver, along with the gain of the codebook signal. In the case of silence having been detected, comfort noise is generated and transmitted in place of the code. Again, the conditional execution is implemented as an FSM. The `seq1` behavior has to be left empty because the conditions are evaluated after the FSM node has been executed.

4.5.8 Filter Memory Update and Post-Processing

The last step in the loop over all four subframes is the update of the filter memories of the synthesis and weighting filters. Its structure is illustrated in Figure 15. The excitation signal $u(n)$ is calculated as the sum of the adaptive codebook vector $v(n)$ and the filtered algebraic codebook vector $c_E(n)$. By filtering the excitation signal $u(n)$ through the LP synthesis filter, a local copy of the speech is produced as if it were synthesized at the decoding size. This speech signal is used to update the memories of the synthesis and weighting filters. In parallel to the synthesis and update, the previously computed gain of the fixed codebook vector is quantized for transmission.

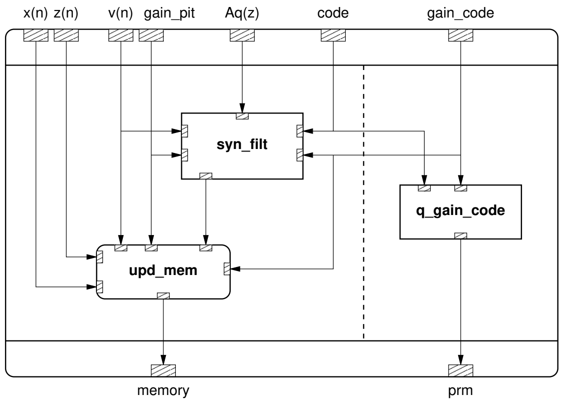

Figure 15: Filter Memory Update.

At this point in the EFR algorithm, the speech frame has been encoded into the block of 57 parameters. In the code, however, they still occupy an array of 57 `Word16` variables. The last step is the encoding of the parameters into a block of 244 bits. This is done in the `post_process` behavior, along with routines that handle DTX-related functions. The `prm2bits_12k2` behavior contains the actual encoding functionality. It takes the parameters as an input, truncates each one to the bitwidth specified by the ETSI standard, and concatenates them to form the output of 244 bits. If no speech was detected, the comfort noise frame is encoded instead. Finally, the states of the DTX functions are added to the output for debugging purposes.

4.6 The Reset Functionality

A reset function is crucial for any embedded system, regardless of implementation in hardware or in software. Its purpose is to bring the system into a defined state and is typically called after power-up. However, it can be necessary to reset a system at any time, for example to restart it or if its state has become corrupted for some reason. Normally, one of the two possible reset methods is used: either synchronous or asynchronous reset. An asynchronous reset will bring the system into its defined state immediately and unconditionally. In contrast, a synchronous reset is tied to a clock signal. It has effect only in conjunction with a clock event, usually a rising or a falling edge of the clock signal.

In the case of the Vocoder, two sources for resets exist. First, an external reset is needed to bring it into a defined state. Second, the EFR standard defines a inband reset function. This provides means of resetting the EFR Vocoder remotely by embedding a special frame in the input data. Upon detection of this frame, the algorithm resets itself. While the external reset is asynchronous, the inband reset function works synchronously.

The SpecC language does not offer specific constructs or keywords for the implementation of a reset function. However, single bit or bool types can be used to implement a synchronous reset. These would be checked at a certain time and, depending on their state, a reset would be executed. An asynchronous reset could be implemented using the events mechanism that SpecC offers. Such an implementation would make it necessary to introduce a reset behavior that would have to run in parallel with the behavior that it is supposed to reset in order to implement a truly asynchronous reset. Besides being time-consuming to develop, the added behavior and ports obscure the true functionality of the code and make it less readable and understandable. Therefore, a different approach was chosen for the Vocoder model.

Every SpecC behavior contains a `main()` method which executes the primary function of the behavior. It is automatically made available for calling from the outside. Additional methods can be made available by implementing interfaces that call the method. While this method is normally used for the implementation of channels it can also be used to model the reset functionality. For the Vocoder, an interface `Ireset` was defined that simply calls the `reset()` function. By adding `implements Ireset` to the behavior declaration of behaviors that need to be resettable, the reset function becomes accessible from the outside.

The reset is exposed up to the highest level of the hierarchy, the coder behavior itself. Upon call it is propagated down to each leaf behavior that implements the Ireset interface, thereby returning the Vocoder to its initial state. Note that this method of implementing a reset function does not abort the processing of a speech frame. Therefore care has to be taken that initialization of the Vocoder does not interfere with the processing, i.e. a reset should only occur between frames.

Apart from external resets through the calling of the reset method, the Vocoder model also implements an in-band reset function as specified by the ETSI standard. The homing frame is detected in the behavior `encoder_homingframe_test` which is a sub-behavior of `pre_process`. It will set a flag if it finds a frame in which all samples have a value of exactly one. If the `pre_process` finds that this flag is set, it sets a `reset_flag` for the other behaviors and then resets itself. While this is not a classical synchronous reset in that it is synchronized with a clock, it is nevertheless synchronous with the input data.

4.7 IP Distribution Model of the Vocoder

The distribution and re-use of components is a crucial part of any strategy to overcome the ever-increasing productivity gap. These components are the intellectual property of the designer or the originating company. Therefore there is a strong desire to protect these IP components from misuse once they are distributed among third parties. Misuse includes making unlicensed copies, reverse engineering, modification etc. At the same time, the customer has to be provided with means to properly evaluate the IP in the target system, even without having access to the source code. This is best accomplished by two separate models. The simulation model allows simulation of the functionality of the IP but does not give away any implementation details. In contrast, the implementation model is the actual implementation of the component for a specific target architecture. It is the valuable part of the IP. Therefore it might not even be given to the customer but rather forwarded directly to the foundry, for example, that manufactures the chip for the customer.

The SpecC system supports this method of distributing IP components through a special IP mode of the SpecC compiler. Using this mode, the designer can compile the simulation model of a component into an object file. Together with the declaration of the behavior, this enables the customer to simulate the component without having access to the source code. The complete interface definition which is part of the behavior declaration makes it possible to replace the simulation model with the implementation model at any time and without modification of the surrounding code.

In the case of the Vocoder, the distributable part is the encoding behavior `coder`. Its declaration has to be made available to the customer. Listing 9 shows this declaration.

The declaration of the coder behavior is identical to the non IP version, as is the re-

```
#include "EFR_Coder_public.sh"

interface coder_reset
{
  void reset(void);
};

behavior coder (
 in bit[SAMPLE_WIDTH−1:0] speech_sample,
 out unsigned bit[BITS_PER_FRAME−1:0] serial,
 in Flag dtx_mode,
 out DTXctrl txdtx_ctrl,
 in event new_sample,
 out event serialbits_ready)
implements coder_reset;


note coder.comment =
  "GSM Enhanced Full Rate speech encoder (
     GSM 06.60)";

note coder.functionality =
  "The EFR Coder maps 160 speech samples in
      13-bit uniform PCM format to encoded
     blocks of 244 bits according to the
     GSM technical specification GSM
      06.60, version 5.1.1.";

note coder.scc_CxxReservedSize = 35876u;
```

Listing 9: Public Interface to the EFR Vocoder Simulation Library.

set interface. The SpecC language provides persistent comments which are used to annotate the functionality of the IP. The note `coder.scc_CxxReservedSize` is a special comment that is used by the SpecC compiler to reserve sufficient memory for the IP library. Its value is calculated by the SpecC compiler during compilation of the behavior in IP mode. The file `EFR_Coder_public.sh` is a SpecC header file that contains the Vocoder specific type definitions.

The IP library file is generated by compiling the `coder` behavior with the IP mode flag set. Instead of an executable program, the compiler generates an object file. This object file is then stripped of its symbols using the tool `strip` and converted to a library with the help of `ar` and `ran`. The resulting file `libEFR_Coder_sim.a` is the IP library of the EFR Vocoder. It can be safely distributed to the customer who can simulate his or her system without having access to the source code.

To incorporate an IP library and run a simulation that includes the IP component, the designer simply imports the behavior declaration which was provided by the IP provider and connects it to the surrounding system. To compile the design the compiler has to be informed about the IP library. This is accomplished with the compiler switch `-lEFR_Coder_sim` which causes the linker to add the Vocoder IP library file. The generated executable then allows simulation of the complete design, including the Vocoder IP component.

5 Conclusion

In this paper the development of a high-level specification model of the Enhanced Full Rate Vocoder is presented. The simulation code for the Vocoder was used as the basis of the project. After an understanding of the underlying algorithm had been gained, the simulation code was analyzed in terms of structure and data flow. Finally, the specification model itself was developed. Simulation runs during and after development were used to verify correct behavior of the model.

5.1 Comments on the SpecC Language

The choice of the SpecC language proved to be very beneficial for the project. Being based on ANSI-C, it allowed parts of the Vocoder simulation code to be incorporated into the specification model without the need for time-consuming and error-prone rewrites. It also enabled an incremental development approach in which parts of the simulation code could be used as a testbench for the part of the specification model that was being developed. The executable nature of the model allows the designer to simulate the system at any stage of the development process in order to verify correct behavior. The rich set of features and constructs offered by SpecC made the development of a precise and efficient specification model possible. However, in some instances, workarounds had to be found in areas where the language was felt to be deficient or awkward.

The EFR Vocoder specification model was the first industrial size project that was realized using the SpecC language. It proved that SpecC is suitable for modeling and simulating complex embedded systems. During the development of the Vocoder specification model, a number of SpecC features were employed to yield a precise and versatile model of the system. At some points, however, some additional features would have been desirable, or problems arose which had to be worked around. In the following, the features of the SpecC language are discussed that were most important and beneficial to the development of the Vocoder specification model. This is followed by features and functions that would have allowed some problems to be solved in a more precise or efficient way.

For the development of the Vocoder model, the constructs for structural hierarchy and well-defined interface ports were the most important features that contributed to a good model. Due to the large size and complexity of the EFR Vocoder, being able to specify a structural hierarchy is essential to both human designers and EDA tools. It enables the human designer to inspect and analyze the model at any level in the hierarchy while hiding unnecessary details that would only be confusing. The benefit for the tools is that they too can be applied at a certain level in the hierarchy. This is important because a number of the computational problems encountered by EDA tools rise exponentially in complexity. The structural hierarchy therefore also helps to limit the computation time of automated design tasks.

Each behavior of the Vocoder model has a list of ports through which it communicates with other behaviors. This allows a very clear and precise description of the data flow in the system. It also allows the separation between computation and communication which is an essential prerequisite for the re-use of components in Embedded Systems. For the Vocoder specification model, the high complexity makes a clearly specified data flow particularly important. Its algorithm makes extensive use of buffers, and data flows not only between neighbor behaviors but also to other behaviors or directly to the output buffer. Analysis of the simulation code showed that it relies heavily on global variables and pointers while also being restricted by the C language syntax. Data flow is very hard to follow. In contrast, all data in the Vocoder model flows through the ports of the behaviors, making the connections immediately clear and easy to understand.

The SpecC language also introduces a number of new data types which are used extensively throughout the Vocoder model. SpecC supports all the ANSI-C types and adds the `bool`, the `bitvector`, the `event` and the `time` types. The purpose of the EFR Vocoder is to compress speech samples. A great amount of the data that is handled by the algorithm has bit widths that are different from standards such as integers. The possibility to define variables as bitvectors with an arbitrary number of bits allows a much more natural and effective modeling of the algorithm. The bitvector type also allows bit slicing and concatenating. Both of these operations are very useful for a compression algorithm.

While a lot of the SpecC features proved to work well, the development of the Vocoder model revealed a few areas in which the language has deficiencies. The most prominent one was the issue of array slicing. The Vocoder relies heavily on buffers for input and output data. During processing of a speech frame, behaviors frequently access only a certain range or slice of the buffer. In the simulation code, this is done by defining pointers and moving these pointers across the appropriate slice of the buffers. However, this is a technique that should be avoided for specifying embedded systems because pointers may cause problems for synthesis algorithms. The best solution would be if an array slicing feature were available, analogous of the bit slicing feature of the bitvector type. A slice of a buffer could then be accessed by specifying `buffer[upper_boundary:lower_boundary]`. The complementary operation would be the concantenation of arrays, again analogous to the concatenation of bitvectors.

The absence of array slicing leaves only three alternatives. First, the whole buffer could be passed every time some of its data is needed. Doing this could lead to possibly large amounts of unnecessary communication, for example when the whole buffer has to be transferred to a specialized hardware unit. Alternatively, the part of the buffer that is needed could be copied into a second buffer. The second buffer would then be passed to the behavior, modified and then copied back into the original buffer. This technique would make additional code and operations for the copying necessary. The code becomes less understandable to the human designer, and the additional operations are potentially redundant if, for example, the respective parts of the model are implemented in software on a single processor. The third alternative is to leave the pointers in place if they cannot be avoided. As mentioned before, this may cause problems for synthesis, although approaches for the synthesis of pointers exist and are a current research topic [18]. If the part of the system that contains the pointers is to be implemented in software, their use has no significant negative consequences. Since this last alternative imposes the fewest disadvantages, it was chosen in the case of the Vocoder specification model.

Another feature that would have been helpful is the generate statement as offered by VHDL for example. The `post_processing` behavior of the Vocoder specification model reduces the bitwidth of the 57 speech parameters and concatenates them to a single frame of 244 bits. With the help of a generate statement and a table with the bitwidths, this task could be accomplished in one or two lines of code. As it stands, it had to be done manually for each parameter, resulting in 57 lines of code.

Further difficulties during the development of the Vocoder model were due to the fact that certain tools are not yet available, namely a SpecC native debugger. While it is possible to debug a SpecC simulation executable with standard tools such as `gdb`, this does pose problems. The extensions that SpecC makes to ANSI-C are themselves implemented in C++. Even if the SpecC source files are available, substantial knowledge of the inner working of the SpecC system is necessary to debug a bitvector, for example. In the abscence of the source files this becomes virtually impossible. A native debugger would also offer better support for the new concepts such as events and the notion of time.

5.2 Comments on the Specification Model

The specification model of the EFR Vocoder satisfies all the requirements for a high-level abstract specification of an Embedded System. It captures the functionality of the Vocoder in a formal and unambiguous way, and its clearly defined structure makes it easy to understand for both human designers and automated CAD tools. While the model contains as few implementation details as possible, some limitations in this respect were imposed by the way the EFR Vocoder algorithm is specified by the ETSI. In order to be type-approved, an implementation of the Vocoder has to produce a bit sequence identical to that of the simulation code, for a given set of input data. Therefore it was necessary to introduce some implementation detail contained in the simulation code into the specification model, most prominently the use of 16-bit fixed-point arithmetic. Ideally the specification model would have used floating-point arithmetic, deferring the decision on the ALU architecture to later stages of the exploration process.

The specification model is the basis for an implementation of the EFR Vocoder. The next step in the design process would be to perform architectural exploration in order to find the most suitable system architecture that satisfies all the constraints. Architectural exploration is an iterative task that is influenced by the designer's experience and aided by tools such as profilers and retargetable compilers. The decision on connecting hardware and protocols is made during communication synthesis, which again is an iterative process. All the remaining work is left to the backend tools such as logic synthesis tools and compilers.

The development of the Vocoder specification model proved to be an interesting and challenging project. Its sheer size and complexity of the EFR algorithm made the application of a structured and well planned development approach mandatory and clearly demonstrated the advantages of the SpecC language for modelling complex Embedded Systems. However, it also showed that real-world projects always require compromises and workarounds, which make them even more valuable as test-cases for new languages and methodologies.

6 References

[1] D. Gajski, J. Zhu, R. Dömer, *The SpecC+ Language*, University of California, Irvine, Technical Report ICS-TR-97-15, April 15, 1997.

[2] J. Zhu, R. Dömer, D. Gajski, *Syntax and Semantics of the SpecC+ Language*, University of California, Irvine, Technical Report ICS-TR-97-16, April 1997.

[3] D. Gajski, J. Zhu, R. Dömer, *Essential Issues in Codesign*, University of California, Irvine, Technical Report ICS-TR-97-26, June 1997.

[4] J. Zhu, R. Dömer, D. Gajski, "Syntax and Semantics of the SpecC Language," *Proceedings of the Synthesis and System Integration of Mixed Technologies 1997*, Osaka, Japan, December 1997.

[5] D. Gajski, G. Aggarwal, E.-S. Chang, R. Dömer, T. Ishii, J. Kleinsmith, J. Zhu, *Methodology for Design of Embedded Systems*, University of California, Irvine, Technical Report ICS-TR-98-07, March 1998.

[6] R. Dömer, J. Zhu, D. Gajski, *The SpecC Language Reference Manual*, University of California, Irvine, Technical Report ICS-TR-98-13, March 1998.

[7] D. Gajski, N. Dutt, C.H. Wu, Y.L. Lin, *High-Level Synthesis: Introduction to Chip and System Design*, Kluwer Academic Publishers, Boston, Massachusetts, 1991

[8] D. Gajski, F. Vahid, S. Narayan, J. Gong, *Specification and Design of Embedded Systems*, Prentice Hall, Englewood Cliffs, New Jersey, 1994

[9] European Telecommunication Standards Institute (ETSI), *Digital cellular telecommunications system; Enhanced Full Rate (EFR) speech transcoding (GSM 06.60)*, Final Draft, November 1996.

[10] European Telecommunication Standards Institute (ETSI), *12200 bits/s Speech Codec for Enhanced Full Rate Speech Traffic Channels; Bit-Exact C Simulation Code (Version 5.1.0)*, June 26, 1996.

[11] K. Järvinen *et. al.*, "GSM Enhanced Full Rate Speech Codec," *Proceedings ICASSP 1997*, pp. 771-774, 1997.

[12] Telecommunications Industry Association (TIA), TR-46, *PCS1900 Enhanced Full Rate Codec US1 (SP-3612)*, Ballot Version, August 1995.

[13] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *The SpecC Methodology*, UC Irvine, Technical Report ICS-TR-99-56, December 1999.

[14] Rainer Dömer and Daniel D. Gajski, *Reuse and Protection of Intellectual Property in the SpecC System*, Proceedings of Asia and South Pacific Design Automation Conference, Yokohama, Japan, January, 2000.

[15] D. Gajski, J. Zhu, R. Dömer, A. Gerstlauer, S. Zhao, *SpecC: Specification Language and Methodology*, Kluwer Academic Publishers, Boston, MA, ISBN 0-7923-7822-9, March 2000, 336 pages.

[16] Synopsys, Inc., *SystemC Reference Manual Release 0.9*, 1999.

[17] CynApps, Inc., *Cynlib: A C++ Library for Hardware Description Reference Manual*, Santa Clara, CA, USA, 1999.

[18] Luc Séméria, Giovanni De Micheli, "Synthesis of Pointers in C: Application of Pointer Analysis to the Behavioral Synthesis from C", *Proceedings of the International Conference on Computer-Aided Design (ICCAD 1998)*, pp.321-326, San Jose, November 98.

[19] The SpecC web page at the University of California at Irvine,
http://www.ics.uci.edu/~specc/.