Virtual Radio Engine: A Programming Concept for Separation of Application Specifications and Hardware Architectures

Riyadh Hossain[1], Matthias Wesseling[1], Claudia Leopold[2]
[1]Siemens AG, Com MP CTO TI2, Frankenstrasse 2, 46393 Bocholt, Germany
[2]Faculty of Electrical Engineering and Computer Science,
[2]Wilhelmshöher Allee 73, 34121 Kassel, Germany
riyadh.hossain@bch.siemens.de, matthias.wesseling@siemens.com, leopold@uni-kassel.de

***Abstract*— The mobile communication market is confronted with an increasing number of communication standards and a corresponding complexity for the mobile terminal applications. To cope with this complexity, the Software Defined Radio approach gains more and more attractiveness. The upcoming hardware platforms supporting a manifold of communication standards all have to compromise between the degree of flexibility and the maximal power consumption. One important architecture feature for these platforms is the parallelization of processing power in different kinds. But this also leads to additional programming restrictions, which requires even more development efforts and specific knowledge about the hardware architecture. To reduce these complexities and to speed up the development process, we suggest to separate the software development into at least two different steps, such that developers only need experience in either the hardware or the software area. In one development step, the application is specified in a hardware-independent way, and in a separate step, the hardware specific implementation can be done without knowledge about the application. To allow an automatism and an optimization in the implementation step, the application has to be specified in a way that the required processing is described in detail, but which still keeps the potential for hardware-specific optimization steps. Within the SDR project at Siemens Communication, we developed a new programming concept called Virtual Radio Engine (VRE) that supports this concept by defining a language and providing compilation techniques.**

***Index Terms*— Code generation, modeling language, system level abstraction, software defined radio.**

I. INTRODUCTION

Software Defined Radio (SDR) is a recent approach for building wireless devices, which is characterized by the use of software modules to control radio functionalities such as modulation/demodulation, signal generation, coding, link layer generation etc. These software modules run on a generic hardware, i.e., SDR builds on the fact that the underlying hardware modules for digital radio systems are considered as programmable [13]. As compared to the traditional way of building wireless devices, which has been to completely implement radio functionalities in hardware, SDR has the advantage of improved re-configurability, i.e., devices can be easily adapted to new communication protocols by just replacing the software. Some of the key requirements for a SDR hardware platform are high processing speed, high speed internal data flow channels, and inter- as well as intra-processor communication paths with the flexibility to permit different data flow topologies. The required processing power can not be obtained through increasing the internal clock frequency due to the low power consumption budget. Instead, it requires multiprocessor architectures, which increase the complexities, as compared to traditional analog and fixed function digital radio platforms, not only on the hardware side, but also in the application development procedure. Several hardware-specific requirements such as mapping and scheduling have to be considered. The complexity can be reduced substantially if the application is described independent from the hardware specifics, and the implementation is done separately. For that, both the hardware and the application need to be described at an abstract level, independent from each other.

Normally, in operating systems, hardware is abstracted through application programming interfaces (APIs). A good API makes it easier to develop programs, as it provides the building blocks from which a program can be composed. Consequently, the API can be thought of as a hardware abstraction layer that hides hardware specifics, and, thus, provides a consistent platform to develop and run applications on. This concept is relatively easy to implement in a PC environment, as the hardware components are well-defined, and, thus, the hardware abstraction layer can be easily described. SDR, in contrast, requires parallel hardware architectures of very different kinds to achieve the required high performance within the low power consumption budget. For example, new architectures like Infineon's "Programmable Baseband Platform for SDR" [14] or Phillips EVP [4], which claim to support SDR, require parallel implementation of arithmetic and signal processing units to achieve their high performance. This hardware can not be described with the *von*

Neuman model, and it is almost impossible to find a single model that describes the variety of parallel hardware platforms. Fortunately, an abstraction can be made in terms of primitives, which are software modules for which the hardware designers provide system-specific implementations. The primitives can be made available to the application developers as a library; however, the implementation is not straight forward like APIs. Implementation of primitives in a specific hardware platform requires one additional step, where hardware specific information (e.g., execution time) of the corresponding primitive is incorporated within the description as well as verified, and thus, the implementation specific optimization can be done depending on the real time requirements.

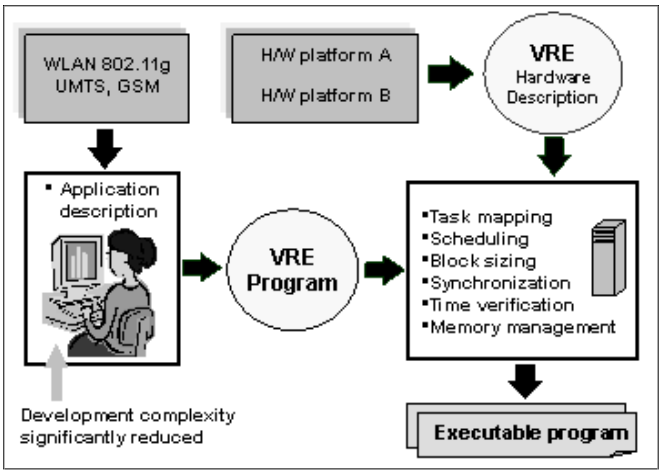

Fig. 1. Application development approach with VRE. Here, the application specification is developed independent from the hardware platform, and the implementation is done separately.

The concept that we suggest in this paper is to separate the description or program of an application from the implementation on a specific hardware. We realize the concept by a tool chain that we call Virtual Radio Engine (VRE). The tool chain is depicted in figure 1. VRE splits the overall design process into two different steps. First, the application is described independent from the hardware details, and then, the implementation is done separately. The platform-independent description of the application (which is typically a communication protocol) is essential for a successful and quick design process [13]. In the VRE tool chain, the hardware-specific information is only needed during the implementation phase. From the application description and hardware-specific information, the VRE compiler generates an executable program for different SDR hardware platforms. Thus, the application is not described in an implementation-specific way; rather it only describes implementation restrictions (in particular restrictions of the block execution order). The compiler is responsible for mapping sub-computations of the application to hardware modules, and it chooses the schedule, i.e. determines the order in which the sub-components are run, including the decision whether they run sequentially or in parallel. As the compiler only has to respect restrictions, but is not bound to a fixed schedule, it has potential for optimizations.

This paper is organized as follows. In section II, we discuss several development environments for signal processing applications and what is still missing there. Then, in section III, we describe why parallelism has to be considered during application development and how this increases the complexity. The concept of the VRE tool chain is presented in section IV, and the VRE language is introduced in Section V. A WLAN experiment using the VRE tool chain is discussed in section VI. Finally, conclusions are drawn in section VII.

II. Environments for The Development of Signal Processing Applications

There are already some products available that generate platform-dependent implementations starting from a higher level description and claim to provide an efficient environment for SDR application development. Some of these tools provide code generation and compilation techniques for a successful and quick development process. The question is, why we need a new approach like VRE, even though there are already tools available for SDR application development. Briefly stated, the reason is that most of the tool chains available in the market do not provide an application description concept that allows an individual description to be realized on different hardware platforms. Moreover, many automatic implementation techniques for parallel hardware platforms are still in their infancy [1]. One example is the tool chain of MathWorks with Simulink/Real Time Workshop/target language compiler. Here, the application is graphically described in Simulink, and then the Real Time Workshop together with the target language compiler generates C code. The generation process is controlled by platform-specific description files [11]. The tool chain works fine as long as there is a simple DSP-like structure, but application development for parallel hardware platforms is not possible. Furthermore, Simulink requires the schedule to be fixed, which restricts portability.

Pieter van der Wolf et al. [8] describe an interface-centric technique for multiprocessor-based embedded software development and implementation. Their goal is to automate the rewriting of source code according to guidelines provided by the user, but they gave less consideration to algorithmic transformations and the efficiency of the source code. Examples of other development environments that are already available and support both SDR and multi-processing include Gedae [10], *Ptolemy* [9], *Waveform Description Language (WDL)* [3], and *MLDesigner* [12]. *Gedae* uses its Primitive and Graph languages to describe an application, and then transforms the description into an efficient implementation on a virtual machine [7]. Their concept of virtual machine does not support the high throughput and good latency that are required for high-performance SDR. *Ptolemy* is an environment for simulation and prototyping of heterogeneous systems [2, 6]. It includes a preliminary code generation framework for heterogeneous multiprocessor platforms. The industrial successor of *Ptolemy* is *MLDesigner*, an integrated platform for modelling and analyzing architecture, function and performance of high level system designs. The problem with *MLDesigner* is that the scheduling of the application

program is part of the description. As the scheduling is hardware-specific, the description can only be realized on one particular hardware platform.

III. Application Development for Parallel Platform Architectures

Parallel platforms pose additional requirements to the implementation of applications. In particular, sub-computations (tasks) must be mapped to parallel units, tasks must be scheduled, and communication and synchronization have to be organized as well. In the implementation, real time constraints, run time information and other parameters need to be taken into account. Some types of parallelisation do need to be considered by the programmer, because they are already handled by low-level tools such as super-scalar devices or, partly, vector units. These types of parallelisation are 'local' and do not influence the overall program structure.

To be able to generate an efficient mapping, we not only need to know which tasks can be run on which parallel units, but moreover the algorithms have to be described in a way that supports parallelisation. A usual C program can not, in general, be separated into independent parts, if this separation is not already considered within the program. On the other hand, the parallelisation should not be described in detail, because an efficient parallelisation depends on the hardware platform, which we want to exclude from the application program. In the VRE language, data dependencies are described, but the scheduling itself is left open, as will be explained in section V.

IV. Development Flow of VRE

VRE's goal is to provide a quick and complete development environment for signal processing applications, especially for SDR, which separates the application description from the hardware-specific implementation details. The VRE approach is divided into two main parts: the VRE programming language and the VRE code generation system. The VRE programming language enables developers to describe the applications in a platform-independent way. The VRE code generator system reads the VRE program and hardware description, and then performs the platform-specific implementation. The responsibilities of the code generator include the efficient mapping of the application algorithms, scheduling, and the insertion of synchronizations.

The development process of VRE is outlined in figure 2. The VRE program only describes the structure of the application and the dependencies that are constraints for the scheduling algorithm. The signal processing algorithms themselves (like FFT, FIR, Viterbi) are neither a part of the VRE program nor of an automatically generated C code of the VRE compiling process. The efficient implementation of the signal processing algorithms can only be obtained with the prior knowledge of the target platform. This means that the signal processing algorithms have to be supplied by hardware-specific libraries, which are developed by the hardware providers who have a better understanding of the hardware architecture. Ideally, each library function should exist in different versions to support different mapping decisions as well as different requirements like throughput, latency etc. This gives a broad range of flexibility to the VRE compiler to devise an efficient hardware-specific implementation.

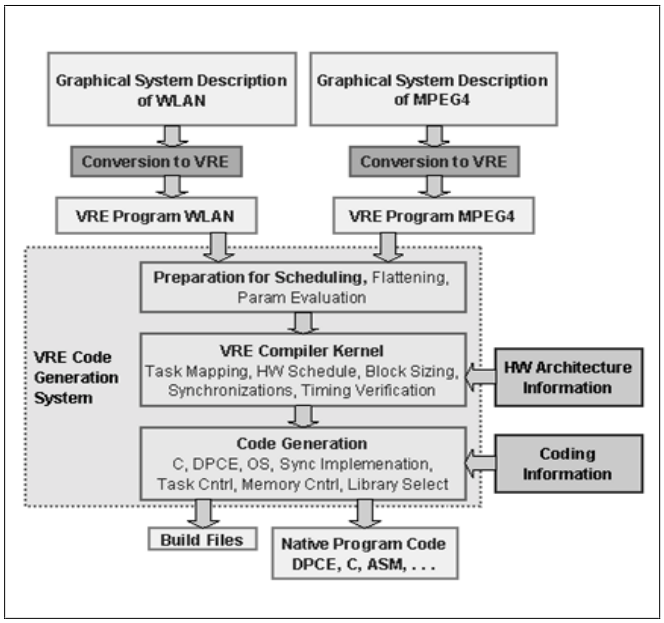

Fig. 2. Development flow of VRE. The applications are described in the VRE programming language without any hardware dependencies. A code generator takes this description as well as a separate description of the hardware as input.

V. The VRE Language

VRE has been designed with the goal in mind that an efficient mapping to a broad range of parallel hardware platforms should be possible. As said before, basic functionalities such as FFT are provided as libraries, which form the building blocks from which VRE programs are composed. We call these building blocks *primitives*; they are considered as black boxes by the VRE compiler. Only the interfaces, i.e., parameters, ports, and memory accesses, are part of the description. The *parameters* customize the block functionalities, i.e., they modify the function executed. The *ports* are interfaces for the input/output data as well as control flow. Each port has a unique name with respect to a primitive block, with properties like data type and size being part of the corresponding port definition. *Memory locations* in VRE are described by name, data type and size. These names are unique in the sense that the same name denotes the same memory location, throughout the program.

The description of the overall system is hierarchically composed of *modules* that contain *primitives* and other modules, and describe a part of the signal processing algorithm. In other words, one can consider the overall system as a tree, with primitives in the leaves, and modules in the interior nodes. At any one level of this hierarchy, lower-level modules are considered as black boxes, and the VRE program for this level describes different kinds of dependencies between the modules. VRE supports three types of

dependencies to describe the application program: *message-coupled dependencies*, *memory-coupled dependencies*, and *control dependencies*.

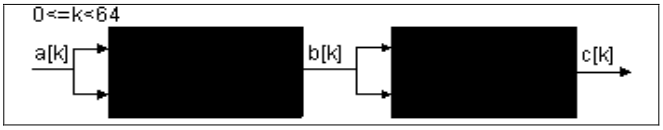

Fig. 3. Example of a data flow description. The arrows show the direction of data flows.

Both message-coupled dependencies and memory-coupled dependencies are described by *signals*, which only tell how the blocks are connected, i.e., they indicate that the modules exchange information. Type, size and other properties of the information exchanged can be recognized from the corresponding port's properties, which are given in text form. A signal may represent either a data flow or a trigger. Message-coupled dependencies refer to direct data flow between modules. The dependencies are directed, i.e., if an output port of block B is connected to an input port of block A, then the execution of block A is dependent on the execution of block B. An example is given in figure 3.


Fig. 4. Comparison of the overall execution times of the module described in figure 3 for two possible (a sequential and a parallel) implementations. In the first case, the block size of data is considered as a vector of 64 values, and in the second case, it is considered as a vector of 16 values.

Data flows impose restrictions on the execution order of the modules, since an operation that produces a data item has to occur before a second operation that consumes it. A different sequence might change the overall result. Even if two blocks are data-dependent, they may be run in parallel as long as the data flow restriction is respected during the implementation. Consider the example depicted in figure 3. There are two modules, "Multiplier" and "Adder", which are connected through data flow signals. Let the inputs and outputs of the modules each be a data vector of 64 values, so that the total number of operation is 128 (64 operations of the "Multiplier" + 64 operations of the "Adder"). Here, a straightforward sequential implementation respects data dependencies if it executes the "Adder" after the 64 operations (b[k]=a[k]*a[k] for k=0..63) of the "Multiplier". Alternatively, the VRE compiler can modify the block size during the implementation, i.e., split both the "Adder" and the "Multiplier" into sub-blocks. Then, the "Adder" can be executed in parallel with the "Multiplier" as soon as the multiplier is finished with its first block operation, which reduces the overall execution time. Such a scenario is depicted in figure 4. Here, a change in the block size of data from a vector of 64 values to a vector of 16 values reduces the overall execution time from 128µs to 80µs.

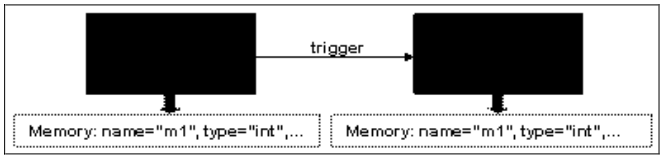

Fig. 5. Example of a data flow description. The arrows show the direction of data flows.

Memory-coupled dependencies occur when two modules access the same memory location. *Memory* is used to import and export data independent from data flow. In this case, it has to be considered what the read/write constellation is, and the corresponding order of memory accesses has to be specified. The VRE language does not allow memory-coupled dependencies between two modules if there is already a message-coupled dependency. Memory-coupled dependencies described different scheduling information than that of message-coupled, as VRE does not support parallel execution of blocks (like figure 4 describes a possibility of parallel implementation of the algorithm chain described in figure 3) if they are accessing the same memory content. Two blocks accessing the same memory are described sequentially. VRE uses trigger flows between the modules to indicate the order of memory accesses. Trigger flows may connect the modules directly, or go through other modules in-between. An example is depicted in figure 5. Here, both "Mod1" and "Mod2" access memory location "m1", where the access in "Mod1" is a write and the access in "Mod2" is a read. As the write must be performed before the read, the trigger is directed from "Mod1" to "Mod2".

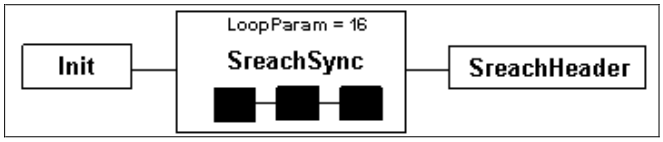

Fig. 6. Example of a control dependency. The signals are just triggers that describe the sequence of operations, in this case the sequence is: 1 X Init, 16 X SearchSnyc, 1 X SearchHeader respectively.

In addition to data dependencies, which we have distinguished into message-coupled dependencies and memory-coupled dependencies, there are *control dependencies*, which occur in loops and branches. An example is depicted in figure 6, which is about searching a synchronization signal. Here, the internal contents of the SearchSync composed module are executing repeatedly to search different parts of the synchronization preamble. After the last iteration, the SearchHeader algorithm is called. This sequence is important, because we need the result of the SearchSync routine as an input for the SearchHeader routine. But this can not be described by message-coupled or memory-coupled dependencies only.

Here, we use a special block parameter ("LoopParam") to describe the internal scheduling behavior of the composed module, i.e., repeatedly execute the contents of the corresponding loop module as long as the current iteration count does not exit the iteration limit.

At present, the VRE language separates message-coupled and memory-coupled dependencies into two different domains: *message-flow* domain and *trigger-flow* domain. In message-flow domain, the connections between the modules are data type signals, and the algorithms are described in terms of message-coupled dependencies. Signals in trigger-flow domain, in contrast, refer to trigger flows and describe memory-coupled dependencies or control dependencies. In general, we describe higher-level algorithms in the trigger-flow domain with different hierarchically composed modules, where the inner construct of each of these modules can be described in either message-flow or trigger-flow domain. Trigger-flow domain is also used for loops, which are described by composed modules. The description of any one module is either in the message-flow domain, or in the trigger-flow domain.

VI. VRE WLAN 802.11 EXAMPLE

To verify the concept of VRE we designed a VRE program for the WLAN 802.11 signal processing, which can be used as a source text to generate (manually or automatically) an implementation for different hardware architecture platforms.

The 802.11 VRE program can be roughly separated into three different areas. The first is the signal processing, where all the data stream operations and the dependencies between these operations are described. The second area is the application control, where the main control for the intra-frame processing structure, like initial chip and symbol synchronization, channel estimation, header detection and data processing is described. The third area covers the MAC layer protocol including the interface to the signal processing, which describes the inter-frame behaviour. The structure of the protocol again is quite different to the signal processing, because it is mainly based on independent processes communicating via messages.

Describing all these areas allows a detailed evaluation of the overall parallelisation restrictions of the application, which can not be seen from the signal processing description only. It is for example important to describe in any way, that the initial frame synchronization processing is never required in parallel to the frame user data demodulation. All three areas required different description structures to allow an efficient implementation. Real time requirements are part of the description and can be part of any of these three areas. From this description we can generate a multithreaded program code, considering as much parallelism as possible, which runs on a Sandblaster SB3000 [5] hardware platform.

VII. CONCLUSIONS

There is already a preliminary version of the VRE tool chain available that partly consists of commercial tools (like MLDesigner for graphically describing the VRE program) and own developments. This version was used for the experiment described in section VI. The experiment shows the potential of the VRE tool chain in SDR application development. The next major step will be the incorporation of Mathwork's Simulink tool into the VRE tool chain, such that we can use both Simulink and MLDesigner to describe the applications graphically. Since both tools are standard industrial development environments, it will increase the intuitive understanding of applications and simplify familiarisation for new programmers. Furthermore, the functionality of the code generator has to be extended such that it supports efficient implementations for different parallel hardware platforms. We will focus on high level mapping and generation of C code tailored to a platform, but continue leaving all hardware-specific register and other low-level optimizations to hardware manufacturers.

REFERENCES

[1] C. Grassmann, M. Sauermann, H.-M. Bluethgen, and U. Ramacher, "System level hardware abstraction for Software Defined Radios, " in *Proc. 2004 Software Defined Radio Technical Conferemce,*Arizone, Nov, 2004, Paper 1.2-3, pp A113.

[2] J. T. Buck, S. Ha, E. A. Lee, and D.G. Messerschmitt, "Ptolemy: A framework for simulation and prototyping heterogeneous system," *Intl. Journal of Computer Simulation*, vol. 4, April 1994, pp. 155-162.

[3] E. D. Willink, "The waveform description language: moving from implementation to specification," in *Proc. Military Communication Conference 2001, Communication for Network-Centric Operation: Creating theIinformation Force.IEEE* ,Oct. 2001, vol. 1, pp 208-212.

[4] C. H. van Berkel, F. Heinle, P. P. E. Meuwissen, K. Moerman, and M. Weiss, "Vector processing as an enabler for software-defined radio handsets from 3G+ WLAN onwards, " in *Proc. 2004 Software Defined Radio Technical Conferemce,*Arizone, Nov, 2004, Paper 2.4-1, pp B125.

[5] D. Iancu, J. Glossner, H. Ye, M. Moudgill, and V. Kotlyar, "Software rake receiver enhanced GPS system," in *Proc. 2004 Software Defined Radio Technical Conferemce,*Arizone, Nov, 2004, Paper 1.2-1, pp A97.

[6] Ptolemy 0.7 User Manual, vol. 1, 1997, chap. 4.1. Available: http://ptolemy.eecs.berkeley.edu/ptolemyclassic/almagest/docs/user/html/domains.doc.html

[7] J. Steed, K. Barnes, and W. Lungdren, "Gedae: A tool for implementing software radio on heterogeneous system, " in *Proc. 2004 Software Defined Radio Technical Conferemce,*Arizone, Nov, 2004, Paper 1.2-5, pp A127.

[8] P. v. der Wolf, E. de Kock, T. Henriksson, Wido Kruijtzer, "Design and programming of embedded multiprocessors: an interface centric approach," *in Proc. of the 2nd IEEE/ACM/IFIP international conference on Hardware/Software codesign and system synthesis*, 2004, pp 206-217.

[9] X. Liu, J. Liu, J. Eker, and E. A. Lee, "Heterogeneous Modeling and Design of Control Systems", Chapter in "Softwate-Embedded Control: Information Technology for Dynamic Systems," T. Samal and G. Balas (eds.), New York City: IEEE press, 2002. Available: http://www.control.lth.se/~johane/publications/csm_ptolemy.pdf

[10] W. Lundgren, K. Barnes, and J. Steed, "Gedae: Autocoding to a virtual machine," Available: http://www.gedae.com/documentation/articles.html

[11] Simulink documentation. Available: http://www.mathworks.com

[12] MLDesigner documentation. Available: http://www.mldesigner.com

[13] Hardware abstraction layer working group report on result of request for information SDRF-04-A-0009-V0.00, version 0.004, Oct 2004. Available: http://www.sdrforum.org

[14] H.-M. Bluethgen , C. Grassmann, W. Raab, U. Ramacher, and J. Hausner "A programmable baseband platform for Software Defined Radios, " in *Proc. 2004 Software Defined Radio Technical Conferemce*,Arizone, Nov, 2004, Paper 3.4-2, pp B155.