22 Programming Models and Implementation Platforms for Software Defined Radio Configuration

Tanguy Risset Riadh Ben Abdallah Antoine Fraboulet Jérôme Martin

22.1 Introduction

Software means programmable. Hence software defined radio means that the radio should now be *programmable*. We know what computer programming means, and we agree, up to a certain level, on how it should be done. But do we know what programming a radio means? Several questions are still open: what will a SDR platform look like in ten years? Will there exist *software radio code*? What will be the technical challenges and commercial issues behind this code?

Programming is more precise than configuring or tuning, it implies a much greater level of freedom for the programmer. But it also means much cheaper implementations in many cases and in particular a re-use of the same hardware for different protocols (i.e. with different programs). This is, to our point of view, the main difficulty of software radio programming: reconfiguration and in particular dynamic reconfiguration. Dynamic (i.e. very fast) reconfiguration is now mandatory because some protocols, 3GPP-LTE (Third Generation Partnership ProgramLong Term Evolution) for instance, propose channel adapting for *each frame*, requiring a setting of the channel estimation parameter in a few milliseconds.

In this chapter we will try to have an overview of the technical difficulties of designing a programming environment for software defined radio. Then we will present one particular solution which aims at defining a virtual machine dedicated to the domain of software defined radio.

22.2 Programming Environment and Tools for SDR

In this section we present existing SDR platforms and give an insight about how they are programmed or configured. This brief review leads to the concept of waveform description language developed in subsection 22.2.2 and reveals the need of a middleware dedicated to SDR programming (subsection 22.2.3). Finally we introduced the Radio Virtual Machine (RVM) concept, which is one possible

choice for SDR middleware, in subsection 22.2.4. Our proposal for such a radio virtual machine will be detailed in section 22.3.

22.2.1 Hardware Platforms for SDR

Unlike for desktop computers, software radio hardware is not standardized yet. The only common point among all SDR hardware platforms is the complexity of their architecture. They are usually composed of many *processing components*, dynamically reconfigurable, and interconnected with fast communication links.

The source of this complexity is, of course, the needed processing power and the need to implement possibly *all* protocols. Software radio attempts to implement in software most of the radio processes that are initially hardwired. In practice, digital treatment are implemented by algorithms that require many giga-operations per second (GOPS) to meet protocols real time constraints. For instance, a turbo-decoding may require 150 GOPS which is far from what embedded processors can achieve.

This problem is solved with the use of dedicated processing elements: FFT, turbo decoder, FIR filters, digital predistortion and matrix inverse for instance. These components, also named IP for *intellectual property*, help to achieve the required computing power. Another solution to this problem is to use Digital Signal Processors (DSP), dedicated to stream processing. Note that the complexity of software radio treatment keeps on increasing. For instance, the 3GPP-LTE protocol provides a data throughput which can reach 300 Mbps. Hence it is very likely that the inherent difficulty of building a software radio hardware platform will stay for a while.

As mentioned in [FSC09], most of existing SDR platforms are prototypes built by public or private research laboratories. Indeed, SDR platform design is a real challenge for hardware architects as well as for software developers : the good trade-off among computing power, power consumption and flexibility (i.e. programmability) is very hard to find.

We present some existing SDR platforms by classifying them in two categories:

1. *DSP centric platforms* that use only software components (DSP, GPP, etc.), and hence are highly flexible but must usually be associated to hardware IPs to meet real time performance.
2. *Heterogeneous platforms* that try to mix dedicated hardware and software components.

DSP Centric Platforms

Many companies (Sandbridge, picoChip, Fujitsu, Icera, Infineon, NXP, etc.) propose integrated circuits based on DSPs, here are some examples:

- The PicoArray processor [DTP+05] from picoChip. This circuit integrates hundreds of small processors. PicoArray can be programmed in ANSI C with a dedicated programming environment. Global computing power can reach

200 GOPS. Associated to some selected IPs (FFT, turbo-decoder, ...), this circuit is able to implement a complete W-CDMA modem.

- X-GOLD^{TM}SDR-20 from Infineon technologies is a signal processing processor for base band processing for multi-standard mobile phones. Infineon proposes hardware/software solutions with this platform that support recent protocols (GSM, W-CDMA, LTE, ...).
- EVP (Embedded Vector Processor) [vBHM$^+$05] from NXP : this architecture is able to support various modes of the LTE protocol.
- In [LLW$^+$07], Lin et al. present a DSP based system composed of 4 SIMD (Single Instruction Multiple Data) vector processors. This architecture can realize two different processing chains: IEEE802.11a and W-CDMA.
- Tan et al. [TZF$^+$09] present an original approach for baseband treatments realized on general purpose processors. They implement IEEE802.11a/b and g physical layers using multi-core architectures.

It is important to realize that although a number of important *software* platforms are mentioned in the literature, many of them have a limited computing power and a bad power consumption, they also need to be associated to dedicated hardware IPs.

Heterogeneous Platforms
These platforms integrate dedicated IPs, usually controlled by a processor. Experimental version may contain FPGAs for implementing recent signal processing algorithms.

- Small Form Factor (SFF SDR) Development Platform from Lyrtech: this board embeds a DSP and an FPGA, for baseband processing, connected to a RF board. A development environment is given for programming this machine.
- Universal Software Radio Peripheral (USRP) : is a hardware platform conceived for the GNU Radio project [TT09]. It connects to a computer with a USB interface.
- Kansas University Agile Radio (KUAR) platform [MES$^+$07]: is an experimental SDR platform that includes a Pentium M processor and a Xilinx Virtex2 FPGA. The board connects to a PC either through a gigabit Ethernet interface or through a PCI-express link.

Even from the small platform list presented here, it seems clear that the work of developing a new protocol for each of these platform is a huge task. Expressing this protocol in an existing language like C or Java will not help because of the granularity of the basic operators used in SDR platforms (e.g: FFT operation). Moreover, there is a real challenge in expressing in a high level language the dynamic reconfiguration which is hardly realized in hardware but will necessarily appear soon. We really need a way to describe protocol physical layers (so-called waveform processing) that can be understand by all the existing SDR platforms. This gave rise to the concept of *waveform description language*.

22.2.2 Waveform Description Language

A waveform description language (WDL) is simply a programming language dedicated to the expression of the physical layer of a communication protocol: how bits are modulated and transmitted on the antenna for emission and the reverse way for reception. It basically describes the waveform that will be sent in the air for a given packet to be communicated. The design of a waveform description language is of course highly connected to the existence of at least one SDR platform able to implement this physical layer. Meanwhile, the long term objective is to have a common WDL for all existing SDR platform.

Many works have pointed out the difficulties in expressing waveforms [GSBR04, KAW+06, Wil01, YJ]. Here is a summary of the properties that a waveform description language should try to respect:

- Be a formal language. Waveform specifications based on large textual documents are often error-prone for developers, the waveform specification should be *compiled* on each SDR platform;
- Implement a clear and extensive Hardware Abstraction Layer (HAL) adapted to most existing SDR platforms. As for processor, a clear hardware abstraction layer will ease the adaptation to different SDR hardware architectures. It also clarifies what is the assumption on the targeted SDR platform and helps in writing specifications that are independent of any hardware platform.
- Use of a component based model which is well adapted to SDR. Many protocols can be expressed by connecting components : FFT, Viterbi decoder, scrambler, etc. These common operators should be easily identified into the language.
- Use an object oriented programming paradigm to ease the mapping between hardware components and software objects semantic.
- Follow the "Write Once, Run anywhere" philosophy, which is a slogan created by Sun to describe benefits of Java virtual machine. This gave rise to the idea of Radio Virtual Machine (RVM).

We now briefly present some attempts that have been made to realize waveform description language. These works are results of academic research, it is worth noting that there is also a military project titled *Advanced Transmission Language and Allocation of New Technology for International Communications and Proliferation of Allied Waveforms (ATLANTIC PAW)* whose goal is to provide a unique standard for expressing waveforms.

Wilink Waveform Description Language

In [Wil01], Wilink presents a Waveform description Languages named WDL, proposed within the *Programmable Digital Radio (PDR)* in UK. It is a behavioral system description: a hierarchical block decomposition using state machines within boxes.

WDL uses a combination of principles defined in various research domains: graphical interface concepts found in block diagram languages such as Ptolemy,

Cossap, SPW. Hierarchical state machines, used for instance in Argos or SDL are used together with synchronization mechanisms inherited from synchronous languages like Esterel. Data types manipulated are defined in object oriented languages as Java or C++.

Figure 22.1 gives an insight of how is expressed a waveform in WDL. It describes a part of the FM^3TR waveform which is an international test waveform initially developed by the Air Force Research Labs.

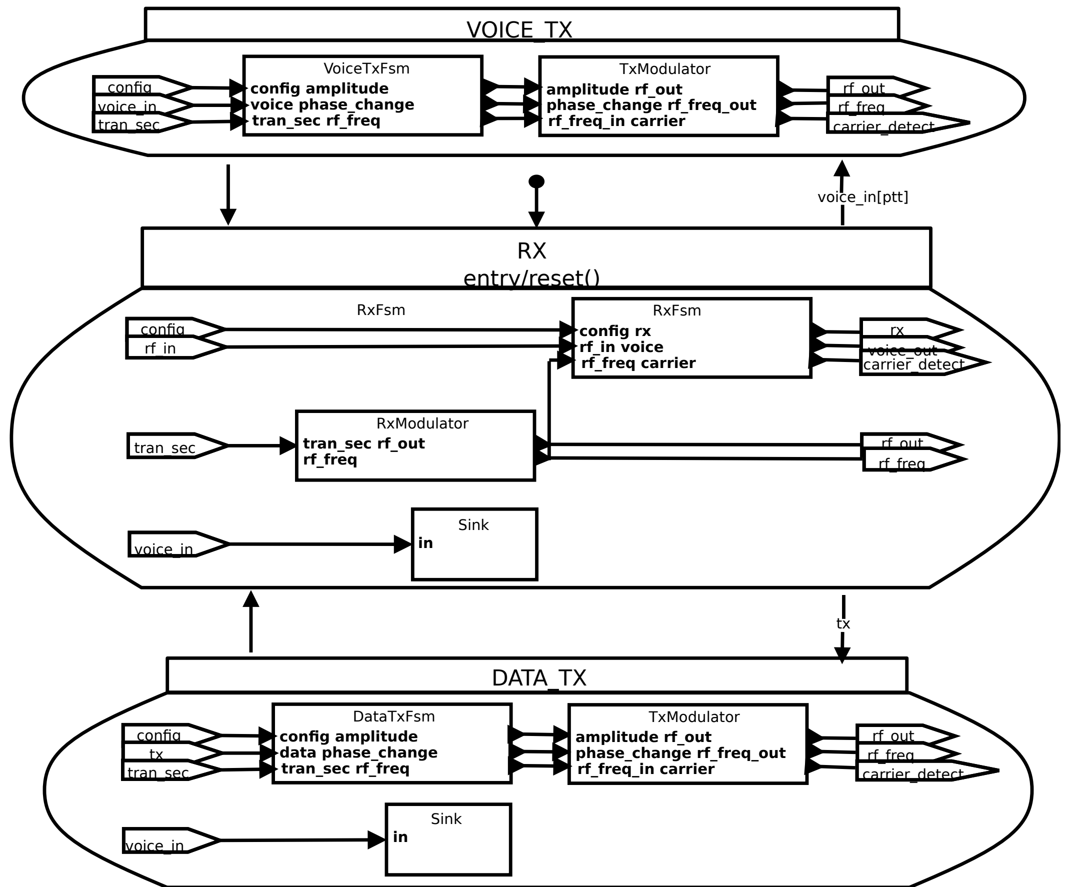

Figure 22.1 Example taken from [Wil01] a WDL description of the FM^3TR physical layer

Figure 22.1 is a graphical representation of an entity: a transmission module. This transmission module is itself refined: it is a state machine, each state being itself a block diagram. At the lowest granularity, computations are expressed in Java-like syntax.

Each WDL block gets an internal schedule, hand-shake mechanisms are used to synchronize with input/output data. The refinement occurs up to a granularity level that is suited to the available library provided by the development

framework associated with WDL. The WDL framework has to be provided for each target architecture. This language does not permit dynamic reconfiguration of hardware. To our knowledge there is no real implementation of WDL showing performance issues.

VANU RDL *(Radio Description Language)*

RDL is a waveform description language developed by VANU, Inc. Chapin et al. present its foundation in [CLM01]. A radio application is described by an oriented graph where nodes are basic signal processing operators provided in the form of software libraries. A RDL *interpreter* is an execution environment that realizes the required processings, it maps the description graph to the targeted platform by configuring the available hardware. The interpreter can also use software blocks, i.e. software implementation of signal processing primitives.

RDL defines two basic elements:

- *Modules* : basic signal processing operators defining the nodes of the graph.
- *Assembly* : i.e. graphs, composed of modules and sub-graphs.

Common types used in signal processing languages such as ports, channels, and streams are available in RDL.

```
module RxConvDecoder {
  parameter EncoderFormat format;
  dataout GsmFrame output;
  datain GsmFrame input;
}

assembly RxConvDecoderDef
implements RxConvDecoder {
  module ConvDecoder convDecoder;
  module DecoderFrameGenerator frameGen;
  module DecoderStreamGenerator streamGen;

  // data flows
  streamGen.mProtectedOutput -> convDecoder.input;
  streamGen.mUnprotectedOutput -> frameGen.mUnprotectedInput;
  streamGen.mHeaderOutput -> frameGen.mHeaderInput;
  convDecoder.output -> frameGen.mProtectedInput;

  // link ports
  frameGen.mOutput -> output;
  input -> streamGen.mInput;
}
```

Figure 22.2 Example of a RDL graph specification

An example is illustrated in Figure 22.2. This assembly uses three modules, each of these modules can be itself refined in other assembly or directly implemented by an available library primitive.

An implementation of the GSM protocol has been realized using RDL [CB02]. The RDL program required 36 modules and 28 assemblies representing approximately 3200 lines of code. Again dynamic reconfiguration cannot be achieved with this language.

E^2R FDL *(Functional Description Language)*

E^{2}R is the acronym for European research project *End to End Reconfigurability* which aims at developing architectures of reconfigurable communicating systems. Burgess et al. [BM] have studied the possibility of defining a language able to unify waveform specifications.

The E^{2}R project has defined a software architecture for radio equipment. This architecture, referred to as "Configuration and Control Architecture", is shown in Figure 22.3. It isolates three abstraction levels: *i*) hardware abstraction, *ii*) system abstraction and *iii*) function abstraction. This work is interesting because it highlights the difficulty of the waveform description language definition and implementation.

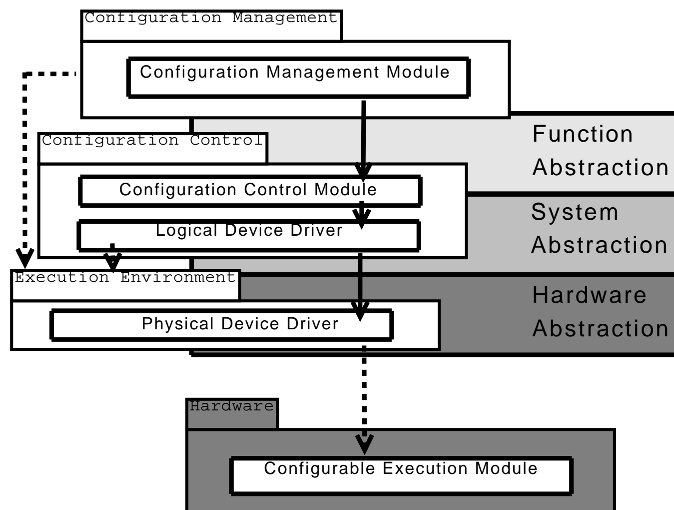

Figure 22.3 Configuration and control architecture of E^{2}R radios

The FDL (Functional Description Language) defined in the E^{2}R project is based on XML and basically proposes a hierarchical composition of components as does RDL. Again, this structural description is useful for describing different protocols but cannot implement dynamic reconfiguration.

In [ZDSS07], Zhong et al. have implemented in software an IEEE802.11a emitter using FDL. They show that the FDL program size is not a big problem but that the execution performance is much worse than with dedicated hardware.

SPEX *Language*
SPEX is a programming language for SDR developed by the Michigan university SDR group. This language is dedicated to DSP-centric platforms and more precisely takes advantage of SIMD DSP. A SPEX description is split in three abstractions: Kernel SPEX, Stream SPEX and Synchronous SPEX.

- Kernel SPEX is an imperative language supporting native DSP arithmetic operations. It is used to define kernel signal processing algorithms (e.g. FIR, FFT, etc.).
- Stream SPEX describes assembly of kernel programs: connections with channels and sequential scheduling.
- Synchronous SPEX allows parallel construction and synchronization with real time constraints.

SPEX is an interesting language but is clearly oriented to pure software implementation: code is compiled and mapped on a multi-core architecture. It is not clear yet whether these architectures will succeed for SDR.

22.2.3 Middleware for SDR Programming

The usage of middleware in networked application is now widely accepted. Middleware requires the definition of standardized interfaces enabling heterogeneous distributed software components to communicate. Middleware is the software that implements an intermediate abstraction between applications and different execution platforms, they usually provide a higher level API than the one provided by the HAL level. Software radio, considered as a particular field of software programming, needs a specific middleware for radio applications. We present here two attempts that have been made to define an SDR middleware.

*Software Communication Architecture (*SCA*)*
The SCA architecture [JTR06, BK07] has initially been conceived in the American military program JTRS (Joint Tactical Radio Systems) for the development of SDR. It is now considered as a middleware dedicated to hardware SDR platforms.

The SCA environment is composed of three main elements: the *Core Framework*, a Corba Object Request Broker and a real time *operating system*. This environment is represented in Figure 22.4.

The SCA framework is the most popular software radio software architecture. Many companies developing middleware for SDR provide implementation of SCA on different hardware platforms, such as for instance Prismtech, Zeligsoft and OIS. However, it is obvious that the use of the Corba specification has an important impact on performances and makes this system not well adapted to actual commercial communication protocols.

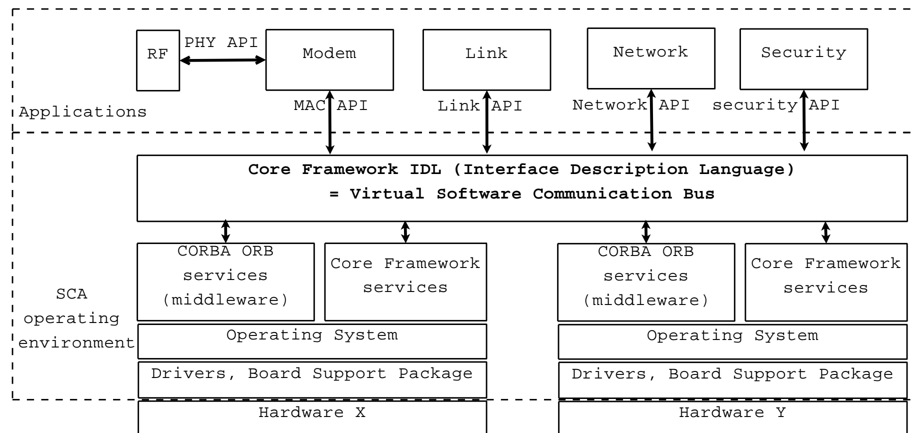

Figure 22.4 Software architecture of SCA

UPC Radio Software Framework

Gelonch et al. from Polytechnical University of Catalonia (UPC) propose in [GRMF05] the P-HAL framework: Platform and Hardware Abstraction Layer. The P-HAL framework abstracts hardware radio platforms by functional services, these services can be categorized in three classes:

- Real time control of radio processes.
- Exchange of data between different processing elements.
- Parameter setting and supervision of functional modules.

By providing a time division in slots, the P-HAL environment tries to bring a simpler real time radio programming environment. In [GMSG08], Gomez et al. present a comparison between P-HAL and SCA on a DSP platform and claim much better performance for P-HAL .

22.2.4 Radio Virtual Machine Concept

From what we have seen above, we would like to see a software defined radio system as a software layer offering application programming interfaces (API) enabling the definition of waveform, and their implementation on a hardware platform. In 2000, Gudaitis and Mitola proposed in [GI00] to apply the virtual machine concept to software defined radio, proposing the term *radio virtual machine* (RVM).

Java virtual machines have allowed the wide spreading of Java applications. Performance weaknesses of bytecode execution has lead to the proposal of just-in-time compiler (JIT) that bridged the performance gap. According to Gudaitis and Mitola a radio virtual machine is a particular virtual machine (VM) with its own programming language, which we call *source code*, that can be compiled into *bytecode* for the VM. This VM will, as it is the case for Java, provide a common

hardware abstraction for software and firmware/hardware developers. This would split the radio application development cycle in two (possibly parallel) phases:

- Software development of the radio application in VM bytecode, common to all platforms.
- Platform specific development to optimize execution of the VM on each particular targeted platform.

Gudaitis and Mitola enumerate a list of important goals that a RVM should achieve:

- Provide a programming language that permits an easy expression of physical layers of most protocols and that can be compiled into an executable form (bytecode).
- Provide an abstraction based on the component model paradigm.
- Avoid Java virtual machine pitfall: provide mechanism to handle real-time constraints and easy access to hardware.
- Include an arbitrary bit-width arithmetic.

Some patents have yet been granted [Fer02], [MKB07], [Bur04] but the field is still wide open for innovation.

22.3 An Existing Radio Virtual Machine Implementation

This section presents a particular prototype RVM implementation that has been realized during R. Ben Abdallah Phd Thesis [Abd10], within a collaboration between three French institutes: CEA, Inria and Insa-Lyon. The goal of this work was to explore the technical viability of using a virtual machine for radio application on a real SDR hardware platform, namely the Magali [CBL+10, CLT+09] chip developed at the CEA Leti laboratory.

As we have frequently noticed above, the main difficulty of SDR programming relies in the dynamic reconfiguration: such a system should be able to configure some of its components within a few hundred microseconds (e.g. within the same frame for LTE advanced protocol). The reconfiguration process is generally triggered with depending on system status and external events such as radio channel impairments. This issue has to be taken into account in the design of an SDR programming model.

22.3.1 Waveform Programming Model

The approach that we have proposed is the following: we introduced a two steps programming model called *reconfigurable khan process network* that formalizes the reconfiguration phase and separates it from the computation phase. Then we propose a computation model which should support most of the existing heterogeneous platform, the main restriction of this computation model is that

it is not well adapted to massively parallel systems where scheduling is done within each of its processing element: our model requires a centralized scheduler.

Computation Model

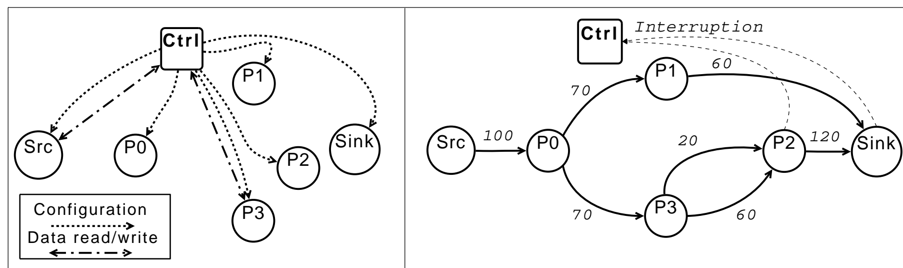

Figure 22.5 Example of a RKPN, during reconfiguration (left) and during computation (right)

Khan process networks (KPN) have been introduced in the seventies by Gilles Kahn [Kah74]. KPN is a distributed computation model with a precise semantic. In KPN, an application is a set of sequential processes communicating through channels which are blocking FIFOs. It is now widely used for modeling signal processing systems. In a KPN, a process cannot test whether a channel is empty or not, it cannot *choose* which channel to read from. This is not an important restriction for modern signal processing programs except during reconfiguration where the channels between processes are changing. The *reconfigurable khan process network* (RKPN) computation model is composed of sequential processes connected by blocking FIFO (i.e. as for a KPN), its behavior alternates between computation phases (i.e. standard KPN computation) and configuration phases during which channels between processes are allowed to change. Figure 22.5 shows the configuration phase and the following computation phase of a RKPN.

We have added the following constraints to this model in order to match with the SDR application and hardware platforms:

- There is a particular node that we call *controller* that controls the configuration of the KPN. This node cannot be source or sink of an FIFO but can receive interruptions from other nodes. This node can also, during reconfiguration phases, access to other nodes memory and of course reconfigure other nodes and connections between them.
- During each computation phase, each node knows in advance the *number* of data unit that it should process. We call that a *static control program*, it helps in optimizing implementation and is usually not an important restriction for telecommunication protocols (as opposed to multi-media algorithms), as soon as reconfiguration is available.

- For convenience, one usually adds two particular nodes called *source* and *sink* to materialize the world outside the SDR (e.g. RF front end on one side and higher OSI level protocols on the other side).

Note that this computation model does not mention the use of a virtual machine, it is adapted to other SDR middleware solutions. It takes into account the reconfiguration requirements of SDR applications. Another novelty is that the data of the signal processing stream can have an impact on the configuration and control of the system. Indeed, the controller can access data stored in nodes (Figure 22.5, left), possibly do some computations with them, and finally configure the network. This is useful for instance when implementing advanced channel adapting algorithms.

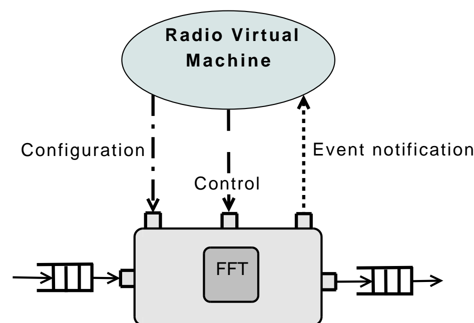

Figure 22.6 Example of radio component: FFT

Execution Model

The Execution model is an abstraction of the hardware platform on which will be executed the SDR program. It is important to emphasize the fact of having a component based model for all IPs of the platform. We impose that an IP should be a *radio component* and have the following interfaces (a radio component model is shown in Figure 22.6):

- Configuration interface: for tuning functional parameters of the IP.
- Communication interface: for the main input/output data stream.
- Control interface: for being started, stopped, checked, etc. by the controller.
- Notification interface: for notifying the controller of particular events (corresponds to an interrupt mechanism).

Note that such a radio component can be a programmable device (DSP or GPP) or a dedicated IP (FFT, matrix inversion, etc.).

The proposed execution model is simply a set of radio component, such as defined above, interconnected by an efficient communication mechanism and associated to a particular IP that is the controller (GPP or dedicated IP). The execution platform will of course contain some memory elements (RAM). We choose the shared memory to communicate data between IPs and the controller

but other communication mechanism could be envisaged. Experience shows that Direct Memory Access modules (DMA) are necessary to achieve acceptable communication performance. An example of target execution platform architecture is shown in Figure 22.7.

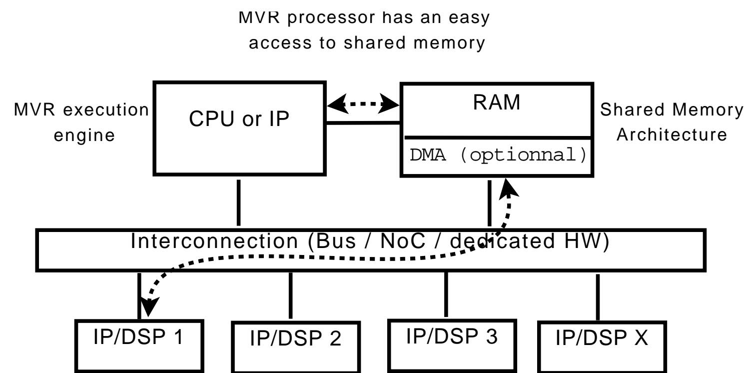

Figure 22.7 Execution model adapted to RKPN

22.3.2 Physical Layer Description Language

The Physical Layer Description Language (PLDL) is the programming language that is proposed in [Abd10] to describe physical layer protocols. Its main component is what we call the RVM API, which is a set of primitives and data structures dedicated to the RVM concept.

The PLDL is adapted to the RKPN programming model, it is executed by the controller which: *i*) allocates and frees radio resources, *ii*) configures radio components, *iii*) controls the execution of components and *iv*) accesses data stored in components. The PLDL program is platform independent, it can be executed on most SDR platform as well as on a desktop PC. One simply has to provide an implementation of the RVM API on each targeted platforms. We use an object oriented syntax for describing the RVM API, in the following **rvm** basically represents the controller and we describe its possible actions as methods of the **rvm** class.

Here is a brief description of the primitives of the RVM API:

Radio Resources Allocation / Release

- *comp_desc* **rvm.allocate (** *comp_type* **)** : Allocates a hardware component or creates an instance of a software component. This method returns a descriptor of the allocated component: *comp_desc*.
- **rvm.free (** *comp_desc* **)** : Releases the hardware component or frees the software component designed by the descriptor *comp_desc*.

Radio Component Configuration

- *core_config_desc* **rvm.build_core_config (** *comp_desc, core_param_list* **)** : Builds a native configuration for the component *comp_desc* using specified functional parameters. Returns a configuration descriptor.
- *com_config_desc* **rvm.build_com_config (** *comp_desc, com_param_list* **)** : Builds a communication configuration for the concerned component (for instance: IP input/output controller configuration). Returns a configuration descriptor.
- **rvm.free_config (** com_config_desc || core_config_desc **)** : Releases the memory used for the specified configuration.
- **rvm.configure (** *comp_desc, core_config_desc, com_config_desc, event_type* **)** : Configures the component specified by *comp_desc* using configurations stored in RVM memory: *core_config_desc* and *com_config_desc*. The parameter *event_type* specifies if the component must send a notification to the RVM for a particular event.
- **rvm.connect (** *comp_desc_src, port_num_src, comp_desc_dest, port_num_dest, data_type* **)**: Interconnects source component output port with destination component input port (i.e. configure a FIFO). The parameter *data_type* may be useful to configure communication channels.

Radio Component Execution Control

- **rvm.start (** *comp_desc* **)** : Activates the behavior of a component previously configured. Note that some hardware components are implicitly triggered by data arrival, but this is not the case for the RVM software components.
- **rvm.stop (** *comp_desc* **)** : Stops the behavior of the specified component.
- **rvm.wait (** *event_type, [comp_desc_event_source]* **)** : Blocks the execution of the RVM until an event notified by a component arrives (i.e. an interruption arrives). Optionally the source of the expected event could be specified.

The PLDL also includes methods for manipulating data of the data flow. In this case, it is up to the RVM programmer to check for memory consistency using the synchronization method **rvm.wait**. This also implies some technical choices such as a global memory addressing scheme.

Data Flow Access Methods

- *coarse_data_struct* **rvm.read (** *mem_ptr, size* **)** : Copy a data bock of size *size* stored at address *mem_ptr* into the RVM local memory. Returns a pointer to the raw data.
- *rvm_data_table* **rvm.convert2rvm (** *coarse_data_struct, data_type* **)** : Converts raw data from data flow to a data type understandable by the RVM language (data_type).

```
          [.....]

print("\n----------\nDATA FIELD DEMODULATION\n----------\n")

-- (0) Initialization --
Ndbps, Ncbps, Nbpsc, coding_rate, modulation = initparam802_11a(rate)

-- (1) Create required instances --
scra1 = scrambler.allocate()

-- (2) Connect the modules --
rvm.connect( vite1, 1, scra1, 1, binary_type )
rvm.connect( scra1, 1, dma2, 1, binary_type )

-- (3) Configure the modules --
param0 = ext_symbol_size
param1 = 16
param2 = symbol_size
param3 = nsym
dma_engine.configure(dma1, "RECEIVE ext_symbol_size; DESTROY 16",
                      NO_IT, param0, param1, param2, param3 )

phase_drift = phase_drift + 64*phase_amount

rotor.configure(roto1, phase_drift, phase_amount,
                 nsym*symbol_size, NO_IT)
fft.configure(fft1, mode_fft, fft_size, nsym*symbol_size, NO_IT)
equalizer.configure(equa1, coef, nsym*symbol_size, NO_IT)
constellation.configure(cons1, Ncbps, Nbpsc, nsym*symbol_size, NO_IT)
deinterleaver.configure(dein1, Ncbps, Nbpsc, nsym*Ncbps, NO_IT)
depuncturer.configure(depu1, coding_rate, nsym*Ncbps, NO_IT)
viterbi.configure(vite1, nsym*Ndbps, NO_IT)

          [.....]

-- (4) Launch modules --
rvm.start( dma1 )
rvm.start( roto1 )
rvm.start( fft1 )
rvm.start( equa1 )

          [.....]

-- (5) Wait for result --
rvm.wait( SIGTER )
print("\nProcess terminated. RVM wakes up.\n")

          [.....]
```

Figure 22.8 Example of a waveform description using PLDL formalism corresponding to (part of) the data field demodulation phase of the IEEE802.11a protocol.

- *coarse_data_struct* **rvm.convert2raw** (*rvm_data_table, data_type*) : Converts data from an understandable data type by the RVM language to a raw format specified by data_type (inverse of **rvm.convert2rvm**).
- **rvm.write** (*coarse_data_struct, mem_ptr*) : Copy the data from the local RVM memory to the system memory at address *mem_ptr* (inverse of **rvm.read**).

Figure 22.8 gives an insight of what might be a waveform description using PLDL. For a detailed description of the language, refer to [Abd10].

22.3.3 RVM Implementation Issues

There exist a lot of virtual machines, including light versions of Java VM (e.g. Squawk[SCC+06], JVM[LY99]). Java might be a good choice provided the targeted platform implement the Jazelle processor [Por05] otherwise Java VMs are too complex. Table 22.1 summarizes pro and cons of the VM we have isolated as potential candidate for being integrated into an embedded system such as a software defined radio system. We have chosen the Lua virtual machine [IDFF96] because it has been conceived to be light, embedded and easily extensible to define domain specific languages.

	Lua	Neko	Python	Squawk	Kaffe	LeJOS	TinyVM	NanoVM	Waba
small memory footprint	x	x		x	x	x	x	x	x
performance	x	x		x		x	x	x	x
extensibility	x	x	x						
memory size	x		x	x	x	x	x	x	x
documentation	x		x	x	x	x			

Table 22.1. Comparison of virtual machine candidate to implement a RVM

We have first implemented as a proof of concept, the RVM API on a standard PC, on which a software implementation of the IEEE802.11a protocol was available [ARFD09]. In this case, allocating a component would create a thread and call a function corresponding to the computation done by the component. The resulting software architecture is shown on figure 22.9 (left).

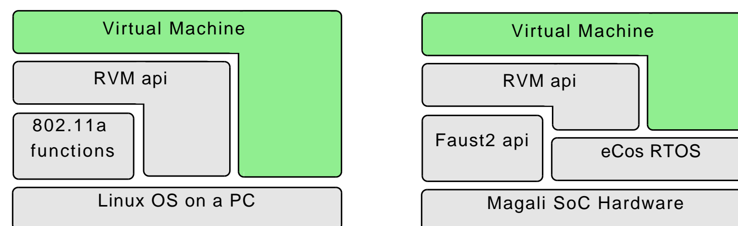

Figure 22.9 Software architecture of the Radio Virtual Machine, on a personal computer (left) and on the Magali Chip (Right)

An implementation of the RVM was then realized on the Magali Chip. Magali is a system on chip (SoC) developed at the CEA Leti, also called Faust2 as being second generation of the Faust SoC [DBL05]. Magali is dedicated to physical and MAC layers of 4th generation telecommunication protocols such as 3GPP-LTE or IEEE802.16e (WiMax). It is composed of an asynchronous network on chip

with a 2D-mesh topology, each router of the network being connected to the components of the system.

Figure 22.10 presents the different components available on the chip. These components are dedicated IPs necessary to realize OFDMA processing: FFT, LDPC, turbo-decoder, etc. In addition to that, Magali contains the following specific components:

- The *smart memory engine* (SME) is a programmable DMA extensively used to move data between components. All the memory available on the chip is accessed through these SMEs.
- The Mephisto processor is a VLIW DSP used to realize efficiently digital signal processing algorithms: channel estimation, MIMO decoding, digital predistortion, etc.
- The ARM1176 is a general purpose processor used for the global control of the Magali platform as well as for MAC layer processing. In our case this processor will be used as the RVM global controller described previously, i.e. the Lua virtual machine will be executed on this processor. This processor can also access directly to the memory without going through an SME.
- Associated to each router is a dedicated component called CCC for Communication and Configuration Controller that is used to regulate data exchange between components and their configurations.

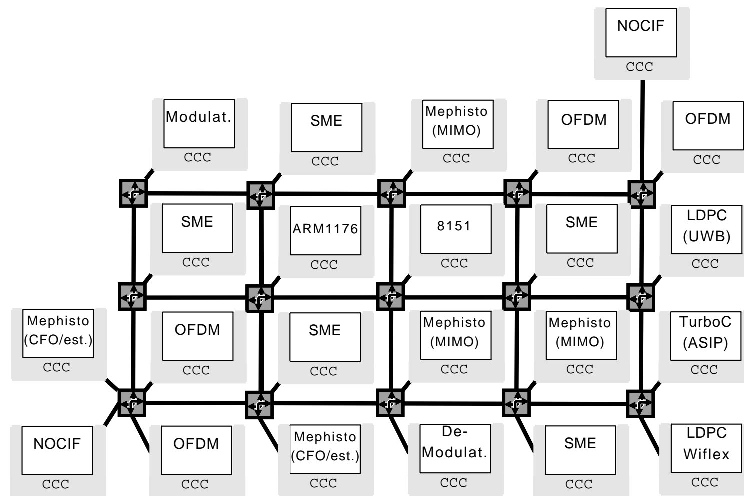

Figure 22.10 The Magali system on chip

As the reader can see the Magali platform is quite complex and it is very difficult to precisely present all the implementation work that occurred for having a radio application running on our radio virtual machine. We simply enumerate the different stages of the implementation and then we will compare this implemen-

tation with a native (i.e. handcrafted) implementation of the same application on Magali.

The reader must be aware that an important choice was made: we decided to implement a *soft* virtual machine. The Lua VM has been ported on the ARM processor (with eCos OS). Another possibility would have to use a *hard* virtual machine, i.e. a dedicated processor for bytecode interpretation. But this would mean to build another chip.

The first part of the work was to adapt the RVM API presented in section 22.3.2 to the Magali platform. The Magali programming environment provides an API called F2 API, the RVM API encapsulates this native API. Then, we ported the Lua virtual Machine to the ARM processor so that programs such as the one presented in Figure 22.8 can be executed on the Magali platform. Finally we wrote the PLDL program for a 3GPP-LTE receiver configured according to a particular operating mode in order to validate and experiment our RVM prototype.

22.3.4 Performance Results for a CFO IEEE802.11a

The Carrier Frequency Offset (CFO) represents the phase shift between emitter and receiver clocks due to hardware imperfections. First, we have implemented the CFO error correction algorithm present in the IEEE802.11a protocol. Then, we have measured the memory footprint and the execution time required by the CFO application for each of the three configurations of implementation on the Magali chip: *i*) native implementation (hand-coded, as it was done without RVM), *ii*) with RVM API or *iii*) with the virtual machine. Figure 22.11 illustrates these three configurations in the case an FFT operator.

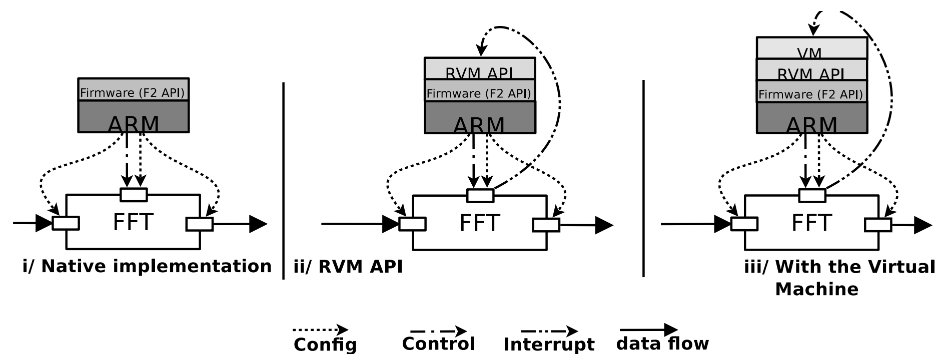

Figure 22.11 Different possibilities for executing a FFT on the Magali platform once the RVM was realized.

The performance evaluation results have been obtained on the cycle accurate VHDL simulator of Magali chip. These results are depicted as following:

The sizes of the executable programs corresponding to the three configurations (native, RVM API and RVM) are presented in table 22.2.

Configuration type	Memory footprint (KB)
Native	96
RVM API	100
Complete RVM	212

Table 22.2. Size of the programs for the three configurations of the CFO of IEEE802.11a

The execution time is presented in Figure 22.12. It is clear that the overhead of the RVM API is not very important. This proves that our PLDL fits well with the Magali platform which is a real SDR platform, whereas adding a virtual machine to enable portability has an important cost: the memory footprint is doubled and the execution time is almost multiplied by ten. For a more precise presentation of the performance results, see [ARFM10]

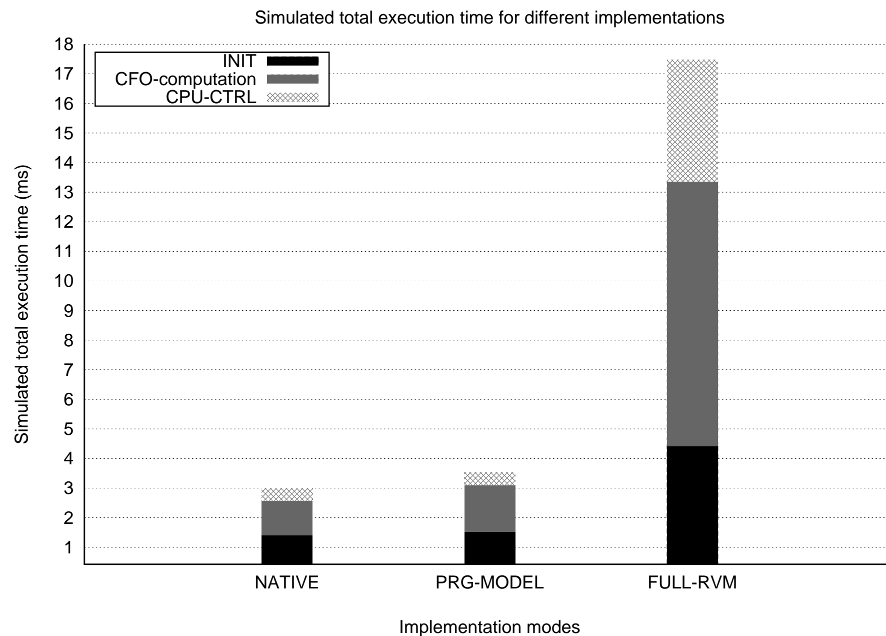

Figure 22.12 Execution time for the three configuration of the CFO of IEEE802.11a

However, this VM implementation is a prototype and can be optimized much more. Analysis of the execution time shows that most of the time overhead is spent in native function calls, which imply access to a hash function, and data type transformation between Lua types and native types. These performance problems can be reduced with the use of well known techniques such as runtime compilation or binary translation techniques [Ayc03, CM96]. Another possibility would be to use hardware accelerators for the chosen virtual machine such as Hard-Int [RJ00], Jazelle [Por05] or picoJava[PS07].

Reference [Abd10] presents a complete 3GPP-LTE receiver implemented on top of the RVM. Real time constraints were not met in this functional demonstrator of the RVM, but as mentioned above, virtual machine optimization techniques can be applied. In any case it is very important to realize that optimization mechanisms present on existing SDR platforms should be taken into account when implementing a waveform description language for a given platform, in order to meet real time and power consumption constraints.

22.4 Conclusion

In this chapter we have briefly presented the technical problems that occur when trying to program a software defined radio system. The complexity and variety of today's hardware SDR prototypes highlights the need of an abstraction layer dedicated to software define radio systems. We have mentioned some attempts to give a common format for software radio programs, then we have investigated more precisely the concept of virtual machine for software radio which seems to be very promising.

Radio virtual machine is attracting because its goal fulfills the requirements mentioned above: write one software radio program which executes on every software radio platform. In the second part of this chapter, we have presented a radio virtual machine prototype developed on the Magali SDR platform at the Leti laboratory. This experiment shows that a radio virtual machine powerful enough to reach real time performances required by modern telecommunication protocols must be optimized from the very beginning of its realization. We have proposed to use just-in-time compilation and binary translation techniques during RVM design and implementation. More generally, memory management and data representation issues should be carefully studied during RVM design.

It is very likely that, as it has been the case for computers or parallel machines, the software tool-chain available on SDR system will have a huge technical and economical impact on the future of communicating objects. Depending on the available computing power, the software models and techniques presented here will not only apply to baseband but also to digital processing part in front-end such as digital pre-distortion, digital up-conversion and down-conversion for instance.

References

[Abd10] Riadh Ben Abdallah. Machine virtuelle pour la radio logicielle. *Thèse de doctorat (PhD Thesis), Laboratoire CITI (Centre d'innovation en télécommunication et intégration de services), INSA de Lyon*, 2010.

[ARFD09] Riadh Ben Abdallah, Tanguy Risset, Antoine Fraboulet, and Yves Durand. The radio virtual machine: A solution for sdr portability and platform reconfigurability. *Parallel and Distributed Processing Symposium, International*, 0:1–4, 2009.

[ARFM10] Riadh Ben Abdallah, Tanguy Risset, Antoine Fraboulet, and Jerome Martin. Virtual machine for software defined radio: Evaluating the software vm approach. *Computer and Information Technology, International Conference on*, 0:1970–1977, 2010.

[Ayc03] J. Aycock. A brief history of just-in-time. *ACM Computing Surveys (CSUR)*, 35(2):113, 2003.

[BK07] J. Bard and V.J. Kovarik. *Software defined radio: the software communications architecture*. Wiley, 2007.

[BM] R. Burgess and S. Mende. Configuration languages-theory and implementation. *E2R Project Whitepaper: http://e2r. motlabs. com/whitepapers*.

[Bur04] R. Burgess. Configuration method, September 28 2004. US Patent App. 10/950,562.

[CB02] J. Chapin and V. Bose. The vanu software radio system. In *2002 Software Defined Radio Technical Conference, San Diego*, 2002.

[CBL+10] F. Clermidy, C. Bernard, R. Lemaire, J. Martin, I. Miro-Panades, Y. Thonnart, P. Vivet, and N. Wehn. A 477mW NoC-based digital baseband for MIMO 4G SDR. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2010 IEEE International*, pages 278–279. IEEE, 2010.

[CLM01] J. Chapin, V. Lum, and S. Muir. Experiences Implementing GSM in RDL (The Vanu Radio Description Language™). In *IEEE Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force*, volume 1, 2001.

[CLT+09] F. Clermidy, R. Lemaire, Y. Thonnart, X. Popon, and D. Knetas. An open and reconfigurable platform for 4g telecommunication: concepts and application. In *12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools (DSD'2009)*, pages 449–456, August 2009.

[CM96] C. Cifuentes and V. Malhotra. Binary translation: static, dynamic, retargetable? In *Proceedings of the 1996 International Conference on Software Maintenance*, pages 340–349. Citeseer, 1996.

[DBL05] Y. Durand, C. Bernard, and D. Lattard. FAUST: On-chip distributed architecture for a 4g baseband modem SoC. In *Design & Reuse IP-SoC, Grenoble, France*, Grenoble, France, December 2005. IEEE Computer Society.

[DTP+05] A. Duller, D. Towner, G. Panesar, A. Gray, and W. Robbins. picoarray technology: the tool's story. In *Design, Automation and Test in Europe, 2005. Proceedings*, pages 106–111 Vol. 3, March 2005.

[Fer02] G.R. Ferris. Digital wireless basestation, July 24 2002. US Patent App. 10/182,043.

[FSC09] Ronan Farrell, Magdalena Sanchez, and Gerry Corley. Software-defined radio demonstrators: An example and future trends. *International Journal of Digital Multimedia Broadcasting*, page 12 pages, 2009.

[GI00] M. Gudaitis and Dr. Joseph Mitola III. The Radio Virtual Machine. *SDR Forum 21st General Meeting. 14-16 Novembrer, Mesa, AZ, 2000*, 2000.

[GMSG08] Ismael Gomez, Vuk Marojevic, Jose Salazar, and Antoni Gelonch. A lightweight operating environment for next generation cognitive radios. *Digital Systems Design, Euromicro Symposium on*, 0:47–52, 2008.

[GRMF05] Antoni Gelonch, Xavier Revès, Vuk Marojevik, and Ramon Frrús. P-HAL: a middleware for SDR applications. In *SDR Forum Technical Conference*, 2005.

[GSBR04] Cyprian Grassmann, Mirko Sauermann, Hans-Martin Bluethgen, and Ulrich Ramacher. System level hardware abstraction for software defined radios. In *Proceeding of the SDR 04 Technical Conference and Product Exposition. SDRForum 2004.*, 2004.

[IDFF96] R. Ierusalimschy, L.H. De Figueiredo, and W.C. Filho. Lua-an extensible extension language. *Software Practice and Experience*, 26(6):635–652, 1996.

[JTR06] JTRS. Software communications architecture specification. standards joint program executive office (jpeo) joint tactical radio system (jtrs), 2006. version 2.2.2.

[Kah74] G. Kahn. The semantics of a simple language for parallel programming. *Information processing*, 74:471–475, 1974.

[KAW+06] T. Kempf, M. Adrat, EM Witte, O. Schliebusch, M. Antweiler, and G. Ascheid. A Concept for Waveform Description based SDR Implementation. In *4th Karlsruhe Workshop on Software Radios (WSR'06)*, 2006.

[LLW+07] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A high-performance DSP architecture for software-defined radio. *IEEE Micro*, 27(1):114–123, 2007.

[LY99] T. Lindholm and F. Yellin. *Java virtual machine specification.* Addison-Wesley Longman Publishing Co., Inc. Boston, MA, USA, 1999.

[MES+07] G.J. Minden, J.B. Evans, L. Searl, D. DePardo, V.R. Petty, R. Rajbanshi, T. Newman, Q. Chen, F. Weidling, J. Guffey, D. Datla, B. Barker, M. Peck, B. Cordill, A.M. Wyglinski, and A. Agah. Kuar: A flexible software-defined radio development platform. In *New Frontiers in Dynamic Spectrum Access Networks, 2007. DySPAN 2007. 2nd IEEE International Symposium on*, pages 428–439, April 2007.

[MKB07] C. Moy, A. Kontouris, and A. Bisiaux. Telecommunication device with software components, May 2007. US Patent 7,212,813.

[Por05] C. Porthouse. Jazelle for execution environments. *ARM Whitepaper, available online*, 2005.

[PS07] W. Puffitsch and M. Schoeberl. picoJava-II in an FPGA. In *Proceedings of the 5th international workshop on Java technologies for real-time and embedded systems*, page 221. ACM, 2007.

[RJ00] R. Radhakrishnan and L.K. John. Microarchitectural techniques to enable efficient Java execution. *Academic Dissertation. University of Texas at Austin*, 2000.

[SCC+06] D. Simon, C. Cifuentes, D. Cleal, J. Daniels, and D. White. Java™on the bare metal of wireless sensor devices: the squawk Java virtual machine. In *Proceedings of the 2nd international Conference on Virtual Execution Environments*, page 88. ACM, 2006.

[TT09] D.C. Tucker and G.A. Tagliarini. Prototyping with gnu radio and the usrp - where to begin. In *Southeastcon, 2009. SOUTHEASTCON '09. IEEE*, pages 50–54, March 2009.

[TZF+09] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G.M. Voelker. Sora: High performance software radio using general purpose multi-core processors. *NSDI 2009*, 2009.

[vBHM+05] Kees van Berkel, Frank Heinle, Patrick P. E. Meuwissen, Kees Moerman, and Matthias Weiss. Vector processing as an enabler for software-defined radio in handheld devices. *EURASIP J. Appl. Signal Process.*, 2005:2613–2625, 2005.

[Wil01] ED Willink. The waveform description language: moving from implementation to specification. In *IEEE Military Communications Conference, 2001. MILCOM 2001. Communications for Network-Centric Operations: Creating the Information Force*, volume 1, 2001.

[YJ] S. Yoo and A. Jerraya. Introduction to hardware abstraction layers for SoC. *Embedded Software for SoC*, pages 179–186.

[ZDSS07] S. Zhong, C. Dolwin, K. Strohmenger, and B. Steinke. Performance evaluation of the functional description language in a sdr environment. In *Proc. SDR Forum Technical Conference 2007*, 2007.