Software Radio Architecture: A Mathematical Perspective

Joseph Mitola, III, Member, IEEE

Abstract—As the software radio makes its transition from research to practice, it becomes increasingly important to establish provable properties of the software radio architecture on which product developers and service providers can base technology insertion decisions. Establishing provable properties requires a mathematical perspective on the software radio architecture. This paper contributes to that perspective by critically reviewing the fundamental concept of the software radio, using mathematical models to characterize this rapidly emerging technology in the context of similar technologies like programmable digital radios. The software radio delivers dynamically defined services through programmable processing capacity that has the mathematical structure of the Turing machine. The bounded recursive functions, a subset of the total recursive functions, are shown to be the largest class of Turing-computable functions for which software radios exhibit provable stability in plug-and-play scenarios. Understanding the topological properties of the software radio architecture promotes plug-and-play applications and cost-effective reuse. Analysis of these topological properties yields a layered distributed virtual machine reference model and a set of architecture design principles for the software radio. These criteria may be useful in defining interfaces among hardware, middleware, and higher level software components that are needed for cost-effective software reuse.

Index Terms—Computability, digital radio, software radio, topology.

I. INTRODUCTION

It is well established that a communications system is the set of devices by which one employs a communications channel to convey information to a recipient. From a radio engineering perspective, the communications device must first encode the information from the source into a suitable electronic representation. The device then must transform this internal form into a waveform compatible with the radio frequency (RF) communications channel. The channel distorts the RF signal, adds noise, and creates time-delayed distorted replicas of the signal. The process is (imperfectly) reversed in the receiver as illustrated in Fig. 1.

In this venerable historical framework, control of the radio needs little attention, generally limited to power on/off, audio volume control, a noise-riding receiver threshold ("squelch"), and a switch to manually select from among predefined RF channels. Several such transmitter(s) and receiver(s) located and working together comprise a radio “node.” The multi-

band, multimode, multithreaded, multipersonality capabilities of software radios require an expansion of this simple model.

A. Essential Functions of the Software Radio

Technology advances have ushered in a new level of physical-layer flexibility of the radio. Multiband technology [1], first of all, accesses more than one RF band of communications channels at once. The RF channel of Fig. 1, then, is generalized to the channel set of Fig. 2. This set includes RF channels, but radio nodes like personal communications system (PCS) base stations [2] and portable military radios also interconnect to fiber and cable; therefore, these are also included in the channel set. The channel encoder functions are expanded to programmable RF/channel access, intermediate frequency (IF) processing, and modem. Antennas and RF conversion that span multiple RF bands (and fiber-optic interfaces) comprise the RF/channel access function. IF processing may include filtering, further frequency translation, space/time diversity processing, beamforming, and related functions. Multimode radios [3] generate multiple air interface waveforms ("modes") defined principally in the modem, the RF channel modulator–demodulator. These waveforms may be in different bands, and may span multiple bands. The source and channel coders of Fig. 1 therefore become the multiple personalities of Fig. 2. A personality combines RF band, channel set (e.g., control and traffic channels), air interface waveform, protocol, and related functions.

Although many applications do not require information security (INFOSEC), there are incentives for its use. Authentication reduces fraud. Stream encryption ensures privacy. Transmission security (TRANSEC) hides the fact of a communications event (e.g., by spread-spectrum techniques [4]). INFOSEC is therefore included in Fig. 2, although the function may be null for many applications.

In addition, the source coder/decoder pair of Fig. 1 must now be expanded to include the data, facsimile, video, and multimedia sources implicit in Fig. 2. Some sources will be physically remote from the radio node, e.g., connected via the synchronous digital hierarchy (SDH) [5], a local area network (LAN) [6], or other network through service and network support of Fig. 2.

These functions are implemented in multithreaded, multi-processor software orchestrated by a joint control function. Joint control assures system stability, error recovery, timely data flow, and isochronous streaming of voice and video. As radios become more advanced, joint control becomes more complex, evolving toward autonomous selection of band,
mode, and data format. Any of the functions may be singleton (e.g., single band versus multiple bands) or null, further complicating joint control. Agile beamforming supports additional users and enhances quality of service (QoS) [7]. Beamforming today requires dedicated processors, but in the future, joint space–time equalization algorithms may time-share a digital signal processor (DSP) pool along with the Rake receiver [8] and other modem functions. Joint source and channel coding [9] also yields computationally intensive waveforms. Dynamic selection of band, mode, and diversity as a function of QoS [10] introduces large variations into computational demand, potentially causing conflicts for processing resources. Channel strapping, adaptive waveform selection, and other forms of data rate agility [11] further complicate the statistical structure of the computational demand. In addition, processing resources are occasionally lost through equipment failures [12]. Joint control integrates fault modes, personalities, and support functions on a limited resource of applications-specific integrated circuits (ASIC’s), field-programmable gate arrays (FPGA’s), DSP’s, and general-purpose computers to yield a reliable telecommunications object [13].

A software radio can upload new air interface personalities. These may modify any aspect of the air interface, including whether the waveform is hopped, spread, or otherwise constructed. The required resources (e.g., bandwidth, memory, and processing capacity), of course, must not exceed those available. Some mechanism for evolution support is therefore necessary to define the waveform personalities, to download them (e.g., over the air), and to assure that each new personality is safe before being activated. To type certify such a radio, one must guarantee that the properties specified by the regulatory bodies are preserved in spite of this high degree of flexibility. The need for such guarantees motivates the study of the mathematical properties of this architecture.

Traditionally, one may model the statistical demand for computational resources versus processing capacity using queueing theory [14]–[16]. Real-time performance can be assured in a fixed architecture using this approach [17]. Plug-and-play, however, creates a variable architecture as modules are introduced into the environment and removed. This raises the complexity of the statistics, particularly in complex nodes such as a base station cell site in which hundreds of users can invoke dozens of variable-bandwidth services in a pool of shared DSP’s. A deeper understanding of the statistical properties of such environments is no doubt needed. Underlying such statistical analyses, there must be a predictable computational-demand relationship between the plug-and-play module and the environment. This calls for a theory of plug-and-play resource bounds for the software radio within which such predictable relationships will exist.

An obvious challenge is to define interface points for plug-and-play hardware and software modules. Industry organizations including the Software-Defined Radio (SDR) Forum (formerly the Modular Multifunction Information Transfer Systems—MMITS—Forum) are in the process of identifying such interface points using generalized applications programmers interfaces (API’s) [18]. A less obvious challenge is to define architecture principles that assure that plug-and-play architectures will have the mathematical properties of controllability and predictability necessary for true plug-and-play services. This paper focuses on that challenge. The relevant properties may be characterized in the following mathematical framework.

**B. Toward a Mathematical Model of Plug-and-Play Architecture**

The thesis of this paper is that the next-generation radio functions of Fig. 2 have a mathematical structure that con-
strains the behavior and interfaces of plug-and-play modules. The mathematical framework abstracts away the details of the radio functions, interfaces, and implementations. It identifies the computational and geometric structure that diverse software, services, and hardware platforms share.

In the proposed framework, modules are modeled as mappings of signals among architecture interface points defined as vertices in a topological space. Topological spaces are very general geometric spaces with a few set-theoretic axioms [19]. Mappings among interface points are represented as edges or arcs in this space. An edge is that which joins two vertices. An RF ASIC, for example, transforms the RF signal at the input vertex into the baseband signal at the output vertex. An arc, on the other hand, is a map that has properties in addition to its vertices, such as the power dissipated by the associated RF ASIC. This model may be used

1) to identify top level plug-and-play interfaces;
2) to predict and control system performance;
3) to define a reference model for standards setting;
4) to derive architecture principles for product evolution.

Mathematical models of multiprocessor task specifications based on topological spaces have been used to prove important properties of shared memory multiprocessors [20]. The model used in this paper is akin to such models. The model in Fig. 3 shows radio hardware and software functional arcs joining the interface points in a topological space. This example shows a dual-band receiver with an analog-to-digital converter (ADC) IF, a channel filter ASIC, and software modem and vocoder (voice coding) functions. Some arcs represent the movement of signals and data from one interface point to another via isochronous, real-time streams (e.g., the high-band RF hardware and modem software of the figure). Others represent the exertion of control over states (e.g., user choice of a channel) and parameters that affect a stream (e.g., the channel filter).

Any important properties of the arcs become dimensions of the topological space. For example, the interface node between the IF ADC and the channel filter would have the value “hardware.” Other dimensions may be continuous in \( R \) the real numbers, such as the range of a control parameter. In addition, there may be infinite-dimensional subspaces such as analog signals. Each subspace represents an aspect of the architecture.

C. Architecture Goals

A successful plug-and-play architecture entails at least the following.

Compatibility: The structure of plug-and-play modules must be compatible with that of the software radio environment—arcs plug into nodes for function composition.

Predictability: Module composition must preserve radio service-defining properties of the system, and when control is exerted, it must not have unintended consequences. Modules must not consume excessive computational resources.

An architecture is a framework in which a set of components is used to achieve specified functions (services) within specified constraints or design rules [21]. Compatibility and predictability are the properties of plug-and-play architecture studied in this paper. A mathematical framework for a plug-and-play architecture should inductively establish the desired properties of a given set of components within specified design rules. Due to the open-ended nature of radio services and technology, the framework must be extensible both to new services and to new implementation platforms. This paper begins the development of this framework with a top-down review of the properties of the software radio. A bottom-up analysis of the computational properties of software radio components then sets the stage for the derivation of the layered virtual machine reference model. Architecture principles that support plug-and-play modularity are developed in the process.

II. DEFINING THE SOFTWARE RADIO

This section defines the software radio from several perspectives: engineering design, topological structure, and computational structure. The programmable digital radio (PDR) is defined and differentiated from the software radio. Topological properties of components are introduced in this section and developed in Sections III and V. In Section IV, the bounded
Fig. 4. Key software radio functions and components.

recursive functions are developed. Section VI derives architecture partitions and design principles.

A. Programmable Digital Radios

There are many ways to impart more than one personality to a radio. The term “PDR” applies to those radios that use a hardware-intensive mix of hardware and software techniques to access more than one RF band with a choice of air interface modes [22]. A PDR’s programmability may be achieved using baseband DSP’s. However, hardware modules or “slices” must be interchanged for the radio to change RF band and air interface mode (“waveform”).

The software-defined radio (SDR) was defined by BellSouth to describe an evolution toward greater programmability of a wireless infrastructure [23]. This evolution includes programmable multiband, multimode radios implemented using the PDR approach. An IS-95/AMPS handset, for example, may employ a code-division multiple-access (CDMA) chip set for IS-95; an analog mobile phone system (AMPS) chip set; a dual-mode RF integrated circuit (RFIC) chip set [24]; a DSP chip [25] for filtering, voice coding, and computationally intensive tasks; and a microcontroller for user interface and system control. Many of the functions are programmable, but the CDMA despreader, for example, is defined in an ASIC, and may not be changed in the field. Over time, functions initially implemented in hardware (e.g., the RF modem) will migrate to software and conversely. The mathematical framework must therefore capture SDR migration between hardware and software.

B. The Software Radio

The software radio imparts multiple personalities to a radio in a specific way [26]. A software radio defines all aspects of the air interface including RF channel access and waveform synthesis (not just selection) in software. In the software radio, then, wide-band analog-to-digital and digital-to-analog converters (ADC’s [27] and DAC’s) transform each RF service band among digital and analog forms at IF as in Fig. 4. The wide-band digitized receiver stream of bandwidth $W_s$ accommodates all subscriber channels, each of which has bandwidth $W_c$ ($W_c \ll W_s$). The mathematical framework differentiates alternative ADC and DAC architectures, with the attendant differences in programmability.

In the software radio of Fig. 4, IF ADC and DAC channels are processed isochronously using programmable digital hardware, and software.² IF processing may include filtering to isolate subscriber channels [28]; digital formation of nulls and beams [29]; joint space–time equalization [30]; integration of space diversity [31], polarization, or frequency diversity channels [32]; and other means of acquiring a high-quality waveform. There may be multiple IF frequencies (in the heterodyne receiver) or the IF frequency may be zero (in the homodyne receiver [33]). IF processing may be null, for example, in a direct conversion receiver [34]. Digital downconversion is the process of using the frequency-domain periodicity of sampled bandpass waveforms to translate the

²Software here means algorithms stored in random access memory of processors which may, in principle, be downloaded over the air. Firmware is software to the degree that it can be downloaded; otherwise, it is part of the hardware personality.
waveform to baseband without analog heterodyning [35].
Preselection filters with the necessary 80+ dB of out-of-band attenuation are becoming available in superconducting RF filters [36]. Local oscillator leakage may also be reduced proportionally so that digital downconversion is becoming more feasible. The mathematical framework represents plug-and-play aspects of such variations via sets of arcs and nodes with subset relationships.

In the software radio transmitter, baseband signals are transformed into sampled channel waveforms via channel modem functions implemented in software to drive high-performance DAC’s. These signals may be preemphasized or nonlinearly precoded [37] by the IF processing software. In some implementations, modem functions, IF processing, and RF channel access may be amalgamated into a single component such as a direct conversion receiver RFIC [34]. In addition, dynamic compilation of software or real-time switching among personalities can allow these discrete functions to be integrated into a single component such as an FPGA [38]. The mathematical framework represents both discrete and integrated implementations. In discrete implementations, each edge in the topological space is also an arc corresponding to a component. In an integrated implementation, a component arc subsumes many topological edges.

C. A Software-Equivalent Model of Hardware

The ADC and DAC define the point at which functions are potentially software defined. We may call this the digital access point. Relatively inflexible analog and digital hardware such as ASIC’s may be necessary in a handset, e.g., for direct conversion of the channel waveform to a bitstream. Although an ASIC may have a few control parameters, the personality of an ASIC cannot be changed in the field (e.g., from one air interface standard to another). To change the air interface personality, one must replace the ASIC, switch among different ASIC’s, or switch among modes of one ASIC. In the tradeoffs among size, weight, power, cost, and flexibility, fixed-function ASIC’s generally consume less power, take less space, weigh less, and cost less per device than the programmable DSP equivalent. Therefore, increased flexibility comes at a price. One way of representing these differences mathematically is to attribute equivalent computational capacity to the nonprogrammable devices.

In the topological model of the hypothetical radio illustrated in Fig. 5 (top), an RF ASIC (A) transforms the analog signal from the antenna directly to input for the baseband modem. The software-equivalent model (B, bottom) postulates an RF ADC with exactly the bandpass characteristics of the RF ASIC. The frequency translation behavior of RF ASIC (A) is modeled as the frequency translation software process of model (B). The filtering behavior of the RF ASIC is modeled by the channel filter software of (B). The sequence of RF ADC, frequency translation, and channel filter arcs constitute the software-equivalent model of this ASIC. The equivalent computational capability of RFIC (A) is the total computational demand of the postulated software processes (B). Since analog hardware has variable performance over time and over different devices, the exact filter model of a given RF ASIC will change over time. Thus, each device is represented in a set of models. The variations are contained within “open balls” [39] in the parameter subspace of the topological model (Fig. 6). One may also aggregate multiple device models into such a parametric model by taking set unions. The set of all such arcs and open balls comprises the topological model of the ASIC. Each model may also have parameter sets in dimensions that characterize or predict such diverse properties as time delays and power dissipation.

Although the sequel focuses on computational properties, this set-theoretic framework is general, capable of representing local properties of a device including interface details, size, weight, and power. Since topological spaces are not necessarily linear, aggregate properties of the radio system in the higher level spaces may have nonlinear relationships to the properties of the components in the lower level spaces.
D. Quantifying Degrees of Programmability

The degree of programmability of an implementation is fundamental to software radio architecture. Since contemporary radios use a mix of processor types, one must characterize this mix precisely. Consider the highest level topological model of a radio (a single arc). This arc may be hierarchically divided into its primitive components. Hardware primitives are the discrete devices, while software primitives are single machine instructions. Primitives that may be redefined via software in the field are labeled. The number of labeled primitive arcs divided by the total number of primitive arcs is a measure of programmability. Since an ASIC’s programmability is limited to the modification of a few parameters, most of its gate-level arcs will not be labeled.

FPGA’s are, in principle, completely programmable. In practice, they are more programmable than ASIC’s, but subject to gate and interconnect constraints. Programmable radios have been based almost entirely on reconfigurable FPGA’s [40]. The field programmability of an FPGA is more constrained than that of a DSP chip, in part because of the likelihood of running out of usable gates on the FPGA as radio functions expand. The topological model of each type of device allows one to characterize programmability more precisely. In the FPGA topology of Fig. 7, two dimensions (corresponding to states and interconnect) represent the allocation of functions to hardware.

Suppose that an advanced timing recovery algorithm, comprising, say, 10% of the FPGA area is to be downloaded to the radio. As shown in the figure, its needs are incompatible with the gate use of the existing timing recovery logic. Assume that one redefines the personality of the FPGA to accommodate the new logic. The download size increases from the 10% needed for the increment to 100% for the entire personality, a 900% increase in the size of the download. It is also possible that a moderately populated (70%) FPGA will be unable to accommodate the 10% download because of hardware constraints such as the required placement of I/O buffers, lack of registers, etc. A similar topological model of a DSP chip is shown in Fig. 8. The additional 10% of DSP code associated with the advanced timing recovery logic (now implemented in software) is accommodated via space available in the nonvolatile random access memory (RAM). The memory map allocates this logic to RAM hardware. DSP may appear easier to program than FPGA’s because RAM allocation is accomplished by loader software while gate allocation in an FPGA generally requires an experienced designer. New waveforms also seem to outgrow the gates of an FPGA more easily than they outgrow the program memory of a DSP chip. On the other hand, the timing constraints of DSP software are more severe than the timing of an equivalent FPGA. Logic in an FPGA runs at nearly the chip’s clock rate, while the DSP code runs from one to three orders of magnitude slower than the system clock. The tradeoff of ASIC hardware allocation and DSP task timing is reflected in these subsets of the topological model.

Complex or reduced instruction set computers (CISC/RISC) provide less hardware acceleration than DSP chips. To quantify
the degree of flexibility of DSP, CISC, and RISC processors, one again defines an appropriate topological space. Let \{ISA\} be the space of single-instruction register-state transformations in a processor with a given instruction set architecture (ISA). From an arbitrary initial state, a DSP has many more edges connecting reachable data states than a CISC processor, which in turn has more arcs than an RISC processor. So, from the perspective of \{ISA\} topology, the RISC processor is the simplest, and thus in some sense, the most general programmable platform. CISC is more complex, and DSP the most complex of the fixed ISA machines. Performance of high-quality code underscores these differences. DSP code that employs zero-overhead loops with full register stacks and processing elements yields higher throughput than CISC code of a processor with the same system clock, memory, and I/O delays. FPGA’s, on the other hand, effectively have a variable ISA that makes them even more computationally efficient than DSP machines, with an attendant loss of flexibility. That is, once we employ an FPGA’s gates for a specific task (e.g., a digital filtering algorithm), those gates in that configuration cannot be used for a different task (e.g., a timing recovery algorithm). A reconfigurable FPGA’s gates may support both functions, provided reconfiguration delays are tolerable. DSP chips use the same gates for both functions time shared through software.

These relationships define a phase space for the software radio. Physicists use a phase space to represent states of a substance (e.g., solid, liquid, and gas) as a function of parameters such as temperature and pressure. The software radio phase space of Fig. 9 represents the states of radio implementations as a function of the digital access point and the degree of programmability.

Fig. 9 places five types of radios in the phase space. The vertical axis represents the bandwidth at the digital access point. The horizontal axis represents the degree of flexibility, the fraction of functionality that may be changed in the field using plug-and-play software. To place a system in the phase space, one examines the ADC’s and DAC’s, placing the digital access point where the functionality is fully programmable. One places the radio on the horizontal axis according to the aggregate degree of programmability of the device.

The HF STR-2000, a commercial product of Standard Marine AB, shown at point [A], employs baseband DSP using the TMS20C30. Most commercial off-the-shelf (COTS) cellular telephone handsets fall near [D]. ASIC’s provide much of the equivalent processing capacity, shifting these designs toward the less programmable end of the axis. Current digital cell site designs [C], similarly, rely heavily on digital filter ASIC’s for frequency translation and filtering, but they access the spectrum at IF. SPEAKeasy II, [D], provides a GFLOP of programmable DSP, shifting this implementation to the right. A product of research at the Massachusetts Institute of Technology, the virtual radio [41] delivers a single channel radio using a general-purpose processor, DEC’s alpha. This is the most general-purpose computing platform reported in the literature. Point [X] represents the ideal software radio with the digital access point at RF and all functions programmed on RISC processors. Although maximally flexible, and thus of research interest, such designs tend to be economically impractical. This phase space quantitatively differentiates software radios in the upper right quadrant from the PDR’s elsewhere in the figure.

A PDR is not a software radio if any crucial aspect of the channel waveform is implemented using programmable hardware (such as a voltage-controlled oscillator) rather than using software (e.g., sin/cosine lookup table). A joint tactical information dissemination system (JTIDS) radio [42] implementation, with a 3 MHz IF from which hop frequencies are generated using a 250 MHz programmable local oscillator, is not a software radio for that waveform. On the other hand, a 250 MHz digitized IF that could set every hop frequency in software would be a software radio. The radio may meet the software radio criterion for a large class of narrow-band waveforms, while failing for wide-band waveforms. The phase space of the topological model thus unambiguously defines the degree to which an implementation is reprogrammable. Given an unambiguous definition, one may consider the components in more detail.
III. TOP-LEVEL COMPONENT TOPOLOGY

Radio components are represented as arcs in the topological model. An arc may be a union or composition of other arcs, defining a natural encapsulation hierarchy for the radio system. At the top level of the hierarchy (the "context" level in object-oriented design), the radio node is a black box mapping air interface, user, and network events to appropriate responses. The functions of Fig. 2 define the second level of partitioning with the component attributes highlighted in Table I. The attributes listed in this table indicate the allocation of functions
to top-level components. In a topological space, each allocated function may be a singleton, a subset, or the null set. This corresponds to the occurrence of a function implemented in a single component, implemented multiple times (e.g., in parallel channels) in multiple components, or not implemented at all. The remarks column highlights functional diversity leveraged by the software radio architecture.

The stream-oriented functional components are source coding and decoding, service and network support, INFOSEC, RF/channel modem, IF processing, and RF/channel access. Each is represented topologically by a pair of arcs between data interfaces, the topological domain, and range of the maps. A topological model of a specific radio consists of those arcs that correspond to the components. For example, the signal-stream topology of a dual-mode handset is shown in Fig. 10.

The critical interfaces among dual-band antenna, IF ADC, baseband modem, and vocoder of this notional device are readily apparent in the topological model. Additional interfaces associated with the functions of Table I are shown in Fig. 11. Analog (audio and video) waveforms comprise the interface between the source set and the source coding and decoding functions. Source bits are encoded, but services and network support adds forward error control structure. If INFOSEC is null, protected bits become clear bits. Interfaces may be null at one level and visible at another. An RFIC, for example, may subsume the IF waveform interface, exhibiting only baseband and RF waveforms in the higher level of the topological space.

Implementation topologies are constrained by these interfaces and related standards as summarized in Table II. Interfaces are represented in a space which includes the interface signal itself and related meta-level characteristics: implementation class (hardware or software), impedance, connector type, bandwidth, etc. The analog stream has infinite dimensionality in both time and amplitude, defined over $\mathbb{R}$. A digitized analog stream consists of an infinite stream of one-dimensional vectors, the samples. But, given a pair of sample buffers of length $N$, this stream is reduced to dimension $2sN$.

In addition, as shown in Fig. 12, the topological model structures the meta-level aspects of these interfaces. Each dimension of the space must be identified, and the associated elements that apply must be specified. The syntactic and semantic definition of the topological spaces of such a generic radio would constitute a knowledge representation language (KRL) [47] for radio (an RKRL). The definition of a complete, extensible RKRL is a significant undertaking that is beyond the scope of this introduction. Important properties of plug-and-play interfaces may be defined without a complete RKRL, however. In particular, general properties of the interfaces and...
TABLE II

<table>
<thead>
<tr>
<th>Interface</th>
<th>Key Characteristics</th>
<th>Topological Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td>Analog Stream</td>
<td>Audio, video, facsimile streams</td>
<td>Infinite dimensional; Filtering constraints comprise open ball(s)</td>
</tr>
<tr>
<td>Source Bitstream</td>
<td>Coded bitstreams and packets. ADCs define a finite window into a quantized, discrete-time sampled waveform</td>
<td>Finite dimensional; frame and data structure defines subspaces. Finite precision defines a Dynamic Range Subspace(^2) for the ADC</td>
</tr>
<tr>
<td>Clear Bitstream</td>
<td>Framed, Multiplexed, Forward Error Controlled (FEC) bitstreams and packets</td>
<td>Finite dimensional; FEC subspaces have rich algebraic properties</td>
</tr>
<tr>
<td>Protected Bitstream</td>
<td>Random challenge, authentication responses, public key, enciphered bitstreams and packets</td>
<td>Finite dimensional; randomized streams; complex message passing for downloads; If null, interface reverts to clear bits</td>
</tr>
<tr>
<td>Baseband Waveform</td>
<td>Discrete time synchronous quantized sample streams (one per carrier)</td>
<td>Digital waveform properties determine fidelity of analytic representation</td>
</tr>
<tr>
<td>IF Waveform</td>
<td>Composite, digitally pre-emphasized waveform ready for up-conversion</td>
<td>Analog IF has infinite dimensional topology; Digital IF may have baseband product topology</td>
</tr>
<tr>
<td>RF Waveform</td>
<td>Power level, shape, adjacent channel interference, etc. are controlled</td>
<td>Analog RF has infinite dimensional topology; Includes spatial and temporal dimensions</td>
</tr>
<tr>
<td>Network Interface</td>
<td>Packaged bitstreams may require ATM, SS7, or ISO protocol stack processing</td>
<td>Synchronous Digital Hierarchy (SDH), Signaling System 7 (SS7) subspaces</td>
</tr>
<tr>
<td>Joint Control</td>
<td>Control interfaces to all hardware and software, initialization, fault-recovery</td>
<td>(Not illustrated in the figure) Parameter spaces; non-linear logic subspaces</td>
</tr>
<tr>
<td>Software Objects</td>
<td>Download from evolution support systems</td>
<td>Represents binaries, applets; includes self-descriptive language subspaces</td>
</tr>
<tr>
<td>Load/Execute</td>
<td>Software object encapsulation</td>
<td>Download topologies are highly nonlinear</td>
</tr>
</tbody>
</table>

\(^2\) A Nyquist-dynamic range subspace has been sampled so as to meet the Nyquist criteria for the bandwidth of the sampled signal; it has been quantized with sufficient accuracy for the two-tone spurious-free dynamic range of the application.

**IV. COMPUTATIONAL PROPERTIES OF FUNCTIONAL COMPONENTS**

A fundamental aspect of software radio architecture is the set of conditions under which plugging in a module results in acceptable use of computational resources. The composability of functions defined over general topological spaces is well understood. Homeomorphisms are topology-preserving maps that may be composed to yield other homeomorphisms provided the domain of one map and the range of the other are topologically compatible [39]. From a topological perspective, computational components can be studied independently of an RKRL. In addition, air interface specifications and radio API's constitute informal RKRL's.

Such interface domain descriptions could promote the assured delivery of services in the software radio’s plug-and-play environment. A plug-and-play interface should effectively separate the module from the rest of the system. Effective separability implies at least the following topological properties:

1) Composition of module functions on the interface topology yields a well-defined system-level function consuming specified resources to deliver the intended service.

2) Performance (e.g., spectral purity, data formats, throughput, response time, etc.) under function composition is within specified bounds.

The separability of modules at plug-and-play interface points is illustrated in Fig. 13. Plug-and-play modules may comprise top-level functional components such as the modem. Or, they may be defined across arbitrary points deep in the hierarchy such as at the vocoder.

The goal is that the system as a whole behave as desired if the plug-and-play modules and the host system each has the prescribed testable local properties. What properties can be guaranteed? The composite system should use bounded computational resources with guarantees against wait-induced fault conditions. This property is necessary for isochronism of voice and video streams and acceptable timing of packet data flows. This property is also essential for type certification by induction.

---

**Domain:**

**Class:** Analog-Stream  
**Implementation:** Hardware  
**Signal Interface:** Coax-DC-Coupled  
**Impedance:** 50 Ohms ...  
**Carrier Frequency:** Baseband

**3dB Bandwidth:** 350 kHz  
**Signals:** [Promoted-Signal]  
**Interface-Signal:** x(t)  
**Control Parameters:**  
- Gain: 0dB to 20 dB  
- Gain Control: AGC ...

---

Fig. 12. Illustrative meta-level topological space for analog stream interface, a topological domain.
the domain of software consists of those inputs over which its results are defined. The range consists of the corresponding results (including side effects and returned values). Under what conditions can well-behaved software be composed with other well-behaved software to yield a well-behaved system? This very general question may be undecidable. But software radios are engineering systems with timing constraints that allow us to prescribe constraints on the topological structure of the software, and to establish conditions under which the composition of software modules consumes finite resources that can be predicted accurately.

The concept in addressing bounded resource use is to construct the largest set of functions for which predictable finite resource consumption may be guaranteed. In addition, the composition of such functions must also be bounded for a proof that defines conditions under which plug-and-play modules will not use excessive resources. Furthermore, any constraints on the radio functions that can be computed with such functions must be specified. The analysis of this section shows that the bounded recursive functions both guarantee predictably bounded resource use and present no constraints on radio functionality. A system constructed using these functions has the ability to maintain predicted throughput across dynamic changes of plug-and-play software modules.

A. Models of Computation

Earlier in this paper, the notion of software-equivalent hardware capability was introduced. To each ASIC and FPGA may be attributed the computational capability of a canonical processor, which may in turn be modeled as a (collection of) von Neuman machine(s). Each procedure encoded in software on a von Neuman processor may be modeled in terms of computability and concrete complexity as a random access machine (RAM) [48]. A RAM consists of a state machine, input, and output arrays; internal registers; the capability to load and store data via direct and indirect addresses; and the capability to increment memory values. The RAM model provides an intuitive but mathematically precise description of a single von Neuman processor. The RAM model has also been proven equivalent to the recursive function and Turing machine models of computing. Important theoretical properties include the ability to simulate in polynomial time capabilities that require exponential time on Turing machines [49]. Let \{RAM\} represent the RAM model of the instruction set architecture \{ISA\} of an arbitrary processor. Theorems outlined below use the recursive function and the RAM models to establish tight upper bounds on processing resources, a necessary condition for predictable throughput.

B. The Primitive Recursive Functions

The recursive function model of computing consists of a set of functions from \( \mathbb{N} \), the natural numbers, onto \( \mathbb{N} \) with closure properties that define classes of functions. The \textit{primitive recursive functions} consist of the following:

1) The \textit{zero} function \( z : \mathbb{N} \rightarrow \mathbb{N} : z(x) = 0 \), which yields the constant zero for any natural number \( x \) in \( \mathbb{N} \).

2) The \textit{successor} function \( s : \mathbb{N} \rightarrow \mathbb{N} : s(x) = x+1 \), which associates a successor to each \( x \) in \( \mathbb{N} \).

3) The \textit{projection} function \( U^n_m : \mathbb{N}^m \rightarrow \mathbb{N} : U^n_m(x_1,x_2,\ldots,x_n) = x_m \) by which arguments may be selected, with closure under:

4) \textit{Composition}: for \( g_1, g_2 \cdots g_n : \mathbb{N}^n \rightarrow \mathbb{N} \) and \( h : \mathbb{N}^m \rightarrow \mathbb{N} \), \( f(x) = h(g_1(x), g_2(x), \ldots, g_m(x)) \), where \( (x) \) is brief notation for \( (x_1, x_2 \cdots x_n) \); equivalently, \( f = h^g(g_m) \).

That is, the set is closed under composition of functions. One can compose functions in a natural way that corresponds to function calls and/or selection of data inputs in a random access machine.

5) \textit{Primitive Recursion}: for \( g : \mathbb{N}^n \rightarrow \mathbb{N} \), \( h : \mathbb{N}^{n+2} \rightarrow \mathbb{N} \), and finite \( y > 0 \), \( f(x,y) = g(x,y) \) if \( y = 0 \), and

\[
  h(x,y-1), f(x,y-1)
\]

for \( y > 0 \).

That is, the set of functions is closed under primitive recursion. Primitive recursion for \( g = 1 \) and \( h(x,0), f(x,0) = h(x) \), \( f \) is simply the conditional execution of \( g \) or \( h \) based on the value of \( y \) [i.e., since \( f(x_1,0) = g(x_1) \), but \( f(x_1,1) = h(x_1,0, g(x_1)) = h(x_1) \)]. This recursive pattern is equivalent to the if–then–else programming construct in terms of effective computability. Since \( f \) appears in its own definition, it requires a pushdown stack of successive arguments. But since initially \( y > 0 \) and \( y \) decreases at each invocation, the procedure will terminate, and a stack which is as large as \([y]\) will not overflow. It is assumed that the stack allocator in a specific \{ISA\} keeps the finite stack from overflowing, provided an upper bound for \([y]\) is known in advance. The knowledge of such bounds is a feature of the software radio application of this theory.

The \textit{primitive-recursive functions} are the smallest set of such functions closed under composition and primitive recursion. Sequential logic, transversal and recursive filters, bit manipulation, and data packing are common primitive-
recursive functions of software radios. They include addition, subtraction, multiplication, and others that do not require iterative search.

C. The Total Recursive Functions

Iterative loops are not primitive recursive, but they are essential to radio software. The simplest model of iteration is bounded minimalization.

6) Bounded Minimalization: Let \( g : \mathbb{N}^{n+1} \rightarrow \mathbb{N} \) be primitive recursive; then

\[
 f : \mathbb{N}^{n+1} \rightarrow \mathbb{N} : f(x,y) = \mu z \{ z < y \} [ g(x,z) = 0 ]
\]

is also primitive recursive.

Read \( \mu z \) as "\( f(x,y) \) is the least \( z \) less than \( y \) for which \( g(x,z) = 0 \)". The class of functions closed under composition, primitive recursion, and bounded minimalization is total. That is, they are defined for all \( \mathbb{N} \). The Fortran do loop exemplifies bounded minimalization. Ackerman’s function [50] is defined everywhere, but uses resources faster than any known function due to the structure of its recursive calling sequences. It is not considered primitive recursive, although it is a total function. Yet, for small finite parameters of Ackerman’s function which are known in advance, one can determine whether the function will exceed an upper resource bound by using a suitable step-counting function. Any primitive recursive function induces a primitive-recursive step-counting function. Let the bounded primitive recursive functions\(^5\) be the primitive-recursive functions with the associated step-counting functions and finite resource bounds. In software engineering terms, this model of computing insists that when a module is tested, its resource use in isolation [e.g., millions of instructions per second (MIPS)] be tested and resource bounds be established as a parameter of the module. Such characterization is regularly used software-development practice for real-time systems. Integer division, exponentiation, generation of primes, finite logic predicates, and finite iteration are examples of bounded primitive functions. These functions have properties attractive for software radio implementations as expressed in the following theorems.

Theorem 1 (Primitive Bounded Resources): Any bounded primitive recursive function is equivalent to a RAM program which terminates in a finite number of steps which can be bounded tightly from above, given bounds \( y_i \) of the minimalizations from which the composite functions are created.

Outline of the Proof\(^6\): Associate a sequence of RAM instructions with each bounded primitive function and closure construct. Each corresponding RAM instruction sequence is primitive recursive and bounded. These RAM instruction sequences are like in-line subroutines. Each such sequence has a small finite number of steps (the bound). Compute the maximum concrete complexity of the recursive function structure from the ISA and a suitably chosen step-counting function [51]. Let the number of instructions from RAM be the step-counting function. The resources used by any bounded recursive function with a given set of calling parameters can be computed from the calling parameters and the step-counting function.

Lemma 1 (Primitive Bounded Execution Time): The dedicated-processor RAM-equivalent of any bounded primitive recursive function executes in an amount of time that may be tightly bounded. One determines the maximum execution time of each class of RAM instructions on a specific ISA. The execution time bound is the product of the number of steps from Theorem 1 times the maximum execution time for that class of instruction. The time used by the function tested in isolation on a specified dedicated machine is thus an alternative step-counting function. From a software engineering perspective, this lemma states that the bounds allocated may be specified in terms of execution time on a dedicated processor.

Relevance: Software radio incorporates isochronous streams for which there is a fixed timing window during which specific functions must be accomplished in order for services to be delivered properly and/or for the system to remain stable. Modules whose structure is not constrained as above may use resources that are bounded on test cases, but that “blow up” in other conditions. All primitive recursive functions can be guaranteed to either: 1) complete within a specified window or 2) be easily computed in advance to be incapable of meeting that timing window (Lemma 1). If the system control algorithm predicts such a fault, it can signal the network or the user and preclude the (plug-and-play) module from being invoked. Such predictable timing properties provide the foundation necessary to allocate computational resources to plug-and-play modules safely. Queueing theory [14] translates service time measured in isolation into statistical bounds on isochronous performance in the service environment where other tasks are sharing the same processor(s). The control algorithm may employ such techniques to establish a statistical bound on aggregate resource use which, if exceeded, could degrade services. But since the resource bounds may be computed, the system should not become unstable.

D. Bounding the Partial Recursive Functions

Bounded primitive recursive functions do not express all of the programming constructs needed in software radios. Notably absent are the while and until loops. These loops search for a condition under which to terminate. This condition may never occur, so the RAM programs may loop forever. Hardware and software processes that wait or search for a condition that may not occur are computationally equivalent to while or until loops. Current research in wait-free computation considers the related problem of defining instruction sequences (“protocols”) which assure that any process that itself is not in a fault condition will terminate in a finite number of steps [58]. The source code of software radios like SPEAKeasy I reveals the relatively widespread use of wait-
prone constructs, e.g., waiting for sufficient signal strength to initiate receiver processing. The process-dispatch loop of the kernel operating system may “consume” infinite resources to deliver resources to applications software. Other infinite loops are computationally equivalent to the partial recursive functions. The computational structure is called (unbounded) minimalization.

7) Minimalization: for \( g: \mathbb{N}^{n+1} \to \mathbb{N}, \)

\[ f(x_n, y) = \mu y[\mathcal{g}(x_n, y) = 0] \]

that is, \( y \) is “the least \( y \) for which \( \mathcal{g}(x_n, y) = 0 \) is zero.”

The search operator \( \mu \) will continue to increment \( y \) and test \( g \) without bound. The partial recursive functions are the smallest set of functions closed under minimalization. Unlike bounded minimalization which consumes finite resources, the extent of which is computable in advance, unbounded minimalization consumes resources until \( g \) is satisfied. For many situations, \( g \) may never be satisfied. For example, a carrier detection loop that is waiting for a signal that will never be transmitted due to an impossible RF value will never terminate. It will keep using resources until someone steps in to force the release of the resources. The partial recursive functions have been proven equivalent to the Turing computable functions, the Post productions, and to the RAM model [52]. These functions compute essentially anything that we know how to compute.

Theorem 2 (Unbounded Loops): Software radio functions implemented with \( \text{while} \) and/or \( \text{until} \) loops or their equivalents (e.g., implemented using go-to programming styles) cannot be guaranteed to use bounded computational resources.

Outline of Proof: Construct a \( \text{while} \) loop that simulates minimalization:

While(\( \neg(\mathcal{g}(x_n, y) = 0) \)), increment \( y \); Return \( y \);

One may show the equivalence of \( \text{until, while, and for} \) loops to unbounded minimalization. They thus may be undefined, and are not total. The equivalent RAM procedure may consume unbounded computational resources, “crashing” the system.

The instruction set \{RAM\} of a programmable processor always includes instructions by which such unbounded loops may be constructed. There are, however, practical ways that radio engineers and real-time programmers have devised for assuring system stability in spite of the regular use of such constructs. One may assign time limits to each search condition. A time-out condition causes an error state that deals with the excessive time used. Such programming techniques informally convert unbounded minimalization to bounded minimalization. The time limits must include the statistical effects of competition for resources with functions that have higher priority.

Radio applications are distinct from general-purpose computing applications in that there is a rich set of timing constraints imposed on the creation, emission, reception, processing, conversion, network interoperation, and user interface of radio signals. These timing constraints are often represented in message sequence charts and state machines. Timing constraints apply to call setup and control; delivery of isochronous voice, video, and multimedia streams; packet networking; and related control. This is a heavily time-constrained applications domain. The constraints imply timing, and hence resource bounds that can be allocated to individual hardware and software components. These bounds constrain the logic (software and hardware) to tightly bound computational structures without the loss of radio functionality. The formal assignment of such bounds is the basis for a theory of plug-and-play software radio stability as follows.

Theorem 3 (Local Feasibility of Bounded Partial Recursion): A single-threaded single processor software radio function may be synthesized in RAM instruction sequences equivalent to a resource-constrained subset of the partial recursive functions.

Outline of Construction (Locally Bounded Partial Recursion): This proof builds on Theorems 1 and 2. Unconstrained \( \text{While} \) and \( \text{Until} \) loops and their “go-to” equivalents which cannot be guaranteed to terminate are precluded. Each such construct from a conventional implementation is allocated a maximum number of iterations (or, equivalently, a maximum processor time). These limits are coded into \( \text{bounded-while} \) or \( \text{bounded-until} \) loops as indivisible operations (in the same way one that a semaphore is an indivisible operation). Hardware that waits until a condition is met is similarly bounded, redesigned to generate a fault interrupt if the condition is not met after a specified time interval. Watchdog timers exemplify this principle.

Hardware and software “read” operations that wait for a response from a port (which may wait forever) are also precluded. Poll and bounded-wait interrupt handshakes are substituted. In these, the system will continue (with the read task suspended if the port has no data available) to be interrupted with a read failure condition if the read timer expires. “Read” is thus allowed in which the reading software task waits a specified time to be interrupted, and then returns an error condition. Write operations generate interrupts in the receiver and acknowledgment to the transmitter. Since there are RAM instruction sequences that implement these time-constrained search/wait loops, they are guaranteed to complete (either successfully or in a time-out state) in an amount of time that can be tightly bounded. This construction constrains the structure of the software away from unbounded minimalization, and hence from partial recursion to the set of functions called bounded partial recursion. Since this construction applies to software within a given von Neuman processor, the proof establishes the local feasibility of bounded partial recursion (i.e., within a single processor).

Theorem 4 (Totality): The bounded partial recursive functions are total.

Outline of Proof: Total functions are those that are defined for all \( \mathbb{N} \). Each of the primitive recursive functions is defined for all \( \mathbb{N} \). The closure under composition, primitive recursion, and bounded minimalization is defined for all \( \mathbb{N} \). Bounded minimalization is defined for all \( \mathbb{N} \) as follows. Associate with each range of arguments of a minimalization a step-counting function with a limit \( t_{\text{max}} \). If the search terminates in fewer than \( t_{\text{max}} \) steps, the loop yields the result of the minimalization. If not, then the result is \( \text{FAIL} \) for all
Lemma 2 (Bounded Stability): The RAM instruction sequences that are equivalent to the bounded recursive functions are total. (Proof: If not, then the RAM model is not equivalent to the recursive function model, which would be a contradiction.)

Relevance: A software radio has to deliver services predictably. When combinations of inputs and states occur for which responses have not been defined, the system’s behavior is unpredictable; hence, it will produce undesired results, and may ultimately go into an unrecoverable “crash” state. Software radios—in particular those with downloadable applications modules—have many modes and control parameters which result in a combinatorial explosion of control states, making it impossible to test every combination of states in advance. However, if the software radio is constructed of modules that are each guaranteed to be total, then any state combinations are also total by induction. Thus, some specific system behavior (non-crash) will be defined under all conditions. The results of such stable computations may or may not be as intended (i.e., “correct”), but if some specific state is defined, behavior will be repeatable. Unintended results, then, may be more readily diagnosed and corrected. Crash states due to undefined unstable behavior, on the other hand, often require heroic labor-intensive debugging efforts. Radio systems fielded with software that is not total appear unreliable to the user.

Theorem 5 (Global Bounded Recursion): Multithreaded multiprocessor software radio functions implemented using RAM sequences that are equivalent to the bounded recursive functions may be guaranteed to run to complete or to cause a resource fault. The number of instructions before a fault may be tightly bounded using polynomial resources to compute the bound.

Outline of Proof: (Existence of Globally Bounded Recursion): Nielson and Nielson [53] present a process algebra for a polymorphic subset of concurrent ML, a parallel multiprocessor language [54, 55]. With this, they show that the number of processes and interprocessor communications channels associated with a subset of concurrent ML is finite. They point out that their analysis is not complete: some programs with finite resource use will be rejected as potentially infinite due to an inability to infer the finiteness of sequences of (unbounded) recursions which terminate. In addition, their semantics admit sequences in which an infinite number of tasks terminate before a finite number begins, again resulting in an erroneous rejection of a finite resource use case.

Substitute bounded recursion into their schema to preclude the partial recursive constructs, rendering the analysis complete. The formal language (the subset of concurrent ML) and semantics constrain the entire system (all processors, applications, and the concurrent ML) to bounded recursion. In addition, the above guarantees that a step-counting function exists, but not that it will be effective in halting a process that is consuming inordinate resources. Therefore, global bounded recursion requires a two-level priority system in which the step-counting function can interrupt the function being monitored. An infinite regression can occur when step-counting functions themselves could consume excessive resources. This may be curtailed in building software radios by constraining the step-counting function to use the real-time clock. The existence proof is therefore a first-order result (it assumes two levels of priority and only isolated clock faults), not a completely general result. But it isolates the risk of exceeding resources to the reliability of the real-time clock (versus the millions of lines of code of the radio applications). It is much easier to make highly reliable clocks than to make error-free totally defined software.

Generalizations: The degree to which Theorem 5 may be generalized to the asynchronous multithreaded heterogeneous multiprocessors of software radios touches on much current research in computer science. The degree of further generalization is, no doubt, limited. Communications paths, data-sharing mechanisms, and process control protocols (e.g., Compare and Swap, Load Linked/Validate/Store Conditional [56]) would have to be bounded in a more general setting than [53] for a fully general proof. Topological models of computation have produced relevant results including necessary and sufficient conditions for wait-free protocols [57], the asynchronous complexity theorem [58], nonuniform and uniform iterated intermediate snapshot models of wait-free computation [59], and related results. In addition, practical nonblocking primitives can be implemented on contemporary machines like the DEC Alpha, MIPS R4000, and PowerPC [56]. The question of task termination has been proven to be undecidable if even one process can be faulty (e.g., fail-stop [60]). Theorem 5 asserts that a mechanism for ensuring bounded recursive interprocessor communications together with locally bounded-recursive software modules guarantees globally bounded resources provided the step-counting function uses a reliable real-time clock. This analysis yields an architecture principle for the software radio.

Architecture Principle 1—Bounded Modules: Software radio architectures which limit (software and hardware) control structures to those that constrain partial recursion to the ISA-equivalent of bounded recursion will consume bounded resources, possibly entering a resource fault condition. Components that conform to this criterion may be called total bounded modules.

Implications: There are many implications to such an architecture principle. Setting resource bounds so tight that the statistical structure of the demand causes them to be exceeded frequently can drive resource use to zero due to thrashing over resource-use fault conditions. Loss of service from misallocating resource bounds is no less painful on the service provider than other types of resource loss, but it is easy to avoid.

In addition, the theorems above do not preclude combinatorial explosion. That is, using these constrained hardware and software structures, one may create data sets and instruction sequences with concrete complexity of $O(2^N)$. One is guaranteed, however, that $N$ is small enough for the exponential resources to be within those allocated in advance. One must know both the structure of the dependency and the bounds on $N$ in advance in order to establish the time-out criteria for each potentially exponential loop in the system. Such
controlled combinatorial explosion balances the needs of an algorithm to search a large space (e.g., in an equalizer) against the finite resources of the system. It will also guarantee that such algorithms will not consume more than the allocated resources. In addition, although the type of constructs that yield Ackerman's function cannot, in general, be ruled out, bounded recursion assures that such a construct would yield a resource fault after a specified amount of resource use. Such limits may be proscribed in the type certification process. Thus, any modules that meet the constraints when tested in isolation will not exceed them when inserted into the plug-and-play environment. In addition, the system will not become unstable because of undefined states since all functions are total. Such guarantees help assure reliable, predictable, plug-and-play software radios.

This section has presented an initial examination of the computational properties of software radio functions. The heterogeneous mix of ASIC’s, FPGA’s, DSP’s, and general-purpose computers can be constructed so as to consume predictable, tightly bounded resources. One must constrain the logic to the bounded recursive constructs with prioritized step-counting fault handling. The theorems provide an initial outline of a theory of resource integrity needed for plug-and-play modules. The constructions from the proofs suggest engineering techniques for applying the theory to development projects. This stability theory must then be combined with queuing theory [16], [26] and other resource management techniques that address the statistical nature of the demand for robust management of processing resources.

V. INTERFACE TOPOLOGIES AMONG PLUG-AND-PLAY MODULES

Constraints on the computational structure of modules can ensure bounds on resource use for plug-and-play. Are there constraints on the interface structure among these modules that further facilitate plug-and-play? The answer requires additional aspects of the topological model.

A. Topological Spaces

Definition (Topological Space): A topological space, denoted \((X, O_x)\), is a set \(X\) and a family of subsets \(O_x\), the “open sets,” which include \(X\) and the empty set \(\phi\), and which are closed under countable union and finite intersection [39]. The topology is the family of subsets \(O_x\) which has the geometric and algebraic structure.

In a topological space, one can represent the geometric properties of interfaces among software radio modules. An interface to an analog source, for example, may be modeled as a subset of functions on \(R\) that obeys certain constraints. The constraints include bandwidth, adjacent channel interference, and minimum and maximum transmitted power. An uncountable number of such waveforms is possible in an analog interface, but regulatory bodies and the hardware limit the waveform to a structured subset of the possible waveforms. Since the space of randomly perturbed signals is also a metric space [61], one can precisely define distance among waveforms, and thus the topological structure of analog interfaces. Fig. 14, for example, shows two analog waveforms in the time domain (a) and in the frequency domain (b).

A range of waveforms is \((A)\) allowed and prohibited \((B)\). Time-domain waveform \(A\) is within the allowed geometry, while waveform \(B\) is not, e.g., due to unacceptable power or RF. Such widely used interface constraints are set theoretic in that they limit the elements in the interfaces to a subset with specified properties. These constraints are also geometric, defining an interior \(A\), the conforming region, and an exterior \(B\), the nonconforming regions. The representations (a) and (b) are homeomorphic because the Fourier transform provides a topology-preserving map among the two.

B. Finite Interface Topologies

If a set \(X\) has a finite number of elements \(|X|\), all subsets are open sets (and are also closed sets). If \(|X| = M\), then the number of topologies that induce a topological space on \(X\) is \(2^{2^M - 2}\), a double exponential. Not all of the candidate topologies satisfy closure under union and finite intersection as required for a topological space [62]. The huge number of possible topologies compels one to define finite interface topologies more compactly.

Definition (Basis): A set \(B \subseteq X\) is a basis for \(O_x\) if the members of \(O_x\) are the union of members of \(B\). A basis is a smaller set than \(O_x\) from which \(O_x\) may be induced by taking unions.

From a hardware perspective, a pin in a connector is an interface point \(x_i \in X\). The set \(X = \bigcup_{x_i} x_i, \phi\) is a basis for \(O_x\). The set \(Y = \bigcup_{x_i} x_i, \{x_1\}, \phi\) which contains three subsets \(\{x_1\}, \phi\) and the union of all the other interface
pins works just as well. The basis with the most subsets is \(\{\phi, \{x_i\}\}\), the \(N+1\) sets that include the empty set and each element of \(X\) taken as a singleton set. If all of the pins are needed for a viable connection, then \(\{X\}\) is the only subset of \(X\) that consummates a connection, and therefore is the only element in the interface topology. In this case, \(O_x = \{\{X\}\}\) and the topology is just the fixed set of required inputs \(X\). This may be called the rigid topology. It is not a topological space because it lacks the empty set. The empty set is not a valid member of the interface set if the interface will not “work” if no pins are present. If the system will “work” with the connector unplugged (e.g., resort to a default or fail-soft mode), then the empty set is a member of the interface topology.

These principles apply to software. An API may specify a call to the synthesize() function, for example, with arguments RF (frequency) and \(W\) (bandwidth). If both are required, then \(Y = \{\text{RF}, \text{W}\}\) and \(O_Y = \{\{Y\}\}\). This is just as inflexible as the equivalent hardware interface. This power-set topology is the discrete topology; it may also be called the flexible API topology. This flexible interface geometry may be implemented if synthesize() uses default values for the missing arguments. Since the power set is the largest set of subsets, the discrete topology maximizes the number of combinations of API parameters over which a function call is valid. The basis for the power set consists of the singleton arguments plus the empty set: \(\{\{W\}, \{\text{RF}\}, \phi\}\).

C. Function-Call Parameter Topologies

The geometric structure of interface spaces may be better understood using additional notions.

Simplex: A simplex is an ordered set of points in a topological space that are adjacent in some sense, such as sharing a relation \(R\).\(^7\) Higher dimensionality simplexes induce lower dimensionality simplexes. Simplexes may be embedded in Euclidean space, but need not be [63].

Complex: A simplicial complex is a union of simplexes that includes the union of all of the lower dimensionality simplexes of a given simplex.

Q-Connected: Simplicial complexes that share a \(q+1\) face are \(q\)-connected [64].

The three vertices of a plane triangle \([A, B, C]\) in Fig. 15 comprise a two-dimensional simplex, adjacent in the sense that they are connected by the points in the plane. Each line segment joining these vertices comprises a one-dimensional simplex (e.g., \([A, C]\)) denoted by the pairs of vertices. A second triangle \(ABD\) that shares one line segment with the first is also a simplex. Each triangle together with its edges and vertices comprise a simplicial complex. The two triangles also comprise a simplicial complex in which the simplexes are 1-connected by the line \([A, B]\).

Interface parameter sets may be modeled as simplexes with a large number of dimensions. In the Synthesize() function above, suppose RF is discrete and has as its explicit basis the range \([400.001, 2400.000]\) MHz and resolution 1 kHz. The RF’s that are valid parameters of Synthesize() constitute a simplex consisting of the 2 million RF’s in this range. Let \(BW\) be defined for \([25, 30, 200]\) kHz. The parameter space \(\{\text{RF}, \text{BW}\}\) contains 6 million points, a simplicial complex consisting of the union of three simplexes of 2 million RF points each, indexed by \(BW\). Then the set of RF’s reachable from Synthesize(925.000, 30) is exactly one, \([925, 30]\). Synthesize(\(BW = 30\)) on the other hand, may reach any of the 2 million point RF simplex through a global default. Synthesize() reaches all 6 million points through a global default, connecting them in a single simplex.

For the example function Synthesize() to have an explicit basis, its definition must include basis tuples expressed as parameter (Type, Range). Parameter indicates the parameter being defined. Type chooses among predefined subspace types (e.g., Discrete, Continuous). Range establishes the valid values. In the example above, an explicit basis could be \(BW\) (Discrete, \([25, 30, 200]\)) and RF (Discrete, \([400.001, 400.002; 2400.000]\)) using Mathcad® range notation. This explicit basis can be enforced by the environment to yield an error result “out of range” for arguments that violate the basis description. Such semantics define Synthesize() as an effectively computable function over all inputs. A product topology on these bases yields the 6 million points of \((BW \times RF)\) as the input space to this function. However, it is often necessary to limit the parameters to subsets of this space. One might limit the 200 kHz bandwidth to the RF’s allocated to GSM, say, 925 to 960 MHz. The simplex

\[
\text{NOT}(200, \{[925.000, 925.200, 926.000]\})
\]

defines the set-complement subspace for which the 200 kHz bandwidth choice is mapped to a fault condition. To the degree that such subspaces can be changed while the radio node is in the field, the service provider has a “future proof” interface. In order to be changed in the field, the “capability” implicit in the interface constraints must be defined in such a way that...
the control algorithms can test and manipulate the constraints, e.g., to decide whether to accept a download. The geometric structure of the interfaces can be expressed naturally in strongly typed languages to define plug-and-play constraints. More appropriately, for plug-and-play environments, they can be expressed in execution time interface parameter sets.

D. Plug-and-Play Interface Geometry

The implementation decisions made during design, development, integration, and test may constrain an interface to a point in the simplicial complex of interfaces as illustrated in Fig. 16.

Each domain and range of a conventional interface consists of a designated point in its simplicial complex. A plug-and-play interface, however, defines an interoperable subset of the interface space. The physical interface subspaces must change as a function of the hardware in which the service is delivered. The logical interface subspaces also may have to change as a function of the software modules configured to deliver the services. To be fully extensible, plug-and-play modules have to be combined dynamically. To do this in a controlled way, the control algorithm(s) must have a way of comparing the range of one function to the domain of the next to determine whether the functions are compatible.

E. Extensible Capabilities

Capabilities may be represented as levels ("tags") such as the type of video teleconferencing interface in the ITU H.320 Recommendation [65]. The level of call control support from Microsoft’s telephony applications programmer interface (TAPI) [66] provides another example of tagged capability levels. The advantage of tags is simplicity. Capability tags are defined during the standards-setting process and documented in the text of the standards. One disadvantage of this approach is the possible misinterpretation of the text of the standard. To counter this chronic problem in telecommunications standardization, ETSI promulgated language that makes the formal specification of the ITU Z.100 specification and description language (SDL) [67] the normative expression of the standard [68], with text providing amplification and explanation. Historically, the text has been normative, while the computer-based representations have been auxiliary.

Another disadvantage of capability tags is that capability cannot be defined dynamically by the radio itself. Computational manipulation of tags by the software radio joint control function is also limited. For control of dynamically defined software radio-based services, the constraints must be computer processable, and the system must be able to act on them during operations. Automated reasoning over the constraint spaces and capabilities of the host system could enable the radio itself to construct the level of capability necessary for a desired service. SDL falls somewhat short of this vision since the application semantics are left to the designer. Dynamic capability requires designer-independent semantics.

It is easy to define designer-independent semantics in a way that is readable by people. To do this in a way that is processable by a control algorithm requires a radio knowledge-representation language (RKRL). Parts of RKRL that are defined \textit{a priori} may be extended dynamically. For example, to add a control that selects from either the $A$ or $B$ buffer in a double-buffered interface of a notional ADC requires extension of the interface topology, adding sets and subsets to $(X, O_X)$ as suggested in the example of Fig. 17.

The meta-level expressions $\langle$Buffering$\rangle$, $\langle$Buffer-Size$\rangle$, $\langle$Buffer$\rangle$, and $\langle$Buffer-flag$\rangle$ introduce the meta-level primitives Buffering, Buffer-Size, Buffer, and Buffer-flag to an extensible RKRL. The RKRL is assumed to include $\textit{a priori}$
TABLE III

**TABLE III**

**SPEAKEasy I Software Modules, DMA: DIRECT MEMORY ACCESS**

<table>
<thead>
<tr>
<th>Module</th>
<th>Source</th>
<th>Ada Module Descriptions/ Functions</th>
</tr>
</thead>
<tbody>
<tr>
<td>At</td>
<td>(127 kB)</td>
<td>C040 interprocessor communications</td>
</tr>
<tr>
<td>BIT</td>
<td>(318 kB)</td>
<td>Built-In-Test packages, including CRC, EEPROM, PID, I/O registers, interrupts &amp; DMA</td>
</tr>
<tr>
<td>Cm</td>
<td>(1.19 MB)</td>
<td>Configuration Management</td>
</tr>
<tr>
<td>ALE</td>
<td>(125 kB)</td>
<td>ALE Receive (Rx) and Transmit (Tx) Functions</td>
</tr>
<tr>
<td>ALE_Rx1</td>
<td>(378 kB)</td>
<td>Automatic Link Establishment (ALE) Receive Modules</td>
</tr>
<tr>
<td>Hvq</td>
<td>(645 kB)</td>
<td>Have Quick Communications Ensemble</td>
</tr>
<tr>
<td>Hvq_Ci</td>
<td>(109 kB)</td>
<td>Control Modules (Initialization, Mode Control, Errors)</td>
</tr>
<tr>
<td>Hvq_Hib</td>
<td>(25 kB)</td>
<td>Globals</td>
</tr>
<tr>
<td>Hvq_Rx</td>
<td>(379 kB)</td>
<td>Receive Mode (Synchronize, TOD, Rx, Active...)</td>
</tr>
<tr>
<td>Hvq_Tx</td>
<td>(131 kB)</td>
<td>Transmit Mode</td>
</tr>
<tr>
<td>Work</td>
<td>(299 kB)</td>
<td>ALE packages &amp; specs</td>
</tr>
<tr>
<td>Hfm</td>
<td>(518 kB)</td>
<td>HF Modern Communications Ensemble</td>
</tr>
<tr>
<td>Hfm_ctrl</td>
<td>(58 kB)</td>
<td>Controls Waveform start/stop messages, RTS Events, PM Query, TX/RX Done (local);</td>
</tr>
<tr>
<td>Hfm_dc</td>
<td>(22 kB)</td>
<td>Data Control Packages, source messages error checking</td>
</tr>
<tr>
<td>Hfm_rx</td>
<td>(289 kB)</td>
<td>Receiver Bit &amp; message operations, text I/O, Rx utilities, data correlation tables, filters, queues</td>
</tr>
<tr>
<td>Hfmx</td>
<td>(149 kB)</td>
<td>Squelch, TX/RX mode, TX templates, RF Control, Timing</td>
</tr>
<tr>
<td>Nbgl</td>
<td>(334 kB)</td>
<td>Narrowband Frequency Hopping Group</td>
</tr>
<tr>
<td>Nbgh</td>
<td>(49 kB)</td>
<td>State Machine, Sync Loss, TX/RX, Waveform, PTT State...</td>
</tr>
<tr>
<td>Nbghp</td>
<td>(105 kB)</td>
<td>Global parameters for NBG package</td>
</tr>
<tr>
<td>Nbgh</td>
<td>(57 kB)</td>
<td>Hop Packages – timing, data request/processing, PTT acknowledge, cryptographic processing...</td>
</tr>
<tr>
<td>Nbgr</td>
<td>(73 kB)</td>
<td>Receiver Packages MFSK, Preamble, Galois (FEC), Dead Bits, Flags, Bitsync, RX flush, Detect/Track</td>
</tr>
<tr>
<td>Nbgl</td>
<td>(49 kB)</td>
<td>TX: Amplitude, Preamble fill, IQ Samples, AM on Voice, Filter, Interprocess Communications (IPC) Messages, SSB, DSB, QAM, OQPSK, Event &amp; Constraint Checking</td>
</tr>
</tbody>
</table>

Semantics for Resource, Type, Set-Extension, Output-Space, Range, None, Double, and Singleton topologies. The set extension augments the interface topology ($X \times O_X$) with new subsets. Range expressions define the new subspaces. Discrete values constrain the subspace to an interface point.

The resource extension indicates that if Buffering = none, then the effect is null. This effectively glues this new subspace to the existing subspaces in which there was no buffering. In addition, the new subspace Buffering(Double) has the rigid interface topology. Topological spaces represent radio capability in a way that is readily processed by control algorithms. The extensibility of interface topologies may be expressed in the following architecture principle.

**Architecture Principle 2—Explicit Extensible Interface Topology:** API’s and hardware interfaces which exhibit an explicit topological basis for each interface parameter space and which are extensible in the field exhibit the level of flexibility necessary for extensible plug-and-play services.

**Implications:** The requirement for an explicit basis assures that the interface topology is defined completely, including error states. The full extensibility of the topological bases in the field requires an RKRL-like representation of radio resources. This is an area of current research in wireless computer-communications systems [69].

VI. ARCHITECTURE PARTITIONS

One goal of a plug-and-play software radio architecture is to provide computing resources from as yet undefined hardware modules to support as yet undefined software modules for as yet undefined services. The following architecture analysis applies the computational and topological properties introduced above to begin to identify the mathematically based partitions of a software radio system.

The source code of a widely published military software radio, SPEAKEasy I [70], is considered. A sequence of virtual machines is then derived that both expand the RAM equivalent instruction set model and constrain the computational geometry, yielding a layered virtual machine reference model for the software radio.

A. **SPEAKEasy I**

SPEAKEasy I resulted in the software radio source code structure summarized in Table III. The as-built code has some strong features—such as real-time performance and accurate handling of timing differences between radio networks. Since an Ada implementation was mandated, the real-time executive is the Ada run-time kernel.

The Ada modules can be viewed as software objects. They include databases, channels, and agents. Databases store personalities, filter parameters, lengthy chunks of compiled code, and data sets to be loaded into a personality at run time.

Channels are abstractions around which modes (e.g., HAVE QUICK) are organized. A channel is supported by several Ada packages that perform the systems-level functions of RF control, modem processing, INFOSEC, and related internetworking. A channel gets the system resources (paths
or threads through the system). It installs its personality on these resources to implement a mode. It then keeps track of the overall state of the processing thread that delivers the associated services. In SPEAKeasy I, lower level modules implement the personalities of the channels. They also serve as hosts for buses, manage IO processes, access timing and positioning data, and control the radio. Channels also keep track of the status of the mode, number of resources employed, volume, data rate, throughput, network parameters such as network number, and assigned time slot(s).

Timing packages manipulate the system clock, time of day (ToD), the day/time format, and the timing resolution. The RF control packages determine RF direction (i.e., transmit or receive); the RF mode (e.g., linear or nonlinear amplification); preemphasis for predistortion; and frequency of transmission. The modem packages include modulation (AM, FM, QAM, USB, and MSK), demodulation, automatic gain control (AGC), packing, and unpacking of protected bits.

The system may also perform a loop-back function for network testing or local diagnosis. The back-end functions include message processing, internetworking, managing protocol stacks, and the user interface. Radio control handles system boot up, initial ToD, current hop, calibration, status requests, and security level. Comparing the Nbg, Hvq, and Hfm mode software shows a common pattern of control module(s), global parameters, transmit and receive (modem) modules, along with specialized modules such as hop generation for frequency-hopping modes.

The as-built code includes four distinct types of software. One type delivers voice, data, and channel bridging services. Another type structures the radio applications using state machines. The infrastructure-type software moves data packets and isochronous streams. The fourth type consists of hardware-specific support software including the operating system. About 30–40% of the application-specific code is infrastructure, 30% structures the radio applications, and 30% implements services. These different functions are almost inextricably intertwined in the SPEAKeasy code. Much of the code is hardware dependent. The partitioning analysis that follows is oriented toward reuse. The most primitive level of code is the hardware-specific operating system.

B. The Hardware-Specific Partition

The hardware–software interfaces define the lowest level partition of a radio system. From a topological perspective, processor hardware connects the states of data in its registers. The set of data states reachable in one clock cycle from any initial state is the processor’s characteristic simplex. A processor with 20 32-bit address registers, cache pointers, general-purpose registers, with multiple direct memory access (DMA) registers and other registers can, in principle, assume any of \(2^{102+20} = 10^{62.6} = 0.358846241 \times 10^8\) states—in the Appendix for magnitude notation. This is the size of the processor’s instantaneous characteristic simplex. Fewer registers yield a smaller simplex. A more complex instruction set architecture yields a larger simplex. An FPGA’s simplex is determined by its use of state memory, which will generally be much larger than a DSP chip of the same area and device feature size. The size of the simplex indicates how many different things the processor could accomplish in a single clock cycle.

1) Paths in Simplexes Induce Partitions: Sequential simplexes are \(q\)-connected to each other by the clock sequence. Over time, a series of RAM instructions traces a path through a sequence of simplexes, the union of which is a simplicial complex (SC). In 1 s, a processor with a 100 MHz instruction clock may assume any of \((10^{32})^{10\times10^8}\) or \(0.87822 \times 10^7\) states. In addition, a software radio with distributed multiprocessor hardware and interconnect devices has additional connectedness among processor simplexes as illustrated in Fig. 18.

At each clock cycle, an edge in the simplex is traversed (e.g., the solid arrow in processor 1’s simplex at time \(k = 3\) in the figure). The path traversed through a sequence of simplexes may be subsumed into a simplicial complex, e.g., \(\text{SC}(P_1, I, P_2; k = 3 \cdots 5)\) in the figure. The dotted arrow in the SC indicates the path traversed by data sent from \(P_1\) to \(P_2\) during clock sequence [3, 5]. The way in which software constrains such paths implicitly defines architecture partitions. A larger number of possible paths indicates greater built-in flexibility, while topological loops in such paths suggest partitions.

2) Interrupt Service Routine (ISR) Topological Loops: Consider hardware-dependent software such as operating system services. This software consumes processing resources in a way that is tightly coupled to hardware-related processor states. A typical ISR, for example, might turn off hardware interrupts, push the processor state onto a stack, test the interrupt condition, set a dispatch pointer, restore the registers, and exit. In a

Fig. 18. Processor simplexes (\(S_1, S_2\)), interconnect simplexes (I), and an equivalent simplicial complex \(\text{SC}(P_1, I, P_2; k = 3 \cdots 5)\).
real-time system, this process could take from 10 to 100 instructions. Windows NT requirements for template data might expand this to 1000 instructions or more. During this sequence, the path through machine’s simplicial complex, normally arbitrary, is constrained to a path that involves states of the interrupt register. In addition, the instruction sequence that is so constrained is defined temporally by the interrupt event and the termination of the ISR. This produces a path in the temporal simplicial complex that begins with a hardware interrupt “set” and a particular configuration of registers and ends with the interrupt “clear” and the identical configuration of registers. Adopt a convenient distance measure on data states such as Hamming distance, the number of bits that are not identical. A topological loop over ([RAM], a t) [RAM], is the shortest path in [RAM] that returns to within ε of a t. An ISR is a topological loop in [RAM] for a t in the above example. Input/output drivers, task schedulers, disk access control, real-time clock support software, and the like have short characteristic topological loops.

3) The Topology of the Kernel Substrate: The set {kernel} comprises the software most closely aligned to hardware states and having the shortest topological loops. The kernel is the lowest level software substrate. Kernel sequences can be modeled as extensions to the processor’s native ISA. A processor with a library of such hardware-dependent modules can be viewed as a machine with an extended instruction set, ISA = {RAM} U {kernel}. The set {kernel} may have a distinguished sequence of {RAM} instructions for the idle state. Idle ⇒ {RAM} in that, during the Idle state, an arbitrary RAM sequence is possible. Let {kernel−} represent the kernel-less idle. Minimizing the number of {RAM} sequences needed for the {kernel} maximizes the {RAM} capacity left for higher level radio services. The simplex of {RAM} U {kernel−} contains a distinguished subspace corresponding to the paths reachable through {kernel}.

C. Topology of Infrastructure Software

Given a set of kernel software, the next step in a bottom-up definition of virtual machines identifies those functions that allocate, set up, and control physical resources to create logical resources. Fig. 19 lists the function calls required for distributed processing. These functions include the structures recommended in ITU’s X.900 open distributed processing reference model [71]. They also include structures unique to radio applications such as frequency distribution. This infrastructure code manages control flow paths; signal flow paths; and timing, frequency, and positioning information. It also includes features needed for isochronous delivery of voice, video, and other real-time streams. The control flow paths mediate message passing among most objects in the system. Error logging, semaphores that manage shared resources, and bus access protocols are examples of control message flows.

The infrastructure functions initialize the system, create and manipulate logical and physical ports, move messages, and perform remote procedure calls (RPC). They also provide

© Mitola’s STATISfaction, used by permission

Fig. 19. Infrastructure software function calls.
multicast services, handle error messages, and support standard protocol interactions such as Internet protocol (IP). Multicast simplifies programming of multichannel operations such as initializing 100 subscriber channels distributed among 25 DSP chips. The control-flow methods listed in the figure constitute a minimum set necessary for multiprocessor software radio infrastructure.

The signal-flow methods listed in the figure, similarly, set up and manage signal flow paths among processes on the same or on different processors. These isochronous streams must meet tight timing constraints. Due to the overhead associated with path setup and tear down, these paths must be opened and closed multiple times without being set up and torn down again. Timing, frequency, and positioning is a very involved process, a complete discussion of which is beyond the scope of this brief treatment. Time references obtained during network synchronization must be maintained on a per-network basis. Since the software radio generally participates in multiple networks simultaneously, it must maintain absolute time per network. This is accomplished not by changing the software radio’s clock, but rather by defining time offset and drift rate for each network. Additional ancillary functions related to queueing data messages (e.g., load, clear, test status, and reset) are also part of infrastructure.

Applying the virtual machine paradigm to this set of software yields a new virtual instruction set {Infrastructure} which consists of {Message-Passing}, {Isochronous Paths}, {Timing}, {Frequency}, and {Positioning} instruction subsets. These extensions build on the facilities of {kernel}. The expanded {RAM} simplex contains {Message-Passing}, etc. Each subspace depends on {kernel} services, the lowest layer of the emerging virtual machine hierarchy.

D. Radio State Machines

State machines control access to many software radio resources. State machines typically control the transmit and receive channels and fault recovery actions. A resource-allocation state machine is illustrated in Fig. 20. Three states are shown in boxes: waiting for instantiation, fetching a waveform, and waiting for a response. The arcs are labeled with conditions that cause one to transition from one state to the next and with actions performed upon such a transition. This control structure may be implemented as a software object with a set of slots (the states) and attached methods. Methods test for state transitions and perform required actions. Recommendations for defining and simulating state machines are provided in the SDL Recommendation Z.100.

Such state machines may operate on several levels. At the top level, each channel (e.g., HAVE QUICK) has a channel-setup state machine that keeps track of resources. The next-level state machine manages the mode, as illustrated in Fig. 21. The channel is initially idle. When it has been set up, it enters the Ready state. If the user (or network) initiates a “talk” sequence, the transmit (Tx) state is entered, upon which the radio will alternate between Tx and Rx states until conditions are met which either suspend or deactivate the channel.

Lower level modem state machines track the states for active plain text receive and transmit, crypto-sync, receiving analog and digital, and bridging fades. The SPEAKEasy state machines also have built-in timing and error-recovery procedures. Automatic gain control (AGC) and squelch (Sq) are adjusted on all transitions in the lower level state machines.

Thus, the state machines schedule both routine processes such as AGC and processes driven by channel conditions such as bridging across fades. States also reflect failure modes. These include sync failure, loss of carrier, and loss of system resources. Such state machines are a central mechanism for controlling system and radio resources in the software radio.

From a geometric perspective, the topological loops of these state machines begin and end on conditions of identical state values. That is, the states “setup,” “transmit,” “receive,” etc., are the constants or fixed points in the evolution of the [RAM] simplicial complex. Periodically, the processor’s register set will return to topologically close states for some suitably defined distance. Since the functions called from these state transitions (e.g., “transmit next block”) employ facilities from {Message-passing}, {Isochronous stream}, etc., these state machines define the next level of virtual machine built on top of the infrastructure machine. The boundary of this layer of virtual machine is based on the function calls attached to the state-machine objects. This new layer of virtual machine is the {State machine} instruction set, consisting of the {RAM} sequences that set up, execute, and control the state-machine
E. Channel Agents

The characteristics of SPEAKEasy I and other software radios have been abstracted to the top-level software object diagram of Fig. 22. These high-level objects may be called channel agents. Each software object has been allocated corresponding functions from the software radio functional block diagram of Fig. 2.

Those agents in the top row of the figure control the system, while those in the lower rows implement the traffic channels and related services. The objects of Fig. 22 implement the functions summarized in Table IV.

Each of these high-level software objects has subordinate objects, to the level of primitive functions like filters, modulators, bit decisions, etc. The protocol and speech processing “back end” of the software radio employs a protocol stack such as ATM, TCP/IP, Mobile IP, etc. Consequently, internetworking to the wireline infrastructure consists of a few monolithic predefined software objects.

These radio functions are relatively generic. Thus, each waveform or mode of operation may be expected to have either its own set of agents or a set of parameters and a script that tell the agent how to treat that specific mode. Radio services, like bridging across waveforms, could be constructed as scripts or Java-like applets that interconnect channel agents and other high-level functions. The semantics of such top-level modules are readily represented in the unified modeling language (UML) [72].

The timing and overall behavior of the channel agents are constrained by the lower level state machines that manage the channels. From a geometric perspective, agent subspaces in the [RAM] simplex subsume the {State machine} subspace. The extended instruction set for channel agents {Channel agents} is more of a category than a virtual instruction set. The top-level radio functions {Antenna and RF Control}, {Channel Control}, {IF Processor}, {Waveform Processor}, etc., each constitute a set of virtual instructions. Since these channel agents must be mutually consistent in order to work together, the class {Channel agents} refers to a mutually consistent set of such instruction set extensions comprising a radio mode.

F. Distributed Layered Virtual Machine Reference Model

The preceding sections have described essential software mechanisms employed in the construction of software radios. When viewed in terms of the topological spaces defined by the paths traced in the simplex of an equivalent RAM processor, layers of successively less machine-dependent software emerge. The distributed layered virtual machine is a way of representing the resulting hierarchy. Each layer is a virtual machine built on subordinate layers. The virtual machine interfaces may be held constant so that the implementation details and intellectual property present within a component at a given layer are hidden from all other layers. The software radio architecture that results from this process is illustrated in Fig. 23.

As the capacity of FPGA’s and DSP’s continues to grow, the processing to support such virtual machines becomes more affordable. The services offered from the top-layer virtual machines thus are independent of the hardware implementations in the bottom-layer virtual machine. The FPGA and DSP hardware vendors or software tool suppliers, for example,
could offer infrastructure layer software. This layer could be implemented in part using an augmented message-passing interface (MPI) [73]. The state machines in the infrastructure and radio applications layers would naturally fall into the domain of SDL. Third-party waveform vendors could offer middle layers. Interfaces among components may be represented in UML or the interface definition language (IDL) [74], possibly including abstract syntax notation (ASN.1) [75]. Systems integrators and service providers could then define their unique added value in the top layers, building on the broad base of industry support at the lower layers. In addition, computer-aided software/systems engineering (CASE) vendors could embrace the wide range of reusable components to assist developers to encompass the entire software radio distributed virtual machine hierarchy for reduced time to market.

The joint control function is distributed over these layers. Joint control at the top layer sets up channel objects and orchestrates services. In the radio applications layer, it consists of state machines that manage the radio channels. In the infrastructure layer, it establishes paths for signal and data flow. And in the hardware layer, it interacts with the operating system. Each layer accepts commands from higher layers, and returns status including fault conditions. With layering of the joint control function, the layered virtual machine insulates the higher layer applications from the details of the hardware.

There are generally significant computational resource penalties in making the translations among the layers. Two factors mitigate these penalties. Compilers, first of all, have become fairly efficient in the use of machine register sets. As radio API's become more widely used, precompilers should emerge which suppress unnecessary levels of function calls and make compile-time translations, enhancing run-time efficiency. In addition, late bindings that are computationally efficient should also emerge. Until they do, the penalties for the layered architecture implementations will remain high. Second, the computational capacity of the underlying chips continues to double every 18 months or so, making the performance of the layered machines acceptable in spite of the inefficiency. In particular, the internetworking functions that a few years ago were implemented in special-purpose hardware are now supported by general-purpose computers. These observations may be summarized in a third architecture principle.

**Architecture Principle 3—Distributed Layered Virtual Machine Reference Model:** Modules partitioned according to the distributed virtual machine reference model will insulate lower layer hardware-dependent modules from upper layer service-defining modules.

The economic incentives for such a model center on integrating markets for economy of scale. The global wireless marketplace now consists of dozens of niches defined by unique hardware platforms and waveform-unique infrastructure. Industry is attempting to organize itself to integrate these diverse markets so that the next generation of wireless will offer low-cost plug-and-play services. The process of defining the required industry standards has already begun. The SDR Forum, for example, has defined an architecture framework based on functional threads in which a generic applications programmer interface (API) is being defined across the virtual machine. The forum is integrating the GloMo radio device API.
VII. CONCLUSION

The computational and geometric properties of the software radio will shape the degree to which plug-and-play is realizable for the global marketplace. Greater understanding of the mathematical properties of the software radio architecture should accelerate progress toward cost-effective plug-and-play services to which developers aspire. The architecture principles are particularly informative.

1) Bounded Recursive Modules: Construct the software radio system—host environment and plug-and-play modules—of bounded recursive modules. These modules will be defined for all inputs, and they will consume predictable system resources. Although the software may contain errors, those errors will be much less likely to cause faults in the delivery of audio, data, and multimedia services. They also will be less likely to cause system crashes than unconfined modules.

2) Explicit Extensible Interface Topologies: Define software radio interfaces using an explicit basis for the underlying topological spaces. Use extensible languages such as UML, SDL, IDL, and ASN.1 until a more domain-specific language such as a radio knowledge representation language (RKRL) emerges.

3) Distributed Layered Virtual Machine: Position plug-and-play modules in the distributed layered virtual machine hierarchy in a way that maximizes the use of existing (lower and higher level) components.

APPENDIX

MAGNITUDE NOTATION

A recursive logarithmic number system makes the large size of simplices and simplicial complexes easier to understand and manipulate. The notation y Mn ("y magnitude n") indicates raising y to the power of 10 n times where n is an integer and 0 < y < 1. The magnitude n of this measure of concrete complexity of a simplicial complex is a strongly nonlinear measure of complexity. For example, $32 = 10^{2.505}$, which equals $10^{1.10^{1.770}}$, which can be expressed in magnitude notation as 0.1775M2, a magnitude 2 number. The size of the processor’s simplex above, $10^{0.26}$, equals 0.358841M3, a magnitude 3 number. The 1 simplicial complex of the same machine has size 0.87822M5. This magnitude 5 number is exponentially bigger than a magnitude 3 number by $10^{19}$. Put another way, the size of this complex is $10^{10}$ raised to a power containing 35 million zeros. For a perspective on the size of this number, the number of neutrino-sized places in the known universe ($10^{20}$ m on each side) over the last 10 billion years is only $10^{26}$ or 0.384836241M3. Using the size of the simplex as a base and raising it to the power $k = 100$ million is not, perhaps, a familiar concept, but it establishes an upper bound on the size of the simplicial complex associated with just 1 s of use of such a processor. This notation is useful in comparing the huge numbers associated with simplices and large simplicial complexes.

REFERENCES


Joseph Mitola, III (M’74), for a photograph and biography, see this issue, p. 312.