Component-Based Design and Service-Oriented Architectures in Software-Defined Radio

Benjamin C. Hilburn

Thesis submitted to the Faculty of the

Virginia Polytechnic Institute and State University

in partial fulfillment of the requirements for the degree of

Master of Science

in

Computer Engineering

Tamal Bose, Co-Chair

Jeffrey H. Reed, Co-Chair

Carl B. Dietrich

April 22nd, 2011

Blacksburg, Virginia

Keywords: Component-Based Design, Service-Oriented Architectures, Software-Defined Radio, Autonomic Systems, Cognitive Radio

Component-Based Design and Service-Oriented Architectures in Software-Defined Radio

Benjamin C. Hilburn

ABSTRACT

Software-Defined Radio (SDR) is a large field of research, and is rapidly expanding in terms of capabilities and applications. As the number of SDR platforms, deployments, and use-cases grow, interoperability, compatibility, and software re-use becomes more difficult. Additionally, advanced SDR applications require more advanced hardware and software platforms to support them, necessitating intelligent management of resources and functionality. Realizing these goals can be done using the paradigms of Component-Based Design (CBD) and Service-Oriented Architectures (SOAs).

Component-based design has been applied to the field of SDR in the past to varying levels of success. We discuss the benefits of CBD, and how to successfully use CBD for SDR. We assert that by strictly enforcing the principles of CBD, we can achieve a high level of independence from both the hardware and software platforms, and enable component compatibility and interoperability between SDR platforms and deployments. Using CBD, we also achieve the use-case of a fully distributed SDR, where multiple hardware nodes act as one cohesive radio unit.

Applying the concept of service-orientation to SDR is a novel idea, and we discuss how this enables a new radio paradigm in the form of goal-oriented autonomic radios. We define SOAs in the context of SDR, explain how our vision is different than middle-wares like CORBA, describe how SOAs can be used, and discuss the possibilities of autonomic radio systems.

This thesis also presents our work on the Cognitive Radio Open Source Systems (CROSS) project. CROSS is a free and open-source prototype architecture that uses CBD to achieve platform independence and distributed SDR deployments. CROSS also provides an experimental system for using SOAs in SDRs. Using our reference implementation of CROSS, we successfully demonstrated a distributed cognitive radio performing dynamic spectrum access to communicate with another SDR while avoiding an interferer operating in the spectrum.

This work was supported in part by the Institute for Critical Applications and Applied Science, by the Defense University Research Program, and by Rockwell Collins.

Acknowledgments

I would like to acknowledge Timothy Newman, who started the CROSS project. CROSS quickly evolved into greater things, and Tim was vital to the CROSS project's success. He is also a great friend.

I would also like to acknowledge Amy Malady, who has enriched my life more than I could possibly explain, and who continues to push me to be my best.

Finally, I would like to thank my committee, whose guidance and advising made the success of this work possible. I would especially like to thank Dr. Jeffrey Reed and Dr. Carl Dietrich, who encouraged me to come to graduate school while I was still an undergraduate.

Contents

List of Abbreviations

CBD	Component-Based Design
CE	Cognitive Engine
CR	Cognitive Radio
CROSS	Cognitive Radio Open-Source Systems
CRS	Cognitive Radio Shell
FOSS	Free and Open Source Software
FSM	Finite State Machine
GPP	General Purpose Processor
IP	Internet Protocol
MANET	Mobile Ad-Hoc Network
OSI	Open Systems Interconnection

PE	Policy Engine
PHY	Physical layer relative to the OSI stack
RX	Receive, Receiver, Reception
SDR	Software-Defined Radio
SML	Service Management Layer
SOA	Service-Oriented Architecture
SOAP	Simple Object Access Protocol
TCP	Transmission Control Protocol
TX	Transmit, Transmitter, Transmission
TXR	Transceive, Transceiver
VTCROSS	Virgnia Tech Cognitive Radio Open Source Systems - DEPRECATED
WSDL	Web Services Description Language
WWW, Web	World Wide Web
XML	Extensible Markup Language

List of Figures

List of Tables

Chapter 1

Introduction and Motivation

Research in wireless and digital communications is a major thrust for many companies, universities, and governments. The method of research, however, is often just as important as the research itself. Additionally, design paradigms are critical to the advancement of technology. A relevant example is the cognitive radio paradigm, which continues to revolutionize the applications and uses of software-defined radio.

To date, most research in wireless and digital communications has been tied to a particular platform; i.e., the work is platform-dependent. The effect of this is that it is often difficult to collaborate with other groups, share research, or even form an accurate comparitive analysis of results from different organizations. When one research group wishes to use something another group produced, they often must reproduce it on their particular platform and architecture. In short, the amount of reuse of one particular implementation is

often quite small.

This situation also represents a closed economy of multiple incompatible SDR domains. Functionality and applications are not interoperable, and radio platform developments are often monopolized by the organizations that created the platform. This is not advantagous to users, whether the users are individual hobbyists, medium-sized emergency response teams, or large government organizations such as the military.

Radio systems have also typically existed within a single node - i.e., even if a radio platform or application was comprised of many components, all of those component instances exist on a single hardware deployment. This represents a contrained use-case, in that the only resources available to the radio were those that were local in the hardware. Components running on neighboring nodes are typically un-available to other nodes, and many times there are inherent interoperability issues - especially if the nodes are using different hardware or software platforms.

The issue of re-use and interoperability is an area of heavy research in the field of software engineering. Perhaps one of the most well-known developments of re-use paradigms is the Object-Oriented Programming (OOP) model. This method of programming is now a primary focus at many universities, and is the basis of many (relatively) new programming languages (e.g., C++, Java, and Python). This desire to improve re-use of code has grown into new methodologies which minimize the amount of time a developer must spend re-implementing previously-implemented functionality and technologies.

Two of the development methodologies that have arisen from this renaissance in software engineering workflows are Service-Oriented Architectures (SOAs), and Component Based Development (CBD). The fundamental ideas behind both of these designs are similiar; the goal of both is to enable collaboration, interoperability, better use of resources, and re-use of existing implementations. Applying these paradigms to SDR also enables distributed radio use-cases wherein components running on different nodes act cohesively as a single radio unit.

For the most part, these ideas exist primarily in the domain of computer science. Research in the field of communications, for the great majority of its history, existed solely in the realm of hardware (i.e., electrical engineering); a radio's implementation was defined by the hardware within it. This changed, however, with the advent of Software-Defined Radio (SDR), which enabled the reconfiguration of a radio through General Purpose Processors (GPPs) and software. In many ways, this represented a new link between the fields of computer science and electrical engineering. The field of computer engineering is often described as sitting in between computer science and eletrical engineering, and more often than not serves to connect the two.

1.1 Previous and Related Work

The field of service-oriented architectures is rapidly expanding, especially in the realm of enterprise Web services and business, where it first originated and has been very success-

ful. To our knowledge, however, no one has applied this paradigm to software-defined radio. The SOA paradigm is not limited by the technology in use, however, and it there is no reason it cannot be as successful in other arenas as it has been for enterprise Web services [5].

In the past, the term 'service-oriented' has been applied to a technology with some connections to software-defined radio: CORBA [5]. As discussed in Chapter 3, a system must really be designed as a service-oriented system from the outset with strict enforcement. While CORBA is successful in many ways, it does not fulfill the vision of SOA in SDR, as defined by this thesis. By this, we mean that CORBA's definition of 'services' really corresponds to independence at a component level, not a service level. As a result, its encapsulation of resources and logic does enable the sort of service-orientation required by the service-oriented architectures described in this thesis. This is further discussed in Chapter 3.

CORBA has, however, been used successfully in the realm of SDR for component-based design. These efforts faltered, however, due to some of the reasons discussed in Sections 2.2 and 2.3. The U.S. Joint Tactical Radio System (JTRS) used CORBA in their Software Communications Architecture (SCA), but the components were not well defined and the program did not meet expectations, ultimately ending in the SCA version 3.0 as being declared unsupported by the JTRS program itself [6].

Component-based design, as a whole, however, has been used to great success within the SDR community. Many SDR architectures and frameworks use some of the characteris-

tics of CBD described in Chapter 2. An FOSS implementation of the SCA is provided by OSSIE, for example, which uses CORBA to define components and component communication protocols [7]. Perhaps one of the most well-known SDR frameworks is GNURadio, the applications for which are defined entirely in terms flowgraphs of 'blocks' (i.e., components) [8]. Both of these SDR architectures have many users, and have been very succcessful.

There are two primary differences in the concepts presented in this thesis regarding CBD, and CBD as it exists in SDR frameworks like OSSIE and GNURadio: SDR platform independence, and distributed radio.

The compononents developed for these SDR platforms work only within that platform; designing a component for OSSIE means that it will not work 'out-of-the-box' with GNURadio. The goal of CROSS is to enable such 'plug-and-play' component technology across platforms, whether the platforms are SDR platforms or hardware platforms. Compatibility and interoperability are key facets of the CROSS design, and hence require strict application of the design methodology described in Section 2.3.

Finally, to our knowledge, there has been no other work that uses CBD to achieve distributed radio systems. CORBA has been used, for example in the SCA, to achieve some degree of distributed radio (e.g., distributed PHY), but we assert that this paradigm is possible even without such middle-ware. Distributed radio is a developing paradigm in terms of radio deployment, and is a step towards even more advanced technologies such as Wireless Distributed Computing (WDC) and wireless cloud computing in MANETs.

1.2 Original Contributions

This thesis of computer engineering applies the concepts of component-based design and service-oriented architectures to software-defined radio, and discusses the prototype implementation developed for such an implementation: the Cognitive Radio Open Source Systems (CROSS) project. This thesis also serves, in a broader sense, as an example for applying new paradigms to communications research to improve designs, results, and enable new technologies.

We define CBD in the realm of SDR and seek to make CBD attainable by SDR researchers by defining six properties to be used as design goals and metrics. We present the relationship between these metrics, and assert which properties enable others, thus simplying the application of the CBD paradigm. By using CBD, we can also achieve distributed radio systems, where multiple nodes act as one radio unit. This is a novel application of component use in SDR, and represents a new development in radio design. Component-based development, and software re-use in general, has historically failed in some organizations, so we also discuss the steps necessary to ensure its success in SDR.

While non-distributed CBD has been applied to SDR in the past, the SOA paradigm has not previously been used (to our knowledge) as a central design idea in SDR. We assert that the application of SOAs to SDR represents the next evolution in software-defined radios in the form of goal-oriented autonomous radio systems. We describe the benefits of using SOAs in SDR, define design terminology, and discuss the technologies necessary

to enable this new form of SDR.

Finally, we discuss our prototype design of a component-based and service-oriented SDR architecture in the form of CROSS. Using CROSS, we have also successfully tested and proved the paradigm of distributed radio systems. We assert that a market-driven evolution of open-source standards is indeed possible, and holds the potential to create a robust economy of SDR functionality. One of the goals of the CROSS project was to identify the required interfaces for certain SDR component types, so we also present our findings regarding these interfaces. The CROSS reference framework source code is heavily documented with Doxygen comments, and in-depth developer documentation is openly available. CROSS is the first step towards fully achieving many of the possibilities discussed in this thesis, and so we also present the positive and negative aspects of our design in hopes that it will serve as a launchpad for further development and research.

1.3 Thesis Organization

This thesis will first provide discussions of the two design methodologies central to our original contributions: component-based development and service-oriented architectures. We then provide an overview of previous and related work regarding CBD and SOAs software-defined radio and communications research. After explaining these ideas, we present our prototype in the form of the CROSS project and discuss its design, implementation, and use. Finally, we conclude with a discussion of potential applications for

such designs and methodologies, future technologies, and thoughts on the future of SDR design and development.

Chapter 2

Component-Based Development

Of the two methodologies discussed in this thesis, Component-Based Development (CBD) and Service-Oriented Architectures (SOAs), CBD is older, better-defined, and better-known in the engineering community. That said, 'better-defined' is still relative, as there are a lot of competing definitions for what CBD means [9], and even what a component is [10]. CBD is often confused with Object-Oriented Programming (OOP).

While many organizations proclaim their use of CBD, very few do it correctly or effectively. This is unfortunate, because CBD has the potential to greatly improve the quality of software in SDR at all levels (i.e., from the SDR platform to the applications running on the platform), reduce development time and costs through increased software reuse, and improve the portability and ease-of-use within SDR platforms. By using CBD, we can also realize a novel use-case: distributed radio deployments.

This thesis asserts that CBD should be more widely applied to the field of SDR, which can only be accomplished by a more wide-spread understanding of the methodology itself. As such, this chapter seeks to accurately and completely define CBD, proposes six characteristics of CBD that should be central to design metrics to ensure the effective application of CBD, and asserts the importance of CBD in Software-Defined Radio design. We also discuss how using CBD enables distributed radio design.

2.1 Understanding Component-Based Development

As previously mentioned, definitions of Component-Based Development (CBD) in the literature vary widely. Before discussing CBD in relation to SDR, we will first clearly define what is being discussed.

One of the most highly cited [9] [11] [10] definitions of a component is given in [12]:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties."

This particular definition is given without any reference to a 'component model', which is a popular method of defining CBD paradigms. Other definitions reflect this model-based approach. A good example of this is from [13], as discussed in [9]:

> A component is a software element that conforms to a component model

> and can be independently deployed and composed without modification according to a composition standard.

The important distinction between the two definitions above is that the second one mandates that a component is only truly defined when described in relation to a specific component model, while the former definition is a general definition for all software components.

For the purposes of this work, all components are defined by the first definition, and the second definition applies when a specific component is created for a particular application. In short, the general idea of a component can be described by the first definition, but a software component is not truly described unless it is accurately defined by the second definition.

The CBD paradigm differs from its predecessor, Object-Oriented Programming (OOP), in that the scope of CBD is much larger. CBD is a paradigm of overall system design, whereas OOP could be used within an individual component. For example, in a large system, built from components written in different languages for different platforms, one of the components may be built using OOP. Using OOP for the entire system, however, would not work as it does not inherently support things like cross-language or cross-platform deployments. OOP is a paradigm meant to increase re-use within software libraries and applications. CBD allows a component created by one group for one application to be used by another group for a completely different application.

In short, "all objects are components, but not all components are objects" [14]. All object-oriented languages, by definition, define an object framework. Objects developed within the same object framework can interoperate with each other, as they share a common access protocol. Components, however, do not necessarily have to be written in an object-oriented framework; each component could be written without object-orientation, and yet the component model could still be successful [14].

2.2 Component-Based Development in Software-Defined Radio

In 1999, an article appeared in *IEEE Computer* that made a case for the rapid adoption of Component-Based Development (CBD) [15]. The CBD paradigm allows for the creation of new programs using pre-fabricated components. This saves development time and money. As explained in [15], "Something must be done to control the costs of developing software products and boosting their quality; any solution requires an industrialization of the process, based on the reuse of standard components...". The requirement for an 'industrializiation of the process' is where CBD comes into play.

Over the next few years, articles began appearing stating that CBD was the future of software development in IEEE-related fields. In 2000, an article in *IEEE Personal Communications* said that CBD "coupled with commercial component marketplaces represents

the next stage in the evolution of software practice," and postulated that all future smart environments would be based on CBD [16]. In 2005, an article appeared in *IEEE Radio Communications* titled, "Is the Wireless Industry Ready for Component Based System Development?" In it, the author describes how CBD is "the Holy Grail" for signal processing design, and could shave months, or even years, off of radio development [17]. The author closes the paper by stating that the market and some critical technologies needed to mature before CBD would be feasible.

Then, in 2007, the same author from [17] published a new article in the *IEEE Communications Magazine* titled "Component-Based Development of Radio Systems and Subsystems: Are We There Yet?" In it, the author states that with the growing number of wireless standards, CBD was more important than ever [6]. After discussing recent advances in relevant technology, the author states that "... it is reasonable to assume that the vision of component-based radio system development is finally within our reach."

While CBD is indeed within our reach, doing CBD **well** is still a challenge. As stated in [15], "... quality is the one issue that stands between us and the realization of the CBD objectives. It seems not to have dawned yet on the industry that components without a draconian attitude to quality at all stages of the process may be worse than the evils they are trying to cure." Component quality is a significant challenge in CBD, and a major risk in its adoption [10]. In order to enforce a high standard of component quality, the industry must adopt a high standard of development. Attempting to make a software module a component without fully understanding what that means will lead to code

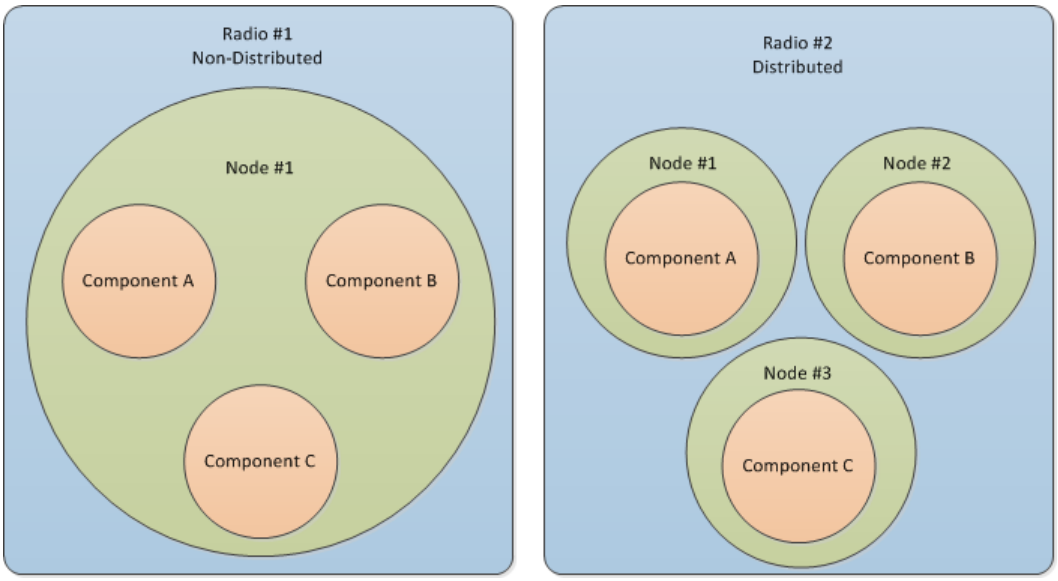
that is unusable by anyone but the author, thus completely defeating the point of CBD. Developers often lack fundamental knowledge and experience in design patterns of their domain, seeking only to produce something that works without regard to its design [18]. This lack of understanding is a critical stumbling block in software reuse and component frameworks.

For CBD to be successful, components must be designed in terms of the CBD paradigm from the outset of the design process [19]. In order for this to happen, the designers must completely understand what CBD is, and what it requires.

The application of CBD also creates possibilities beyond improved re-use and software quality. By enforcing CBD design and implementation, we can realize a new form of SDR deployment in the form of distributed radio. By 'distributed', we don't necessarily mean 'distributed computing' in the computer science sense (i.e., where a single job is distributed to multiple computers to speed up computation time), although that is certainly a possibility. By 'distributed', we are referring to the deployment locations of the components comprising the radio system. This idea is illustrated in Figure 2.1 below.

Figure 2.1 depicts two radios, Radio #1 and Radio #2. Radio #1 is a typical component-based radio. It is comprised of three components, all of which reside on the same hardware node. Applying a strong standard of CBD to SDR, however, allows us to realize the paradigm illustrated by Radio #2 - a distributed SDR. Radio #2 is comprised of the same components that Radio #1 is made of, only each of the components exists on a different hardware node.

Figure 2.1: Two SDRs, both comprised of the same components, but one is local to one node, and one is distributed.



The idea of distributed radio is enabled by the strict enforcement of CBD design principles and a common communications framework. These design principles are presented and discussed in the next section.

2.3 Using Component-Based Development

Clearly, there are both good and bad ways of implementing the CDB paradigm in designs. Simply knowing what CBD is does not necessarily guarantee success in its use. In 2003, the president of TopCoder Software, David Tanacea, wrote an opinion piece in ComputerWorld titled, "Component-Based Development: Why Hasn't the Vision Met Reality?" [20]. In it, he asks why CBD has not been as successful as it ought to be given it's virtues. He cites three primary reasons:

1. A ”lack of discipline and expertise” in the ”beginning stages of software development - research and discovery.” [20]

2. ”Proper application development isn’t done rigorously.” By this, Tanacea means that there is always a rush to start coding, and not enough attention paid to ”[breaking] down an application to its lowest level.” [20]

3. Metrics are required to tabulate the benefits of CBD. This will also ”[make] the discipline of application development easier to enforce.” [20].

In this thesis, we attempt to simplify the the process of using CBD in design methodologies, and incorporate it into larger ideas (e.g., service-oriented architectures). As such, we propose a new way of viewing CBD that focuses on ’best practices’. These are most easily described as desirable characteristics of a component. Note that not all components necessarily need to meet every criteria - the subset of applicable characteristics depends heavily on the component model in use. That said, all of the characteristics are related to each other, and it is difficult to tinker with one without affecting the others.

Desirable characteristics of a component are shown in Table 2.1.

Each of these characteristics will now be described in detail.

#1: Independence - Components should each be completely independent from each other (within the component model). As such, they must encapsulate both state and functionality [11]. Without a high level of independence, it is much more difficult to re-use or port components to other applications, which would defeat the purpose of CBD.

Table 2.1: The six desirable characteristics of a component-based design.

#	Characteristic
1	Independence
2	Immutable Interface
3	Polymorphism
4	Reusability
5	Parameterization
6	Dynamic Binding

#2: Immutable Interface - The level of independence described above necessitates that the component's interface is the only means of control, co-operation, and interoperability. Thus, as explained in [19], a component's interface "... fundamentally defines the service contract between itself and the rest of the world." Maintaining an immutable interface guarantees that versioning, language, platform, etc., are all non-issues. If the interface to a component changes, however, there is a break in compatibility and inter-operability, which is almost never desirable.

#3: Polymorphism - This is the characteristic that describes the ability of a component to have different implementations of the same interface. One implementation of a component should be interchangeable with any other, regardless of platform, programming language, etc.; thus, this characteristic is key in the definition of a 'component model', as it clearly makes the distinction between a component and a component implementation.

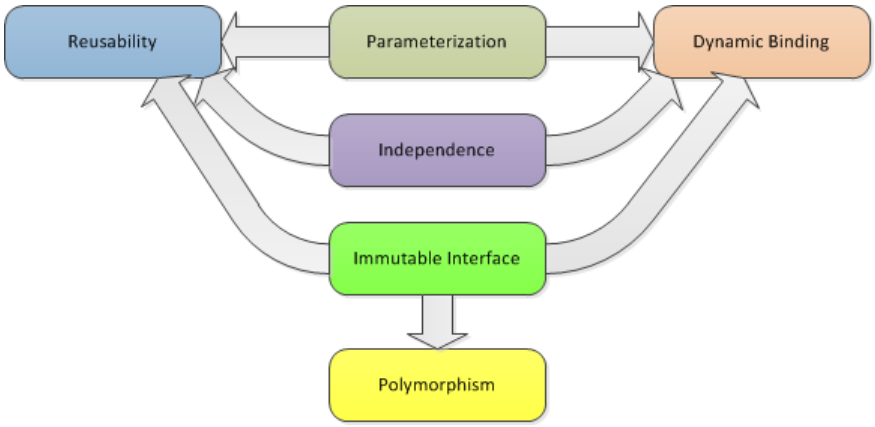
#4: Reusability - This is one of the core benefits of CBD. The component should not only be reusable by the original developer, however, but by third-parties; the users of the component need not necessarily be the developers [9]. This also facilitates the possibility of component marketplaces - essentially, software development with COTS components.

#5: Parameterization - A parameterized component can be configured for specific purposes by simply passing it the appropriate arguments during it's instiation / configuration / etc. This greatly improves component reuse, and enables 'one-size-fits-all' implementations. Additionally, it gives specific application deployments a level of configurability that they would lack otherwise. This is the same idea that drives the field of software radio - by configuring the system in software with parameters, the system has a much greater range of applications.

#6: Dynamic Binding - This is perhaps the most ambitious of the characteristics. This allows an application to dynamically choose which components to use based on what is available to it. This could occur once when the application starts running, or it could take place continuously in real-time.

Clearly, all of these properties are interconnected in one way or another. The degree to which some of the properties exist in the component model can actually determine the effectiveness (or even feasibility) of another. These relationships are visualized in Figure 2.2.

Figure 2.2: How the different characteristics affect each other, where the arrows indicate an enabling relationship.



In Figure 2.2, the arrows indicate that one characteristic enables, or improves the effectiveness, of another. This relatioship is represented in the direction of the arrow. So, for example, an immutable interface, CBD Characteristic #2, enables dynamic binding, Characteristic #6.

The relationship shown in Figure 2.2 is an original contribution of this thesis, and has some important implications. As such, it requires some detailed explanation.

One of the more important, yet subtle, take-aways from Figure 2.2, is that three of the characteristcs enable three others. The 'Parameterized', 'Independence', and 'Immutable Interface' characteristics together enable 'Reusability', 'Polymorphism', and 'Dynamic Binding'. The latter three characteristics don't affect each other, and don't have a reverse-enabling property.

The 'Independence' characteristic affects 'Reusability' and 'Dynamic Binding'. The relationship with 'Reusability' is fairly straight-forward - a component is far less reusable if its

use is dependent on other componentslibraries. In fact, this defeats the very goal of CBD. 'Independence' affects 'Dynamic Binding' in that if a component isn't a stand-alone object, then dynamically binding functionality to it is likely infeasible without pre-defined binding instructions; this in turn greatly limits the benefits of dynamic binding itself.

Designing a component to be 'Parameterized' greatly improves 'Reusability' and 'Dynamic Binding'. For example, if the same general functionality is needed, but can be performed over different data types (e.g., a FFT over complex floats or complex doubles), it is more efficient to have one parameterized component that can do both than have two separate components. Thus, that one component is far more reusable in other implementations, and is an easy provider of the desired functionality for multiple scenarios, thus improving its 'Dynamic Binding' characteristics.

The 'Immutable Interface' property affects three others. Its relatioship to 'Polymorphism' is the most derivative, in that the very definition of polymorphism (in Computer Science) requires an immutable interface for implementations to sit behind. An 'Immutable Interface' aids 'Reusability' in that other developers are far more likely to be able to use your interface if the API is stable; a moving API is a nightmare to maintain. Finally, it enables 'Dynamic Binding' in the same way it enables 'Reusability' - a standard interface to a certain type of component makes dynamicautomatic binding much more feasible.

This relationship mapping gives some indications of what an optimal design methodology, with respect to CBD, might be. By focusing the design process on parameterization, independence, and a strong immutable interface, the other three characteristics can also

be achieved. This somewhat simplifies the design process, and is crucially important in the design of frameworks. This will be discussed further in later chapters.

Thus, while CBD may seem like a complex concept to fully embrace in a design, the methodology for achieving all six of the stated characteristics is actually much simpler.

Chapter 3

Service-Oriented Architectures

Service-Oriented Architectures (SOAs) first originated as a way of building applications for the Internet. SOAs are a relatively recent technology - the Simple Object Access Protocol (SOAP), widely recognized as the first SOA protocol specification, was first submitted to the W3C in May of 2000 [21]. One year later, the W3C published the Web Services Description Language (WSDL) [22]. Vendors and manufacturers have since adopted the service-oriented methodology for web applications at a rapid rate [23].

At first glance, SOAs and CBD might seem like the same general idea with a different name. Indeed, there is even some conflict in describing these two design methodologies in the available literature. Some authors consider SOA to be a 'newer version' of CBD, some authors see SOA as an umbrella for CBD, and yet others see SOA as something separate that works well when used in conjunction with CBD. This work takes the latter

point of view.

Due to the origins of SOAs, many of the descriptions, specifications, and definitions used for SOA concepts refer to web services and applications. Although this thesis is not about web services or applications in any way, these definitions will remain unchanged when quoted or paraphrased in this section to reflect the original context. This section will then make the case for applying SOA to Software-Defined Radio (SDR).

3.1 Understanding Services and Service-Oriented Architectures

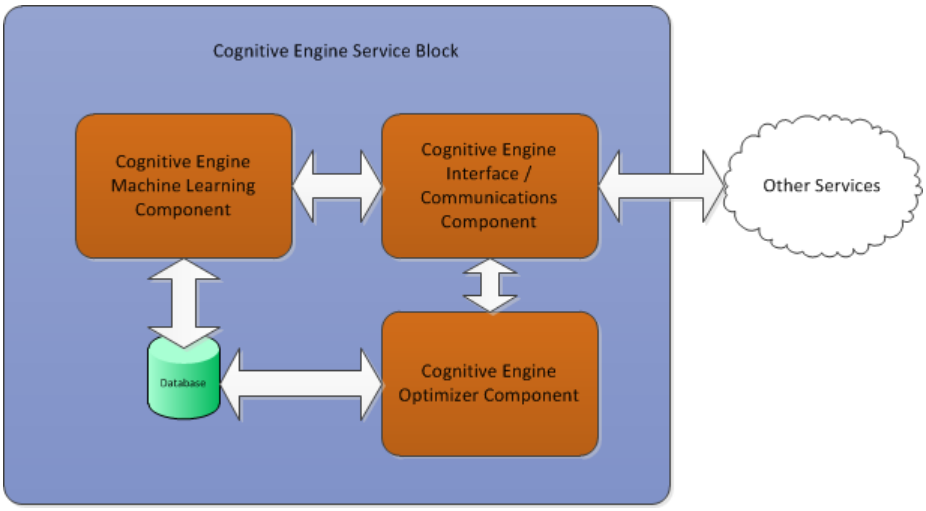
Similar to the definition of a component (see Chapter 2.1), a service is an independent block which, when combined with other services, powers an application [23]. Another good definition is provided by [24], which states that a service ”is a discrete domain of control that contains a collection of tasks to achieve related goals”.

The primary difference between a service and a component is that a service is ”[completely] autonomous from other services” [23]. This is different from the ”Independence” characteristic of components described in the previous section. Components must be independent from each other within the same component/usage model. Services, in order to fit into a Service-Oriented Architecture (defined in more detail shortly), must be completely independent and only bound to other blocks (i.e., services) by a ”standard

communications framework" [23]. This level of separation between blocks (i.e., the high level of independence between services) enables completely platform-independent implemenatations of logical blocks [23].

A shorter comparison can be stated thusly: a service block must be completely autonomous from other service blocks, whereas a component block is not necessarily fully autonomous without other blocks. In fact, a service block can be comprised of component blocks. This concept is visualized in Figure 3.1.

Figure 3.1: An example Cognitive Engine Service, comprised of various CE-related Components.



In essence, service-orientation cleanly partitions and represents resources [25]. So in Figure 3.1, all of the resources and logic for the cognitive engine are contained within that service, which then interacts with other services in the network.

An architecture meant to support service-oriented encapsulations of logic and resources is thus a service-oriented architecture. A SOA requires these encapsulations to "exist au-

tonomously yet not isolated from each other" [25], i.e., blocks connected by a common communications protocol [23]. The architecture itself is the technical description of a networked collection of services, which are then collectively used by applications to achieve the applications' requirements.

Perhaps one of the best ways to think about SOAs on a grand design scale is as the next evolution of the client-server paradigm. As Thomas Erl describes in his Service-Oriented Architectures textbook [25]:

> In the same manner in which main-frame systems were succeeded by client-server applications, and client-server environments then evolved into distributed solutions based on Web techonologies, the contemporary, Web services-driven SOA is succeeding traditional distributed architectures on a global scale.

The service-oriented paradigm been very successful in the realm of Web services. We will now make the case for the use of SOA in the area of Software-Defined Radio.

3.2 Applying Service-Oriented Architectures to Software-Defined Radio

This thesis asserts that by applying Service-Oriented Architectures (SOAs) to Software-Defined Radio (SDR), we can dramatically improve software reuse, interoperability, and autonomic systems (i.e., self-management).

SDR is an extremely diverse field of research in terms of the involved software platforms, hardware platforms, and applications. SDR research targets everything from small-scale sensor networks to massive cellular telephone networks. As a result of this great diversity in SDR deployments, much of the same technology gets re-implemented over and over again each time a new platform is introduced. Software reuse is a common goal in design methodologies, and service-orientation offers a way to improve this reuse across vendors and platforms.

Some groups have attempted to address this problem by creating flexible SDR frameworks, and providing a library of components that can then be re-used for applications developed for that particular framework. These projects are discussed in more detail in the next chapter of this thesis. These solutions all use different component models (described in the above chapter), which means each of the components only work within that specific SDR framework. As a result, a specific application developed using components (e.g., a Dynamic Spectrum Access (DSA) application using a cognitive engine) only works when deployed with the exact SDR framework with which it was designed. Since the component models are different from platform to platform, re-use only extends throughout the one SDR platform.

Standardized service-orientation in SDR frameworks would help alleviate this problem, and thus reduce the time and money spent 're-inventing the wheel' with each deployment. Not only would individual services be usable in different platforms, but applications, either in-part or in-full, could be reused on different platforms. An extension of this

reusability is interoperability of services, and thus applications.

Interoperability between various systems is another great benefit of SOAs. Standardized service descriptions and designs enable interoperability on a large scale because service vendors can be completely ignorant of other services and potential applications, but still create a compatible service implementation. NATO stated that standardized Web services are a major enabler for interoperability between the various military systems across the NATO member-countries [26]. This same operability, extended to SDR platforms, would create incredible new opportunities.

As an example, if service descriptions were standardized, then an application running on one SDR platform could operate using two different services on two other systems, both of which are using a different SDR platform. A service developed on one platform would work with any other service developed on any other platform seamlessly - as long as the service description (described further in Section 3.3) was standard. This is distinct from 'reuse' in that reuse refers to a service being portable between platforms, whereas interoperability refers to a service being capable of working with another service regardless of the platform on which each service runs.

These benefits do not come without challenges, though. Specifically service description and service discovery. These are difficult problems that must be solved before the above application scenario is possible. They are described and discussed later in Section 3.3.

These two properties, reusability and interoperability, together create a sort of free mar-

ket in terms of functional implementations on SDRs. Extensive service reuse and interoperability between services offers the possibility of significant cost and time savings to developers [27]. Vendors could design and implement services, from which developers could select the services that best fit their needs and applications. This sort of free-market of services has worked well in the Web services area, and given the growing interest in SDR, could provide a strong secondary-economy within the field of SDR.

Improved radio self-management is another possible benefit of using SOAs in SDRs. Self-managed radio has become an area of great interest recently, especially with the rise of the ideas like 'smart radio', 'adaptive radio', and 'cognitive radio'. The shared principle between all such concepts is that the radio has some degree of autonomous self-management - it is able to control / adapt / learn / react / etc. without human interference. Well-defined and delimited services enable dynamic recomposition of applications [27], which represents a great opportunity for self-managed radios.

Service-orientation and service-management are beginning to be recognized as avenues for scalable autonomous management [27] [28]. As an example, the U.S. Army identified peer-to-peer service-discovery as an important aspect of their Future Combat Systems initiative [27]. Services, on their own, do not immediately create self-managing applications; they merely enable the concept. In order to truly realize self-managed applications, there needs to be some sort of managing controller.

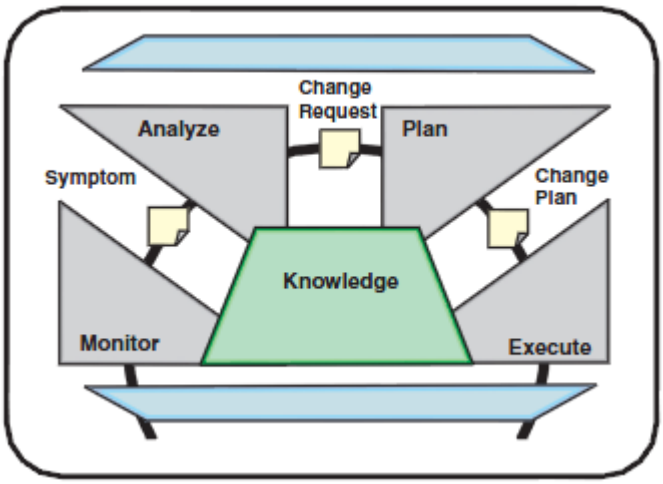
In 2003, IBM identified what they deemed as the main properties of self-management [4] [29]. These properties are shown in Table 3.1:

Table 3.1: The four properties of autonomic computing, as defined by IBM [4].

Property	Description
Self-Configuration	Autonomic systems configure themselves based on high-level goals. The user "[specifies] what is desired, not necessarily how to accomplish it." [4].
Self-Optimization	Autonomic systems optimize their own resource usage - this can happen in a proactive or reactive manner.
Self-Healing	Autonomic systems detect and diagnose problems on their own. Obviously, it is best if the system can also fix the problem, but that is not always possible, depending on what went wrong. The system is thus reactive to failures or signs of a possible failure.
Self-Protection	Autonomic systems protect themselves from malicious attackers and user mistakes. This must be done, principly, in a proactive manner.

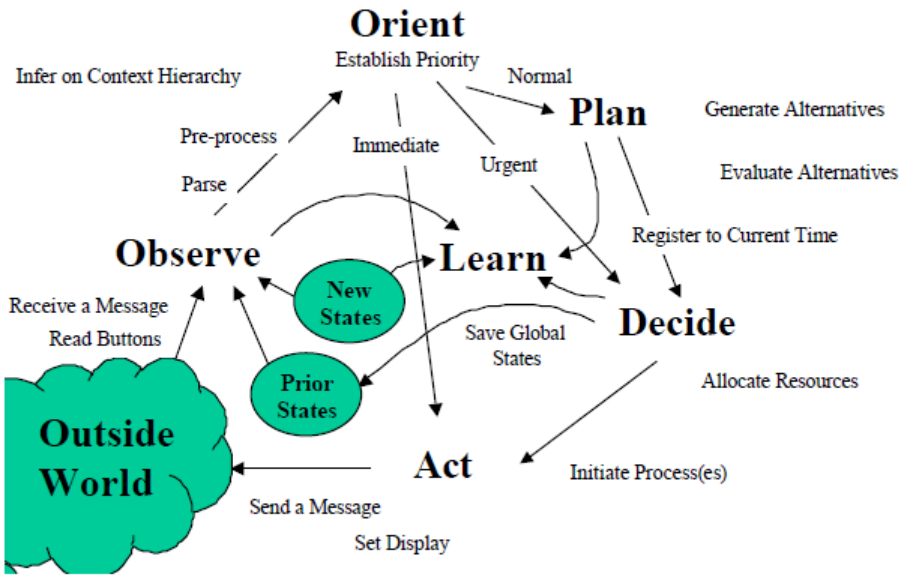
IBM also proposed a model for an autonomic control loop, to be used by an autonomic system to achieve the four properties listed above. This loop is reproduced from the original document in Figure 3.2.

Figure 3.2: The IBM 'MAPE-K' Autonomic Loop, reproduced from original IBM Publication, [1].



The loop depicted in Figure 3.2 is summarized by IBM with the acronym 'MAPE-K', which stands for the five steps in the loop: Monitor, Analyze, Plan, Execute, and Knowledge. For readers familiar with Mitola's vision of Cognitive Radio, this loop looks, and sounds, strikingly familiar to the cognitive engine control loop. This control loop is reproduced from Mitola's Ph.D dissertation [2] in Figure 3.3.

Figure 3.3: The 'Cognition Cycle' for Cognitive Radios, reproduced from Mitola's Ph.D dissertation [2].



The 'cognition cycle' consists of the steps: Observe, Orient, Plan, Decide, Act, and Learn - only slightly different from IBM's 'MAPE-K' control loop. This similarity reflects the obvious connection in purpose: self-management of autonomic systems. Or, more specifically, this thesis is concerned with self-managed radios.

Service-oriented architectures provide an avenue for self-management and autonomy that goes beyond what Mitola envisioned in a cognitive radio using his cognition loop.

Achieving the full scope of IBM's vision for autonomic systems isn't plausible at a fine-grained level - the system is simply too complex. Service-orientation, however, vastly simplifies the system from the viewpoint of self-management, and makes the design of an autonomic unit much easier to realize. As explained by Erl, SOAs are "fundamentally autonomous", and applications comprised of these services are inherently "self-reliant ... that exercise their own self-governance" [25]. A service-oriented, self-managed radio can pursue high-level goals, without any manual action from the user. It could fend off attackers, heal itself, and configure itself per its environment and usage scenario.

These high-level goals represent a new paradigm in radio management. Strictly defining what a high-level goal is would defeat the purpose of their flexibility. Defining the structure of the goal, however, is necessary for their use in design. Thus, we propose the following definitions for autonomic radios:

This thesis proposes that autonomic radios are the next evolution of cognitive radio. Where cognitive radio seeks autonomy in the RF-related (PHY layer) portions of the radio, autonomic systems seek autonomy in all layers of the system. An autonomic system may use a cognitive engine, i.e., the cognition radio paradigm, in part of its PHY layer implementation, but it also uses other modes of autonomic computing to manage the rest of the system to achieve the four properties listed in Table 3.1. This thesis also asserts that a scalable, inter-operable way of achieving this goal in radio systems - especially those that are resource constrained, e.g., mobile and portable nodes - is through the use of service-oriented architectures.

Table 3.2: Definitions for goal-oriented autonomic radios.

Term	Definition
Goal	A single goal that an autonomic radio can pursue. The goal defines an end condition, the realization of which is defined by a measurable metric (i.e., a 'hard' metric). A goal can be an 'active' goal, where the radio actively makes decisions to achieve it, or it can be a 'reactive' goal, where the radio only makes relevant decisions when a certain event(s) occurs.
Goal-Set	The set of goals that the radio is currently pursuing, or can pursue. The goals in a goal-set may have priorities, where some are more important than others.
Goal-Oriented	A autonomic radio is goal-oriented if its operation is defined by its pursuit of successfully achieving a goal-set.

Some of the defining aspects of SOAs - service independence from application and client systems, active service discovery, inherently distributed in nature, etc., - make SOAs an excellent model for mobile and portable radios operating in dynamic environments, such as MANETs. As a mobile radio enters / leaves the communication area of other service-aware nodes, it becomes aware of the services available on those nodes, and can begin using them. This design methodology could even be applied to connecting mobile nodes to high-bandwidth backbones, for example, where the backbone access-point would broadcast that it provides access to that backbone as a service, and mobile nodes could discover it and use it as needed.

A goal-oriented radio must have some sort of system for managing the services available to it, thus achieving goals by leveraging the appropriate services, and completing its goal-

set. The ideal implementation of such a management system would require some sort of optimization / machine learning and artificial intelligence to fully achieve. This area of research exists under the umbrella term ”autonomic systems”, and is currently receiving a lot of attention from the research community. It is also possible to create user-defined intelligence, where decision tracks are pre-decided based on conditionals in the system. The design and implementation of such a system is much easier than creating a fully autonomic system. CROSS provides a prototype design of a user-defined management agent, which is discussed in Chapter 5.

3.3 Making Service-Oriented Architectures Successful

In order for Service-Oriented Architectures to be successful, there is one critical key: standardized service descriptions. The method in which services describe themselves to each other must be an agreed standard [24], otherwise, services will only be interoperable within the same vendor.

One of the goals of the CROSS project, discussed in Chapter 5, is to make progress in discovering the requirements of such service descriptions in the field of software-defined radios. While standards usually come from groups such as the IEEE, this is not totally necessary, and has not always been the default [24]. A mode of standardizing something, which is becoming more and more popular, is releasing the standards document as something that is Free and Open Source Software (FOSS), and letting the market choose its

own standard [24].

First, we will clarify what we mean by 'standardized service descriptions'. This is **not** the same thing as standardized interfaces. If service interfaces were standardized and immutable, it would greatly limit vendors' ability to improve on the status quo. In short, immutable service interfaces would define the maximum potential for each service. This is obviously not desirable. There must be a way, however, for services to communicate with each other, and thus know each others' interfaces.

This can be accomplished through standardized service descriptions. If the method in which a service describes itself to other services is standardized, then services should always be able to understand each other. Each other service that one service is aware of is mapped to a particular interface, thus enabling communication. This is accomplished in the Web services field using the W3C-standardized Web Services Description Language (WSDL). While WSDL is not necessarily applicable to SDR due to the vast differences in requirements, it serves as an excellent example in how to create a successful service description standard.

In addition to the mode of service description being standardized, the mode in which services share these descriptions amongst each other must be standardized. For Web services on the Internet, this occurs using the Simple Object Access Protocol (SOAP). This is perhaps the most difficult of the requirements for SOA in SDR to achieve. Standardizing a description language is platform-independent, but standardizing the mode in which descriptions are shared is not. In the domain of Web services, this mode of communica-

tion is already standardized in the form of TCP/IP; such a standard does not exist in the realm of radio, though.

This, however, is perhaps not as great a problem as it first appears. Pragmatically, services are only available to each other when the nodes the services reside on can talk to each other. In terms of radio, this means that the radios must have the necessary RF capability to communicate with each other (e.g., the approprate TX/RX chain). Given this assumption, a mode of negotiating a communications channel and sharing of service descriptions becomes feasible.

The ability to discover new services and immediately understand their interfaces is critical to the success of SOA in SDR. The environment that autonomic SDR operates in may be subject to frequent changes in the RF spectrum. This is even more true for mobile and portable radios, such as those operating in a Mobile Ad-Hoc NETwork (MANET). The services available to a system at start-up may not be the same services available to it some time later, as the mobile radio system moves around geographically (or even as the RF environment changes due to jamming, other users, etc.,). Thus, "design-time discovery is insufficient because services discovered at design time may not be available when needed" [26].

Chapter 4

The Cognitive Radio Open Source Systems Project

The Cognitive Radio Open Source Systems (CROSS) project began at Virginia Tech because we needed a way to solve a problem, and could not find the necessary functionality in existing software. The CROSS project was never explicitly funded during its primary development period, but rather grew out of necessity.

When CROSS first began, the primary goal was to achieve the vision of component platform-independence and distributed radio. By this, we mean component platform-independence above the SDR platform; in short, creating a new component model for all SDRs that could also be distributed. By appyling Component-Based Development (CBD) to the realm of SDR, we were able to achieve these goals, and realize many other benefits. During this

development, it became apparent that there was also potential to create an incredible new paradigm - a service-oriented radio. This developed into applying SOAs to SDR to create autonomic radios, the idea described in Chapter 3, and realized by the CROSS Service Management Layer (SML).

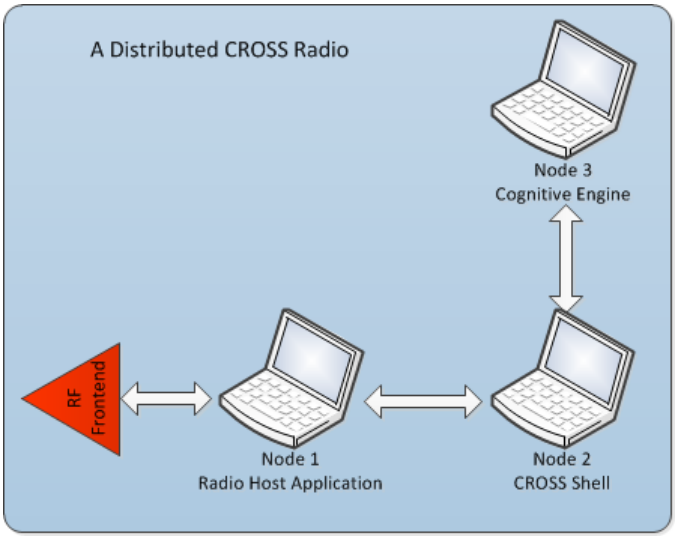
The CROSS project itself is free and open source, available under the Apache License (available at [30]). Hence, development can occur in any group, and forks can be created by anyone. Historically, the project was termed 'VTCROSS' in an attempt to distinguish the team at Virginia Tech from outside parties also developing and using CROSS. This practice was dropped in late 2009, but not before 'VTCROSS' was used in some publications and presentations. The term 'VTCROSS' is now deprecated in favor of 'CROSS'.

4.1 Overview

The goal of the CROSS project is to develop a prototype modular architecture that allows portability, inter-operability, and distributed radio deployments using components that are possibly developed for different hardware and software platforms, or even in different programming languages [31]. In pursuing the first two goals, we hope to enable better collaboration among cognitive radio researchers, and streamline the research and development process. In seeking to create an architecture that enabled distributed radios, i.e., a radio comprised of components deployed to different nodes, we intend to enable a new paradigm in radio deployments. The easiest way to think of a distributed radio is

as multiple nodes acting a single cohesive radio unit. This concept is illustrated in Figure 4.1 using CROSS components (which are themselves described in more detail later in this chapter).

Figure 4.1: Three distinct nodes, represented by laptops, each running a different CROSS component and forming a single radio unit.



In Figure 4.1, there are three separate nodes, represented by images of laptops. Node 1 has access to an RF front-end, and is running an SDR application. By using CROSS, this Node can use components running on the other two nodes, Nodes 2 & 3, to form a single, cohesive radio unit. These three nodes, when connected together with CROSS, represent a single radio from a high-level. This capability is extremely advantageous when there are power-constrained mobile nodes that perhaps need the resources of other platforms. Using CROSS, this use-case becomes feasible.

The goals of CROSS can be achieved by using the principles of Component-Based Design (CBD), as discussed in Chapter 2. The project's deliverable is, very specifically, an

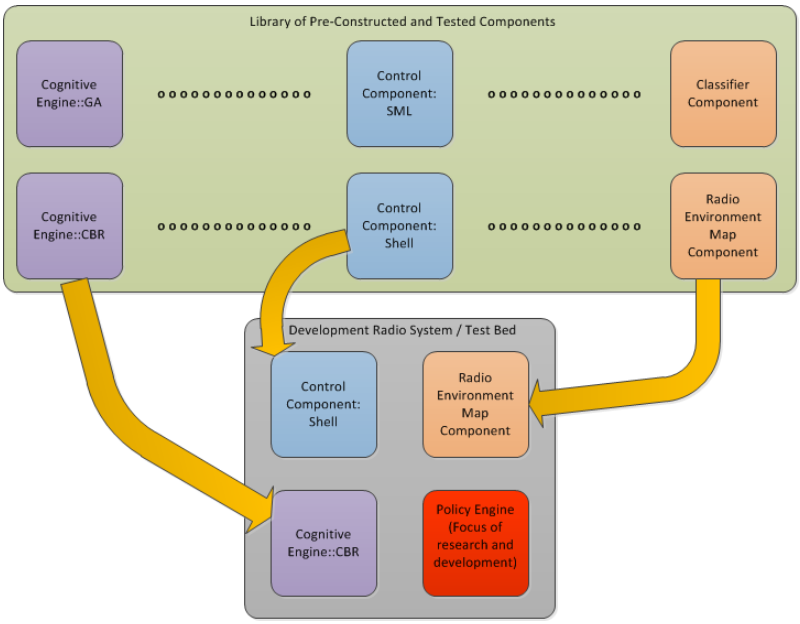
architecture - not necessarily an implementation of it. Thus, the primary requirement in achieving this goal is to define the necessary components, the functionality that each component type encapsulates, and the mode of inter-component communication. The difficult part was thus determining what components were needed to enable any arbitrary SDR application, and what information needed to pass to/from each component type.

One of the major advantages of CROSS is that it enables the creation of a component 'market', so to speak. With component types, descriptions, and communications defined, different groups/vendors can produce libraries of components for use by third-parties. This, in turn, allows researchers to focus on the one particular component of a radio in which they are interested. If a radio developer is working on a new type of policy engine, he/she can build the rest of the radio from other components, developed by other parties, and spend as much time as possible focusing on the new policy engine component. This sort of development process is illustrated in Figure 4.2. This is the same idea and benefit behind shared libraries in software engineering, except in this instance, it is a shared library of components rather than functions.

The development process illustrated in Figure 4.2 is one of the major advantages of CBD. Given a free market of SDR components, quality, selection, and any associated development costs could be dramatically reduced.

Finally, although the CROSS project is centered around the creation of an architecture, we also provided prototype implementations for most of the component types. These

Figure 4.2: Constructing a radio system out of mostly pre-built components so the developer can focus on the component important to him/her.



implementations are specific to our uses, although the architecture remains flexible and available to all platforms. Any discussion of implementation refers to our example implementation, developed on and deployed to GNU/Linux systems.

4.2 Nomenclature

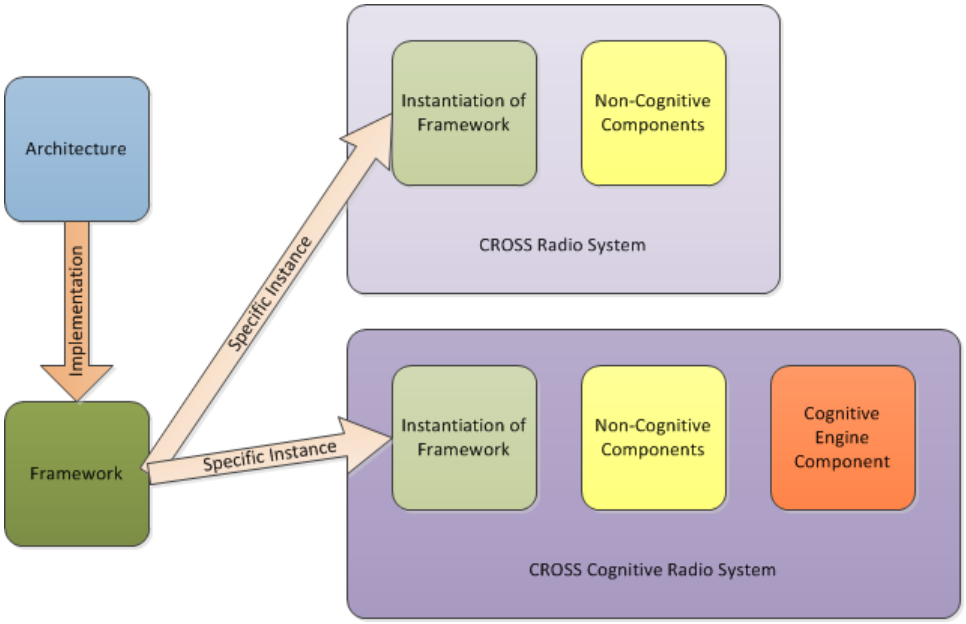
Due to the distributed nature of CROSS radios, we use specific naming conventions to clearly distinguish between different aspects of the radio.

CROSS, on its own, is an architecture. From this architecture, anyone can implement it as a framework. Using this framework, the user can build a complex radio. On its own, however, the CROSS framework does not comprise a radio.

Since CROSS radios can be distributed in nature, simply referring to such a radio as 'the radio' can be misleading. It is not clear whether the reference is to the radio host platform, the RF-frontend, the entire CROSS system, or just a single CROSS component. For that reason, we refer to the entire working radio as a 'CROSS radio system' - the 'system' keyword denoting that the radio itself is comprised of many components, some complex and some simple. By default, a CROSS radio system is not necessarily cognitive. It only becomes a cognitive radio once a cognitive engine component is connected to the system. Once this occurs, we call the entire radio a 'CROSS cognitive radio system'.

This arrangement of architecture, framework, components, and radio systems is visualized in Figure 4.3.

Figure 4.3: A visualized distinction between the CROSS architecture, framework, components, and radio systems.



In Figure 4.3, the CROSS architecture is represented by the blue block on the left. An

implementation of the architecture is represented by the green block below it. This implementation could be used to create a radio system instance with, or without, a cognitive engine - represented by the cognitive SDR and non-cognitive SDR on the right. The following section is dedicated to describing the architecture design itself.

4.3 The CROSS Architecture

The CROSS architecture currently defines five types of components, several of which are optional in a radio system. The five primary components of a CROSS radio system are summarized in Table 4.1.

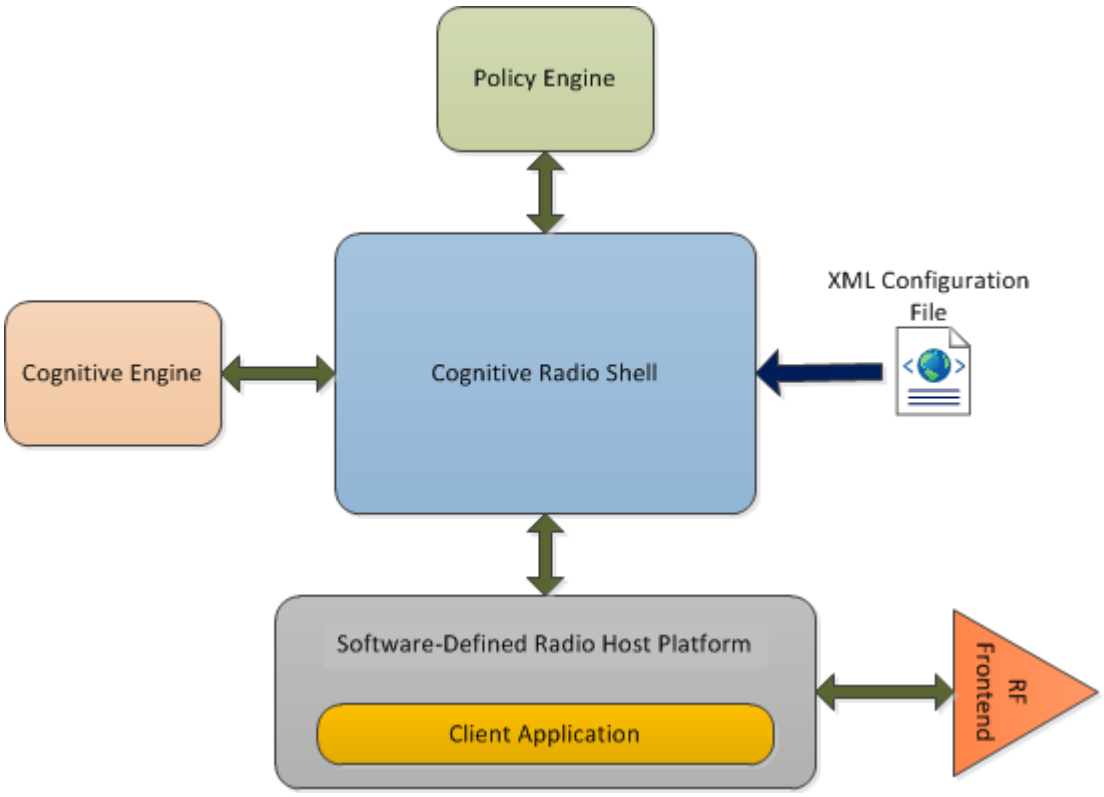
Table 4.1: The core component types of the CROSS architecture.

Name	Type	Description
Cognitive Radio Shell	Control	Central control & message passing.
Cognitive Engine	Core Processing	Optimization processing component.
Policy Engine	Core Processing	Gateway component for policy processing.
Service Management Layer	Control	Enables goal-oriented autonomic radio operation.
SDR Host Platform	Application	The SDR application controlling the radio system.

Of the components listed in Table 4.1, the only absolutely required components for a CROSS radio system are the host platform and the Cognitive Radio Shell (CRS), both of which, along with the other core-components, are described in more detail in below

sections. A CROSS radio system, when **not** using the Service Management Layer and only using one component from each of the type catagories, can be visualized as shown in Figure 4.4.

Figure 4.4: A block diagram of a basic CROSS radio system, including one cognitive engine and one policy engine.



In Figure 4.4, the radio system is using one of each of the core component types with the exception of the SML. Also, note that the image does not clearly designate the locations of each of the components relative to each other. Again, CROSS allows a fully distributed radio system, where each of the components could be on separate hardware nodes - including the RF Frontend. The following sections provide more detail and discussion for each of the components listed in Table 4.1.

4.3.1 Software-Defined Radio Host Platform

The SDR host platform is not a component that would be included in an implementation of CROSS, but it is a necessary component of a CROSS radio system. The host platform is where the client application (sometimes referred to as the 'host application') that interfaces with the rest of the radio system is running. Typically, this is also where the radio hardware (i.e., RF front-end) is located, although this is not a requirement. Even if the CROSS radio is left to manage itself using some form of autonomic control, there must be some 'master application' somewhere that started up the system. This 'master application' is the client application running on the host platform. Without the client application and host platform to get a CROSS radio system running, any CROSS components would sit idle, unaware of the existence of the radio system as a whole. The client application also has responsibilities like shutting down the radio system, telling the Shell where the default configuration file is at boot-time, and other such clerical duties.

Beyond clerical duties, the host application can be anything from a standard TCP/IP stack that uses CROSS to optimize network parameters, to a DSA application that is capable of gathering information about current spectrum use and adapting accordingly. In both of these application examples, the host application holds some sort of control over the radio's operation. Alternatively, the host application can release all operational control over the radio to some sort of control component (e.g., a goal-oriented autonomic component). Where CROSS defines an architecture, the host application describes how the architecture

is used.

4.3.2 Cognitive Radio Shell

The Cognitive Radio Shell (CRS) has several core responsibilities that are integral to component communication and integration within the radio system. It acts as a message passer, configuration parser, and as the interface component from the application to the rest of the radio system.

Before describing the CRS further, we would like to note that the name of this component is not indicative of its duties. The vision of CROSS evolved over time, and the name of this component reflects two facets of previous designs that no longer exist: that CROSS would only be used for cognitive SDRs, and that this component would act as a direct interface between the user and the radio (hence, the term 'Shell', a la a Unix Shell). Neither of these situations exist anymore, but the name persists for historical reasons.

In terms of radio configuration, the CRS is responsible for reading the radio configuration file (more details about this file will be presented later) at boot-up, or at the direction of the host application. The radio configuration file is an XML (Extensible Markup Language, see [32]) file that specifies the operating parameters, environmental parameters, and objectives of the radio's operation (these will all be described in more detail later). The CRS stores this configuration data, and passes it to components that connect to the system as necessary. For example, if a parameterized cognitive engine (see CBD characteristic

#5 in Chapter 2) connects to the radio system, it may require some of this configuration information to become operational.

Another major responsibility of the CRS is to act as a gateway between the host application and the rest of the CROSS radio system. Anytime the host application issues a command to the CROSS radio, this command goes first to the CRS, which then takes the appropriate action. Thus, the CRS is the central control component of a CROSS radio. This single interface to the radio system makes accessing a CROSS system extremely simple, even if the radio itself is relatively complex (e.g., distributed, part of a MANET, etc.,).

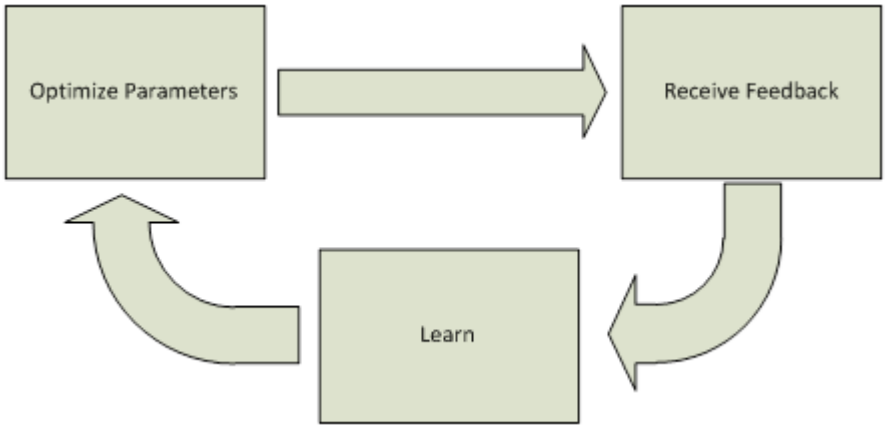
Finally, the CRS is responsible for maintaining the radio system itself - an extension of its already-stated duties of the central command-and-control component. A CROSS radio system is not necessarily static in nature - components may connect and disconnect due to the movement of mobile nodes, channel quality, etc. The CRS is responsible for keeping track of what components are currently connected to the radio, where they are, and how to talk to them.

4.3.3 Cognitive Engine

A full explanation of Cognitive Engines (CEs) is outside the scope of this thesis, but in short, a cognitive engine is responsible for optimizing the PHY layer parameters of a radio. This was first described by Dr. Joe Mitola in his Ph.D dissertation [2]. The full cognition loop, as defined by Mitola, is reproduced in Figure 3.3 in Chapter 3. A simplified

loop, for purposes of a summary description, is shown in Figure 4.5.

Figure 4.5: A simplified operation loop of a cognitive engine.



As depicted in Figure 4.5, the role of a CE is to optimize parameters, and then learn what worked and what didn't through received feedback. The machine learning internal to the CE is dependendent upon the CE deployment, and can be anything from a genetic algorithm to a case-based reasoner. Additionally, the parameters being optimized by the CE vary from case to case - they could be anything from the TX power to the bit rate. In short, cognitive engines can come in many shapes and sizes.

A primary goal in the development of the CROSS framework was to create an architecture that would work with any type of CE implementation. Each cognitive engine implementation has its own benefits and weaknesses, and some might be more appropriate for certain radio systems, or even host applications at a finer level of granularity, than others. In the event that multiple CEs of different types are connected to the same CROSS radio system, the CROSS radio can selectively use the one that best fits the current host application goals.

A major difference between the CROSS system and other systems with more integrated cognitive engines is that the host application dictates when CROSS cognitive engines are required to push new parameters. We do not require the system to constantly generate parameters and push them to the host application; this is of course an **option**, if the application desires it - it just isn't a requirement. This decision, push versus pull, is left up to the host application. Since a CROSS radio system can be distributed over a network (although this isn't necessary), this keeps network bandwidth open when it isn't needed, and allows for more flexible implementation models. It can also reduce power consumption in power-constrained nodes by reducing the required processing.

4.3.4 Policy Engine

Policy engines, also called policy reasoners, are the enabling component of the policy-managed radio paradigm. A full discussion of policy-managed radio is outside the scope of this work, but the fundamental concept is that transmission parameters are checked with a policy engine to ensure compliance with local regulations and laws regarding radio transmission. The policy-managed radio model is the keystone of white-space radios, as identified by the FCC [33] and first investigated by DARPA in the (now obsolete) XG Radio program.

The CROSS architecture provides for policy engine components as one of the core component types. In general, PEs within the CROSS architecture act as a validation phase for

the output of the CEs. When a PE connects to the radio system and the radio is directed to use it by the host application, the CRS will double-check all transmission parameters produced by any CEs with the PE.

The PE determines whether the parameters conform to the active policies and returns a decision array denoting the invalid parameter values (if any) and the reason they were denied. The decision array allows a more fine-tuned approach to policy feedback. Instead of a simple yes-or-no result, the PE can inform the system which values were not acceptable and why. Essentially, the PE is a gateway to transmission when new transmission parameters are produced.

As with all other components in the CROSS system, the PE can be implemented any way the developer chooses. It could be a custom PE with a small policy database and simple decision engine, or a PE could implement the XG policy reasoner and use policies written in OWL (Web Ontology Language, [34]). Again, the only requirement is that the PE implements the CROSS component communication model.

4.3.5 Service Management Layer

The Service Management Layer (SML) is an extremely complex component, and is the only other control component in CROSS other than the CRS. The SML, when connected to a CROSS radio and activated, turns the radio into a goal-oriented autonomic radio, as described in Chapter 3. Essentially, the SML gives the radio the capability to perform

complex missions that may depend on numerous services provided by different components. More complex radio systems may need to execute several different tasks, with the output of one task determining the next task to execute. These complex systems can be created using the SML and building an SML XML configuration file that describes the decision models of the SML.

Section 4.5 describes the SML and its use in more detail.

4.3.6 Non-Core Component Types

When CROSS was first designed, it was created with a more narrow purpose than what it evolved into. It became apparent, after some time, that the core component types listed in Table 4.1 simply were not enough to achieve CROSS's full potential. For more advanced, and not necessarily cognitive radio oriented, applications, other components were necessary - e.g., classifiers, detectors, radio environment maps, etc.

The CROSS architecture, however, does not in any way disallow or somehow prohibit the creation of these components - they just weren't in the original CROSS architecture description. Thus, all components that do not fall into one of the catagories described in Table 4.1 are simply called a 'non-core component type'. The CROSS architecture supports these, but CROSS's state as a learning-experience prototype has made the nomenclature regarding these components somewhat confusing.

4.4 Configuring a CROSS Radio System

As mentioned in Section 4.3.2, a CROSS radio is configured via an XML configuration file. While the CROSS architecture does not explicitely prohibit any fields within the configuration, it does require three specific fields for definition. These fields are shown in Table 4.2.

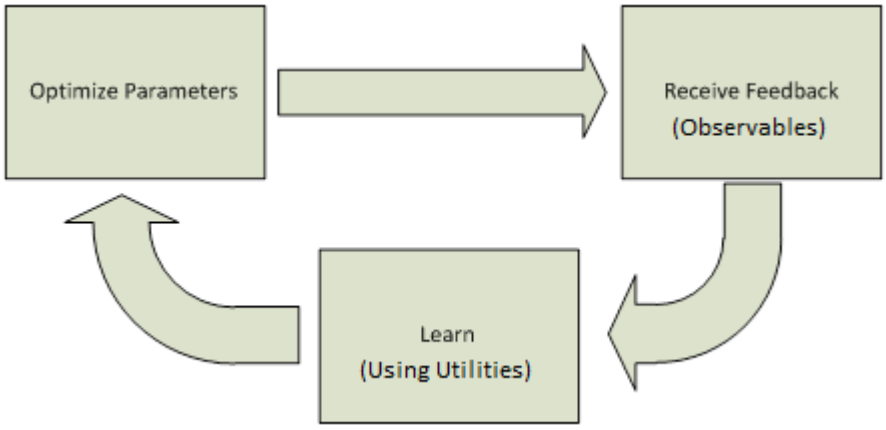
Table 4.2: The three required CROSS configuration fields.

Name	Description
Utilities	The hueristics by which the radio measures its success.
Parameters	The parameters that the radio can optimize / change in order to achieve the Utilities.
Observables	The feedback data with which the radio evaluates the success of its parameter changes.

The three fields in Table 4.2 provide the basic data and informational types necessary for machine learning (such as the machine learning that takes place in a cognitive engine). We previously presented a simplified cognition loop for cognitive engines in Figure 4.5. Adding some subtext to this image to tie it to Table 4.2 produces Figure 4.6.

The loop in Figure 4.6 visualizes how utilities, parameters, and observables are used. It is important to note that the CROSS architecture places no restrictions on how these fields are defined, and what qualifiers are used to define them. As long as the client's CROSS deployment understands the configuration, the fields themselves are arbitrary. By

Figure 4.6: A simplified operation loop of a cognitive engine described using CROSS's configuration fields.



doing this, we achieve Characteristic #5 of CBD (as defined in Table 2.1) - a parameterized system.

An example of a CROSS configuration file defining utilities, parameters, and observables in shown in Figure 4.7.

Figure 4.7: An example CROSS XML configuration file defining parameters, observables, and utilities.

```
<engine name="dsa" filename="dsa.sql">

    <!-- utilities : QoS metrics -->
    <utilities>
        <utility name="interference" units="dB" goal="min" />
    </utilities>

    <!-- radio parameters -->
    <parameters>
        <parameter name="channel" units="none" min="1" max="4" step="1">
            <affect utility="interference" relationship="n/a" />
        </parameter>
    </parameters>

    <!-- link/channel observations -->
    <observables>
        <observable name="energy">
            <affect utility="interference" relationship="n/a" />
        </observable>
        <observable name="communication_time">
            <affect utility="interference" relationship="n/a" />
        </observable>
    </observables>
</engine>
```

In Figure 4.7, the configuration filed defines a single utility, ”interference”. It defines two subfields - the units, and the goal of the optimization algorithm (again, these are all arbitrary, it is up to the optimization engine deployment to make use of these fields). The configuration file then defines a single parameter for tuning - the channel the radio is operating on. Note that it also explicitly states what utilities measure the success of this parameter. Finally, the configuration file defines observables that the radio can use to evaluate performance - here, they are channel energy and communication time. Finally, note that this entire configuration file is geared for a single cognitive engine for a dynamic spectrum access application - thus, it is encapsulated in a single 'engine' tag.

By using this system of utilities, parameters, and observables, we can successfully achieve the goals of parameterization and increased re-use, as defined in Chapter 2.

4.5 The Service Management Layer

The Service Management Layer (SML) is an optional component defined by the CROSS architecture. In a radio system without the SML, the radio operates as per the instructions of the host application, often not taking action without explicit direction to do so.

The SML, when attached to a CROSS radio system and given control over the radio, turns the radio system into a goal-oriented autonomic system (these concepts were discussed previously in Chapter 3). When operating with a SML, the radio is not expecting control instructions from the host application other than clerical instructions like shutting down

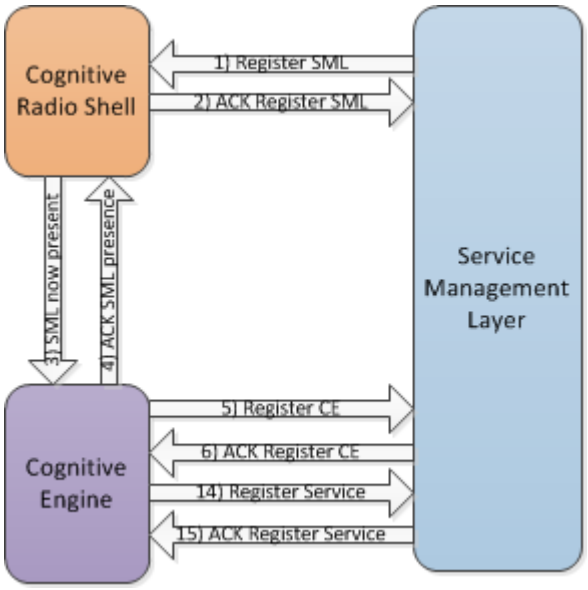
the radio, or re-assuming control from the SML. For all radio operational purposes, when active, the SML is in control of the radio's actions.

There are, however, some other requirements to the SML's operation. When using the SML, the radio system becomes a Service-Oriented Architecture (SOA). As indicated by the SML's name, the SML works by managing services available to the radio system - i.e., services provided by components currently connected to the radio system. This, in-turn, means that the components must be service-aware - they must know what services they are capable of providing, and be able to communicate in a manner appropriate to a SOA. In fact, the two main reasons that the SML is an optional component are that: not all radio systems need to be SOAs, and component implementations can become more difficult when they are required to be a part of a SOA.

When the SML is introduced into a CROSS radio system, it takes control of the system's operation and turns the radio architecture into a Service-Oriented Architecture (SOA). The CRS informs all of the already-connected components of the SML's location, and then relinquishes control of the radio to the SML. All of the other components are then responsible for registering their available services with the SML, which then becomes responsible for controlling the radio in pursuit of one or all of the radio's goals. This process of SML registration, followed by components registering their services, is depicted in Figure 4.8.

Figure 4.8 demonstrates a SML joining a CROSS radio system consisting only of a CRS and CE. Specifically, Figure 4.8 shows the necessary clerical communication for the radio

Figure 4.8: The process by which a SML joins a CROSS radio system.



to re-organize itself based on the new presence of a SML.

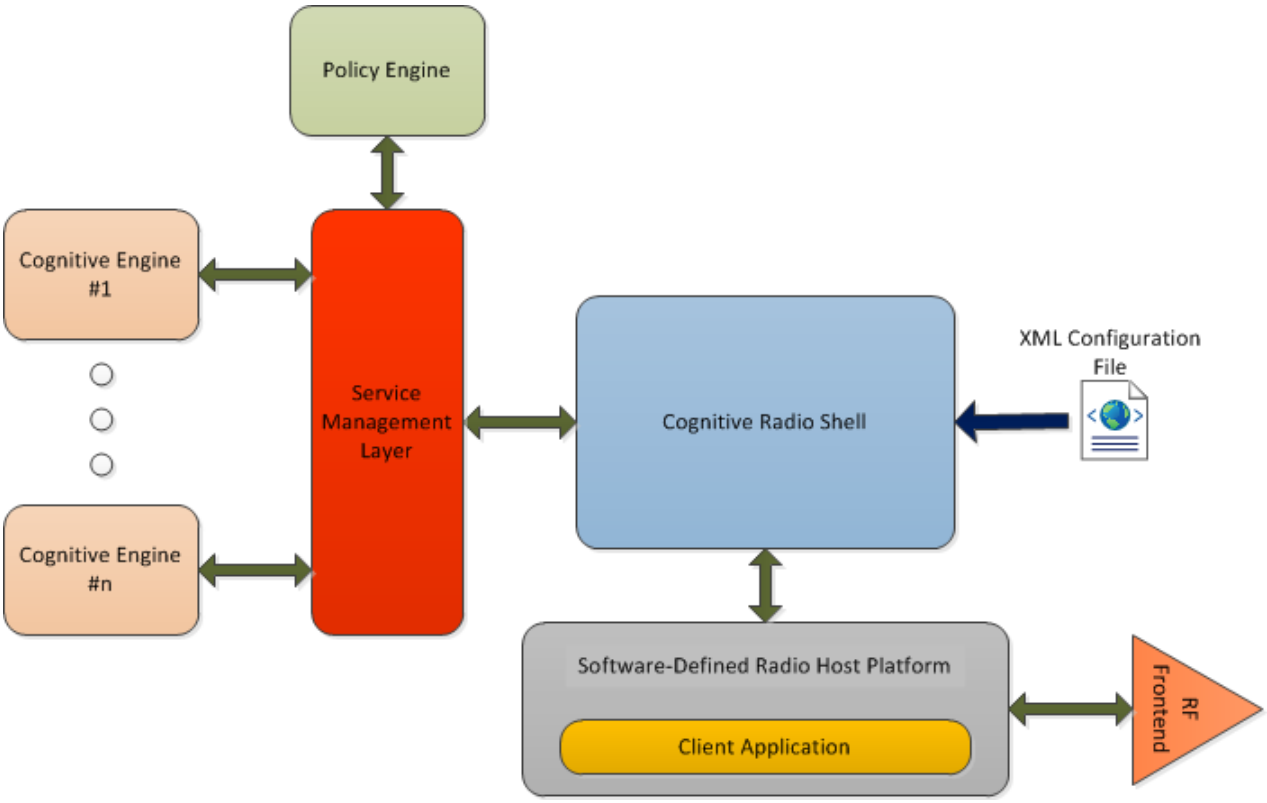
An example of a fully-configured SML-controlled CROSS radio system is shown in Figure 4.9.

In Figure 4.9, the CROSS radio system has re-configured itself to work with the SML. Note that the image shows the existence of an arbitrary number of cognitive engines, to which the SML delegates tasks as necessary. These tasks are driven by the SML's pre-configured goals. Describing these goals is described in Section 4.5.1.

4.5.1 SML Missions

In CROSS, the autonomic system goals, as described in Chapter 3, are referred to as 'Missions' (this, some of the other names in CROSS, is a relic of previous design facets). Thus,

Figure 4.9: A CROSS radio system with an active SML and multiple cognitive engines.

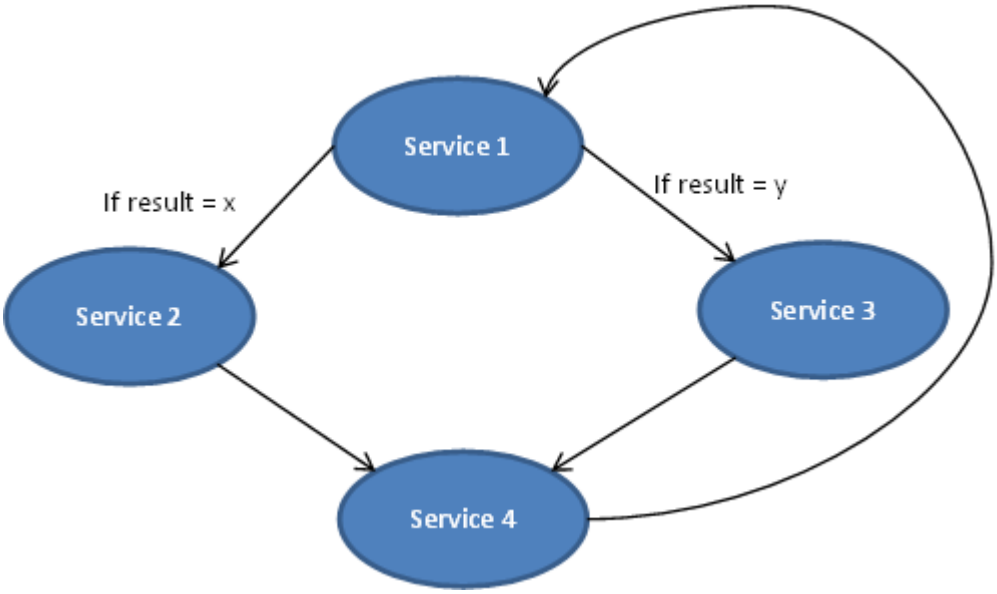


in defining SML Missions, the user is defining the goals of the service-oriented autonomic system.

Using these mission descriptions, the SML delegates tasks to the various components of the radio system in pursuit of accomplishing the specific mission objectives; the SML thus is able to use many, perhaps fundamentally different services, to achieve a higher-level radio objective. Each service can be executed based on feedback from the previous service, and it us to the SML to invoke certain services as necessary. This invocation of services, when pursuing a single mission (i.e., the SML is not actively pursuing multiple missions at once), can be visualized as a Finite State Machine (FSM). This is depicted in

Figure 4.10.

Figure 4.10: Visualizing the SML's operation as a finite state machine.



In Figure 4.10, the SML is aware of four services, and in order to achieve it's mission goals, it must use Services #1 and #4, and either #2 or #3, depending on the output of #1. The SML is capable of analyzing the output from Service #1, and making the appropriate decision.

One of the few responsibilities of the host application while the SML is controlling the radio is to tell the SML which mission(s) it should pursue. This can be changed on-the-fly by the host application, as long as the SML has been configured for the requested mission.

Describing services, missions, and mission goals are described in Section 4.5.2.

4.5.2 Configuring SML Missions

All SML configuration is done via XML files. The XML configuration files define the missions available to the radio, and describe which services are used to accomplish each mission. The configuration file also describes the data-flow between services, dependant on the feedback from the previous service in the flow (i.e., the previous state in the FSM). A very simple mission configuration, without branches, is shown in Figure 4.11.

Figure 4.11: A simple SML mission definition in XML.

```
<mission name="DSACommunication" id="0">
    <services>
        <service name="SpectrumScan"></service>
        <service name="OptimizeParameters"></service>
    </services>
</mission>
```

The mission defined in Figure 4.11 is called ”DSACommunication”, and is assigned the identification number #0. The mission requires the use of two services, ”SpectrumScan” and ”OptimizeParameters”. The SML passes the service output from the former service to the latter. It is important to note that the mission definition does not define which components provide which services - only which services are required.

A more complex mission with branches in the FSM in shown in Figure 4.12.

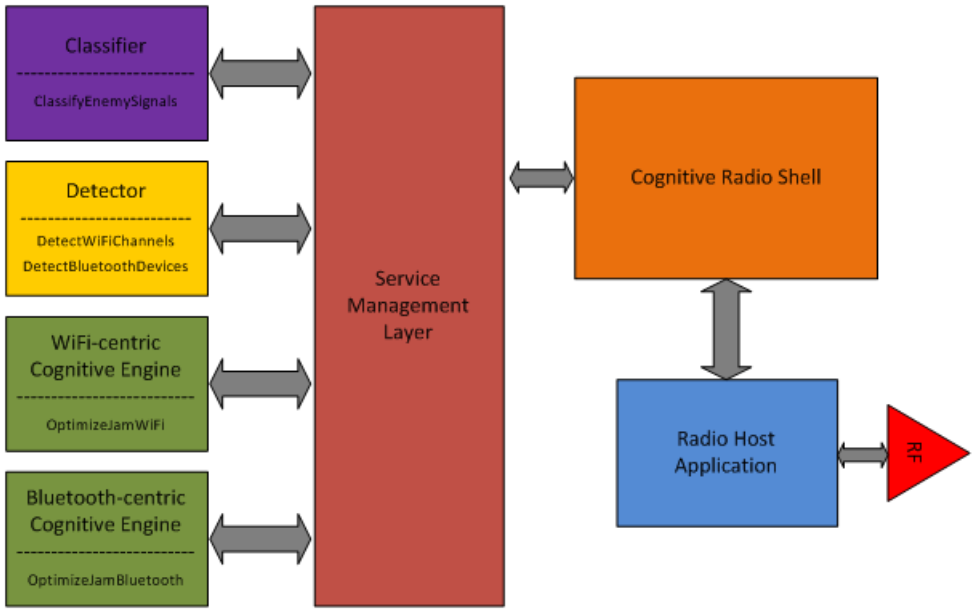
In Figure 4.12, the mission goal is to jam all enemy radio signals, assuming the enemy is using only WiFi and Bluetooth. The first step is classifying the enemy signals, and based on the results of this classification, the SML decides to either execute the appropriate services to jam WiFi or Bluetooth signals. The CROSS radio system performing this mission

Figure 4.12: A more complex SML mission definition in XML.

```
<mission name="JamAllEnemy" id="2">
    <services>
        <service name="ClassifyEnemySignals"></service>
            <if value="wifi">
                <service name="DetectWiFiChannels"></service>
                <service name="OptimizeJamWiFi"></service></if>
            <if value="bluetooth">
                <service name="DetectBluetoothDevices"></service>
                <service name="OptimizeJamBluetooth"></service></if>
    </services>
</mission>
```

might look something like Figure 4.13. Note that this image includes non-core component types, the detector and classifier.

Figure 4.13: An example CROSS radio system for accomplishing the mission described in Figure 4.12.



Given a CROSS radio system like the one shown in Figure 4.13, the SML would seek to accomplish the mission goal of jamming all enemy signals, as described in Figure 4.12. Note that in the event that a mission requires a service that is not provided by any connected component, the SML is unable to pursue that mission. The SML can be configured

with as many missions as the user wishes, as long as the services listed in the missions are provided to the radio system by a component.

4.6 CROSS Interfaces

The CROSS Client Application Interface, also called the CROSS Interface, is what the client application on the SDR host platform uses to hook into the CROSS radio system. These commands are issued to the CRS from the client application. Where the commands are issued from depends on the implementation - all that matters is that they are issued over a socket connection to the CRS. It is also important to note that even when the SML is in control of the radio, commands issued by the client application still pass through the CRS. This was previously shown in Figure 4.9 from Section 4.5.

The CROSS Interface is summarized in Table 4.3.

Table 4.3: The commands provided by the CROSS Client Application Interface.

Command	Data	Description
load_configuration	Configuration File	Tells the radio to load the specified configuration file.
get_utilities	None	Requests the list of utilities loaded from the configuration file.
get_parameters	None	Requests the list of parameters loaded from the configuration file.
		Continued on next page...

Table 4.3 – Continued

Command	Data	Description
get_observables	None	Requests the list of observables loaded from the configuration file.
get_cognitive_engines	None	Requests a list of the currently connected cognitive engines.
get_policy_engines	None	Requests a list of the currently connected policy engines.
get_smls	None	Requests a list of any connected SMLs.
get_connected_components	None	Requests a list of all connected components.
get_component_info	Component	Requests all available information regarding the specified component.
get_optimal_parameters	Parameters	Requests optimized parameters from the appropriate cognitive engine.
update_parameter _performance	Observables	Sends feedback to the radio system in the form of observables regarding the performance of parameters.
send_experience	Experience	Sends training data to the radio for cognitive engine training.
deactivate_component	Component	Deactivate the specified component.
activate_component	Component	Activate the specified component.
shutdown	None	Shutdown the CROSS radio system.

The interface listing shown in Table 4.3 is obviously extendable dependent upon the CRS implementation - this is simply the prototype command listing provided by CROSS.

While the underlying network connecting the components in any CROSS deployment is arbitrary, we have prototyped and tested the CROSS Interface by creating a shared library on a Linux system that implements the functions in Table 4.3. This is discussed further in Section 4.7.

4.7 The Current Status of CROSS

As mentioned previously, CROSS is a prototype architecture developed to help us solve other difficult engineering problems. During the development process of CROSS, we explored the benefits of CBD and SOA, as described in Chapters 2 and 3. The CROSS project has achieved its goals wonderfully, but has not been updated since it reached an operational stage (as of the time of this thesis).

As part of the CROSS project, we have created a reference implementation of the CROSS architecture, complete with the core component types and required interfaces. Our CROSS components exist within a strict C++ class hierarchy, defined primarily by abstract base classes which serve as parent classes within the object-oriented class tree. This implementation is completely Free and Open Source Software (FOSS), which enables any developer using our implementation of CROSS to quickly develop a new component without re-implementing the functionality that already exists in other components.

This reference code is written for a GNU/Linux system using the GCC toolchain. When this code is compiled for the host platform (on which the client application is running),

a shared library is generated which can be used within applications on that system to control the CROSS radio. This library is a direct implementation of the CROSS Interface, described in Section 4.6.

To demonstrate the power of this framework, we created a Dynamic Spectrum Access demo using our CROSS implementation.

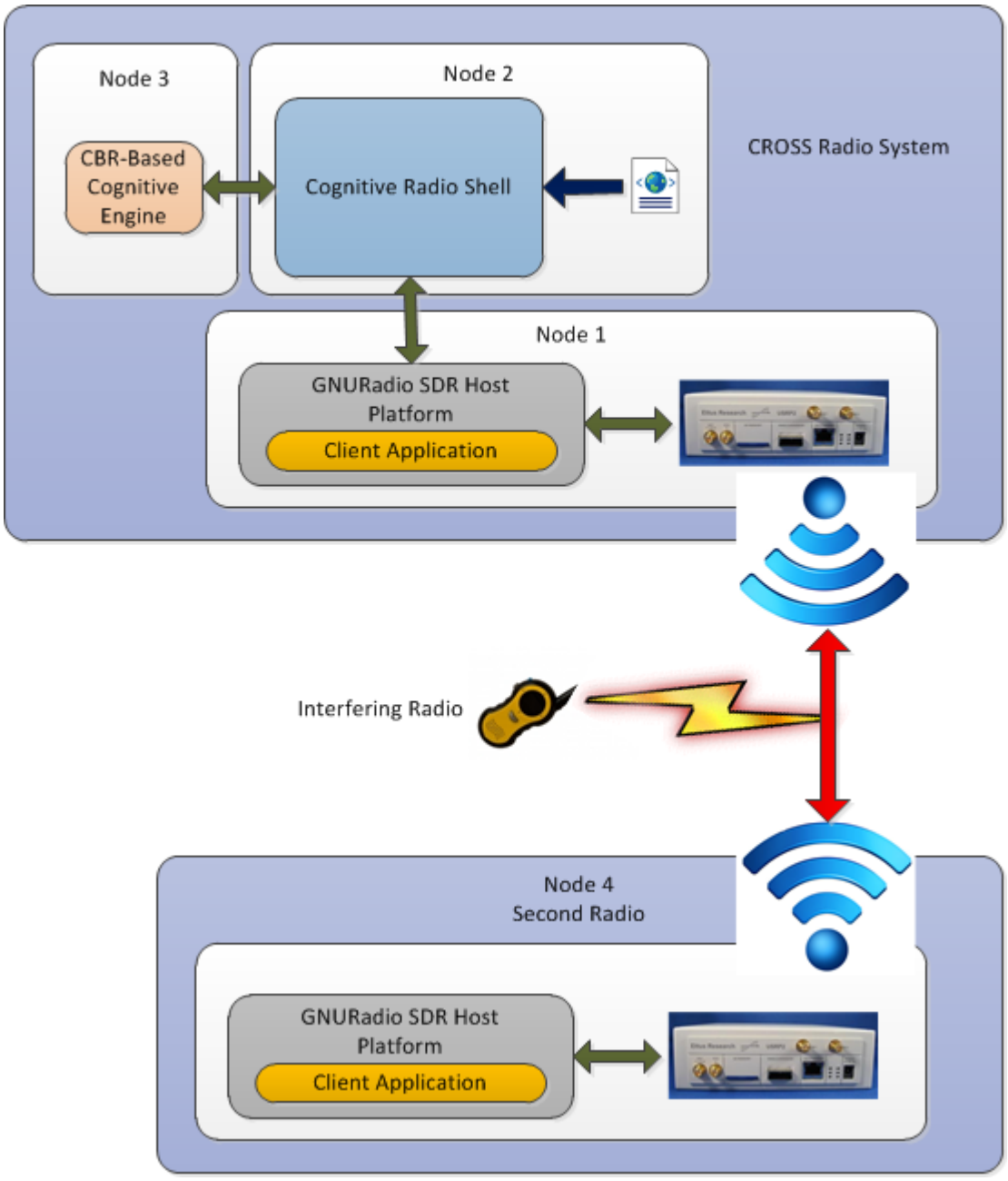
4.7.1 The CROSS DSA Demonstration

Using our reference implementation codebase, we demonstrated a distributed CROSS radio that performs Dynamic Spectrum Access (DSA) in the FRS radio bands to communicate with another radio. DSA is one of the most popular applications for cognitive radio, and the concept has now evolved into what are called 'White Space Devices'.

In our demo, there is one CROSS radio system, another radio that is communicating with the CROSS radio (transmitting and receiving), and an interferrer (in our case, the interferer is someone with an FRS-band walkie-talkie transmitting over the channel). The SDR platforms are both GNURadio, and RF frontends are Ettus Research USRP2s [35]. The CROSS radio system uses a Case-Based Reasoning (CBR) cognitive engine to optimize its channel selection; it tends to avoid channels that it previously experienced interference on. The CROSS radio and secondary radio communicate over a channel, and evacuate when another user (i.e., the walkie-talkie interferrer) appears on the channel. The radios then rendevous on another channel, where the CROSS radio chooses channels based on

its CBR optimized parameter output. This demo layout is visualized in Figure 4.14.

Figure 4.14: The experimental setup of the CROSS DSA demonstration.



As shown in Figure 4.14, the CROSS radio was distributed across three separate nodes. The CBR-based CE was on one node, the CRS was on another node, and the host plat-

form, client application, and RF frontend were on a third node. All three of these nodes acted as one cohesive radio using Ethernet as the communications layer between them. The second radio with which this CROSS radio was communicating was simply running a GNURadio script that sent and received dummy data, and cycled through available channels upon channel evacuation. While the CROSS radio and secondary radio were communicating, we would suddenly cause interference on the channel using a walkie-talkie. Both radios would evacuate the channel. The CROSS radio would report to the CBR CE that the channel parameter had failed, and request a newly optimized channel. The CE would then respond with a new channel parameter, and the CROSS radio would tune to that channel and begin sending beacons. The secondary radio, in the meantime, was simply cycling through channels looking for the CROSS radio's beacons. Upon seeing one, it would respond, and resume communications.

The CROSS radio system operated using the specified CROSS interfaces in Section 4.6, and the source code for the framework implementation is available at [31].

4.8 Learning from the CROSS Prototype

In terms of an architecture and framework meant to prototype the ideas of CBD and SOA, CROSS has been very successful. That said, it was a prototype design, and as such, there are a number of things that were done well, and a number of things that could be improved upon in the future or in the next iteration of this technology. The positive things

to come out of the CROSS project are summarized in Table 4.4.

Table 4.4: The positive aspects of the CROSS design.

Aspect	Description
Configurability	Using XML to define observables, parameters, and utilities is an excellent way to achieve parameterization of components.
Component-Based Design	Using CBD allowed us to actually realize a completely distributed radio system that was independent from the SDR and hardware platforms of the system.
Interface Requirements	We identified the basic requirements of certain component types and created simple, but powerful, interfaces.
Service-Orientation	The idea of a service-oriented radio grew out of CROSS, and from there, the concept of autonomic radio systems evolved.

Equally important are the aspects of CROSS that can be improved upon in the future. These aspects include some things that we actively made a decision about and later realized a better option, and others that were side-products of unrelated decisions. These aspects are summarized in Table 4.5.

Table 4.5: Aspects of the CROSS design that could be improved.

Aspect	Description
Cognitive Radio Confusion	When the CROSS project began, it was intended solely for cognitive radio applications. As a result, much of the nomenclature and component types were designed specifically for cognitive radios. This made improving the architecture's flexibility for non-cognitive radios confusing in the future.
Message Passing Bottlenecks	Using a single component as a hub for message passing bottlenecks the system somewhat. The components should have been designed to talk directly to each other after a start-up period to cut down on latency.
SML Configuration	There are better choices for the SML configuration file than XML. Alternatively, the SML XML tags need to be changed. Ideally, the configuration file should reflect a finite state machine inherently.
SML Metrics	The SML configuration file should require metrics to evaluate the degree of success or failure of the radio when operating autonomously.
ASCII Commands	Using ASCII for commands worked great during the prototype stages. In an operational system, however, using strings slows down the system compared to bit comparisons in packet headers.

Many of the aspects detailed in Table 4.5 are easy to improve, especially if taken into consideration early in the design process. Somewhat ironically, the positive aspects of CROSS caused it to grow so quickly that many of the negative aspects are items that just did not get fixed during the rapid growth period. That said, there is no reason that we

cannot maintain backward-compatibility when remedying these issues.

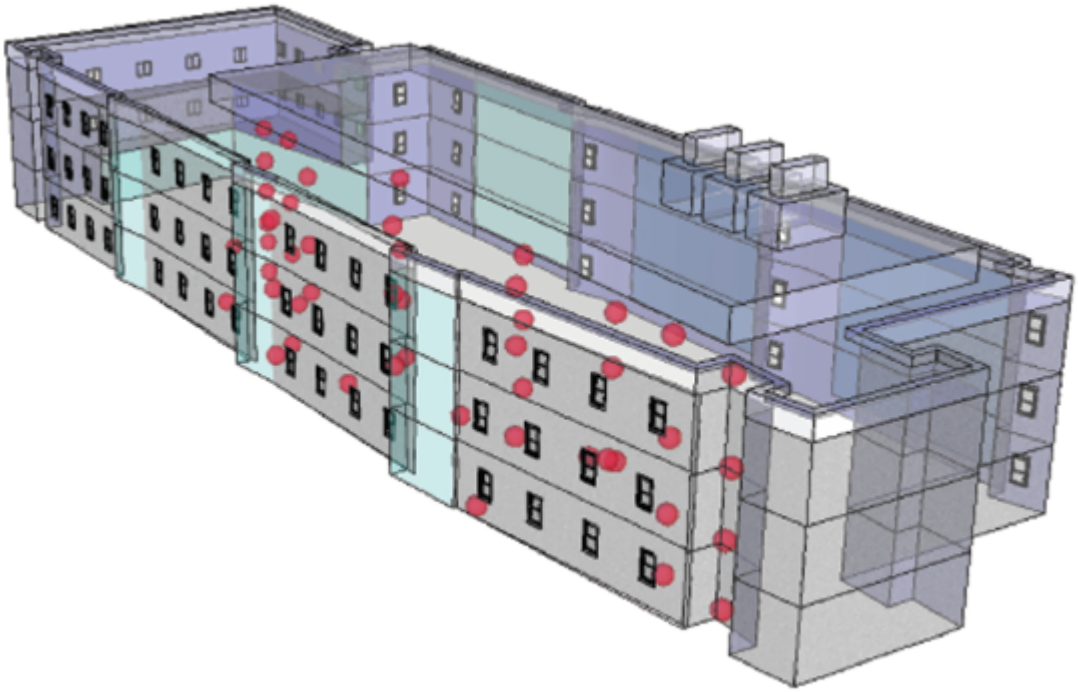
4.9 Known CROSS Deployments

Since it's development, the CROSS architecture (and in some cases our reference implemantion) has been used by a number of parties for various projects. This section is titled as '**Known** CROSS Deployments', because since CROSS and our reference CROSS framework are free and open source software, there may be other groups using the architecture and/or framework that we are unaware of. This section will detail the uses that we are aware of.

The largest deployment that we are aware of is in the COgnitive Radio Network Testbed (CORNET), located at the Institute for Critical Technology and Applied Science (ICTAS) at Virginia Tech. CORNET is a network of 48 software radios, deployed over four floors of the ICTAS building at Virginia Tech's Blacksburg campus. The testbed is openly available for use by people both within and outside of the university, and can be accessed and controlled easily through SSH. A diagram of the node deployments in the building can be seen in Figure 4.15.

Our reference CROSS framework is installed on each of the nodes by default, which enables the creation on many different distributed radios within CORNET. It also has the advantage of enabling researchers to test components they have created and are stored locally (i.e., outside of CORNET) with the RF frontends provided by CORNET. For exam-

Figure 4.15: The node deployment of CORNET in the ICTAS building [3].



ple, a researcher at a different university could create a CROSS component on his / her local laptop, connect it to CORNET, and create a radio system using the component, and CORNET hardware. This enables easy prototyping of components, and makes SDR much more accessible to researchers as they are not required to have their own RF hardware to test their SDR components. To our knowledge, this capability is unique to the CROSS + CORNET testbed deployment. More information about CORNET can be found in [3].

CROSS has also been used within Virginia Tech for projects related to cognitive radio applications, including Dynamic Spectrum Access and position location applications. We are also aware of CROSS being used in the defense industry for cognitive radio-related

projects.

Chapter 5

Conclusion

In this work, we successfully applied the concept of component-based design to software-defined radio to create a modular and distributed radio architecture. We used the knowledge gained from this process to define six properties that can be used as design goals and metrics for sucessfully using the CDB process. We presented the relationship between these metrics, and asserted which properties enable others, thus simplying the application of the CBD paradigm. We discussed how products of this process can enable a new type of software-defined radio: a distributed radio - where components running on many different hardware nodes act as one cohesive radio unit. This is a novel application of component use in SDR, and represents a new development in radio deployments.

We also presented the novel idea of autonomic radio systems, enabled by service-oriented architectures. We asserted that the application of SOAs to SDR represents the next evolu-

tion of software-defined radios in the form of goal-oriented autonomous radio systems. We described the benefits of using SOAs in SDR, defined design terminology, and discussed some of the difficulties in achieving this concept.

Finally, we presented the CROSS prototype - an architecture implementing CBD and exploring SOAs in SDR. CROSS has proven the usefulness of the design methodologies discussed in this thesis, and we successfully demonstrated a distributed radio performing the cognitive radio application of Dynamic Spectrum Access. We presented the lessons learned from CROSS, from both the good and bad aspects of the design, in hopes that our experience serves as a starting point for the next phase of research in this area.

5.1 Future Work

We hope that the ideas, concepts, and lessons presented in this thesis serve as a launchpad for future endeavors in the field of software-defined radio. The application of CDB to SDR makes many challenging SDR deployments feasible, and opens the door to new opportunities in SDR applications. Distributed radio deployments are the first step towards complex use-cases like Wireless Distributed Computing (WDC), or mobile cloud computing. Extending this technology by improving the protocol used for inter-component interfaces would be a great step towards solving even more complex problems.

The concept of using SOAs in SDR is perhaps the most novel idea presented by this thesis, and there is a large amount of future possible work in this area. While many facets of

the concept of goal-oriented autonomic radios, as described in Chapter 3, are challenging, perhaps one of the most critical issues for mobile and ad-hoc networks is service discovery. The other major challenge is in creating a fully autonomous service-management agent that minimizes the input required from a user (at configuration time, not real-time) to make decisions based on its environment and goals. These problems incorporate research from many areas, and must be confronted to successfully realize mobile fully-autonomic radios.

Service-orientation could also be used in the area of testing and verification for software radios. By designing services that handle the testing of software radios, and the verification of their security, these quality-assurance processes could be vastly simplified during the development of new software radio systems. Designing these functions as services would also allow the radio to connect to the service provider while the radio is deployed, and perhaps have maintainence performed remotely and automatically. This use of service-orientation holds great promise, and could serve as an excellent test application for the SOA paradigm in SDR.

The application of CBD and SOA to software-defined radio can be an immense enabler of future technology, regardless of whether or not it involves CROSS. While CROSS serves as an excellent example and learning experience, it isn't necessary to use CROSS to benefit from the ideas of this thesis. Further exploring and developing the concept of autonomic radios is simultaneously the most ambitious and most promising, in terms of possible applications, of the future work enabled by this thesis.

5.2 Publication List

- Hilburn, B.C.; et. al.; "A Survey of Channel Selection Techniques for Cognitive Radios," Software-Defined Radio Forum Technical Conference, 2010. SDRF, 2010. Washington, D.C.

- Hilburn, B.C.; et. al.; "CROSS - A Distributed and Modular Cognitive Radio Framework," Software-Defined Radio Forum Technical Conference, 2009. SDRF, 2009. Washington, D.C.

- Hilburn, B.C.; Newman, T.R.; Bose, T.; "Sector-based policy generation and enforcement for cognitive radios," Military Communications Conference, 2009. MILCOM 2009. IEEE, pp.1-7, 18-21 Oct. 2009

- Newman, T.R.; An He; Gaeddert, J.; Hilburn, B.; Bose, T.; Reed, J.H.; , "Virginia tech cognitive radio network testbed and open source cognitive radio framework," Testbeds and Research Infrastructures for the Development of Networks & Communities and Workshops, 2009. TridentCom 2009. 5th International Conference on, pp.1-3, 6-8 April 2009

Bibliography

[1] IBM, "An architectural blueprint for autonomic computing." IBM, White Paper 3rd Ed., June 2005.

[2] J. Mitola, "Cognitive radio: An integrated agent architecture for software defined radio," Ph.D. dissertation, Royal Institution of Technology, Stockholm, Sweden, 2000.

[3] T. Newman, S. Hasan, D. Depoy, T. Bose, and J. Reed, "Designing and deploying a building-wide cognitive radio network testbed," *Communications Magazine, IEEE*, vol. 48, no. 9, pp. 106 –112, 2010.

[4] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41 – 50, Jan. 2003.

[5] N. Gold, A. Mohan, C. Knight, and M. Munro, "Understanding service-oriented software," *Software, IEEE*, vol. 21, no. 2, pp. 71 – 77, 2004.

[6] L. Pucker, "Component-based development of radio systems and subsystems: Are we there yet? [trends in dsp]," *Communications Magazine, IEEE*, vol. 45, no. 6, pp. 44

–46, 2007.

[7] Open source sca implementation :: Embedded, ossie. [Online]. Available: http://ossie.wireless.vt.edu/

[8] Gnuradio. [Online]. Available: http://gnuradio.org/redmine/wiki/gnuradio

[9] K.-K. Lau and Z. Wang, "Software component models," *Software Engineering, IEEE Transactions on*, vol. 33, no. 10, pp. 709 –724, 2007.

[10] S. Mahmood, R. Lai, and Y. Kim, "Survey of component-based software development," *Software, IET*, vol. 1, no. 2, pp. 57 –66, 2007.

[11] R. Stets, G. Hunt, and M. Scott, "Component-based apis for versioning and distributed applications," *Computer*, vol. 32, no. 7, pp. 54 –61, Jul. 1999.

[12] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming*, 2nd ed. Addi, 2002.

[13] G. Heineman and W. Councill, *Component-Based Software Engineering; Putting the Pieces Together*, eds., Ed. Addison-Wesley, 2001.

[14] B. Wallace. (2010, May) A hole for every component, and for every component a hole. [Online]. Available: http://existentialprogramming.blogspot.com/2010/05/hole-for-every-component-and-every.html

[15] B. Meyer and C. Mingins, "Component-based development: from buzz to spark," *Computer*, vol. 32, no. 7, pp. 35 –37, Jul. 1999.

[16] C. Herring and S. Kaplan, "Component-based software systems for smart environments," *Personal Communications, IEEE*, vol. 7, no. 5, pp. 60 –61, Oct. 2000.

[17] L. Pucker, "Is the wireless industry ready for component based system development?" *Communications Magazine, IEEE*, vol. 43, no. 6, pp. s6 – s8, 2005.

[18] D. C. Schmidt. (1999, January) Why software reuse has failed and how to make it work for you. [Online]. Available: http://www.dre.vanderbilt.edu/~schmidt/reuse-lessons.html

[19] P. Houston, F. Wilkie, and T. Anderson, "Component-based development, corba and rm-odp," *Software, IEE Proceedings -*, vol. 145, no. 1, pp. 22 –28, Feb. 1998.

[20] D. Tanacea. (2003, February) Component-based development: Why hasn't the vision met reality? Op-Ed. ComputerWorld. [Online]. Available: http://www.computerworld.com/s/article/78077/Component_Based_Development_Why_Hasn_t_the_Vision_Met_Reality_?taxonomyId=11&pageNumber=1

[21] W3C. Soap specifications. W3C. [Online]. Available: http://www.w3.org/TR/soap/

[22] ——. Wsdl specifications. W3C. [Online]. Available: http://www.w3.org/TR/wsdl

[23] T. Erl, *Service-Oriented Architecture: A Field Guide to Integrating XML and Web Services*, I. Pearson Education, Ed. Prentice-Hall, Inc., 2004.

[24] S. Jones, "Toward an acceptable definition of service [service-oriented architecture]," *Software, IEEE*, vol. 22, no. 3, pp. 87 – 93, 2005.

[25] T. Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, I. Pearson Education, Ed. Prentice-Hall, Inc., 2005.

[26] F. Johnsen, T. Hafs ande, A. Eggen, C. Griwodz, and P. Halvorsen, "Web services discovery across heterogeneous military networks," *Communications Magazine, IEEE*, vol. 48, no. 10, pp. 84 –90, 2010.

[27] N. Suri, M. Marcon, R. Quitadamo, M. Rebeschini, M. Arguedas, S. Stabellini, M. Tortonesi, and C. Stefanelli, "An adaptive and efficient peer-to-peer service-oriented architecture for manet environments with agile computing," in *Network Operations and Management Symposium Workshops, 2008. NOMS Workshops 2008. IEEE*, 2008, pp. 364 –371.

[28] D. Wu, S. Ci, H. Luo, H. Wang, and A. Katsaggelos, "A quality-driven decision engine for live video transmission under service-oriented architecture," *Wireless Communications, IEEE*, vol. 16, no. 4, pp. 48 –54, 2009.

[29] D. F. Bantz, C. Bisdikian, D. Challener, J. P. Karidis, S. Mastrianni, A. Mohindra, D. G. Shea, and M. Vanover, "Autonomic personal computing," *IBM Syst. J.*, vol. 42, pp. 165–176, January 2003. [Online]. Available: http://dx.doi.org/10.1147/sj.421.0165

[30] (2004) Apache license v2.0. Apache Software Foundation. [Online]. Available: http://www.apache.org/licenses/LICENSE-2.0.html

[31] "Cross." [Online]. Available: http://www.cornet.wireless.vt.edu/trac/wiki/Cross

[32] Extensible markup language, xml. W3C. [Online]. Available: http://www.w3.org/XML/

[33] Federal Communications Commission, "Facilitating opportunities for flexible, efficient, and reliable spectrum use employing cognitive radio technologies," FCC, NPRM and Order Docket No. 03-322, December 2003.

[34] Web ontology language, owl. W3C. [Online]. Available: http://www.w3.org/2004/OWL/

[35] Usrp2 datasheet. Ettus Research. [Online]. Available: http://www.ettus.com/downloads/ettus_ds_usrp2_v5.pdf