ARCHITECTURE FOR AN OPEN-SOURCE COGNITIVE RADIO[*]

Erich Stuntebeck (Georgia Tech, Atlanta, GA; eps@gatech.edu);
Timothy O'Shea (NC State University, Raleigh, NC; tjoshea@ncsu.edu);
Joseph Hecker (Clemson University, Clemson, SC; heckerj@clemson.edu);
T. Charles Clancy (Laboratory for Telecommunications Sciences, College Park, MD; clancy@ltsnet.net)

ABSTRACT

OSCR, a framework for the implementation of a cognitive radio, is designed to facilitate the integration of a cognitive engine with one or more existing Software Communications Architecture (SCA) based radios. It consists of two components: a multiplexer, which acts as the cognitive engine's point of control for each individual radio within the system, and an SCA resource within each radio, which translates between the radio's native control API and the OSCR API. OSCR is designed to integrate multiple radios, which may all have differing capabilities, under a single cognitive engine. To demonstrate the usefulness of OSCR, used it to construct a cognitive radio composed of a basic SCA-compliant software radio and a cognitive engine designed to maximize channel capacity by monitoring channel statistics and varying radio parameters. Implementation details and results of experiments with this system are provided.

1. INTRODUCTION

The research community's interest in cognitive radio has spawned numerous proprietary and incompatible environments for the evaluation of algorithms and protocols. There is a need for a universal architecture allowing the reuse of cognitive radio algorithms across a wide variety of radio platforms. In this work, we present such an architecture, which we call OSCR – The Open Source Cognitive Radio. OSCR is designed to facilitate the integration of a cognitive control engine with one or more SCA-compliant software radios. Since the SCA does not dictate a standard API for the control of software radios, a cognitive engine must currently be modified for compatibility with each different radio that is to be supported. OSCR integrates an SCA resource within each radio which presents a standardized control API – the OSCR API – to the outside world. Thus, the OSCR framework allows any cognitive control engine using the OSCR API and any SCA radio which has been loaded with the OSCR Radio Interface to be combined into a cognitive radio. This framework will allow researchers to easily exchange cognitive control engines for evaluation and testing on different software radio platforms.

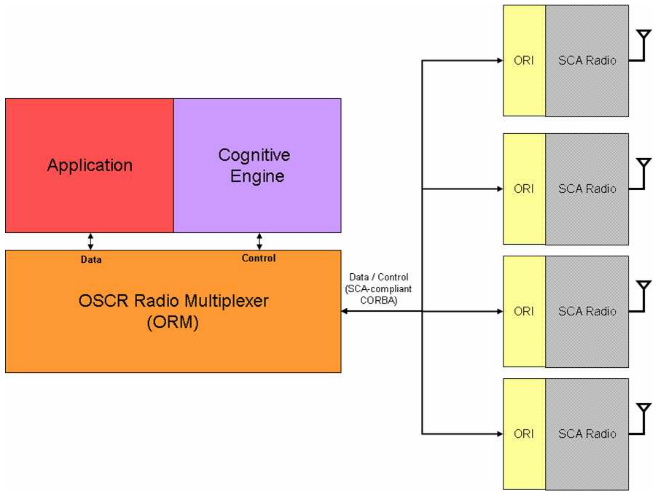

Fig. 1. An OSCR system consisting of multiple SCA radios.

To demonstrate the usefulness of OSCR, we have created a simple SCA-compliant software radio using the OSSIE [1] SCA implementation. This radio was then combined with a cognitive control engine, designed using SOAR [2], whose goal was to maximize the capacity of a noisy channel given a constantly changing noise environment. The cognitive engine was able to modify the modulation type, signal constellation size, and coding parameters of the transmitting radio in order to achieve this goal.

The remainder of this paper is organized as follows. Section 2 describes each component of the OSCR framework and how it is utilized within an OSCR-based cognitive radio. Section 3 describes the utilization of OSCR for the channel capacity maximizing cognitive radio. The paper is concluded in Section 4.

2. THE OSCR FRAMEWORK

OSCR allows a user to design a cognitive engine once, using the software of their choice and the OSCR radio control API (implemented as a C++ library), and then have it control any SCA-compliant radio which has been OSCR enabled. The framework consists of two software components – the OSCR Radio Interface (ORI) and the OSCR Radio Multiplexer (ORM). The architecture of an OSCR-based cognitive radio

[*]This work was completed while E. Stuntebeck, T. O'Shea, and J. Hecker were with the Laboratory for Telecommunications Sciences, US Department of Defense, which funded this research. The opinions expressed in this document represent those of the authors, and should not be considered an official opinion or endorsement by the Department of Defense or US Federal Government.

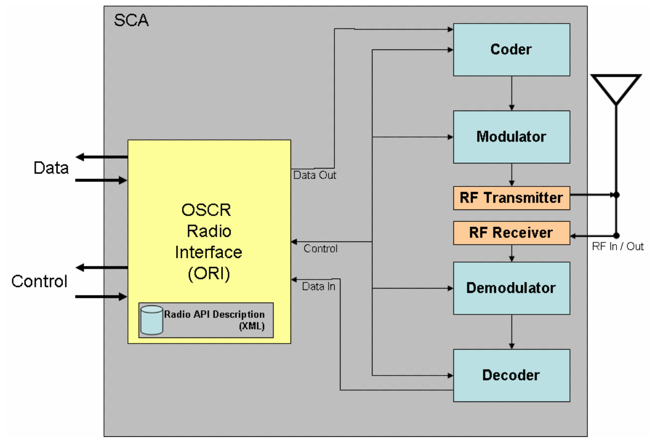

Fig. 2. Internal architecture of an OSCR compatible SCA-compliant software radio.

```
<component name="Modulator"
           interface="OSCR.ModulatorControl">
  <optionset name="modulation_scheme"
             function="set_modulation_scheme"
             type="int">
    <option name="QAM" value="1">
      <optionset name="constellation_size"
                 function="set_constellation_size"
                 type="int">
        <option name="QAM4"   value="4"/>
        <option name="QAM16"  value="16"/>
        <option name="QAM64"  value="64"/>
        <option name="QAM256" value="256"/>
      </optionset>
    </option>
  </optionset>
</component>
```

Fig. 3. XML description of settable properties in the modulator of an SCA-based radio. A description of each radio's API in this format resides within the ORI and is passed between the ORI and the ORM.

system is illustrated in Fig. 1. In this section, we describe the two components of the framework in detail.

2.1 OSCR Radio Interface

The OSCR Radio Interface is the interface between a generic SCA-compliant radio and the OSCR framework. It serves two functions. First, it acts as a translator between the OSCR API and the radio's native control API. The skeleton ORI provided with OSCR must be custom modified to match the control API of each SCA radio that will be used within an OSCR system. Second, it acts as the communication gateway between the radio and the OSCR framework via SCA-compliant CORBA. In order to be compatible with SCA-compliant radios, the ORI is implemented as an SCA resource.

An illustration of the internal architecture of an SCA-compliant radio to be utilized in an OSCR system is provided in Fig. 2, which represents the architecture of the radio we designed for the experiments described in Section 3. The resource coexists with the other SCA resources, which together comprise the SCA radio. The two functions of the ORI are now described in detail.

2.1.1 Interface Translator

Since the SCA does not dictate a common API for the software control of an SCA-compliant radio, it is necessary to provide a component in the OSCR framework which maps the radio's control API to the common OSCR control API. The ORI, which resides within the SCA domain of each OSCR radio, is responsible for receiving radio control information from the ORM and relaying control messages to the appropriate radio resource (see Fig. 2). In order to do this, the ORI must be modified for compatibility with the potentially different control API of each radio controlled by the ORM. For example, two different radios, although both SCA-compliant, may have different APIs for varying transmit power, modulation scheme, coding rate, etc. The ORI accepts as input control commands in the OSCR API format, translates these into the radio's native control API, and then carries out the requested actions in the native control API.

The control API between the ORI and ORM consists of only two methods: `setParameter` and `getStatistic`. Using only these two calls, the ORM can control any radio feature or fetch any radio operational statistic (e.g. SNR) through the ORI. Since an OSCR cognitive radio is capable of integrating multiple radios under a single cognitive engine, it is possible to have radios with different capabilities controlled by the same cognitive engine.

The cognitive engine must be aware of the capabilities supported by each radio. This is accomplished through the use of a description of the radio's capabilities in an XML format which the ORI on each radio stores. A portion of a radio description that describes the modulator component of a radio is provided in Fig. 3. The information, which is transmitted to the ORM upon request, describes all parameters of the radio which can be set by the cognitive engine, as well as all operational statistics which can be fetched. Upon receipt of this description of the radio, the ORM translates it into a hierarchical structure of objects for use by the cognitive engine in controlling the radio.

As an example of the flexibility given by the use of XML to describe each SCA radio, consider the portion of a radio description given in Fig. 3. The XML in this figure describes parameters which may be varied for the modulator of a particular radio. The `optionset` tag describes one parameter which may be varied – in this case, the modulation scheme, and within the modulation scheme, the constellation size. The `option` tags within the `optionset` describe the discrete values which this parameter supports – here, a modulation scheme of QAM and QAM constellations of size 4, 16, 64, and 256. This XML carries information for use by both the ORM and

]]>

cognitive engine, as well as the ORI. Notice that the `optionset` tag requires a `function` argument – this is the function in the radio's native API which is used to set the given parameter. The `value` argument associated with each discrete `option` is the value which must be passed to the function within the radio's native API in order to select that particular option. The `name` field carries a description of each option and is intended to be utilized by the cognitive engine. Thus two different radios may both support QAM-64 modulation and although they have different APIs to select this option in the modulator, the cognitive engine is able to select "QAM64" for both without needing to worry about the specifics of controlling each radio.

2.1.2 Communication Gateway

In addition to its function as a translator from the radio's native control API to the OSCR control API, the ORI is responsible for communication of both control and data with the ORM. Communication between an ORI and the ORM is accomplished using CORBA. Since the SCA dictates the use of CORBA for communication internal to an SCA-compliant radio, it was a natural choice for extending the radio's communication externally to the ORM. Both control information and data to be sent by or received from the radio are transmitted between the ORI and the ORM using CORBA. A substantial benefit of using CORBA for communication is that it allows OSCR systems to be highly distributed. Each radio in an OSCR system may run on a different hardware platform and can easily be connected to the ORM through a network. Our demonstration of the OSCR system described in Section 3 utilized one Linux-based machine to act as a transmitting software radio, one to act as a receiving radio, and one to act as the ORM and cognitive engine.

2.2 OSCR Radio Multiplexer

The OSCR Radio Multiplexer serves as the external interface of the OSCR framework. It allows a single cognitive engine to control an unlimited number of software radios. The set of software radios which the ORM controls is configured through an XML file. Fig. 4 illustrates the configuration of one radio within this file. In order to communicate with a remote SCA radio, the ORM must have information on the CORBA naming service the radio utilizes and the CORBA domain name within which the radio resides. Additionally, it must know the CORBA name of the SCA DomainManager, the SCA application name for the radio, and the CORBA name of the device on which the radio operates. Finally, the ORM must be aware of the specific CORBA name given to the ORI within the radio. This information allows the ORM to contact the radio over CORBA and communicate with the ORI.

```
<sdr>
  <name>OSCR Controlled SDR</name>
  <description>SDR #1</description>

  <corba>
    <nameServiceLoc>corbaloc::localhost/NameService
                                    </nameServiceLoc>
    <domain>DomainName1</domain>
  </corba>

  <sca>
    <domainManager>DomainManager</domainManager>
    <application>OSCR</application>
    <device>GPP1</device>
  </sca>

  <oscr>
    <interface>RadioInterface1</interface>
  </oscr>
</sdr>
```

Fig. 4. XML description of an SCA radio to be controlled by the ORM.

The ORM facilitates the cognitive engine's control of multiple software radios by providing a common control API. It also provides an interface for the application-level transmission of data through the cognitive radio. These two functions are now described in detail.

2.2.1 Cognitive Engine Interface

A cognitive engine designed around the OSCR framework speaks directly to the ORM when generating control messages for radios. Communication between the cognitive engine and the ORM is accomplished through C++ function calls, and thus the OSCR framework should be linked with the cognitive engine at compile-time.

The ORM must provide the cognitive engine with information about the capabilities of each radio under its control. In order to do this, it connects with the ORI within each radio and requests a copy of the radio's XML description (see Fig. 3). This description is then parsed and turned into a hierarchical structure of objects which the cognitive engine can use to control each radio.

The cognitive engine is permitted to access a distinct `sdr` object within the ORM for each software radio under the multiplexer's control. Within this object, control functions are divided into the radio's two functional units – the modulator and the coder. Each of these functional units contains parameters which may be varied by the cognitive engine. These parameters are represented by `ResourceVariable` objects. Each of these objects contains a set of discrete values which the parameter can be set to, represented as an `Option` object. For example, in Fig. 3. we saw that the "modulation scheme" parameter had one option – "QAM". When choosing certain options for a given parameter, it may be necessary to choose additional options. For example, choosing QAM for a "modulation scheme" parameter may require setting the constellation size. In this case, the `Option` for QAM would contain a

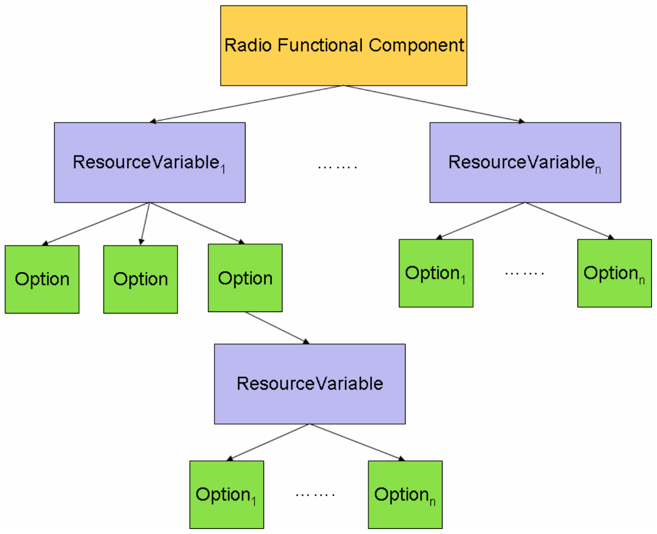

Fig. 5. Control object hierarchy of an OSCR software radio.

sub `ResourceVariable` object which must also be set by the cognitive engine.

2.2.2 Application-level Data Interface

In addition to its function of providing the external OSCR control API to the cognitive engine, the ORM also serves as a means of transmitting and receiving application-level data through the OSCR cognitive radio. To do this, the ORM creates a network socket on a user specified port number. Any standard socket-based application wishing to transmit or receive data through the cognitive radio may then connect. The ORM then monitors the currently selected software radio for incoming data and forwards any to the socket for receipt by interested applications. It also monitors the socket for incoming data from connected applications, which it then sends to the currently selected software radio using the OSCR API.

OSCR is designed such that application-level data flows are limited at any given time to one of the multiple radios under the control of the ORM. The radio through which data flows are routed can be selected by the cognitive engine at any time. This allows the cognitive engine to utilize a set of radios with differing properties and capabilities and to always route data through the best radio given its goal and current conditions.

3. SAMPLE OSCR APPLICATION

To demonstrate the usefulness of the OSCR framework, we developed an application utilizing the OSCR API, which demonstrates communication between an SCA software radio and a cognitive engine. In this example, we applied the cognitive engine to the problem of channel capacity maximization in the presence of noise. We utilize one ORM for the transmitter and one for the receiver. The cognitive engine is allowed to communicate with both ORMs through a wired communication channel, whereby it switches the modulation scheme of the transmitter and informs the receiver of the switch as necessary. Although this setup is not ideal for a real-world application (where in-band control signaling would likely be optimal), it nonetheless demonstrates the functionality of the OSCR API. This section describes the implementation of the cognitive engine, the SCA-compliant software radio, and the results obtained in the goal of channel capacity maximization in the presence of varying noise.

3.1 Cognitive Engine Implementation

The cognitive engine was implemented using the SOAR [1] cognitive architecture based on the OPS5 production system. Two scenarios were designed to demonstrate the versatility of the OSCR API. The first determines a solution based on the assumption that the radios are being operated in an AWGN channel. In this case, the settings that maximize channel capacity can be directly calculated from noise statistics at the receiving radio using the Shannon-Hartley law [3]. The second scenario examines communication in a non-AWGN channel by formulating maximization of channel capacity as a unimodal hill-climb [4]. In this case, the cognitive engine must communicate with the receiving radio in order to obtain error statistics. The API is designed in such a way that the cognitive engine need only read and write communication parameters and statistics that are relevant to determining a solution. Implementing the API in this manner allows the cognitive engine to be viewed as a dynamic component in the radio system that can be exchanged as necessary with cognitive engines implementing alternative control algorithms (e.g. a genetic algorithm) [5]. The OSCR API allows both methods of cognitive control to follow the same protocol for interfacing with SDR components.

At initialization of the cognitive engine, each radio is polled for the components to which it has access. In this simulation, each radio is equipped with a modulator and a coder. Each component is then queried for a listing of options and sub-options which may be set for that particular component. Every parameter that is polled is stored in the knowledge base of the radio agent. Each modulator contains a listing of modulation types, QAM and PSK in this example. Each modulation type contains constellation sizes of which that modulator is capable as options. Each coder contains a set of error-correcting convolutional codes. The codes are identified by their rate and free distance.

Once the radio agent's knowledge base is populated, the cognitive engine determines the best settings with which to achieve its goal of channel capacity maximization. The cognitive engine polls either the sending or receiving multiplexer for the statistic needed in the calculation.

3.1.1 AWGN Channel Capacity Maximization

In the AWGN channel scenario, signal to noise ratio is returned and used to calculate the theoretical maximum capacity by:

$$C = W \log\left(1 + \tfrac{S}{N}\right)$$

where W is the bandwidth of the available channel and S/N is the signal to noise ratio. For the given SNR, the achievable channel capacity C' is then calculated for every combination of possible settings. Assuming a block length n, channel-bit error probability, p_c, capable of correcting t errors at a coding rate r, is approximately equal to:

$$C' = \frac{kr}{T_s}\left(1 - \tfrac{1}{n}\sum_{j=t+1}^{n} j\binom{n}{j} p_c^j (1-p_c)^{n-j}\right)$$

where T_S is the symbol time and k is the number of bits per symbol and is equal to $\log_2(M)$. The variable p_c is approximated as P_E / k, where P_E is the probability of symbol error for a given modulation type expressed as:

$$P_E^{QAM}(M) = 2\left(1 - \frac{1}{\sqrt{M}}\right) Q\left[\sqrt{\frac{3E_s}{(M-1)N_0}}\right]$$

$$P_E^{PSK}(M) = 2Q\left[\sqrt{\frac{2E_s}{N_o}} \sin\left(\tfrac{\pi}{M}\right)\right]$$

where E_S/N_0 is the ratio of average symbol energy to noise power density spectrum. This value can be expressed in terms of known system quantities:

$$\frac{E_s}{N_0} = \frac{\log_2(M) rSWT_s}{NC}$$

The setting that achieves the greatest theoretical corrected throughput without exceeding the maximum channel capacity is selected. The cognitive engine then outputs the coding and modulation parameters to the radio through the OSCR API.

3.1.2 Non-AWGN Channel Capacity Maximization

In the case of a non-AWGN channel, direct calculation of the parameters which calculate channel capacity is no longer possible. To do this, we utilize a hill-climbing algorithm in our cognitive engine. In this scenario, the solution space of the communication settings has been formulated to be convex in the case of channel capacity maximization. A similar execution cycle occurs: initialization, statistic acquisition, parameter output. In this scenario, the receiving radio is polled for the bit error rate. The channel capacity of the current settings is calculated by:

$$C' = \frac{kr}{T_s}(1 - B)$$

The differences between the two formulations of the problem completely lie within the cognitive engine implementation. All API function calls are implemented in the same manner as they were in the last example, with the only difference being the desired statistic. The advantage of the second problem formulation is an increase in the functional intelligence of the system, underscoring the seamless connectivity between an arbitrary cognitive engine and the SDR components.

3.2 SCA Software Radio Implementation

Our SW Radio implementation takes in a binary data stream adds forward error correction and provides samples of M-QAM modulated intermediate frequency (IF) output. In a complete system this IF would then be up-converted to a higher sample rate, passed through a DAC, amplified, and fed to an antenna. For our purpose of demonstrating the usefulness of the OSCR framework, however, we have implemented only forward error correction and modulation and we simulate AWGN channel effects directly on the IF sample stream. Our SW Radio is implemented in a way both to comply with SCA specifications and to build off of the implementation of this as interpreted by the OSSIE project. This results in a very modular design in which each component of the radio may be developed independently and linked together using SCA Ports, which make use of CORBA Interfaces.

Each individual radio component which is involved in the actual data-path contains three significant SCA Ports. These ports handle the data-in stream, the data-out stream, and a control interface. These components are then tied together by the Radio Interface component to form a single Software Radio entity.

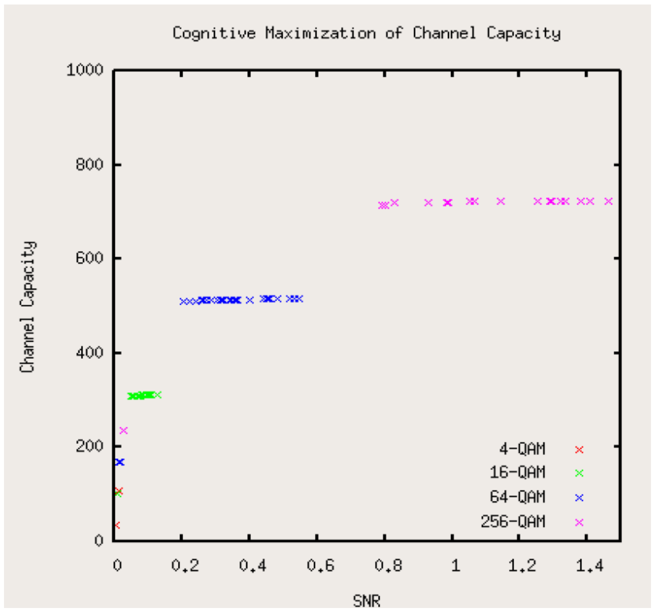

Fig. 6. Illustration of the cognitive engine's decisions to change modulation types at various SNRs based on its goal of maximizing channel capacity.

The component which adds forward error correction to the stream provides several different convolutional codes with different rates to allow for adjustment for higher capacity or more robust communication as needed. We provide a one-sixth rate code with constraint length 15, two one-half rate codes with constraint lengths of 7 and 9, and a rate-one null coder. The coding scheme may be updated at any time during operation by first pausing the data stream and then updating the coding mode on coder and corresponding decoder components. This is done through a CORBA interface specific to each component which is part of an SCA port.

Our demodulation block works by first scaling the constellation size by a max power value, symbols are then modulated by I and Q carrier signals so that we can average out I and Q values for the symbol and fit them to the nearest points on the constellation. During this process, we collect the symbol error ratio which is used to calculate the SNR statistic.

OSSIE provides all of the mechanism for loading and connecting these components to form a single software radio or "OSSIE waveform". We simply define a list of SCA port connections, define the port translation in the Radio Interface, and load the waveform. It can then be used by the SDR Multiplexer and any of the available cognitive control engines or otherwise.

3.3 Experimental Results

We were able to successfully run both of our cognitive engine approaches through this design. The architecture

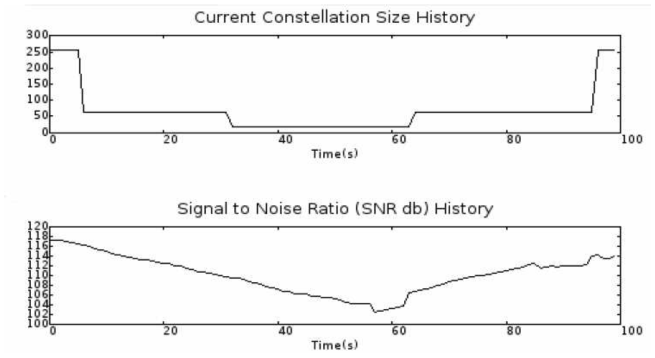

Fig. 7. The lower portion illustrates SNR values during a test. The upper portion shows the cognitive engine's chosen constellation size at that SNR.

allowed us to spend almost all of out time developing the cognitive engine simply implementing logic, and not management overhead. This architecture was also able to easily adapt to either of our cognitive engine objectives providing a lot of flexibility for these and future objectives. The figure below shows a trace of the different capacities provided by different coding and modulation combinations for different SNR levels.

We were also able to observe the behavior of the SNR based cognitive engine operating over time and see that it provided timely updates as the noise level was varied. This can be seen in the results in Fig. 7.

4. CONCLUSION

We have designed and implemented OSCR, an open source cognitive radio framework. OSCR provides an SCA-based software radio with a control API and allows users to easily integrate a cognitive engine, designed using the software of their choice, with one or more SCA-based radios which include the OSCR interface. We successfully demonstrated the usefulness of the framework by developing two SCA radios and using them to communicate over a noisy channel while under the control of a cognitive engine. It is our hope that this framework will be useful to and expanded upon by the cognitive radio research community.

5. REFERENCES

[1] Open-Source SCA Implementation::Embedded, *URL: http://ossie.mprg.org.*
[2] SOAR, *URL: http://sitemaker.umich.edu/soar/.*
[3] T. Cover, J. Thomas. *Elements of Information Theory*. John Wiley and Sons, 1991.
[4] S. Russell, P. Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2002.
[5] T. Rondeau, B. Le, C. Rieser, C. Bostian. "Cognitive Radios with Genetic Algorithms: Intelligent Control of Software Defined Radios," *SDR'04*.