INTERFACING A REASONER WITH AN SDR USING A THIN, GENERIC API: A GNU RADIO EXAMPLE*

Jakub Moskal (Northeastern University, Boston, MA, USA; jmoskal@ece.neu.edu);
Mieczyslaw M. Kokar (Northeastern University, Boston, MA, USA;
mkokar@ece.neu.edu); Shujun Li (Northeastern University, Boston, MA, USA;
shli@ece.neu.edu)

ABSTRACT

We present a Cognitive Radio Framework (CRF), an extension of the Ontology-Based Radio, which makes use of the LiveKB [5] component to interface an inference engine with a SDR. LiveKB provides a generic, SDR-independent interface for accessing radio's knobs and meters, which enables the CRF to support knowledge reusability across different SDRs. We explain how the SDR experts can benefit from this design; we describe the challenges encountered while implementing the prototype, and demonstrate how we integrated the CRF with the GNU Radio toolkit.

1. INTRODUCTION

Developing the Cognitive Radio (CR) architecture on top of a SDR platform requires two features: 1) access to the radio's contextual information (meters) that comes from its self-awareness and from sensing its environment, and 2) the ability to alter the radio's operational behavior by modifying its parameters (knobs). Since different SDRs offer different knobs and meters, and different ways to access them, it is a challenge to design a CR architecture that would work with all or at least with most of the radio platforms.

A common solution to the problem of interfacing multiple software components is to introduce a public Application Programming Interface (API). This allows the developers to implement just one API and enable their software to interface with many components, developed independently. To ensure the quality of the API and to attract the majority of a community, APIs are standardized by organizations, rather than by single companies. Despite their obvious benefits, there are at least two drawbacks of using the standard APIs: 1) it takes a significant effort to change them, and 2) creating a new version that is not backwards-compatible results in losing the interface among the components that implement different versions of the standard. Currently, there is no standard API for the interface between a cognitive engine and SDR, though different groups in the community have published public APIs to serve this purpose. Among them is CR API [1] from the VTCROSS framework, the PAAL [2,3] layer from the Community-Based CR Architecture and Open Source Cognitive Radio (OSCR) Radio Interface (ORI) [4].

We proposed [5] the LiveKB architecture – a platform-independent extension of the Ontology-Based Radio (OBR) architecture, which utilizes ontologies and policies to enable the CR paradigm. Although the OBR was never fully implemented, it has been shown that it can be used to analyze the multipath structure [6], negotiate the length and structure of the equalizer training sequences [7], and dynamically extend network coverage and reachback [8]. Instead of relying on radio-specific APIs to interface the reasoner, the proposed architecture requires a thin, generic API for connecting with the inference engine that uses the software model of the radio platform to identify access paths to particular radio parameters (variables). This architecture is capable of maintaining the interface with the SDRs that change their APIs without the need of recoding the reasoner API due to the fact that the interface information is provided dynamically via the software model.

In this paper we present an improved design of the architecture, now called the Cognitive Radio Framework (CRF), with the focus on the reusability of expert knowledge. We describe the challenges encountered during its implementation and demonstrate how it can be used with a GNU Radio toolkit [9] and the Universal Software Radio Peripheral (USRP).

2. THE COGNITIVE RADIO FRAMEWORK (CRF)

The CRF architecture (Figure 1), although initially designed as an extension of the OBR [7], introduces numerous improvements to its predecessor. It consists of the following components: Monitor Service (MS), Reasoning Component (RC), Data In/Out (DIO) and LiveKB. The first two are inherited from the OBR: MS is responsible for passing

* Approved for Public Release, Distribution Unlimited

control messages between DIO and RC, and RC is used to process the expert knowledge. As a result of reasoning, new control messages can be generated and passed to DIO. DIO processes incoming messages and dispatches data to the application layer and control messages to MS. It also combines the outgoing data and control messages and broadcasts them using the SDR. All messages within CRF are passed asynchronously, using buffers. It is also noted that control messages need to be transferred between radios in a reliable fashion, thus DIO must implement a reliable transfer protocol, at least for the delivery of the control messages. Finally, LiveKB reads the SDR's meters and sets its knobs according to the reasoner's requests.

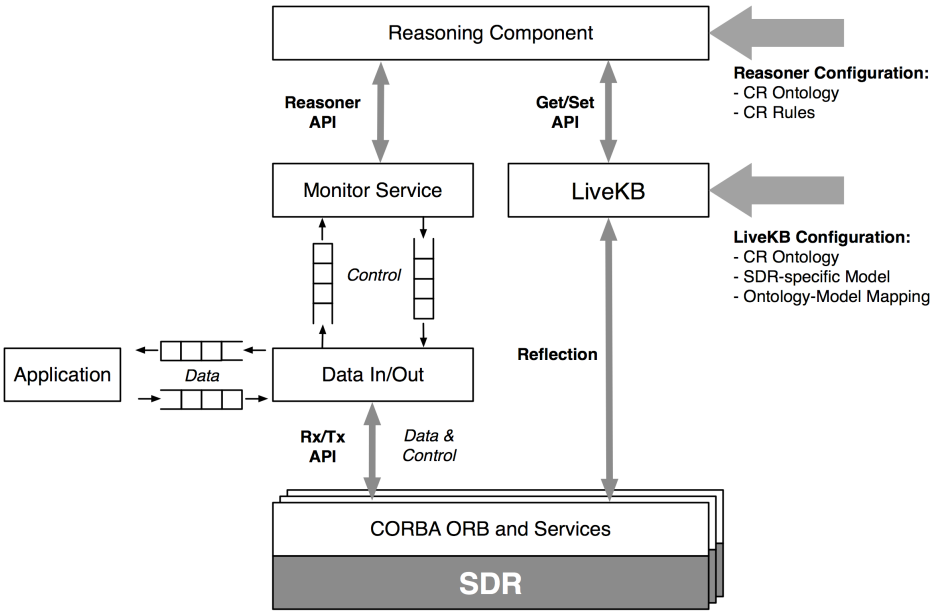

Figure 1. Cognitive Radio Framework (CRF) architecture

Since both the RC and LiveKB components are domain-independent, they are required to be configured with the domain-specific information. RC needs to be provided with the expert knowledge in the form of CR ontology and rules, and LiveKB requires the CR ontology, SDR's software model and a mapping between the two.

2.1. CRF Interfaces

The main objective in designing CRF was to support the reusability of knowledge, thus our goal was to rely on SDR-specific interfaces as little as possible. Three APIs are involved in the design of the CRF architecture: Rx/Tx, Reasoner and Get/Set.

2.1.1. Rx/Tx API

The only radio domain-specific interface used in CRF is the *Rx/Tx API* – the interface between the radio's physical layer and the DIO component. It is used to transmit and receive data to and from the SDR, respectively. Unfortunately, there is no standard way to access SDR and thus, similarly as the existing architectures, CRF utilizes a custom interface. SDR is required to act as a CORBA server providing clients means to transmit and receive data using the callback mechanism. Rx/Tx API is used only to transmit/receive data. Thus, currently, the API includes only the following methods (using the IDL syntax):

```
interface SDRListener {
    onRx(in string msg, in boolean corrupted);
};

interface SDR {
    tx(in string msg);
    void addRxListener(in SDRListener l);
    void removeRxListener(in SDRListener l);
};
```

Developing a universal CR architecture would require that the transceiver API was developed by the wireless community as a standard. There is an ongoing effort to develop such an API [10], however, it goes well beyond the needs of the CRF architecture. Nevertheless, Rx/Tx API could be replaced with the standard and still support the reusability. It is noted that the dependence on a particular transceiver interface does not affect the reusability of expert

knowledge, because the ontology and rules are associated with the radio's parameters via LiveKB's generic API, not via the Rx/Tx API.

2.1.2. Reasoner API

The Reasoner API allows MS to pass control messages to the reasoning component and initiate inference. There are some public APIs that are available to serve this purpose, among them is the Description Logics Implementation Group (DIG) interface [11], OWL API [12], Storage and Inference Layer (SAIL) [13], Jena API [14] and the currently being developed API4KB [15]. Existing APIs for accessing the reasoner functionality differ significantly: they support different query languages and data formats. Each API is designed with a different philosophy in mind and it requires a fair amount of effort to add support for a new API to an existing piece of software. The design of a new API or a choice of one of the existing reasoner interfaces was not the focus of this research and will be a part of the future work. As of now, the interface consists of merely two methods, one to configure the reasoner and one to start it.

2.1.3. Get/Set API

The LiveKB component provides a thin and generic *Get/Set API*, which allows the reasoner to access and adjust radio's parameters. This API, utilized from within the reasoner's procedural attachments, is entirely domain-independent and consists of merely two methods: `get(propertyName)` and `set(propertyName, newValue)`. The names of the properties passed to these methods correspond to the terms from the CR ontology, provided to the reasoner and to the LiveKB dynamically, at runtime.

LiveKB uses the provided ontology-model mapping to translate abstract requests from the reasoner to the SDR-specific method invocations. Since the model is provided to LiveKB dynamically, it has no prior knowledge about it and it must use *reflection* to invoke the SDR methods. Reflection is a mechanism that allows programs to see their own structure at runtime. The only requirement is that the SDR's knobs and meters are available through CORBA and registered with CORBA's Naming Service.

The use of the generic interface provided by the LiveKB component yields several benefits: 1) it does not require the domain expert who writes the rules to know any radio-specific knobs and meters API, 2) the list of available parameters is not fixed – new parameters can be accessed in the future without the need to recode the interface, and 3) the expert knowledge captured in abstract, ontological terms can be reused across different SDRs, because it is expressed in SDR-independent terms.

2.2. Roles and artifacts

Since both the reasoner and LiveKB are domain-independent, they need to be configured with radio-specific information. This task needs to be carried out by domain experts and by the SDR vendors. The CRF architecture provides both parties with a reasonable separation of

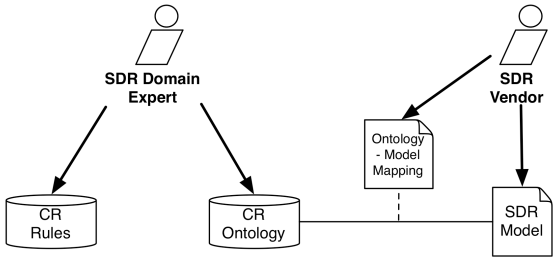

Figure 2. CRF roles and artifacts

concerns. Figure 2 shows both roles and artifacts that they are responsible for producing.

Domain experts express the knowledge related to the CR in two forms: CR Ontology and CR Rules. On the other hand, in order to make their products compatible with CRF, SDR vendors provide the SDR software model and a mapping between the model and the CR ontology.

The artifact that binds the efforts from both roles is the mapping between the software model and the ontology. To maintain the interface between LiveKB and the SDR, it needs to be updated upon changes made to the ontology or to the model. For this reason, we recommend standardizing the CR ontology in order to decrease the amount of necessary maintenance of the mapping. Such a standard ontology would give the vendors the freedom to develop their radios according to their needs, yet still making it possible to be compatible with the CRF framework. There is an ongoing effort [16, 17] to develop such ontology. It is noted, that updating the mapping does not require any recoding of the interface because the mapping is provided to CRF at runtime.

2.3. Behavioral description

The operation of CRF can be broken down into two stages: initialization and runtime. To better understand what happens at each stage, we give a brief behavioral description for each of them.

2.3.1. Initialization stage

During the first stage, LiveKB is provided with the CR ontology, the SDR's software model, and a mapping from the model to the ontology. Next, the reasoning component is supplied with the CR ontology and rules.

The initialization of both components can be done offline – using an external configuration file, located in the local file system. Alternatively, it could be performed online

– by executing specific methods. However, the online configuration would require implementing a dedicated API, which prevents legacy software from being used without modification. It is likely, that the CRF implementation will permit both styles of initialization.

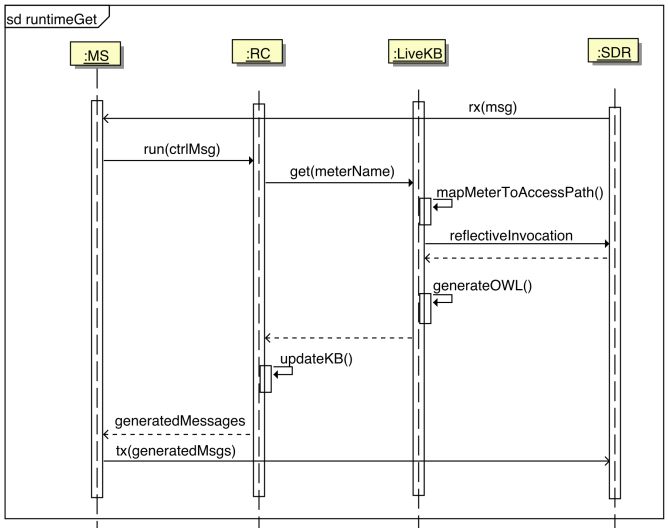

Figure 3. CRF during the runtime stage

2.3.2. Runtime stage

Once LiveKB and the reasoner are initialized, Data In/Out can start exchanging messages with other radios. When a control message is received, it is passed to MS, which then runs the reasoner. Depending on the current state of the radio's parameters, some rules might fire one of the procedural attachments provided by LiveKB – `get` or `set`. When this happens, LiveKB uses the provided mapping and translates the name of the parameter to the access path, specific to the underlying SDR. Using reflection, LiveKB executes methods on the access path and updates or retrieves a value of the desired parameter. Finally, LiveKB must also update the ontology to reflect the changes made to the radio's parameters.

We illustrate the CRF runtime stage using a UML sequence diagram (Figure 3). This diagram corresponds to the use case when the radio receives a control message and passes it to the MS component. MS runs the reasoner, which in turn triggers a rule, which utilizes LiveKB's `get` procedural attachment. As a result of reasoning, new control messages are generated and passed to the MS for broadcast.

3. CURRENT STATE OF THE IMPLEMENTATION

At the present time a limited functional prototype of CRF has been implemented. We were able to use the same ontology and rules to interface a mockup software radio component implemented in two different languages – Java and Python. In addition, we successfully integrated CRF with a GNU-Radio based SDR (described below) and were able to read and change values of some parameters while the radio was in operation.

CRF requires that the SDR objects are available via the CORBA middleware, thus CRF itself could be implemented in any language that supports this technology. In our prototype all CRF components are implemented in Java and we chose BaseVISor [18] for the implementation of the reasoning component. Since BaseVISor is written in Java, delegating procedural attachments to the LiveKB component was straightforward. The asynchronous message passing between MS, DIO and the application layer was implemented using the Java Message Service (JMS). JMS is a message-oriented middleware, which implements the publish/subscribe paradigm, suitable for our needs.

We have not yet implemented the mechanism for the mapping between the model and the ontology. Because of that limitation we still assumed a tree structure of the model and the names of CORBA attributes were required to be unique in the global scope and correspond directly to the names of properties in the OWL ontology.

Before the implementation can be completed, some decisions still need to be made at the design level. Most importantly we need to settle on the choice of APIs for controlling the reasoner and for the SDR platform. It would be best if the latter were a standard developed by the wireless community, e.g., the Wireless Innovation Forum.

3.1. Prototype example

We illustrate how a prototype mapping between a model and an ontology was implemented using a trivial example. Figure 4 shows a sample CR ontology, which includes two OWL classes: `SimpleSDR` and `Channel`, one OWL object property `hasChannel`, two OWL datatype properties `hasId` and `hasMaxThroughput`, and one

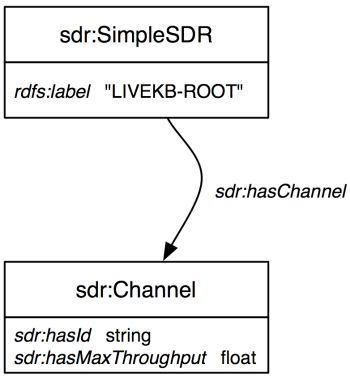

Figure 4. Sample CR ontology used in the prototype implementation

annotation `rdfs:label`, which denotes the root object of the SDR. More classes could be easily added, as long as there is only one that is annotated with the "LIVEKB-ROOT" label and the classes form a tree structure without cycles.

Because the mapping between the ontology and the model is assumed to be in 1-1 relation, the corresponding IDL model must look as follows:

```
interface SimpleSDR {
   attribute Channel hasChannel;
};

interface Channel {
   attribute string hasId;
   attribute float hasMaxThroughput;
};
```

In a 1-1 mapping, all OWL classes must have corresponding interfaces in the IDL, and all OWL properties must correspond to IDL attributes. Using the prototype, any software that implements the IDL definitions could be monitored and controlled from within the rules that make use of the ontological terms and pass them to the `get` and `set` procedural attachments implemented by LiveKB.

4. CRF WITH GNU RADIO AND USRP

CRF has been successfully tested with the GNU Radio toolkit and the USRP hardware. GNU Radio is a popular open source software development toolkit that provides signal processing runtime and processing blocks to implement software radios using low-cost RF hardware and commodity processors [10]. The toolkit offers a Python interface to implement waveform applications, although the performance-critical signal processing part is implemented in C++. It is a suitable platform for testing the CRF framework for two reasons: 1) it allows for rapid application development due to the use of a dynamic language, and 2) it is available on a range of operating systems.

USRP belongs to a family of products that allows for creating a high bandwidth SDR with a use of a general-purpose computer. GNU Radio is its natural companion because the toolkit was originally designed for that hardware, although USRP can be utilized without it.

Since numerous CR use cases [6,7,8] require a reliable exchange of control messages and since the GNU platform we used does not support this feature, our DIO component implemented the *Selective Repeat (SR)* protocol [19], used for the reliable data transfer. The DIO component sends and receives packets from the physical layer of the GNU radio using the CORBA middleware; it performs retransmission when the PHY packets are not acknowledged and buffers packets that are acknowledged out of order. Once acknowledged in order, packets are delivered to the appropriate JMS buffers. Currently, SR is implemented to work only between two nodes; in the future we will extend this protocol to allow for message exchanges between numerous radios.

In our experiments we modified the *tunnel.py* transceiver flow graph provided with the GNU Radio toolkit, to act as a CORBA server. Clients, written in any language that supports CORBA, can receive and transmit packets directly from the PHY layer. A second CORBA object, representing the model of the SDR, is registered with the CORBA's Naming Service and allows the LiveKB component to get and set values of some parameters while the flow graph is running. At the configuration stage, both the CR ontology and the SDR's IDL model are passed to LiveKB.

This setup allows the application layer to transmit data between radios, but also enables the modification of their operational parameters from within the rules. For instance, the following rule, written using the BaseVISor syntax, was successfully fired and resulted in modifying the value of the radio's carrier sense threshold in runtime:

```
<rule name="AdjustCST">
  <body>
    <Individual rdf:type="rad:SDRModel">
      <rad:hasCST variable="currentCST"/>
    </Individual>
    <equals>
       <currentCST/>
       <param>30</param>
    </equals>
</body>
  <head>
    <set>
      <param>rad:hasCST</param>
      <param>35</param>
    </set>
```

```
  </head>
</rule>
```

Although the rule merely increments the value of the threshold by an arbitrary value, it could be easily extended to include a condition, e.g. decrease the threshold given that the average time to send one packet has reached some value. This rule, however, demonstrates that there is no SDR-specific knowledge necessary to write the rule; all terms come from the ontology. Thus, domain experts may develop rules focusing on the abstract terms, regardless of the underlying SDR platform.

5. DISCUSSION AND FUTURE WORK

The CRF architecture supports the reuse of ontologies and rules written by domain experts. It enables the use of rules written by domain experts, rather than programmers and allows the rules to capture the generic domain knowledge, rather than the knowledge of a given radio implementation. We advocate the standardization of the CR ontology, rather than developing standard APIs. This has several advantages: 1) ontology is not a part of the hard-coded design; it is provided dynamically, thus changes to it do not require recoding of the architecture; 2) ontology is an abstract representation of the knowledge about the entire domain, not about a particular technology or a platform, thus once standardized, changes to it should occur less frequently; 3) ontologies can be created by domain experts without the knowledge of a particular software or hardware, thus it provides a better separation of concerns.

As part of the future work we will extend the implementation to allow arbitrary mapping between the model and the ontology. Moreover, we will settle on the choice of the API for the SDR platform and the reasoning component. The next important step will be to integrate CRF with the SCA-based SDRs and execute CR scenarios on heterogeneous radios using the same set of ontologies and rules.

6. ACKNOWLEDGMENTS

This work has been partially supported by an ONR contract No. N00014-05-C-0367 through VIStology, Inc. and a DARPA contract through System Planning Corporation (Agreement No. SRA-0775). The views, opinions, and findings contained in this article are those of the author and should not be interpreted as representing the official views or policies, either expressed or implied, of the Defense Advanced Research Projects Agency or the Department of Defense.

7. REFERENCES

[1] B. Hilburn, W. Rodgers, T. R. Newman, and T. Bose, “CROSS - a distributed and modular Cognitive Radio framework”, *SDR Technical Conference*, Dec 2009

[2] A. Ginsberg, J.D. Poston and W.D. Horne, “Experiments in Cognitive Radio and Dynamic Spectrum Access using An Ontology-Rule Hybrid Architecture”, *Second International RuleML-2006 Conference*, 2006

[3] A. Ginsberg, W.D. Horne and J.D. Poston, “Community-Based Cognitive Radio Architecture: Policy-Compliant Innovation via the Semantic Web”, *New Frontiers in Dynamic Spectrum Access Networks, DySPAN 2007,* pp.191—201, 17—20, Apr 2007

[4] E. Stuntebeck, T. OShea, J. Hecker, and T.C. Clancy, “Architecture for an open-source cognitive radio”, *SDR Technical Conference*, Nov 2006

[5] J. Moskal, and M.M. Kokar, “Interfacing a reasoner with an SDR: A platform and domain API independent approach”, *SDR Technical Conference,* Dec 2009

[6] J. Wang, D. Brady, K. Baclawski, M. Kokar, and L. Lechowicz, “The use of ontologies for the self-awareness of the communication nodes”, *SDR Technical Conference*, Nov 2003

[7] J. Wang, M.M. Kokar, K. Baclawski, and D. Brady, “Achieving self-awareness of SDR nodes through ontology-based reasoning and reflection”, *SDR Technical Conference,* Nov 2004

[8] S. Li, M.M. Kokar, and J. Moskal, “Policy-driven Ontology-Based Radio: A Public Safety Use Case”, *SDR Forum Technical Conference*, Dec 2008

[9] E. Blossom, “GNU Radio: Tools for exploring the radio frequency spectrum”, *Linux Journal*, no. 122, June 1, 2004

[10] E. Nicollet, and L. Pucker, “Standardizing Transceiver APIs for Software Defined and Cognitive Radio”, *RF Design Magazine*, pp.16-20, Feb 2008

[11] S. Bechhofer, R. Moller, and P. Crowther, “The DIG description logic interface”, *International Workshop on Description Logics*, 2003

[12] S.Bechhofer, R.Volz, and P.Lord, “Cooking the semantic web with the OWL API”, pp.659–675, 2003

[13] J. Broekstra, A. Kampman, and F. van Harmelen, "Sesame: A Generic Architecture for Storing and Querying RDF and RDF Schema", *International Semantic Web Conference,* Jun 2002

[14] J.J. Carroll, I. Dickinson, C. Dollin, D. Reynolds, A. Seaborne, and K. Wilkinson, “Jena: implementing the semantic web recommendations”, *Proceedings of the 13th international World Wide Web conference,* May 2004

[15] Object Management Group, “API4KB Request for Proposal”, http://www.omg.org/cgi-bin/doc?ontology/10-03-01.pdf

[16] S. Li, M.M. Kokar, and D.Brady, “Developing an ontology for the cognitive radio: Issues and decisions”, *SDR Technical Conference*, Dec 2009

[17] WIF Forum MLM Working Group, “Description of Cognitive Radio Ontology v.1.0”, 2010

[18] C. Matheus, K. Baclawski, and M.M. Kokar, "BaseVISor: A Triples-Based Inference Engine Outfitted to Process RuleML and R-Entailment Rules", *In Proceedings of the 2nd International Conference on Rules and Rule Languages for the Semantic Web*, Athens, GA, Nov 2006

[19] J.E. Kurose, and K.W. Ross, *Computer Networking – A Top Down Approach*, Addison Wesley, Boston, 2008