CODIPHY - Composing On-Demand Intelligent Physical Layers

Aveek Dutta[1], Dola Saha[2], Dirk Grunwald[1,2], Douglas Sicker[2]
[1]Department of Electrical, Computer and Energy Engineering
[2]Department of Computer Science
University of Colorado
Boulder, CO 80309-0430 USA
{Aveek.Dutta, Dola.Saha, Dirk.Grunwald, Douglas.Sicker}@colorado.edu

ABSTRACT

In this paper we present CODIPHY or Composing On-Demand Intelligent Physical Layers that aims to solve two fundamental problems in practical cognitive radio networks: Collaboration between two radio physical layers (PHY) with varying capabilities to agree on a common communication protocol and provide a method to compose a functioning software defined radio (SDR) from a set of pre-compiled libraries. Both solutions use an ontology based description of the internal structure of the radio subsystems and use the high-level dataflow represented by the ontology to target heterogeneous platforms. CODIPHY isolates the various domains of radio engineering but still allows them to share domain knowledge to achieve a common goal of radio adaptation. Automating this process through declarative specification and collaborative learning is the goal of this paper. We present a generic methodology to facilitate the concept of CODIPHY and present examples from the radio PHY domain.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous; D.2.8 [**Software Engineering**]: Metrics—*complexity measures, performance measures*

General Terms

Theory

Keywords

Software Defined Radio, Architecture, Design Automation, Cognitive Radio, Physical Layer

1. INTRODUCTION

Modern Cognitive Radios (CR) are based on "tuning knobs" that can implement many different radio waveforms, but they are fundamentally limited and cannot radically change the physical layer. Crosslayer techniques in recent research [16, 5, 17] require adaptation of the PHY for the changing requirements of the protocol. If all the radio nodes in a network have the same radio architecture, hardware and processing power, any low level PHY adaptation can be achieved by using a common firmware upgrade, usually in the form of a hardware image (*FPGA bitfile*) or as a software executable.

However, modern radio systems have heterogeneous computing platforms often combining specialized processor and hybrid FPGAs [10, 15, 13, 21]. This plethora of platform choices point towards a more radical form ofdynamic software radio reconfiguration where a system not previously designed for a particular waveform can be modified for new waveforms. That reconfiguration can occur either when designing a particular waveform (*e.g.* by a researcher) or during operation. Therefore we need a method to specify what is needed, but not over-specify the solution to the point where the features or benefits of a specific platform can not be exploited. We believe that this requires a method for hierarchical description. For example, at a high level, a radio PHY design may want to specify a particular common correlator for packet detection. However, if a system involved in the cognitive network does not "know" that specific design, the design specification can be "lowered" or refined to a more concrete form.

We propose to use an ontology specification, a tool of the semantic web, to capture these hierarchical specifications. The goal is two fold: a structured way of representing a hierarchy of components, their properties and the relationships between them and secondly, to be able to extract information about the domain using queries, such that a radio agent can learn about other agents and collaborate if needed. The use of ontologies in cognitive radio has been studied by Kokar et al. [7, 11] and research [9] has shown that ontology can be used by multiple radios to decide on a particular access policy. However, the use of ontology to define the internal structure of a radio PHY hasn't been studied before.

For example, radio A can query radio B if it supports a particular modulation format. In the absence of such a modulation in Radio B, it can obtain the description for that particular modulator and compose the internal structure for the new modulation. Part of this problem can be solved by parameterizing the modulator. However, this approach is infeasible once the complexity of the system increases, such as in Orthogonal Frequency Division Multiplexing (OFDM) based protocols. Also, this leads to over-provisioning of resources as it is very hard to predict what new functions will be required in the future for harmonious co-existence of heterogeneous CRs. Using an ontology the radios can understand each other and compose newer subsystems using components from their own library or download from a server without involving the designer or the user. As the subsystems get complex and more application specific, this process can be elevated to a kernel level reconfiguration because at that point it gets too complex to formally express and reason on those complex internal structures. Therefore, to support

SRIF'13, August 12, 2013, Hong Kong, China.

modern protocols, we need to be able to *compose* the radio PHY, rather than build fixed function pipelines that are brittle when exposed to such novel crosslayer implementations. Hence the term CODIPHY - Composing On-Demand Intelligent Physical Layers.

Using suitable back-end compilation, the ontology and the dataflow it represents, is translated to the preferred implementation language required by the target platform, thus allowing specific optimization to be targeted during the implementation phase. Automating this process of expressing the PHY and code generation for onward implementation is the goal of CODIPHY. Now machines can process the radio dataflow and use a rich library of building blocks to compose the radio when it is required to modify the physical structure or the dataflow between various components.

2. HIERARCHICAL KNOWLEDGE REPRESENTATION

In this section we design a knowledge representation system for a radio PHY that is able to provide information about the specifications of the radio at various levels of granularity. Depending upon the specific requirement of a radio agent, the components can be chosen either as a high level aggregate with proper parameters or as a low level dataflow graph. The system will allow a non-expert to query the knowledge base and decide on which level of granularity to use for a particular implementation. The levels in the knowledge representation system are as follows:

- System level: At this level, we specify a very broad description of radio from a system perspective. For example, is it a CDMA or an OFDM radio? We also specify the various capabilities of the radio like bandwidth, sampling frequency, tuning range etc. These are required to initiate a collaborative adaptation of PHY using CODIPHY.

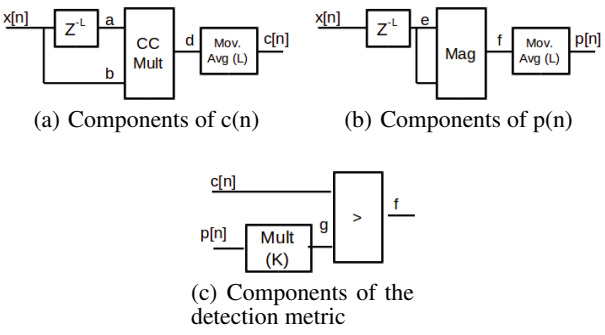
- Subsystem level: This is still at a high level but now delves deeper into a particular family of radios. For example, what are the subsystems of the radio? Does it have a specific type of correlator?

- Specification level: Specifies the parameters of a particular subsystem. What are the inputs and output variables and what are the programmable parameters of a subsystem? This information is typically derived from the mathematical equations that define that subsystem.

- Representation level: This level defines how the subsystem is represented to the outer world. It can be represented as a mathematical equation, a state machine, as a mapping function etc. Knowing the representation helps the radio agent to make incremental changes to an existing radio pipeline and build subsystems by minor modifications.

- Dataflow level: Dataflow level is the lowest level of expressing a subsystem. At this level, the entire subsystem is a graph of interconnected components that are provided as rich set of pre-compiled design files tied together to form the subsystem.

Each of these levels provide enough information to build a functioning radio. One of the goals of CODIPHY is to compose the radio pipeline from a set of pre-compiled components that require a component based design approach [12]. These components are tied together either by electrical wires in the hardware or form sequential statements to be executed on a processor architecture. The components range from basic arithmetic and logic units to aggregates are required for specialized task like Fourier transform, forward error correction and filtering.



(a) Components of c(n)

(b) Components of p(n)

(c) Components of the detection metric

Figure 1: Components derived from packet detection unit

In this paper, we focus on the dataflow level and present an ontology based description of an OFDM transceiver. We also provide the necessary compilation tools required to clone the subsystems between two heterogeneous radio platforms.

2.1 Dataflow Level

The dataflow level contains the knowledge of each component within a subsystem and all the connections between the components and their data types. As an example, we show how a packet detection unit can be decomposed into components and aggregates, and represent it in an ontology such that radio agents can understand each others internal structures and compose a new subsystem. Equations 1 and 2 compute the autocorrelation energy between input signal, $r(n)$ and its delayed version $r(n+L)$, and the signal energy during that autocorrelation window respectively, where $L = 16$ for 802.11a/g.

$$c(n) = \sum_{i=0}^{L-1} (r^*_{n+i}.r_{n+i+L}) \quad (1)$$

$$p(n) = \sum_{i=0}^{L-1} |r_{n+i+L}|^2 \quad (2)$$

If $c(n) > K*p(n)$ is satisfied, then the system detects an OFDM packet. The threshold K is a design parameter that is user-defined.

Analyzing eq.1 and eq.2 reveal distinct components that compute the metric $c(n)$ and $p(n)$. Figure 1(a) shows the components for $c(n)$. It suggests that the input signal $r(n)$ is multiplied with the complex conjugate of $r(n)$ that is delayed by L samples. The summation in eq.1 is indexed over the variable L and is independent of the input sample index n. Hence it is a 'sliding window average' component that computes a moving average of L values. The flow of the results are designated by variables, a, b and d. Similarly, figure 1(b) shows the components used to compute $p(n)$. The metric computation requires a comparator that compares the product of K and $p(n)$ with $c(n)$ to produce the binary result f, which is shown in figure 1(c).

It is to be noted that all the components have been identified at a high level and we do not focus on how these components are implemented. For example, the moving average unit can be implemented as a single stage comb filter or as a FIR filter with constant coefficients. From a functional point we assume that these imple-

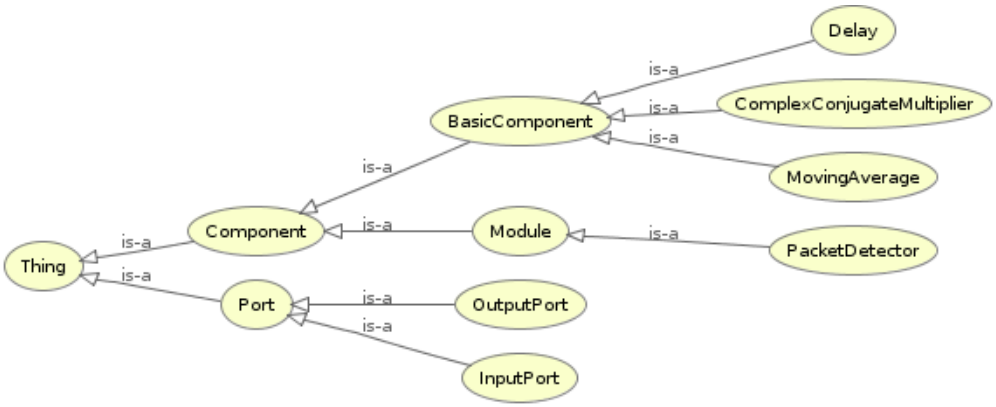


Figure 2: Classes in the Packet Detector ontology

mentation specific structures will be provided for composing the radio.

Using this example we show that any DSP system can be completely represented using a set of components and aggregates, which is a direct reflection of the variables and constants provided in the mathematical function. We also generate an implementation from this high level functional description and target different hardware platforms and automating this process is the ultimate goal of CODIPHY.

3. ONTOLOGY OF THE RADIO PHY

In this section, we describe the implementation of the ontology of the dataflow level of the knowledge base. We have used the OWL language [20] and the Protégé [3] open source tool to build the ontology. In the domain of CRs, *classes* describe the various components and aggregates required for processing radio samples like adders, multipliers, FFT etc. In building DSP subsystems, *instances* of these classes are used to form a hierarchy of components that are connected by *object properties*, which define the dataflow. It is also possible to specify certain qualitative attributes in the *data properties* like data type, precision, latency and resource consumption (for FPGA), that are useful in optimizing the implementation. Therefore, an ontology for CR PHY focuses on the dataflow between various instances of the defined classes and their taxonomic hierarchy (subclass-superclass).

3.1 Taxonomy of the Ontology

The ontology specifies a hierarchy of subsystems. A module may be a sub-module of a bigger module and also that module can have multiple instances within the radio with different parameters. For example, an FIR filter can be used multiple times in the radio pipeline but with different weights.

To make the ontology inter-operable between different radios, we define the taxonomy of classes used to represent a PHY. The top level classes in the taxonomy are *Component* and *Port*. Components have sub-classes 1) *BasicComponent*: superclass for all fundamental computation units, defined by the domain experts, required to implement various DSP functions, and 2) *Module*: represents the subsystems, which are a collection of instances of various basic components. Equations 1 and 2 represents the mathematical structure of the OFDM packet detector, and figure 1 represents the components and their interconnects of the same unit. Figure 2 shows the hierarchy of the different classes created for representing the partial Packet Detector specified by eq.1. The 'is-a' relationship denotes the hierarchy of the classes. In this example the *PacketDetector* is a Module that contains instances of the classes *Delay*, *ComplexConjugateMultiplier* and *MovingAverage*, which are in turn subclasses of *BasicComponent*. The class hierarchy in CODIPHY is constant for all ontologies that define a radio PHY. However, the specific naming of the instances is left to the designer of the ontology. This is also useful in a component based radio implementation, where the instances of the basic components are mapped to pre-defined design files or functions that can be stitched together to implement a radio. Therefore, the ontology specifies *what* is required to build a radio rather than specifying *how* to build it.

3.2 Specifying the Dataflow

The instances of the classes are related to each other using certain properties. These properties are part of the taxonomy and remain constant for all implementations of the ontology. The selection of properties and how they relate the instances will depend on how a reasoning engine would classify the ontology to answer queries from the user and other radio nodes. In CODIPHY, we deal with those queries only that will allow an external agent to understand what components are present in a module, *e.g.*, Packet Detector, and how they are connected to each other, so that if required the querying agent can re-create the packet detector without human intervention. Figure 3 shows a partial ontology of the OFDM packet detector and the relationship among various instances of the classes defined in the ontology.

The main properties that are used to relate the components are:

- *hasBlock* and *isABlockOf*: These two properties relate an instance of a module with instances of basic components that are contained within that module.
- *hasIndividual*: This property relates instances to their respective classes.
- *hasInputPort* and *isInputPortOf*: Relates the instances of components with instances of their input ports.
- *hasOutputPort* and *isOutputPortOf*: Relates the instances of components with instances of their output ports.
- *isConnectedTo*: Relates various instances of the Port class, *e.g.*, relates the output port to that of an input of another component.

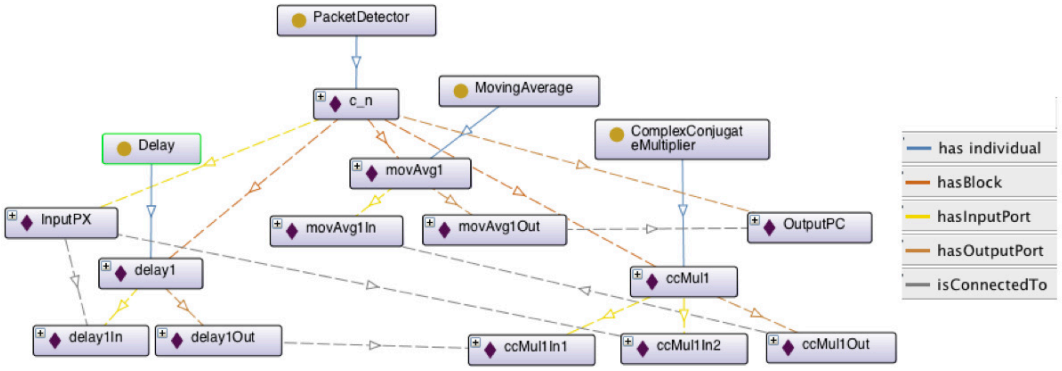


Figure 3: Dataflow of the Packet Detector subsystem

Query	Class	Instance	Knowledge Acquired
Module	PacketDetector	c_n	Subsystem
isInputPortOf value c_n	InputPort	InputPX	Input Port of the Module.
isOutputPortOf value c_n	OutputPort	OutputPC	Output Port of the Module.
isABlockOf value c_n	Delay ComplexConjugateMultiplier MovingAverage	delay1 ccMul1 movAvg1	Components of the Module.
isInputPortOf value delay1 Repeat this query for each component in Module.	InputPort	delay1In	Input Port of each Component.
isOutputPortOf value delay1 Repeat this query for each component in Module.	OutputPort	delay1Out	Output Port of each Component.
isConnectedTo value ccMul1In1 Repeat this query for all ports in Module.	OutputPort	delay1Out	Connections between ports.

Table 1: Querying the dataflow of the Packet Detector subsystem

3.3 Learning through Queries

The usefulness of representing the physical layer using ontology is in its ability to allow learning about the PHY by a machine, *e.g.*, a CR is willing to communicate with the other but is unsure about the capabilities of the other radio. Ontology provides the common language that all CR nodes can use to understand and learn about its own structure and when required, it can let other nodes know about its own capabilities. Understanding the radio PHY independent of the platform, operational knowledge or specific implementation is very beneficial in a heterogeneous environment. With this knowledge distribution mechanism, radios can adapt to form homogeneous communication environment.

If a radio node needs to collaborate at the dataflow level, to implement a common subsystem, it will learn the dataflow of this subsystem from the ontology of the other radio. This is done by a structured querying mechanism. Algorithm 1 shows the automated querying process at the dataflow level. First, the subsystem is searched and if it matches the intended subsystem required for collaboration, then the queries are processed for that. At this point, we are interested in knowing the instances of various classes of components in the dataflow ontology. Once the instance of the subsystem (or Module in Taxonomy) is found, we query the input ports, output ports and the basic components of the subsystem, and update the data structures *PortIns*, *PortOuts* and *Comps*. This gives us the knowledge of the input and output ports of the subsystem, as well as the components or aggregates contained within the subsystem. Then, for each of the component within the subsystem, we query the input ports and the output ports and update the *PortIns* and *PortOuts* data structures. Once the list of all the ports have been generated, we query the connections of each input port and each output port. This gives us the knowledge of how the data flows from one port to another. Through this process of querying, we also get the information of the data type of each port and the programmable variables of each component, which are required in the code generation phase.

Continuing with the example of the OFDM packet detector subsystem, we design a set of queries that will reveal the components present in it and their interconnections. We used DL-Query and OWL APIs [14] to query the ontology. Table 1 shows the queries and the corresponding knowledge obtained about the dataflow between various components. Once the knowledge about all the components and their interconnections are obtained, CODIPHY proceeds to the code generation phase that generates an implementation of the design learned through querying.

4. COMPOSING THE RADIO PHY

In §3, we have shown a method to construct an ontology to represent a physical layer and to learn the relationship between various components and their instances by simple queries. However, composing an operational radio requires many stages of design and optimization and tight coordination of multiple devices and circuits, like the ADC/DAC, RF front-end, I/O interface etc. We believe that the engineering problem of implementing a fully functional radio can be broken down into smaller systems and the baseband is the most important part of it. Ontological description of a system of systems can be constructed to solve the system level problem. So

Algorithm 1: Algorithm to Automate the Query Process at Dataflow Level

```
Input: subSys
/* Module Level Query                                   */
Query: Module;
Result: mods[];
for m in mods[] do
    if subSys == m then
        /* Input Ports of the Subsystem                 */
        Query: isInputPortOf value subSys;
        Result: all input ports of subSys;
        Update: PortIns[subSys].name;
        Update: PortIns[subSys].dataType;
        /* Output Ports of the Subsystem                */
        Query: isOutputPortOf value subSys;
        Result: all output ports of subSys;
        Update: PortOuts[subSys].name;
        Update: PortOuts[subSys].dataType;
        /* Components of the Subsystem                  */
        Query: isABlockOf value subSys;
        Result: all components of subSys;
        Update: Comps.funcName;
        Update: Comps.vars;
        for c in Comps[] do
            /* Input Ports of each Component
               */
            Query: isInputPortOf value c;
            Result: all input ports of component c;
            Update: PortIns[c].name;
            Update: PortIns[c].dataType;
            /* Output Ports of each Component
               */
            Query: isOutputPortOf value c;
            Result: all output ports of component c;
            Update: PortOuts[c].name;
            Update: PortOuts[c].dataType;
        end
        for inP in PortIns[] do
            /* Connections from all Input
               Ports                                    */
            Query: isConnectedTo value inP.name;
            Result: all connections from inP;
            Update: Connects.From = inP;
            Update: Connects.To;
        end
        for outP in PortOuts[] do
            /* Connections from all Output
               Ports                                    */
            Query: isConnectedTo value outP.name;
            Result: all connections from outP;
            Update: Connects.From = outP;
            Update: Connects.To;
        end
    end
end
```

far we have focused on the dataflow level description of the radio that can be used to construct a functioning baseband radio.

Radio processing can be done on any platform but often require domain specific optimizations, *e.g.*, implementing an FFT engine on a stream processor is completely different from that in a FPGA. Our vision of the methodology of CODIPHY is that the device specific implementation and optimization is not possible to solve from a high level description like an ontology. Instead, it is safe to assume that there will be some abstraction between the domain experts and the composing agent. Domain specific languages allow radio agents to reap the operational benefit of the products created by the language while hiding the functional or implementation details. This is typically done by employing function calls, APIs or pre-compiled quasi-static libraries. In composing the PHY from the ontology we use such an approach.

```
void c_n(int* inputPX, int* outputPC)
{
    int delay1Out;
    Delay(inputPX, 64, &delay1Out);
    int ccMul1Out;
    ComplexConjugateMult(delay1Out, InputPX, &ccMul1Out);
    int movAvg1Out;
    MovingAverage(movAvg1In, 64, &movAvg1Out);
    *OutputPC = movAvg1Out;
}
```

Figure 4: Generated C Code for Packet Detector subsystem

4.1 Composing software executable

From the list of components and their connections, we are able to generate a software code, where the components denote function calls to pre-defined functions in the library. Algorithm 2 shows how a C code is generated from the information of components, input ports, output ports and its connections. These information are direct output of the querying process described in algorithm 1. The function declaration requires the input ports of the subsystem, and their data type as input arguments of the function call. All the input ports and output ports are variables in the function, which will be assigned values later. The sequential processing required in the processor architecture, requires tracking of where the data is available when the code is generated. Initially, the data is available only at the input ports of the subsystem. Then, from the *Connects* list, connections are made to the ports from the input port of the subsystem. Next, if all input ports of a component has data available, then the code for that component is generated, which is a function call from the library provided by the domain experts. After the code is generated, the data is now available at the output port of that component. This process of assigning connections and then generating the code for the component is repeated until the code for all the components have been generated.

Figure 4 shows the code generated for the packet detect unit from the dataflow given in the ontology as shown in figure 3, by using the automated querying mechanism described in algorithm 1, and the code generation process described in algorithm 2. It is evident that the generated code is not an optimized version, and requires domain expertise to optimize this implementation. However, the objective of CODIPHY is not to generate optimized code, instead it describes a backbone to represent the domain knowledge in a hierarchical fashion, such that a radio agent can learn and reconfigure its radio with varied parameters. The optimization engine for each hardware type can reside on top of CODIPHY to generate optimized code base for the target platform. We show the code generation process to verify that we can implement the dataflow in correct format in the ontology.

4.2 Composing hardware descriptions

The high level description of radio subsystems using a collection of components and their interconnections facilitates the generation of structural HDLs. The set of basic components and aggregates, as defined in the ontology are mapped onto HDL entities while the

Algorithm 2: Algorithm to Generate C Code from Queries

```
Input: Comps, PortIns, PortOuts, Connects, MotherComp
/* Function Declaration                          */
Print PortOuts[MotherComp].dataType;
Print PortOuts[MotherComp].name;
for pin in PortIns[MotherComp] do
    Print pin.dataType; Print pin.name;
end
/* Variable Declaration                          */
for c in Comps do
    if c not equal to MotherComp then
        for pin in PortIns[c] do
            Print pin.dataType; Print pin.name;
        end
    end
    for pout in PortOuts[c] do
        Print pout.dataType; Print pout.name;
    end
end
/* Initialize Data Ready Port & List of
   Code Generated Components                     */
for pin in PortIns[MotherComp] do
    dReady.append(pin);
end
cgen=[];
/* Body of the Code                              */
while len(cgen) < len(Comps) do
    /* Make connection if data ready             */
    for cons in Connects do
        if cons.From in dReady and cons.To not in dReady
        then
            dReady.append(cons.To);
            Print cons.To, " = ", cons.From , ";" ;
        end
    end
    /* Generate code for a component if
       data is ready                             */
    for c in Comps do
        if c not in cgen then
            if (∀ pin ∈ PortIns[c]) in dReady then
                Print PortOuts[c].name, "=", c.funcName, "(",
                PortIns[c].name, c.vars, ");" ;
            end
            cgen.append(c);
            dReady.append(PortOuts[c]);
            break;
        end
    end
end
/* Return Statement                              */
Print "return", PortOuts[MotherComp].name;
```

ports and their interconnections are translated to physical wires. In this implementation, we have used the Xilinx System Generator (SysGen) API [22] to generate synthesizable designs using Matlab function scripts. The APIs use the *xBlock, xSignal, xInport, and xOutport* objects to construct System Generator models. The various *Individuals* of basic components in the ontology are mapped onto the library blocks of System Generator. The code example below, shows the programmatic synthesis of an *adder* with two inputs 'a' and 'b' and one output 's' with a latency of '1' clock cycle. The library component used in this case is 'AddSub'.

```
[a, b] = xInport('a', 'b');
s = xOutport('s');
adder = xBlock('AddSub', struct(...
            'latency', 1), {a, b}, {s});
```

Subsystem	Modules	Compo--nents	Input Ports	Output Ports	Conne--ctions
TxController[1]	1	29	53	53	55
IFFT[2]	0	7	16	18	12
Modulator	6	63	119	71	102
InsertGuard	3	16	66	32	60

[1] FSM to parse control and data frames[6]

[2] IFFT treated as atomic unit

Table 2: Ontology of the Transmitter

New library elements have been created for aggregates that are not a part of a standard installation which shows the that this technique can be generalized for composing any subsystem of a radio.

A Matlab M-function is generated automatically from the query results using a Python script that contains the instantiation of the various blocks and the signals that connect them. This M-function is used to generate a System Generator model using APIs. The algorithms to synthesize the Matlab script is the same as for synthesizing C-code as shown in algorithm 2, except for hardware synthesis the order of execution is not necessary as all the components run in parallel and simultaneously for every system clock pulse.

The HDL is generated using the compile option in System Generator to produce synthesizable VHDL code that can be implemented using standard tools. We use this method to ensure the correctness of the generated HDL and also provide a platform for users to test the subsystem for functional correctness as well.

5. RESULTS

In this section, we present results from an example radio implementation using the high level knowledge representation system discussed in §3. We have chosen 802.11a/g as the physical layer to be *composed* from an ontology specification and generate executable code to target a general purpose processor as well as FPGAs. Although radio is typically a combination of many systems, in this example, we focus on the baseband signal processing systems only. As a design choice, we omit the binary computation systems as they are relatively low in complexity and are beneficial to run in software form. While the receiver subsystems are more complex and require hardware acceleration, fast software implementations are slowly becoming more practical [19].

Composing physical layers from an ontology specification involves three important steps: 1) Creating the ontology to reflect the components and their interconnections for various subsystems, 2) An interface to retrieve the knowledge in a meaningful way using structured queries, and 3) Automatically generate code for multiple platforms. We report the design complexity and salient features of each of these three categories.

5.1 Knowledge Representation System

Building the ontology is fundamental to making CODIPHY practical. Using domain expertise in hardware design and DSP algorithms, we have identified a set of components and aggregates that are considered *atomic* from a system perspective and are available to the radio design agent as pre-compiled libraries. The dataflow of a subsystem is obtained by analyzing the mathematical equation that specifies a computational flow, transforming the input samples to meaningful output. Table 2 and Table 3 show the complexity of the ontology for various transmitter and receiver systems respectively.

5.2 Querying the ontology

The dataflow of a subsystem of the radio can be obtained by querying the knowledge base or the ontology. The number of

Subsystem	Modules	Compo--nents	Input Ports	Output Ports	Conne--ctions
Packet Detect	6	30	65	39	68
Correlator	1024	5152	8252	6178	8240
FFT[1]	1	20	49	34	44
Equalizer	14	137	289	180	299
Demodulator	0	32	67	34	55
Decoder	7	82	179	116	177
CRC Check	6	45	93	57	91

[1]FFT treated as atomic unit

Table 3: Ontology of the Receiver

Module	Lines of Code
TxController	78
IFFT	21
Modulator	163
InsertGuard	43

Table 4: Lines of C Code Generated

queries required to get all the information in dataflow level depend on the size of the subsystem.

$Total\ number\ of\ queries = (1 + Blocks + Components + InputPorts + OutputPorts)$

Hence, the number of queries required for each module can be computed from tables 2 and 3. The output of the queries is the intermediate form, from which target-specific code is generated. It is stored in the file system which is used by the code generation process.

5.3 Composing radio subsystems

The current toolflow of CODIPHY supports automatic code generation in C and VHDL. To generate an executable C code, each of the components in the ontology are mapped onto pre-defined functions with *inPorts* and *outPorts* as the input and output variables respectively. Each function is assigned to a new variable thus making them explicitly independent. It is our belief that further optimization will make this code more efficient by performing static and dynamic analysis. We do not address these optimization and leave that to domain experts. As of now, CODIPHY supports all the components and aggregates required to generate C code for the 802.11a/g transmitter and the library to compose the receiver is under construction. We report the code sizes for various transmitter blocks in Table 4. Although this implementation focuses on the sample domain subsystems, CODIPHY is generic enough to include libraries for binary domain systems as well and the methodology is same for that.

The hardware utilization for 802.11a/g largely depends on how the subsystems are designed. Various architectures are available to implement particular DSP subsystems. It is not our goal to build the most optimized system, but rather provide the toolflow to generate synthesizable design from a set of pre-compiled components. The interconnection of these components or the architecture is left to domain experts. We choose the dataflow represented in prior implementations of 802.11a/g in [6, 4]. Table 5 and Table 6 shows the hardware utilization for transmitter and receiver respectively on a Virtex V (LX110) FPGA.

Module	Slices	LUTs	BRAM	DSP48s
TxController	120(0.69)	280(0.41)	5(3.91)	0(0.00)
Mod	57(0.33)	97(0.14)	0(0.00)	0(0.00)
IFFT	1349(7.81)	2794(4.04)	4(3.13)	9(14.06)
InsertGuard	245(1.42)	616(0.89)	6(4.69)	0(0.00)

Numbers in parenthesis indicate percentage utilization

Table 5: Hardware Utilization of the Transmitter

Module	Slices	LUTs	BRAM	DSP48s
Packet Detect	536(3.10)	1805(2.61)	0(0.00)	12(18.75)
Correlator	1816(10.51)	2884(4.17)	0(0.00)	2(3.13)
FFT	875(5.06)	1745(2.52)	5(3.91)	30(46.88)
Equalizer[1]	1197(6.93)	3356(4.86)	0(0.00)	4(6.25)
Demodulator	36(0.21)	100(0.14)	0(0.00)	0(0.00)
Decoder[2]	1319(7.63)	2644(3.83)	3(2.34)	0(0.00)
CRC check	114(0.66)	129(0.19)	0(0.00)	0(0.00)

Numbers in parenthesis indicate percentage utilization

[1] Includes channel estimation, [2] Includes De-interleaver and Viterbi decoder

Table 6: Hardware Utilization of the Receiver

6. APPLICATIONS OF CODIPHY

In this paper, we present CODIPHY as a methodology to *compose* complex radio physical layers from a high level specification. The key concept is to introduce clear abstractions in a multidisciplinary environment, which facilitates knowledge sharing. CODIPHY also helps in re-targeting radio design to multiple platforms and architectures. This ensures longevity of the design and reduces time to prototype a radio PHY. The benefit of CODIPHY goes beyond code generation for heterogeneous platforms to enable a larger realm of research. We discuss some of those in this section.

Case 1: Hierarchical inference – CODIPHY assists in design and implementation of radio PHY for users with varied expertise. The various levels of the knowledge representation system discussed in §3 provide design information at various levels of granularity. Each level provides a clear path that leads to implementation using either pre-compiled subsystems or by generating code from atomic components.

Case 2: Collaborative adaptation of radio PHY – Using the querying mechanism, radio agents learn the internal structures of another radio. This collaboration at the physical layer happens in real-time and radio systems can reconfigure themselves by building subsystems that was not originally built for that radio. Thus, *cloning* of radio PHY facilitates MAC-PHY crosslayer implementations in a large scale network because radios can learn, adapt and automatically reconfigure to agree on a particular PHY design.

Case 3: HW/SW Co-design – Platforms like Zynq [21] present great opportunity for efficient radio implementation by employing a hardware, software co-design approach. Hybrid architectures provide the flexibility of software programs while utilizing the fast parallel computation model of FPGAs. Fast interconnects between the two domains make it more practical. The multi-target code generation infrastructure of CODIPHY allows radio designers to partition the design based on high level constraints like speed, power and area. By including these design trade-offs in the ontology specification, each subsystem is constrained accordingly, leading to efficient hybrid architectures.

Case 4: Partial Reconfiguration – The underlying architecture that accepts CODIPHY as the methodology for implementing radio PHY, employs a modular approach. Instead of subsystems being connected by hard synchronization logic, making them brittle, a producer-consumer paradigm of data transfer will make the design more practical. Now, radio designers use partial reconfiguration techniques [1] to *swap* components in real-time that reduces the reconfiguration time. Also, by making the architecture, latency insensitive, independent development of DSP algorithms is possible.

We believe that adopting CODIPHY for future heterogeneous cognitive networks will be beneficial in realizing the greater vision of dynamic radio adaptation. Through CODIPHY, cognitive radios can not only agree on high level policies, but enforce them using proper waveforms at the physical layer through dynamic reconfiguration.

7. RELATED WORK

In order to make radios self aware and adapt to the changing environment, researchers have used ontology to represent the relationship between various radio processes [7]. Authors layout some practical requirement for the language to represent this cognitive engine. In [9], a simple implementation of this has been shown by representing the programmable components of the radios as *knobs* and using a reasoning machine to decide on which parameters to change based on the knowledge obtained from the environment. However, as radios and waveforms become more complex, the number of tunable parameters increase to the extent that parametrizing the variables become nearly impossible. Fixed hardware pipeline and dataflow is no longer optimum and a way to compose radio structures on-demand is provided by CODIPHY. An important aspect of cognition is *understanding your neighbors* and facilitate coexistence in a network topology. An example of radios of different capabilities can collaborate to achieve a better communication link has been shown in [8, 18]. The Wireless Innovation Forum has developed the first Ontology based cognitive radio that is capable of transmitting audio waveform using CDMA technology [2]. The pre-cast radio suffers from similar brittleness as an partially programmable ASIC as it requires re-designing the entire pipeline to ensure proper synchrony between components. Therefore, abstractions at different design stages is the key to develop cognitive radio for the longer run.

8. CONCLUSION

In this paper, we propose a method to use knowledge representation techniques to represent the physical layer of a cognitive radio. By using ontology as the framework we are able to realize collaborative learning between heterogeneous radio nodes to adapt to a common protocol. Not only radios can agree on new access policies but also compose the subsystems required to implement those. We also present a code generation technique that uses the ontology to generate implementation code for multiple targets.

9. REFERENCES

[1] {X}ilinx {P}artial {R}econfiguration of {FPGAs}: http://www.xilinx.com/tools/partial-reconfiguration.htm.

[2] Description of the Cognitive Radio Ontology. *Wireless Innovation Forum*, (September), 2010.

[3] I. P. Conference, N. Drummond, M. Horridge, and H. Knublauch. Protégé-OWL Tutorial. (July):1–41, 2005.

[4] A. Dutta, J. Fifield, G. Schelle, D. Grunwald, and D. Sicker. An Intelligent Physical Layer For Cognitive Radio Networks. In *WICON '08: Proceedings of the 4th international conference on Wireless internet*, New York, NY, USA, 2008. ACM.

[5] A. Dutta, D. Saha, D. Grunwald, and D. Sicker. SMACK: a SMart ACKnowledgment scheme for broadcast messages in wireless networks. *SIGCOMM '09: Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, 39(4):15–26, 2009.

[6] J. Fifield, P. Kasemir, D. Grunwald, and D. Sicker. Experiences with a platform for frequency agile techniques. In *DYSPAN*, 2007.

[7] B. M. M. Kokar and L. Lechowicz. Language Issues for Cognitive Radio. *Proceedings of the IEEE*, 97(4), 2009.

[8] S. Li and M. M. Kokar. Now Radios Can Understand Each Other : Modeling Language for Mobility. (1), 2008.

[9] S. Li, J. Moskal, M. M. Kokar, and D. Brady. An implementation of collaborative adaptation of cognitive radio parameters using an ontology and policy based approach. *Analog Integrated Circuits and Signal Processing*, 69(2-3):283–296, July 2011.

[10] Y. Lin, H. Lee, M. Woh, Y. Harel, S. Mahlke, T. Mudge, C. Chakrabarti, and K. Flautner. SODA: A Low-power Architecture For Software Radio. In *ISCA '06: Proceedings of the 33rd annual international symposium on Computer Architecture*, pages 89–101, Washington, DC, USA, 2006. IEEE Computer Society.

[11] K. B. M. Kokar, D. Brady. Role of Ontologies in Cognitive Radios. In B. A. Fette, editor, *Cognitive Radio Technology*, chapter 13, pages 401–428. Elsevier Science Publishers B. V., 2009.

[12] J. Mitola III and H. Man. Semantics in Cognitive Radio. *2009 IEEE International Conference on Semantic Computing*, pages 261–266, Sept. 2009.

[13] P. Murphy, A. Sabharwal, and B. Aazhang. Design of WARP: A Flexible Wireless Open-Access Research Platform. In *Proceedings of EUSIPCO*, 2006.

[14] U. of Manchester. OWL API.

[15] G. Panesar, D. Towner, A. Duller, A. Gray, and W. Robbins. Deterministic parallel processing. *Int. J. Parallel Program.*, 34(4):323–341, 2006.

[16] H. Rahul, N. Kushman, D. Katabi, C. Sodini, and F. Edalat. Learning to Share: Narrowband-Friendly Wideband Networks. In *SIGCOMM '08: Proceedings of the ACM SIGCOMM 2008 conference on Data communication*, pages 147–158, New York, NY, USA, 2008. ACM.

[17] S. Sen, R. Roy Choudhury, and S. Nelakuditi. No time to countdown. In *Proceedings of the 17th annual international conference on Mobile computing and networking - MobiCom '11*, page 241, New York, New York, USA, Sept. 2011. ACM Press.

[18] M. M. . K. Shujun Li and J. Moskal. Policy-driven Ontology-based Radio: A Public Safety Use Case. *Proceedings of the Software Defined Radio Technical and Product Exposition Forum*, 2008.

[19] K. Tan, J. Zhang, J. Fang, H. Liu, Y. Ye, S. Wang, Y. Zhang, H. Wu, W. Wang, and G. M. Voelker. Sora: high performance software radio using general purpose multi-core processors. In *NSDI'09: Proceedings of the 6th USENIX symposium on Networked systems design and implementation*, pages 75–90, Berkeley, CA, USA, 2009. USENIX Association.

[20] W3C. OWL Web Ontology Language - http://www.w3.org/TR/owl-features/.

[21] Xilinx. ZYNC Processing Platform.

[22] Xilinx. Xilinx System Generator for DSP Reference Guide, 2012.