INVITED PAPER

Language Issues for Cognitive Radio

Computer languages that may be useful for expressing cognitive radio concepts are identified and evaluated in this tutorial paper.

By Mieczyslaw M. Kokar, *Senior Member IEEE*, and Leszek Lechowicz, *Member IEEE*

ABSTRACT | This paper discusses various aspects of formal languages in the context of cognitive radio. A bottom up approach is taken in which an example of a specification of a feature of cognitive radio in a selected language is shown, followed by an example of a feature that cannot be expressed in the language and the identification of a capability that needs to be added to the language in order to cover the identified gap in the expressiveness. Following this pattern, we start with a language based on description logics and add the capability of expressing rules, then functions, and finally behavioral aspects. The running example used in this paper—conveying the description of a component to be synthesized by another radio—covers two major aspects of a cognitive radio: an ability to express own capabilities and an ability to interoperate with other cognitive radios.

KEYWORDS | Cognitive radio; formal language; formal semantics; radio ontology

I. INTRODUCTION

The notion of cognitive radio was first introduced by Mitola [1], [2]. His analysis started with a statement of need for radios to negotiate various aspects of communication etiquette. Mitola's conclusion was that a cognitive radio must be self-aware (know its own structure, both hardware and software) and have a language in which to describe requests and replies to/from other radios and network. He also described a language that could be used to represent this kind of knowledge (models of radio and communication environment).

The Software Defined Radio Forum (SDRF) [3] defines cognitive radio as follows.

a) Radio in which communication systems are aware of their environment and internal state and can make decisions about their radio operating behavior based on that information and predefined objectives. The environmental information may or may not include location information related to communication systems.
b) Cognitive radio [as defined in a)] that utilizes software defined radio, adaptive radio, and other technologies to automatically adjust its behavior or operations to achieve desired objectives.

Kokar *et al.* in [4] pattern the definition of cognitive radio upon the definition of a cognitive agent, i.e., a system that can:

- reason, using substantial amounts of appropriately represented knowledge;
- learn from its experience so that it performs better tomorrow than it did today;
- explain itself and be told what to do;
- be aware of its own capabilities and reflect on its own behavior;
- respond robustly to surprise.

All of these definitions stress the fact that cognitive radios must be able to express their knowledge of themselves and of the communication environment. Thus, either directly or indirectly, they all call for the need of a language in which these concepts can be expressed. Thus in this paper we discuss not whether a language is needed to achieve the cognitive radio functionality but what kind of language it needs to be.

At least two paths can be taken to specify requirements for a language. In one approach one would specify some general requirements for a language, and then such requirements could be refined. The problem with this approach is that for this to be possible, one would need to have pretty good understanding of the expectation for such a language. The concept of cognitive radio is still under

Manuscript received November 11, 2008. Current version published April 15, 2009. The work of M. Kokar was supported in part by the Defense Advanced Projects Research Agency under programs XG, DTN, WANN and WNaN.
The authors are with the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA 02115 USA (e-mail: mkokar@ece.neu.edu; llechowi@ece.neu.edu).

Digital Object Identifier: 10.1109/JPROC.2009.2013028

0018-9219/$25.00 ©2009 IEEE

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

development, and thus it is difficult to put forward requirements that would capture all the future needs.

In this paper, we are taking a different approach. We first start with an attempt to satisfy the needs with an existing language—Web Ontology Language (OWL)[1]—and then identify areas that cannot be expressed in this language. This leads us to consider a more expressive language (a rule language) that covers some of the gaps that cannot be covered in OWL. Following the same path of reasoning, we come to the conclusion that the target language for cognitive radio must include the ability to introduce new functions. And finally, we show that in addition to this, the language must be capable of specifying dynamics (behaviors) of radio components.

Computer languages can be classified into two groups—*imperative* (or *procedural*) and *declarative*. In Section II, we give a brief overview of the main differences between these two groups and then argue that the flexibility of the functionality of cognitive radio requires the inference power of a formal declarative language with formal, *computer-processable semantics*.

The rest of this paper is organized as follows. In Section III, we discuss the role of ontologies in the functioning of cognitive radio and show how OWL can be used for expressing some properties of such radios. Then in Section IV, we show that structures cannot be expressed in OWL and that at least the power of a rule language is necessary to achieve such functionality. We continue our analysis in Section V, recognizing the need for the ability to reason about functions. In Section VI, we discuss the problem of expressing behavioral aspects of radio components. In particular, we discuss the problem of specifying dynamics, e.g., delays. In Section VII, we provide an overview of three efforts whose goal is to develop a standard language for radio/network communications. In Section VIII, we conclude this paper with a discussion of the language issues that have and have not been covered here.

II. PROCEDURAL VERSUS DECLARATIVE

Computer languages can be classified in many different ways. It would be difficult, if not impossible, to come up with a complete and disjoint classification that would capture all of the existing languages. Thus in this paper, we list some of the properties of languages rather than attempt to provide a classification. The basic distinction in the properties of computer languages is whether a language is *imperative* (also known as *procedural*) or *declarative*. Examples of imperative languages are C, C++, Java, and Fortran. Examples of declarative languages include Prolog, SQL, OWL, and SWRL. A program written in an imperative language can be viewed as a list of statements (operations) that manipulate the state of the program [5]. The operations then are executed in a sequence according to the control structure. The control structure is partially captured by the ordering of operations in the list. The rest of the control structure is embedded in the program in the form of control statements like *if-then-else, do-while,* and *do-until*. Thus an imperative program includes the information on both *what* needs to be done and *how* it needs to be accomplished (in what sequence). An imperative program is said to provide an *algorithm*. The "what" information is explicit, although it is distributed over all the statements of the program.

A *declarative language,* on the other hand, represents a somewhat different paradigm of programming. In this paradigm, the program expresses *what* should be accomplished rather then how. The "how" is left to a generic *inference engine*. Thus a declarative program is a collection of facts (referred to as *clauses*) expressed in a declarative language and a goal provided by the user. The generic inference engine then tries to find a solution to the *goal* (or satisfy the *query*). Logic programming, whose example is Prolog, is a representative of the declarative programming paradigm.

Both imperative and declarative programs take some data on input and produce some output. One more difference between the two paradigms shows up in the way modification to a program is achieved. In the procedural approach, modification involves replacement of a program. Typically this means that a system needs to be taken offline and the modified piece needs to be recompiled and linked with the whole system, after which the system can be restarted. In the declarative paradigm, on the other hand, the only part that needs to be modified is the fact and rule base (clauses). A clause can be removed and another clause can be added during the system operation. This is possible because the generic algorithm (the inference engine) is not modified. The clauses can be treated (almost) as input data.

It is important to notice, however, that any modification, whether of an imperative program or of the clauses in a declarative program, carries the risk of inserting errors into the system; thus in either case, such modification requires testing. In the procedural case, this is the new (modified) program that needs to be tested. In the declarative case, the clauses can be tested with an inference engine (not necessarily the same one), and thus the testing procedure can be performed offline before the modification of clauses in the running program. More discussion on the advantages of declarative languages can be found in [6].

So now comes the fundamental question—which kind of computer language is more appropriate for cognitive radio? Unfortunately, there is no simple answer to this question. In an attempt to answer this question, let us look at some of the aspects of the definition of cognitive radio. In particular, we can see that cognitive radio must have the following capabilities:

- be *aware* of its own state and the state of the environment;
- *tell* other radios and network of what it knows and what it wants;

[1] http://www.w3.org/2004/OWL/.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

- *reflect*, i.e., be able to draw conclusions from the facts that it is aware of;
- *react* to surprise, i.e., react to the circumstances it has not seen before.

When we match these requirements with the features of the two types of computer languages, the first reaction seems to be that a declarative language should be able to satisfy all of them. In particular, it is easier for a radio to express its own knowledge in a declarative language rather than in a procedural one since it can be accomplished by stating only the "what" without saying "how." For instance, a radio can tell another radio "do this and this." It would be much more difficult to actually tell the other radio how to accomplish this.

In order to satisfy the "awareness" requirement, it is important to realize that awareness includes not only the knowledge of particular facts but also the ability to understand the implications of the facts to the operation of the radio. For instance, just knowing that the current operating frequency is within the 700 MHz band, without also knowing whether it is currently assigned to analog TV or for public safety, cannot be considered as a case of full awareness. On the other hand, if the radio 1) has a declarative knowledge base that relates various variables, like frequencies, bands, band allocations, error rates, and such and 2) has a generic inference engine that can infer the implications of various operating states and environmental conditions (like the understanding of whether it can access the TV band), then this will satisfy the awareness requirement to a much higher degree.

The self-awareness also may need to include the concept of *reflection,* which is addressed by most declarative languages and by some procedural languages too. It is important to point out here that although a program keeps values of its variables, it does not automatically follow that the program *knows* its own variables. We could require (and this is rather reasonable) that for a program to know its own variables it should be able to answer such queries like—what are your variables? And then, what is the type of that variable, and what is its value?

The ability of reacting to surprise is perhaps the most prominent feature of cognitive radio that points in the direction of the declarative paradigm. If all the required information is available at design time, the programmer can encode this information into a program using an imperative language, which would most likely provide a better performance than one obtained by using an inference engine. This approach will not be too effective if (costly) modifications to the program are frequently required.

In the declarative paradigm, whatever knowledge (clauses) that is available in the knowledge base can be utilized for finding answers to unexpected goals (queries). This is achieved through the searching and matching algorithm of the inference engine. In the procedural paradigm, on the other hand, not only the facts but also the sequence of statements have to be provided by the programmer. Thus, although in both paradigms some knowledge must exist in the program, the fact that a declarative program can find an answer without explicit control knowledge makes this paradigm more appropriate to the achievement of the goal of reacting to surprise. Note, however, that adding new knowledge dynamically to a running program implies that the program must be able to (correctly) interpret the added knowledge by itself. This, in turn, implies that the declarative language in which the knowledge is expressed must have a computer processable semantics. In other words, there must be a logic associated with the language. This leads us to *formal languages* (i.e., languages with formal syntax) that also have *formal semantics*. Formal syntax means rules for deciding whether a given string is in the language or not [strings that are part of a language are also called *well-formed forms* (wffs)] and for generating wffs out of other wffs. Formal semantics, on the other hand, refers to *interpretations,* i.e., mappings from the terms of the language to a mathematical domain (a set of individuals) and from sentences to *truth values*.[2] Together with a set of *inference rules* and an inference engine, this constitutes a *formal system*. In this paper, we discuss formal languages with formal semantics within the context of a formal system.

An inference engine can take a set of sentences in the language and apply the inference rules of the formal system to derive new sentences. A formal system is desired to be *sound*, i.e., given a consistent set of true sentences, it can derive only true sentences, i.e., the sentences that map to the truth value of "true" by the interpretation function (although researchers in the semantic Web community are working on ways of handling inconsistencies and unsoundness, e.g., [7]). This feature is particularly important when automatic inference is carried out by an inference engine, without a human in the loop. Without this characteristic, the system would not have a basis for deciding which of the inference results should be admitted and which had to be ignored. Another desirable (but hardly achievable) feature of a formal system is the *completeness* requirement, which refers to the ability to infer all possible true sentences using the rules of inference.

Lastly, one aspect of computer languages that is very important is the engineering aspect—the time and memory requirements for a given language. The flexibility of declarative languages comes with a price—the high time complexity. The worst case time complexity of inference in most of the declarative languages discussed in this paper falls in the complexity class termed *undecidable* [8]. The consequence of this fact is that inference algorithms are not complete and may return an "unknown" answer given the time and/or memory limits are exceeded.

[2]Note that this is only one of many ways of defining a semantics for a language.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

III. ONTOLOGIES AND OWL

A formal system is just a system for sound inference. In order to make it useful to a specific domain, the things of interest for the domain must be represented in the language so that inference can be applied to the sentences about these things. For the domain of cognitive radio, some knowledge of the wireless communications domain is necessary. The issue of knowledge representation has been investigated for many years now by the artificial intelligence (AI) community. Various competing representation formalisms were developed over the years. Recently, the AI community has converged on an abstraction for knowledge representation called *ontology*. In one sense, ontology is "a science or study of being" [9] (onto), i.e., about what exists in reality. Ontology in this sense is a branch of philosophy. AI, on the other hand, uses ontology in a different sense (although still somewhat related to philosophy). According to an AI definition (see [10]), "an ontology is an explicit specification of a conceptualization." Thus it is just knowledge of a domain represented in a declarative formalism. These are definitions that associate the names of things in the universe of discourse (e.g., classes, relations, functions, or individuals) with human-readable text describing what the names mean, and formal axioms that constrain the interpretation and well-formed use of these terms. It is a statement of a logical theory. In this paper, we use the term "ontology" in the sense of AI.

Another consensus that the AI community has reached is that a common language is desirable for representing ontologies. This trend can be observed in various communities. For instance, the software engineering community has settled on one language—the Unified Modeling Language (UML). Although there is no such a strong agreement on a standard ontology representation language, the OWL has collected the largest number of practitioners and supporters so far. It is worthwhile mentioning that the semantic Web community is working on various ways of modifying OWL, in the direction of both weakening and increasing its expressivity.

In this paper, we will present our attempts at using OWL for cognitive radio. Rather than waiting for a fully approved standard (which may never happen), for the purpose of this paper we took OWL as it exists now. In the rest of this paper, we show what can and what cannot be achieved with OWL. We also show what kind of expressiveness extensions are needed to satisfy the cognitive radio requirements listed in Section I.

To introduce the concept of ontology (as used in this paper), we use a simple example of an ontology that is relevant to the cognitive radio (CR) domain shown in Fig. 1.

An *ontology* for a domain specifies the concepts of the domain, attributes of the concepts, and relationships among the concepts. Concepts are specified as *classes*, interpreted as sets of *individuals*. Typically, classes are represented graphically by rectangles. For instance, in this

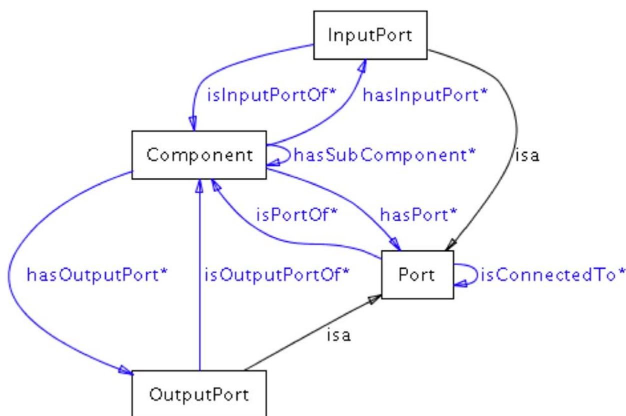

Fig 1. ***A simple ontology: Component and Ports.***

example, Component and Port are classes. Relationships are represented as arrows. For instance, isInputPortOf is a relation (usually called *property*).

Classes are organized into a hierarchy of classes by the *subclass* relationship. For example, InputPort and OutputPort are both subclasses of the Port class. This means that each individual of InputPort and OutputPort is also an individual of Port. The subclass relationship in the notation we use here is represented by an arrow from the subclass to the superclass with an annotation "isa."

Properties are relationships among individuals. There are two kinds of properties. *Data type properties* are attributes that individuals have, e.g., the number of ports that a component has. A data type property is a characteristic of a single individual, where that characteristic is a data value such as a number. An *object property* is a relationship among various individuals. For example, a component can have input ports and output ports. This is shown in the ontology as arrows from the class Component to the classes InputPort and OutputPort. The arrows are annotated with the name of the properties—hasInputPort and hasOutputPort, respectively. The class at the tail of the arrow is called the *domain* of the property, while the class at the head of the arrow is called the *range* of the property. An ontology will generally have many different kinds of data type and object properties. As with classes, one kind of property may be regarded as a set of elements called *facts*. For example, when a particular component c has an input port p, this fact is represented by the triple $(c, hasInputPort, p)$. Properties can be organized in a hierarchy by the *subproperty* relationship.

The following fragment of the OWL code represents the ontology shown in Fig. 1. It is represented here in the XML syntax. First it states that this is a legal RDF document and lists the XML namespaces and their abbreviations. The ontology shows declarations of classes (delimited by the owl:Class tags). Then it lists the properties (delimited by the owl:ObjectProperty tags)

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

with specifications of domains and ranges of all the properties. It also states that some of the classes satisfy the owl:disjointWith property, i.e., they have no individuals in common. All the classes, subclasses, and properties in this listing can be directly traced to Fig. 1.

```
<?xml version="1.0"?>
<rdf:RDF
xmlns="http://www.llech.com/RadioTest1.owl#"
xmlns:rdf="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:xsd="http://www.w3.org/2001/XMLSchema#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:owl="http://www.w3.org/2002/07/owl#"
xml:base="http://www.llech.com/RadioTest1.owl">
<owl:Ontology rdf:about=""/>
<owl:Class rdf:ID="Port">
    <owl:disjointWith>
        <owl:Class rdf:ID="Component"/>
    </owl:disjointWith>
</owl:Class>
<owl:Class rdf:ID="OutputPort">
    <rdfs:subClassOf rdf:resource="#Port"/>
    <owl:disjointWith>
        <owl:Class rdf:ID="InputPort"/>
    </owl:disjointWith>
</owl:Class>
<owl:Class rdf:about="#InputPort">
    <rdfs:subClassOf rdf:resource="#Port"/>
    <owl:disjointWith
        rdf:resource="#OutputPort"/>
</owl:Class>
<owl:Class rdf:about="#Component">
    <owl:disjointWith rdf:resource="#Port"/>
</owl:Class>
<owl:ObjectProperty rdf:ID="hasPort">
    <rdfs:domain rdf:resource="#Component"/>
    <rdfs:range rdf:resource="#Port"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="isConnectedTo">
    <rdfs:range rdf:resource="#Port"/>
    <rdfs:domain rdf:resource="#Port"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasInputPort">
    <rdfs:range rdf:resource="#InputPort"/>
    <rdfs:domain rdf:resource="#Component"/>
    <rdfs:subPropertyOf
        rdf:resource="#hasPort"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasSubComponent">
    <rdfs:range rdf:resource="#Component"/>
    <rdfs:domain rdf:resource="#Component"/>
</owl:ObjectProperty>
<owl:ObjectProperty rdf:ID="hasOutputPort">
    <rdfs:subPropertyOf
        rdf:resource="#hasPort"/>
    <rdfs:range rdf:resource="#OutputPort"/>
    <rdfs:domain rdf:resource="#Component"/>
</owl:ObjectProperty>
</rdf:RDF>
```

To illustrate the use of OWL in the software radio domain, consider a component [quadrature modulator (QM)] shown in Fig. 2.

QM is one of the most fundamental building blocks used in radio engineering [11]. QM shifts a carrier by 90°, multiplies the original carrier by one signal and the shifted by another, and adds the two signals together. In this way, QM effectively encodes two different signals in the same band. Since the two carriers are orthogonal, the encoded information can be extracted at the receiver by synchronous demodulation. QM is frequently used to implement various functional modules in modern communication systems, for example, quadrature amplitude modulator (QAM), phase-shift keying modulator, and others. Practical examples of its use are shown in Fig. 3.

The choice of QM in our running example can be argued as being too simplistic and too disconnected from the context of its use for a real-life radio engineering example. Clearly, from the radio engineering perspective, it would be more convincing to show an example of the specification of the QAM instead. However, such a component would be too complex for showing the fundamental concepts related to ontologies, rules, and functions, mainly because a formal specification of this component would require many pages of code. For this reason, we use QM as our running example throughout this paper.

In order to describe the QM component, our ontology needs to have additional concepts. In particular, we need the notion of BasicComponent specialized to various kinds of basic components, like Adder, Multiplier, or Phase-Shifter. For components that are not basic, we introduce the concept of Module. An extension of the ontology of Fig. 1 is shown in Fig. 4.

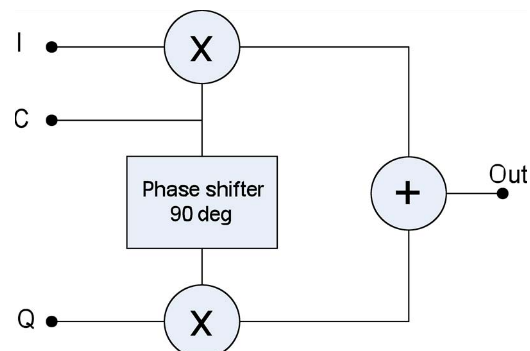

Fig 2. ***A simple component (quadrature modulator).***

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

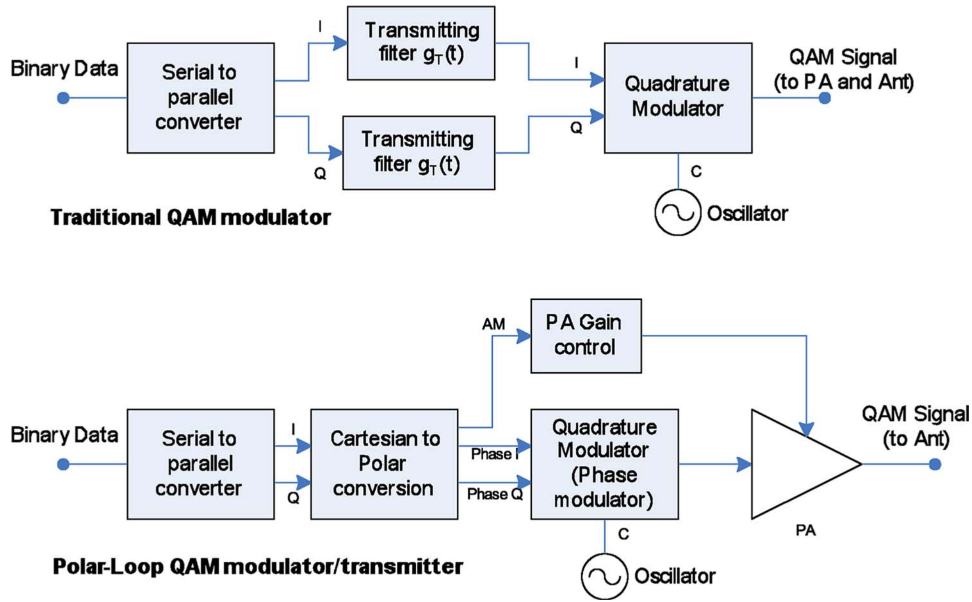

Fig 3. *Example uses of QM in radio engineering.*

The component shown in Fig. 2 consists of four basic components: one adder, one phase shifter, and two multipliers. These components are captured as subclasses of BasicComponent of the ontology. QuadratureModulator has three ports for external input and one output port. Moreover, although this is not shown in the figure, each of the basic components has at least one input port and an output port of its own.

In Fig. 5, we show a graphical representation of a partial description of an individual of the Quadrature-Modulator class. The arrows annotated with 'io' stand for "individual of," i.e., the assertions that a given individual belongs to a given class. For the sake of readability of the description, only some of the ports and the connections among them are shown. As can be seen in this figure, the ports include six individuals of InputPort and three individuals of OutputPort. The input port InputPC, representing the C input, is connected to the input port Mul1_Pin1 (input port of Multiplier 1) and PhSh1_Pin (input port of Phase Shifter). The rest of the connections can be traced in the similar way. A similar scenario, although in the context of Web services, has been shown in [12].

The main purpose of this exercise was to show that individuals of classes of complex components can be represented in OWL. This means that a CR node can describe its internal structure, i.e., the composition of complex components (modules) can be represented. This kind of capability may be very useful for the operation of the CR. However, under some circumstances, a more advanced capability is required. To make this point more concrete, consider the following interoperability scenario [13].

The synthesis of more advanced concepts from the facts in the base ontology is the foundation of the

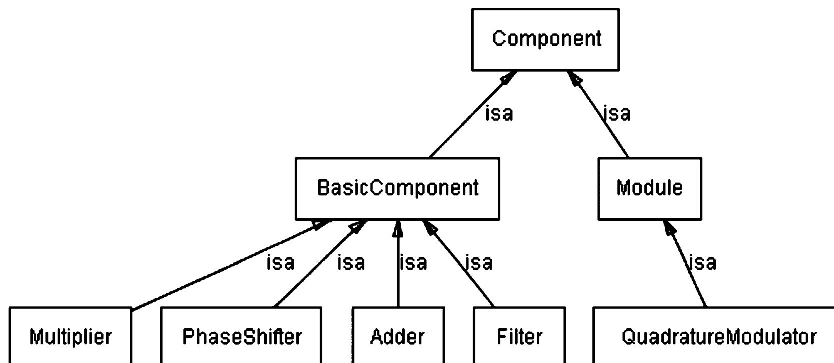

Fig 4. *A simple ontology: Subcomponents.*

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

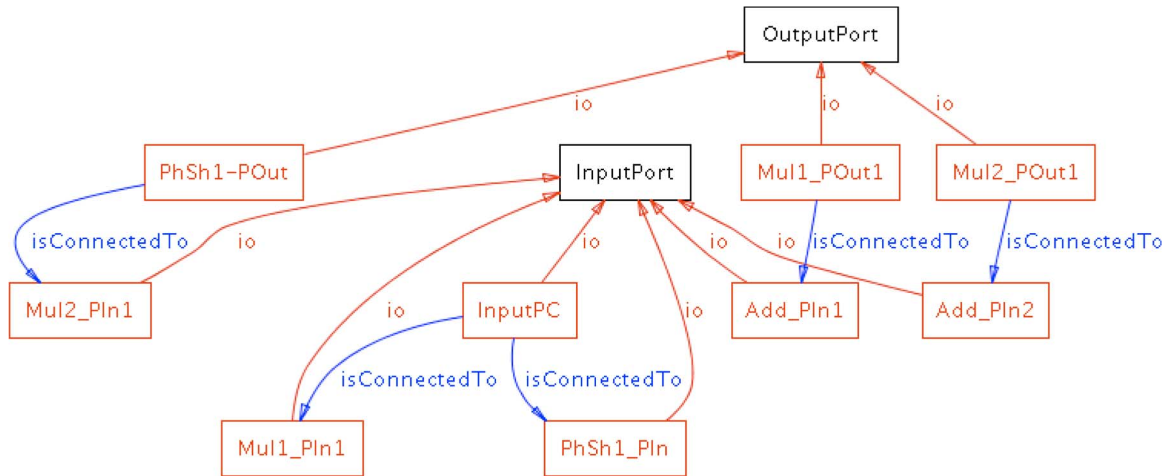

Fig 5. ***An individual of QM.***

interoperability between cognitive radios. One of the uses for this idea is achieving interoperability between CR nodes at the specific protocol level.

For example, one of the nodes (node A) might decide that switching to a specific modulation scheme might provide more optimized communication results in particular circumstances. That node then sends a request to its peer (node B) to make a switch. If the peer "knows" the chosen modulation scheme, it might choose to comply with the request or might decide to reject it. If it does not "know" the proposed modulation scheme, it might in return send a query for an explanation of what exactly that modulation scheme is. If the response to that query contains concepts that are still not understood by the node, it can continue querying for concepts all the way down to those defined in the base ontology. While the scenario in which a node does not know a communication protocol might be extreme, the assumption that a node knows everything would be at the other end of the spectrum. In other words, assuming that communicating nodes, especially CR nodes, know all the possible concepts about which they can exchange information would impose extremely demanding (if achievable at all) requirements on the capabilities of cognitive radio.

Continuing with our example, assume that Node A requests Node B to use a QuadratureModulator but Node B does not have such a module in its library. If the ontology is built as an extension to an upper layer ontology (e.g., SUMO,[3] BFO,[4] DOLCE,[5] or GFO[6]) then Node B might be able to find at least some partial (more general) information about the requested class since the class might be a subclass of one of the classes in the upper ontology. Otherwise, Node A needs to explain to Node B how to build such a component using the four basic components. The straightforward way to do this is to send the definition of class QuadratureModulator to Node B so that Node B can construct a structure that is an individual of the class. In other words, Node B, after receiving the description of the QuadratureModulator class, would have to formally prove (using its own reasoner) that the constructed structure (MyComponent) is an individual (or *type*) of QuadratureModulator. Putting this in OWL terms, Node B would have to prove:

```
<Module rdf:ID="MyComponent">
<rdf:type rdf:resource="#QuadratureModulator"/>
</Module>
```

Unfortunately, the expressive capabilities of OWL do not allow for a representation of QuadratureModulator that would guarantee the correct decision. In other words, it is impossible to construct an OWL description that would capture all components that are considered to be quadrature modulators and none of the components that are not considered quadrature modulators. This is due to the fact that OWL does not have a construct for expressing composition of properties (relations). In this case, the quadrature modulator has two multipliers. We can express in OWL the fact that a particular class (such as QuadratureModulator) is in a relationship with another class (such as Multiplier) using a property (in our case, hasSubComponent). Moreover, we can say that there are two individuals of Multiplier in a quadrature modulator. We cannot, however, distinguish the relationship with one of the multipliers from the relationship with the other one, i.e., we can express in OWL the fact that the quadrature modulator is in the relationship with InputPort, but it is impossible to

[3]http://www.ontologyportal.org/.
[4]http://www.ifomis.org/bfo.
[5]http://www.loa-cnr.it/DOLCE.html.
[6]http://www.onto-med.de/en/theories/gfo/index.html.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

differentiate between the particular relationships for input ports I, Q, and C. The implication of this is that the definition of QuadratureModulator is not restrictive enough, i.e., various configurations will satisfy the definition for as long as they have the necessary components. However, they will be classified as individuals of this class even if they do not have the correct connections.

IV. RULES

The limitation of OWL described in the previous section can be resolved by extending the expressive power of OWL using other, more expressive languages and more powerful reasoning mechanisms. The limitation of OWL described above has been known in the semantic Web community as the inability to define the "uncle" relationship. In other words, it is impossible to express in OWL that an uncle is a person who is the brother of someone's father. The main issue here is that OWL does not have a way to capture the "who is" part of the above statement. In order to be able to do this, we would need the notion of variables that link two different relations. In this example, we would need a variable, say, ?X, which would represent the father, who would then have a child and a brother. This means OWL lacks the capability of expressing the composition of relations.

Other languages, e.g., Prolog [14]—probably the most known declarative language (also classified as a *logic programming* language)—allows logical statements that are *Horn clauses*. A Horn clause is a disjunction of *literals* in which at most one literal is positive. A *definite* Horn clause has exactly one positive literal and thus is of the form

$$\neg A_1 \vee \neg A_2 \vee \ldots \vee \neg A_n \vee B.$$

This can be rewritten in an equivalent form as an implication (also often referred to as a *rule*)

$$A_1 \wedge A_2 \wedge \ldots . \wedge A_n \Rightarrow B.$$

The symbols used in these two formulas represent negation ($\neg$), logical OR operation ($\vee$), logical AND operation ($\wedge$), and logical implication ($\Rightarrow$). In operational terms, this rule states that whenever $A_1, A_2, \ldots, A_n$ hold, so does B. Using Prolog notation, this formula can be represented as

$$B :- A_1, A_2, \ldots, A_n.$$

In this notation, the implication arrow is represented as ":-" while "," represents the logical AND. The literals can be either propositional variables or first-order logic atomic formulas. Since variables are allowed in these formulas, we can represent the composition of relations, e.g.,

$$\text{uncleOf}(?Z, ?Y) :- \text{fatherOf}(?X, ?Y), \text{brother}(?X, ?Z).$$

Returning to our example, by using rules (with variables), the description of the QuadratureModulator class can be made more precise. For instance, we could reinforce the class definition by adding the predicate hasQMConnections ('?' as the first character in a name indicates that it is the name of a variable):

```
hasQMConnections(?QM) :-
    Module(?QM), hasSubComponent(?QM, ?M1),
    type(?M1, Multiplier), hasInputPort(?M1, ?InPortM1),
    hasInputPort(?QM, ?I), isConnected(?I, ?InPortM1), ...,
    hasSubComponent(?QM, ?M2), not(?M1=?M2), ...
```

This is not a complete rule. The intent here was to show how the following facts about a quadrature modulator could be expressed using rules:

1) that a quadrature modulator has two different multipliers M1 and M2;
2) that the input port I of the quadrature modulator is connected to the input port of M1 (but not to M2).

The net result is that once the ontology is extended with this kind of rule, the inference engine can verify that all the necessary connections are in place and thus that a given structure is in fact a quadrature modulator. In our experiments, we used BaseVISor [15], an engine that can perform inference both with a subset of OWL axioms (so called "R-entailment" [16]) and over rules. The need to extend the expressivity of OWL by adding rules has also been recognized in [17]–[19], where both OWL and rules were used to express spectrum access policies. In these experiments, Jena[7] was used as an inference engine for reasoning over both the ontology and the rules.

A. Negation

While rules can make definitions of classes and properties more precise, there is yet another aspect that needs to be understood in order to avoid errors. This is the issue of various types of negation.

OWL is based on the *open world assumption* model in which facts that have not been explicitly asserted to be true are not presumed to be false; they are simply unknown. Negative facts, in this approach, have to be explicitly proven. In other words, both the positive and the negative facts are treated in the same fashion (symmetrically). In this approach, facts that have already been proven to be true (or false) remain true (or false, respectively)

[7]http://jena.sourceforge.net/.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

independently of new facts that might become part of an existing base of facts. Only the facts whose truth values were unknown can be modified to either true or false. Due to this incremental nature of inference, the reasoning in this kind of systems is called *monotonic*. Negation in this approach is referred to as *logical negation*.

Some of the rule languages, on the other hand, take a different approach called the *closed world assumption* (CWA). In this approach, if a fact cannot be proven to be true, it is taken to be false. In case new facts arrive, the inference system must modify its conclusions, i.e., modify the truth values of some of the facts. Reasoning in this kind of systems is called *nonmonotonic*. Perhaps the most known example of this kind of reasoning is seen in databases, where if a fact is not in the database, it is assumed to be false. For example, if a flight is not listed in an airline's database, it is inferred that it does not exist. This is consistent with the name—the world is assumed to be closed, i.e., all that is relevant about it is in the database. Negation in this approach is referred to as *negation as failure* (NAF).

Both kinds of negation—NAF and logical—are useful in modeling real-life problems [20]. If we admit only one type of negation in a formal language, we can have difficulty modeling various situations and reasoning about them using automatic inference engines. For instance, if we accept only logical negation, we will not be able to infer many of the negative facts that, by default, are known to hold. For instance, if a CR has policies that tell it under what circumstances it can transmit, it will not be able to infer that it cannot transmit in case any of these circumstances do not hold. But, on the other hand, if we accept the CWA, then all the facts that are not currently in the database of the CR's facts will be inferred to be false. For instance, under CWA, the CR will infer that the node it is communicating with does not have a quadrature modulator component (unless it has an explicit statement in its knowledge base that it does—and it is rather unlikely that a node would have complete knowledge about all the nodes it communicates with).

So what is a way out of this impasse? Since both logical negation and NAF are needed and since the combination of OWL and rules are necessary to provide more expressive power for modeling real-life problems, there is a need for a language that combines the features of both OWL and rules with a semantics that provides the meaning to both types of negation. Such a necessity has been recognized by the semantic Web community, and research efforts are under way to achieve such a goal (see [21]). In the meantime, the burden of avoiding logical inconsistencies while modeling real-life problems is put on the developers of ontologies, rules, and inference engines.

An example of this problem in the CR domain is the description of structure of radio components. A complete knowledge of a component must include both what subcomponents it has and how they are connected with each other, as well as that there are no other connections except those explicitly stated. For example, consider one of the input ports (port I) on the quadrature modulator shown in Fig. 2. The description of the QuadratureModulator component must express the fact that an individual of InputPort is connected to exactly one multiplier. We can capture this by two rules that define the isExclusiveInputPortOf(?P,?C) and isNonExclusiveInputPortOf(?P,?C) predicates

isExclusiveInputPortOf(?P,?C) :– inputPortOf(?P,?C),
not(isNonExclusiveInputPortOf(?P,?C)).

isNonExclusiveInputPortOf(?P,?C) :–
inputPortOf(?P,?C), inputPortOf(?P,?D), not(?C = ?D):

The predicate isNonExclusiveInputPortOf is asserted when a given input port is in the inputPortOf relationship with more than one individual. The complementary predicate isExclusiveInputPortOf can evaluate to true if isNonExclusiveInputPortOf is false. In the open world model, we could not make such a deduction because failure to prove something does not imply the opposite. In the open world, the failure to prove this might be the consequence of the lack of complete knowledge about the connections within this particular component. Thus, in this example, we had to assume the closed world model. Consequently, the "not" predicate in the body of the rule is a case of NAF. If more information about this port becomes available at a later time, the derived fact (that the port is an exclusive port) may be negated, leading to an inconsistency (both the fact that the port is exclusive and is not exclusive in the same knowledge base). The reasoner must ensure that such a situation does not take place. One way to avoid this is to monitor for such nonmonotonic changes and remove the facts that have been derived through the use of NAF.

It is also worth noting that OWL provides the capability to state that a given class is "closed," i.e., that a class includes only a given collection of individuals. This is termed as the "closed domain reasoning" [21], which is not the same as the closed world reasoning. In the closed domain reasoning we make assumptions about the domain and in the closed world reasoning we make assumptions about our knowledge about the domain. In the above example, we assumed that we knew everything about the connections within that component, i.e., we were making an assumption about the completeness of our knowledge about the domain.

V. FUNCTIONS

Augmenting OWL with a rule language enables cognitive radio nodes to exchange information, for instance, about the structure of their components (class descriptions). In general, nodes can exchange information about concepts that are not explicitly defined in an ontology but can be

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

expressed using OWL and terms from the ontology. For instance, a node can query other nodes about the structure of unknown components and then use this knowledge for reconstructing components locally. Moreover, nodes can use an automatic reasoner to prove that the component it created locally is indeed the individual of the class described in the recipe received from the remote node.

The knowledge of the structure of components, however, is insufficient for some reasoning tasks that involve components [22].

- Software components that are structurally different may be equivalent in terms of their functionality. For example, a software component implementing the function $f(a, b, c, d)$ according to the scheme $f(a, b, c, d) = (a + b)(c + d)$ is functionally equivalent to the module implementing the function f′ according to the scheme $f'(x, y, v, u) = (xv + xu + yv + yu)$, in spite of the fact that their respective internal structures are different. An example of such a situation was shown in Fig. 3, where the classical QAM modulator followed by a power amplifier is equivalent to the polar-loop QAM modulator. Obviously, since those two circuits work using different principles, their structures and functional subcomponents are different.
- The same functionality using two different data types is seen as two different structures, as the structure-based approach does not allow for easy abstraction of the functionality from the data type.
- The structure-based approach does not allow for an easy "understanding" of the functionality, which might lead to implementation inefficiencies. For example, a CR node receiving the description of the quadrature modulator (Fig. 2) expressed in terms of a base ontology and rules might not be able to realize that an alternative, more efficient implementation of such structure might exist, e.g., one that uses a specialized Multiply-Add hardware unit that is available for the node (see Fig. 6).

All of these shortcomings support the requirement that the modeling language for cognitive radio should be capable of describing the "functionality" of components. In formal terms, this means that the language should support functions. Unfortunately, OWL has a very limited capability in this respect. While it is possible to declare properties that are functional, it is not possible to quantify over functions. It is possible to state that two or more properties are equivalent properties, but OWL does not provide any mechanism that would enable inferring that two functions, like those listed in the examples above, are equivalent. Although rule languages can provide definitions of functions, they still do not resolve the above issues. Although rules can provide definitions of particular functions, they do not provide quantification over functions and do not provide capabilities of expressing equivalence of functions. In order to address those issues, a more expressive language has to be selected.

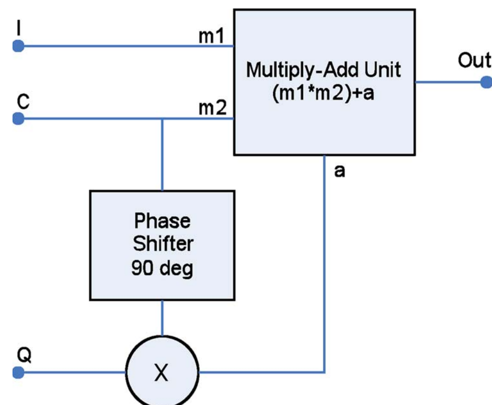

Fig 6. ***A composite module functionality equivalent to QM in Fig. 2.***

Quantification over functions takes us into the realm of *second-order logic* [23], which in turn is extended by *higher order logic* [24]. It is interesting to notice that higher order logic has been used in hardware verification (see [25]). While functions are at least partially covered by a number of declarative languages, for our discussion we choose a higher order logic language Metaslang, the language supported by the Specware tool [26]. The main reason for the selection of this language was that it supports composition, using constructs of category theory like *morphism* and *colimit* [27]. Specware also integrates two theorem provers (Snark [28] and Isabelle [29]) that can be used to prove conjectures on functional equivalency of components. Since Metaslang is based on the principles of category theory, it seems to be a good candidate for linking multiple languages and multiple inference engines into a formal hybrid inference system. The category theory concept of colimit is applied to the category of *specifications* (also referred to as *Spec*) to compose specifications that are related through specification morphisms [30]. Pavlovic and Smith [30] define specification as a finite presentation of a theory. The *signature* of a specification defines concepts that describe individuals, operations, and properties in some domain. The *axioms* included in the specification put constraints on the meaning of symbols.

In order to show how functionality can be expressed in Metaslang and then reasoned about within the Specware framework, we first need to introduce some basic elements and concepts of the language. Additionally, we show how the composition of specifications works in Metaslang. Moreover, we show how the same abstract specification can be easily refined to concrete specifications, i.e., how the specification of *Samples* can be refined to either real or complex samples.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

The following code presents an example of two simple specifications.

```
BinRel = spec
    type E
    op le: E*E -> Boolean
endspec

PreOrder = spec
    import BinRel

    axiom reflexivity is
       fa(x) x le x

    axiom transitivity is
       fa(x,y,z)
          (x le y) && (y le z) = > (x le z)
endspec
```

BinRel is an abstract specification defining an abstract type E and a binary operation *le*.

Those two elements are specified but not defined. *PreOrder* is a refinement of the first specification and it adds two axioms that constrain the functionality of the op *le*. Specification morphisms map one specification into another in such a way that all theorems from the source specification are preserved in the target.

```
Antisymmetry = spec
    type X
    op binOp: X*X -> Boolean
    axiom antisymmetry is fa(x,y)
       binOp(x,y) && binOp(y,x) = > x = y
endspec

m_BinRel_Antisymmetry =
    morphism BinRel-> Antisymmetry
       {E +-> X, le +-> binOp}
```

In the above example, the morphism *m_BinRel_Antisymmetry* maps *BinRel* into *Antisymmetry*. It maps type *E* of *BinRel* into type *X* of *Antisymmetry* and the op *le* into *binOp*.

A directed graph formed from specifications (nodes) and morphisms (edges) is called a *specification diagram*. The three example specifications create the following diagram.

```
BinRelDiag = diagram {
    n1 +->
       BinRel,
    n2 +->
       PreOrder,
    n3 +->
       Antisymmetry,
    e1: n1->n2 +->
       morphism BinRel -> PreOrder {},
    e2: n1->n3 +->
       m_BinRel_Antisymmetry
}
```

This diagram has three nodes ($n1, n2, n3$) representing three specs (*BinRel*, *PreOrder*, and *Antisymmetry*, respectively) and two edges $e1$ and $e2$. Note that since *PreOrder* imports *BinRel* (in other words, *BinRel* is a part of *PreOrder*), the morphism from *BinRel* to *PreOrder* is a trivial one.

Specware can produce the colimit of the specifications on the diagram. For example, *PartialOrder* can be defined as a colimit of *BinRelDiag*.

```
PartialOrder = colimit BinRelDiag
```

An inspection of the resulting specification is shown below. As we can see, the *PartialOreder* spec "produced" by the colimit operation includes all of the axioms of the specs used in this composition.

```
spec PartialOrder
type {X, E}
op {binOp, le}: X * X -> Boolean
import translate (BinRel) by
    {type E +-> {X, E, E}, op le +-> le}
axiom reflexivity is fa(x:E) x le x = true
axiom transitivity is fa(x:E, y:E, z:E)
    x le y && y le z = > x le z
axiom antisymmetry is fa(x:X, y:X) binOp(x, y) &&
binOp(y, x) = > x = y
endspec
```

One of the shortcomings of the structure-based interoperability scenario—the difficulty of separating the functionality from the underlying data type—can easily be solved in Specware through the use of specification refinements. For example, a Multiply-Add unit processing real samples represented by single precision floating-point numbers will be composed quite differently than a unit processing pairs of integers representing complex samples. There is, however, some commonality between those two functional units that could and should be captured at some abstract functionality level. There are two benefits related to that—first, the common functionality is represented only once, so we avoid the inefficiency related to representing and maintaining the same functionality twice. The second, even bigger benefit is that Specware is aware of the fact that those two different modules at some abstract level are *the same* and that this knowledge can be used in refinements and proofs of theorems.

In the example shown below, an abstract specification *Samples* is refined into two concrete specifications *IntSamples* and *CplxIntSamples* (for real and complex samples, respectively). Any software module using *Samples*

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

can easily be refined into a module using real samples or complex samples. First we show the specification *Samples*. It first introduces the type *Sample* and *NonZeroSample*, including the constants *zero* and *one*. Then it shows the signatures of the operations *add*, *multiply*, and *minus*. Lastly, it presents some of the axioms that define these three operations.

```
Samples = spec
   type Sample
   type NonZeroSample = (Sample | nonzero?)

   op Sample.zero: Sample
   op Sample.one: Sample

   op Sample.nonzero?: Sample->Boolean
   def Sample.nonzero?(x) = x ~= Sample.zero

   op Sample.multiply: Sample*Sample->Sample
   op Sample.add: Sample*Sample->Sample
   op Sample.minus: Sample->Sample

   axiom Sample_mul_ax is
     fa(a:Sample, b:Sample)
       Sample.multiply(a,b) =
Sample.multiply(b,a)
   axiom Sample.add_ax is
     fa(a:Sample, b:Sample)
       Sample.add(a,b) = Sample.add(b,a)
   axiom Sample.mul_add_ax is
     fa(a:Sample, b:Sample, c:Sample)
       Sample.multiply(a, Sample.add(b,c)) =
         Sample.add(Sample.multiply(a,b),
                    Sample.multiply(a,c))
endspec
```

Below, we show the specification of samples that take integer values. It imports the *Samples* spec and then defines the constants *zero* and *one* as integer values of 0 and 1, respectively. Moreover, it identifies the operations of *add*, *multiply*, and *minus* with their counterparts in the *Integer* type.

```
IntSamples = spec
   import Samples
   type Sample = Integer

   def Sample.zero = 0
   def Sample.one = 1
   def Sample.multiply(x,y) = x * y
   def Sample.add(x,y) = x+y
   def Sample.minus(x) = -x
endspec
```

The *CplxIntSamples* spec shown below defines the meaning of the constants and operations for the type of complex-valued samples. It also expands the *Samples* spec by the operation *conj* standing for "complex conjugate,"

```
CplxIntSamples = spec
   import Samples
   type Sample = {re:Integer, im:Integer}
   def Sample.zero = {re = 0, im = 0}
   def Sample.one = {re = 1, im = 0}
   def Sample.multiply(x,y) =
     {re = (x.re * y.re − x.im * y.im)
      im = (x.re * y.im + x.im * y.re)}
   def Sample.add(x,y) =
     {re = (x.re + y.re), im = (x.im + y.im)}
   def Sample.minus(x) =
     {re = −x.re, im = −x.im}

   op Sample.conj: Sample -> Sample
   def Sample.conj(x) =
     {re = x.re, im = −x.im}
endspec
```

In the code fragment below *Adder_Int* and *Adder_CplxInt* are refinements of *Adder* with concrete data types *IntSamples* and *CplxIntSamples*, respectively. Those specs are the result of specification substitution operation (square brackets), which is a simplified form of colimit.

```
MorphInt =
   morphism Samples -> IntSamples { }

MorphCplxInt =
   morphism Samples -> CplxIntSamples { }

Adder = spec
   import SampleSpec#Samples
   op Adder.Func: Sample*Sample -> Sample
   def Adder.Func(x,y) = Sample.add(x,y)
endspec

Adder_Int = Adder[MorphInt]
   Adder_CplxInt = Adder[MorphCplxInt]
```

The use of a theorem prover in Specware, together with the fact that all transformations between specifications are formalized in Metaslang, makes it possible to prove the equivalence of the functionality of two software modules. In the following example, the specification QuadratureMod is expressed with elements from the base ontology. The receiving CR node composes the specification Quad2, which uses a composite module MAC, not present in the base ontology. The reasoner is able to prove (see conjecture Quad2_conj below) that the function Quad2.Func used by the receiving CR node is equivalent to function QuadratureMod.Func in the original specification.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

```
QuadratureMod = spec
  import CplxIntSamples
  op QuadratureMod.Func:
      Sample*Sample*Sample -> Sample
  def QuadratureMod.Func(I,Q,C) =
    Sample.add(Sample.multiply(I, C),
      Sample.multiply(Sample.conj(C),Q))
endspec

MAC = spec
  import CplxIntSamples
  import Adder_CplxInt
  import Multiplier_CplxInt

  op MAC.Func: Sample*Sample*Sample -> Sample
  def MAC.Func(m1,m2,a) =
    Adder.Func(Multiplier.Func(m1,m2),a)
endspec

Quad2 = spec
  import CplxIntSamples
  import Adder_CplxInt
  import Multiplier_CplxInt
  import PhShifter90Deg
  import MAC
  import QuadratureMod

  op Quad2.Func: Sample*Sample*Sample -> Sample
  def Quad2.Func(I,Q,C) =
    MAC.Func(I, C, Multiplier.Func(
              PhShifter90Deg.Func(C), Q))

  conjecture Quad2_conj is
    fa(I:Sample, Q:Sample, C:Sample)
    Quad2.Func(I,Q,C) =
    QuadratureMod.Func(I,Q,C)
endspec

Quad2_p0 = prove Quad2_conj in Quad2 options
"(use-resolution t) (use-paramodulation t)"
```

The main point of the above discussion was to show an example of the capability of reasoning about functions. While we chose to use Meataslang, this goal could be achieved in a number of different declarative languages. One of the possible candidates is Common Logic (CL), which recently has become a standard of ISO/IEC [31]. According to [31], CL is a first-order language that "permits 'higher order' constructions such as quantification over classes or relations while preserving a first-order model theory, and a semantics which allows theories to describe intensional things such as classes or properties." Thus CL is not a higher order language like Metaslang is, since its de facto expressivity is limited to first order; it is referred to sometimes as a "reified first-order logic" [32], [33]. While it is a known fact in mathematics that second-order logic cannot be reduced to first order, the practical implications of this fact on the use of CL in cognitive radio could be posed as an experimental question. In this paper, we do not delve into this issue but instead use a higher order language. The most important reason for this selection is Metaslang's ability to represent in a very compact and elegant way the composition of logical theories (specs), as was shown in this section.

VI. BEHAVIORAL ASPECTS

Metaslang is a functional language and, like other functional languages (e.g., Haskell), does not easily support the so-called *side effects*. In the functional programming paradigm, the result of application of a function always depends only on the input parameters. It cannot depend on the previous results of that or any other function(s). This is a serious limitation in case it is to be used to model *behavioral aspects* of systems, where the results of computation depend on the *state* of the system, like in *dynamical systems* [34]. This limitation long has been recognized and, as a result the concept of *monads* [35], has been used to address this problem. The main idea here is to use the monad concept to capture and pass some state information of a computation among functions. Even though monads can be useful for dealing with a limited set of states within a context of a functional program, their use to simulate memory elements and/or globally scoped variables, as is required in the context of software defined radio, introduces additional overhead that normally does not exist in imperative languages.

Functional languages also do not provide any obvious way to express and reason about time-dependent information. In logical terms, this refers to relating the truth of formulas at distinct time points. The simplest example of this kind of a need in the domain of cognitive radio is the need to reason about *delays* introduced by the various processing components. For instance, in the discrete time domain, we can say that a *unit delay* means that if the value of a signal for the time index t is $s(t) = a$, the value of another signal s' related to s by the *delay* operation, for the (next) time index $t+1$, will be exactly a, i.e., $s'(t+1) = a$. We can say that delay means if the sentence $s(t) = a$ is true, then the sentence $s'(t+1) = a$ is also true. Note that the intent here is to say that such a relationship between two time-dependent sentences should hold for all time instants $t \in T$. Thus we want to express such facts without explicitly referring to the time index.

Other examples of descriptions involving temporal information are: signal should be held at a given level *until* some event happens; signal should have a given value *after* an event happens; a given combination of values should *never* happen; a given event should happen *before* or *after* another event; a given relationship should *always* be satisfied (here "always" refers to time); the signal should have a given value *within* a given interval of time.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

Note that by chaining the various predicates indicated above (marked by italics), the descriptions of temporal relations can be arbitrarily complex. For instance, we can state that the signal *never* drops to 0 *before* it stays at 1 *within* a given time interval. We might also need to combine such sentences using the logical connectives, like logical OR, logical AND, negation, implication, or equivalence.

Moreover, the special predicates listed above may be logically related, e.g., a sentence involving some combination of the predicates may be logically equivalent to another sentence involving another combination. For instance, the sentence "the signal will *eventually* drop to zero" is equivalent to the sentence "*not always* the signal will be one." This suggests that one can define a logic that involves a collection of special (temporal) predicates. For this, a formal language should be defined and formal semantics should be established. The semantics would have to incorporate a specific model of time. Then inference can be carried out within such logic by the means of automatic inference engines, like in any similar formal system.

The logics that deal with time-dependent inference are called *temporal logics*. The most popular logic in this group is called *linear temporal logic* (LTL) [36]. As the name indicates, the structure of time in this logic is assumed to be linear (a linearly ordered set). LTL uses discrete time, i.e., it assumes that events in the system can happen only at the discrete moments in time. LTL is particularly useful for modeling behaviors of clock-driven systems, i.e., systems in which progression of events is controlled by a clock signal (either explicitly—as in case of hardware—or implicitly—as in case of a computer program running on a processor clocked with certain frequency). Thus LTL may be useful for the domain of cognitive radio.

The predicates in this logic include $X\varphi$ (φ must hold at the neXt time instant); $G\varphi$ (φ must hold on the entire subsequent time); $F\varphi$ (φ will hold eventually, in the future); $\psi U\varphi$ (φ holds at the current or a future position and ψ has to hold *until* that position; at that position ψ does not have to hold any more); $\psi R\varphi$ (φ is true until the first position in which ψ is true, or forever if such a position does not exist, i.e., ψ *releases* φ). Only three of these predicates are necessary. The rest can be expressed in terms of the other three.

LTL has found applications in the area of verification of distributed and reactive systems [37], where properties of systems such as reachability, safety, and deadlock are proven in a formal way. Two approaches used in this process are *deduction* (theorem proving) and *model checking*. Since deductive inference for this domain is undecidable, model checking is used more often to verify that a property holds in a given model [38]. But some use a combination of the two approaches; see, e.g., [39]. In this paper, we are not discussing the verification scenario but instead assume that verification would be done offline and only the resulting properties (in terms of an underlying, shared ontology) would be communicated among the participating radios. The automatic inference task would involve reasoning about composition specific to the communications domain.

To exemplify this kind of domain-specific reasoning, in this paper we focus only on one aspect of temporal reasoning: the one that is captured by the *X* operator of LTL. However, instead of implementing a temporal logic, we follow the approach that is typical in engineering. Towards this aim, we formalize the concept of *unit delay* and then prove properties of systems composed of components that include delays. In particular, *unit delay* allows us to model the temporal behavior of a system in terms of clock intervals, which removes explicit time values from the system description.

In the example specs below, we first define the function UnitDelay.Func whose domain and range is the type `Sample`. Then we state two commutativity axioms for this operation for functions of one and two arguments, respectively. It means that the composition of a function *f* with the *unit delay* is equal to the *unit delay* composed with *f*. Note that in this spec, we quantify over all functions with domain *Sample* and range *Sample*—the feature that requires that functions be covered by the language (as discussed in Section V). This specification of unit delay is not complete; it captures only the properties of this concept that are necessary to prove the conjectures shown later.

```
UnitDelaySpec = spec
  import Samples

  op       UnitDelay.Func: Sample -> Sample

  axiom UnitDelay_commutativity is
    fa(f:(Sample->Sample), x:Sample)
      UnitDelay.Func(f(x)) =
        f(UnitDelay.Func(x))

  axiom UnitDelay_commutativity2 is
    fa(f:(Sample*Sample->Sample),
         x:Sample, y:Sample)
      UnitDelay.Func(f(x,y)) =
        f(UnitDelay.Func(x),
          UnitDelay.Func(y))
endspec
```

In Section V, we specified the "ideal" Adder and Multiplier components, i.e., components without any delay. Now we specify delays introduced by these two components. As we can see in the specs below, we state that they introduce a one-step delay.

```
AdderDelay = spec
  import UnitDelaySpec

  op Adder.Func: Sample*Sample -> Sample
  def Adder.Func(x,y) =
    UnitDelay.Func(Sample.add(x, y))
```


Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

```
endspec

MultiplierDelay = spec
    import UnitDelaySpec

    op Multiplier.Func: Sample*Sample -> Sample
    def Multiplier.Func(x,y) =
       UnitDelay.Func(Sample.multiply(x,y))
endspec
```

The specification *MACSpec* shown below describes the behavior of a component that implements the multiply-add functionality with a two-clock-cycle delay for each input set (it is a typical behavior for a MAC unit implemented in field-programmable gate array). The second spec in the example—*CompositeMACSpec*—describes a module being a composition of three other modules—*Adder*, *Multiplier*, and *UnitDelay*. That spec contains a conjecture (*CompositeMAC_conj*) that is used by the theorem prover [28] to prove that functionalities of *MACSpec* and *CompositeMACSpec* are equivalent.

```
MACSpec = spec
    import UnitDelaySpec

    op MAC.Func: Sample*Sample*Sample -> Sample
    def MAC.Func(m1, m2, a) =
       UnitDelay.Func(
          Sample.add(
             UnitDelay.Func(
                Sample.multiply(m1, m2)),
                UnitDelay.Func(a)))
endspec

CompositeMACSpec = spec
    import AdderDelay
    import MultiplierDelay
    import MACSpec

op CompositeMAC.Func:
    Sample*Sample*Sample -> Sample
def CompositeMAC.Func(m1, m2, a) =
    Adder.Func(Multiplier.Func(m1,m2),
               UnitDelay.Func(a))

    conjecture CompositeMAC_conj is
       fa(m1:Sample, m2:Sample, a:Sample)
          CompositeMAC.Func(m1, m2, a) =
             MAC.Func(m1, m2, a)
endspec

p0 = prove CompositeMAC_conj in CompositeMACSpec
options "(use-resolution t) (use-paramodulation t)"
```

The above example shows two things. First, by using an abstract specification with a set of axioms, we are able to express time dependencies between different components of the system without introducing time explicitly. Secondly, it also proves that one can use automatic inference engines to reason about abstract specifications, i.e., specifications of concepts, like classes. However, Metaslang cannot be used to reason about individuals of such classes. For this purpose, some additional capabilities are needed. One of the efforts in this direction is the development of the system called Accord [40]. Accord is an extension of Specware currently being developed by the Kesterel Institute. The motivation for extending Specware was the observation that the modeling of state becomes very important as the system architectures move towards distributed and embedded systems. Additional motivations were increased computational efficiency that can sometimes be achieved through imperative programming as well as more direct connection to common programming languages [40].

In Accord, the behavior is encapsulated in a *module*, which is an extension of Metaslang's spec. Accord is backwards compatible with Metaslang, and so every Metaslang spec can be directly embedded into an Accord module. Accord is built on ideas of evolving specifications (e-specs). Since a state can be seen as a data structure and a state transition as a finite change to that structure, the states and their transitions can be seen as specs and conditional spec morphisms, respectively. In Accord, the behavioral aspect of a module is encoded in one or more procedures *(proc)*. Each proc contains *modes*, which encode *abstract states*. For each procedure, there is an implicit entry mode (initial state) and an implicit exit mode (final state). *Steps* specify *abstract transitions* and are usually guarded by logic expressions. Accord implicitly defines steps to the initial mode and from the final mode to the exit. Other imperative elements introduced in Accord include global *variables* and *signals* (exceptions). Accord is still in a very early stage of development at this point. However, when it matures, it might become an attractive option for implementing inference about declarative programs that capture some aspects that are normally best addressed in imperative programming.

To close this section, we should mention the efforts to capture state information in languages like UML[8] or SDL. However, we do not devote much attention to these languages since they do not have formal semantics. Consequently, formal inference cannot be carried out over the specifications of systems expressed in these languages.

VII. LANGUAGE STANDARDIZATION EFFORTS FOR THE COGNITIVE RADIO DOMAIN

A call for the development of a language for wireless systems was presented in [41], emphasizing the need to

[8] http://www.omg.org/spec/UML/2.1.2/Superstructure/PDF/change-barred/.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

exchange information among various stakeholders, such as network operators, enterprise networks, system and component vendors, software vendors, regulatory agencies, and end users, and to provide means for the end-to-end reconfigurability. The paper states:

> In an end-to-end reconfigurable system, there are various tasks that require interactions with different components in the network, and that require such components to communicate with each other. Example tasks relate to network configuration, device reconfiguration (involving download of applications/firmware), and network operation (provisioning new services, upgrades, roll out of new AIS's, etc.). Depending on the nature of the specific task, network elements need to be queried, controlled, modified and managed, at varying levels of detail.

To achieve such interoperability goals, a standardized language that allows for exchange of information is needed. Two nodes can efficiently exchange information if they have a common language.

The need for a standard declarative language for the domain of CR has been recognized by a number of organizations. Below we mention three organizations and their efforts: Functional Description Language (FDL) from the End-To-End-Reconfigurability (E^2R) organization, Modeling Language for Mobility (MLM) from the Software Defined Radio Forum (SDRF), and Policy Language and Policy Architectures for Managing Cognitive Radio for Dynamic Spectrum Access Applications from SCC 41 1900.5, an IEEE standards organization.

FDL [42] is envisioned to be the language software defined radios would use to communicate functional description data among themselves. FDL is based on XML and defines the functionality of the system as a collection of processes linked together by communication channels. The descriptions in that language are platform-independent and are interpreted by the Configuration Control Module (CCM) in terms of specific software and/or hardware solutions available locally. It is envisioned that when a CCM encounters a description that is not locally available, the node would be able to access a centralized database and download the missing software module.

FDL enables SDR software modules to be defined as hierarchical flow of signals between processes communicating via one-to-one and one-to-many channels. The top level of the functional description is the *algorithm* element, which might include one or more processes. It also defines how those processes communicate by defining sets of input and output ports per process and then linking them to the channels. FDL supports hierarchy so processes can contain subprocesses and subprocess ports can communicate with ports of the higher level. That provides the potential for describing the functionality at an arbitrary level of granularity enabling structure-based composition similar to what was demonstrated in our experiments with OWL and BaseVISor rules described in Section IV. FDL's capabilities go beyond simple composition, as it incorporates some time constraints such as time baseline (clock), latency, time deadlines (max delay), bandwidth, etc.

However FDL (as it stands now) is not a formal system. FDL is defined in terms of XML, where the domain-specific XML tags need to be interpreted by the programmers. Since FDL does not have a formal semantics, it cannot be used by an inference engine for automatic derivation of facts about FDL descriptions.

The SDRF has a working group (MLM Working Group) whose charge is to develop the MLM [43]. The group has developed a number of use cases that explicate and justify the need for a formal language in which various aspects of the life cycle of communication nodes, including mobile, could be described and shared among the nodes and the stakeholders. The next step after the use cases will be the development of ontologies relevant to the communication domain. The developed ontologies will be captured in various formalisms. The classes and the properties will be formalized in OWL, and rules will supplement the OWL descriptions. Other formalisms, including other formal languages and mathematics, will also be considered.

The SDRF approach to the development of its language is somewhat different from the E^2R effort in the sense that from the very beginning, the stress is on expressing everything in a formal declarative language with formal semantics so that automatic inference can be carried out by inference engines.

The third effort that is directly relevant to the CR domain is the effort by SCC 41 to develop a standard for a collection of languages for the dynamic spectrum access domain. At the time of this writing, SSC 41 has created the IEEE 1900.5 Study Group on Policy Language for Managing Cognitive Radio and Dynamic Spectrum Access Applications. The charge of this group is to define a family of interrelated languages that would serve the needs of the domain described above.

While it is not possible to predict, with a high degree of confidence, whether these three efforts will converge and will produce a unified language and a unified set of ontologies for the CR domain, it is encouraging that the three organizations behind these efforts have working relationships and have already started interacting with the intent of unification and the integration of both the ontologies and the languages.

VIII. CONCLUSION

The main motivation for this paper was to identify various formal, declarative languages and their expressivity features that may be relevant and useful to the implementation of the concept of cognitive radio. A running

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

Table 1 Summary of Languages and Features

Language	Expressivity	Relevance to CR Domain Advantages/Limitations
OWL	Classes, individuals, binary relations	Composition can be described only at the individuals level
OWL plus Rules	Composition of relations	Compositions can be described at the class level
CL	Functions, limited quantification	Description of functionality; limited function inference capability
Metaslang	Functions, full quantification, colimits	Equivalence of functionality; refinement; abstract specifications; composition of specifications
LTL	Temporal aspects of the system	Temporal aspects (time delays, etc.)
Accord	Behavioral descriptions, system's state	Dynamic behaviors (state and temporal aspects)

example was selected, and then various features were discussed in a progressive manner. The languages discussed in this paper and their expressivity features are summarized in Table 1.

First, we discussed the useful role ontologies expressed in a formal language can play in cognitive radio. Then we showed that some of the aspects that may be needed for cognitive radio are missing in OWL, and that a more expressive language would be needed to fill the need. In particular, we showed that rules complement OWL and allow us to describe architectures of types of components. This was followed by the discussion of negation, a logical concept that seems to be very simple in principle but is very difficult to implement within one language that includes both OWL and rules. The main issue is that the negation in OWL is the logical negation, while in rules it is negation as failure. Since this problem has not been solved as yet, the best we could do in this paper is to provide the background information and then indicate some directions of possible partial solutions. The next step was to show that reasoning about the functionality of various components requires functions to be first-class objects in the formal language. We showed an example of proof that structurally different components can be functionally equivalent. We discussed the problems of representing and reasoning about behavioral aspects of components. The issue is that most formal languages are functional, i.e., each function's value depends only on the inputs, while representation of behavior requires memory—results of previous computations—and dealing with temporal aspects. Dynamical systems are one class of behavioral systems. We provided a brief overview of the use of linear temporal logic to represent behaviors. Then we focused on one aspect of temporal behavior—the delay. We showed that automatic inference about behavioral aspects can be carried out using inference engines.

The range of issues related to the use of formal languages for cognitive radio is probably unbounded. Consequently, we were not able to cover all of them, and it is not even possible to provide a complete list of the issues that have not been covered. Here we mention only some of them.

1) OWL can express only binary predicates (properties). However, in practice, we need to use n-ary predicates, too. We did not discuss how to deal with these.
2) Although we discussed the distinction between closed-world reasoning and open-world reasoning, we have not discussed the distinction between closed-world querying of open-world knowledge bases versus closed-world reasoning.
3) We have not discussed the issue of expressing integrity constrains. This relates to the distinction between constraints on the modeled world (e.g., as expressed in OWL) versus constraints on the knowledge base.
4) We only barely touched upon the issue of nonmonotonic reasoning.
5) We have not discussed query languages.
6) We have not discussed in any detail a very important aspect—the computational complexity of inference within particular languages. In particular, we have not discussed the issue of running an inference engine on limited computing resources of a CR.

In the last section of this paper, we provided references to three ongoing efforts to the standardization of languages relevant to the cognitive radio domain. We believe these efforts will address the problems we identified in this paper and identify solutions to some of them. ■

Acknowledgment

Interactions with P. Marshall, D. Hillman and G. Denker have provided valuable input to this work. Moreover, collaboration with the Software Defined Radio

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

Forum, especially with M. Cummings and B. Fette, provided valuable information on the understanding of the applicability of formal declarative languages in the cognitive radio domain. The authors would like to express their deep gratitude to the anonymous reviewers who have provided many very insightful comments that have improved the completeness, consistency, and readability of the presentation.

REFERENCES

[1] J. Mitola, III, "Cognitive radio: An integrated agent architecture for software defined radi," Ph.D. dissertation, Royal Institute of Technology (KTH), Stockholm, Sweden, 2000.

[2] J. Mitola, III and G. Q. Maguire, Jr., "Cognitive radio: Making software radios more personal," *IEEE Wireless Commun.*, vol. 6, pp. 13–18, Aug. 1999.

[3] Software Defined Radio Forum, "Cognitive radio definitions," Working Doc. SDRF-06-R-0011-V1.0.0, Nov. 8, 2007.

[4] M. M. Kokar, D. Brady, and K. Baclawski, "Chapter 13: Roles of ontologies in cognitive radios," in *Cognitive Radio Technology*, B. Fette, Ed. Oxford, U.K.: Newnes, 2006, pp. 401–433.

[5] T. W. Pratt and M. V. Zelkowitz, *Programming Languages: Design and Implementation*, 3rd ed. Englewood Cliffs, NJ: Prentice-Hall, 1996.

[6] J. W. Lloyd, "Practical advantages of declarative programming," in *Proc. 1994 Joint Conf. Declarative Program (GULP-PRODE'94)*, Peñiscola, Spain, 1994.

[7] P. Haase, F. van Harmelen, Z. Huang, H. Stuckenschmidt, and Y. Sure, "A framework for handling inconsistency in changing ontologies," in *Proc. 4th Int. Semantic Web Conf.*, 2005. [Online]. Available: http://www.cs.vu.nl/~frankh/abstracts/ISWC05.html

[8] H. Enderton, *A Mathematical Introduction to Logic*, 2nd ed. Boston, MA: Academic, 2001.

[9] R. Corazzon. (2008). *Ontology: A resource guide for philosophers*. [Online]. Available: http://www.formalontology.it/

[10] T. Gruber, *What is an ontology?* [Online]. Available: http://www-ksl.stanford.edu/kst/what-is-an-ontology.html

[11] J. G. Proakis, *Digital Communications*, 3rd ed. New York: McGraw-Hill, 1995.

[12] D. Preuveneers and Y. Berbers, "Automated context-driven composition of pervasive services to alleviate non-functional concerns," *Int. J. Comput. Inf. Sci.*, vol. 3, no. 2, pp. 19–28, Aug. 2005.

[13] L. Lechowicz and M. Kokar, "Achieving dynamic interoperability of communication: Transfer of ontology and rules between nodes," in *Proc. Software Defined Radio Tech. Conf. (SDR'06)*, 2006.

[14] A. Colmerauer and P. Roussel, "The birth of prolog," in *Proc. 2nd ACM Conf. History Program. Lang. (SIGPLAN)*, 1992, pp. 37–52.

[15] C. Matheus, K. Baclawski, and M. M. Kokar, "BaseVISor: A triples-based inference engine outfitted to process RuleML and R-entailment rules," in *Proc. 2nd Int. Conf. Rules Rule Lang. Semantic Web*, Athens, GA, Nov. 2006.

[16] H. ter Horst, "Combining RDF and part of OWL with rules: Semantics, decidability, complexity," in *Proc. 4th Int. Semantic Web Conf. (ISWC 2005)*, 2005, vol. 3729, pp. 668–684.

[17] J. D. Poston, W. D. Horne, M. G. Taylor, and F. Z. Zhu, "Ontology-based reasoning for context-aware radios: Insights and findings from prototype development," in *Proc. 1st IEEE Int. Symp. New Frontiers Dyn. Spectrum Access Netw. (DySPAN)*, 2005, pp. 634–637.

[18] A. Ginsberg, J. D. Poston, and W. D. Horne, "Experiments in cognitive radio and dynamic spectrum access using an ontology-rule hybrid architecture," in *Proc. 2nd Int. Conf. Rules Rule Markup Lang. Semantic Web (RuleML)*, Athens, GA, 2006. [Online]. Available: http://2006.ruleml.org/group3.html#3

[19] A. Ginsberg, W. Horne, and J. Poston, "Community-based cognitive radio architecture: Policy-compliant innovation via the semantic web," in *Proc. 2nd IEEE Int. Symp. New Frontiers Dyn. Spectrum Access Netw. (DySPAN)*, 2007.

[20] M. Kifer, *Negation in knowledge representation*, unpublished notes, Jun. 9, 2007

[21] B. Motik, I. Horrocks, R. Rosati, and U. Sattler, "Can OWL and logic programming live together happily ever after?" in *Proc. 5th Int. Semantic Web Conf.*, 2006, vol. 4273, pp. 501–514.

[22] L. Lechowicz and M. M. Kokar, "Composition, equivalence and interoperability: An example," in *Proc. Software Defined Radio Tech. Conf. (SDR'07)*, 2007.

[23] S. Shapiro, *Foundations Without Foundationalism: A Case for Second-Order Logic*. Oxford, U.K.: Oxford Univ. Press, 2000.

[24] C. E. Brown, *Automated Reasoning in Higher-Order Logic: Set Comprehension and Extensionality in Church's Type Theory*. Oxford, U.K.: College Publications, 2007.

[25] T. F. Melham, *Higher-Order Logic and Hardware Verification*. Cambridge, U.K.: Cambridge Univ. Press, 1993.

[26] J. McDonald and J. Anton, "SPECWARE—Producing software correct by construction," Kestrel Inst., Tech. Rep. KES. U.01.3., Mar. 2001.

[27] D. R. Smith, "Composition by colimit and formal software development," in *Algebra, Meaning, and Computation: A Festschrift in Honor of Prof. Joseph Goguen*, K. Futatsugi, J.-P. Jouannaud, and J. Meseguer, Eds. Berlin, Germany: Springer-Verlag, 2006, vol. 4060, pp. 317–332.

[28] M. E. Stickel, R. J. Waldinger, M. Lowry, T. Pressburger, and I. Underwood, "Deductive composition of astronomical software from subroutine libraries," in *Proc. 12th Int. Conf. Autom. Deduct. (CADE-12)*, Nancy, France, Jun. 1994, pp. 341–355.

[29] T. Nipkow, L. C. Paulson, and M. Wenzel, *Isabelle/HOL: A Proof Assistant for Higher-Order Logic*. New York: Springer, 2002, vol. 2283.

[30] D. Pavlovic and D. R. Smith, "Software development by refinement," in *Proc. 10th Anniv. Colloq. Formal Methods Crossroads: From Panacea to Foundational Support (UNU/IIST)*, 2003.

[31] *Information Technology—Common Logic (CL): A Framework for a Family of Logic-Based Languages*, ISO/IEC Standard 24707, 2007.

[32] C. Bock, M. Gruninger, D. Libes, J. Lubell, and E. Subrahmanian, "Evaluating reasoning systems," National Inst. of Standards and Technology, Tech. Rep. NISTIR 7310, 2006.

[33] P. Hayes and C. Menzel, "Simple common logic," in *Proc. Workshop Rule Lang. Interop. (W3C)*, Washington, D.C., 2005. [Online]. Available: http://www.w3.org/2004/12/rules-ws/paper/103/

[34] L. Padulo and M. A. Arbib, *System Theory: A Unified State-Space Approach to Continuous and Discrete Systems*. Philadelphia, PA: Saunders, 1974.

[35] P. Wadler, "Comprehending monads," in *Proc. 1990 ACM Conf. LISP Funct. Program.*, Nice, 1990.

[36] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems: Specification*. Berlin, Germany: Springer-Verlag, 1991.

[37] Z. Manna and A. Pnueli, *Temporal Verification of Reactive Systems: Safety*. Berlin, Germany: Springer-Verlag, 1995.

[38] G. J. Holzmann, *The Spin Model Checker: Primer and Reference Manual*. Reading, MA: Addison-Wesley, 2004.

[39] N. S. Bjørner, A. Browne, M. A. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma, and T. E. Uribe, "Verifying temporal properties of reactive systems: A STeP tutorial," *Formal Methods Syst. Design*, vol. 16, no. 3, pp. 227–270, 2000.

[40] J. McDonald and D. R. Smith, *Accord Language Manual Version 4.2*. Palo Alto, CA: Kestrel Institute, 2007.

[41] M. Cummings and P. A. Subrahmanyam, "Perspectives of a metalanguage for configurable wireless systems," in *Proc. Software Defined Radio Tech. Conf. (SDR'04)*, 2004.

[42] C. Dolwin, R. Burgess, and B. Steinke, "Power efficient and real-time configuration of resources in an end-to-end reconfigurable system," in *Proc. Software Defined Radio Technical Conf. (SDR'05)*, Anaheim, CA, 2005.

[43] B. Fette, M. M. Kokar, and M. Cummings, "Next-generation design issues in communications," *Portable Design Mag.*, no. 3, pp. 20–24, 2008.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.

ABOUT THE AUTHORS

Mieczyslaw M. Kokar (Senior Member, IEEE) received the M.S. and Ph.D. degrees in computer and system engineering from Wroclaw University of Technology, Wroclaw, Poland, in 1969 and 1973, respectively.

He is an Associate Professor in the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA. His technical research interests include information fusion, ontology-based information processing, software defined radio, self-controlling software, modeling languages, and formal methods. He teaches various graduate courses in software engineering, formal methods, and artificial intelligence. He is the author of numerous publications in his areas of interest. He is a Cochair of the Software Defined Radio Forum Work Group on Modeling Language for Mobility.

Dr. Kokar is a member of ACM. He is a member of the IEEE 1900.5 Working Group on Policy Language and Policy Architectures for Managing Cognitive Radio for Dynamic Spectrum Access Applications.

Leszek Lechowicz (Member, IEEE) received the M.S. degree in electrical engineering and in computer science from Gdansk University of Technology, Poland, in 1994 and 1996, respectively. He is a doctoral candidate in the Department of Electrical and Computer Engineering, Northeastern University, Boston, MA.

His research interests include software defined radio, self-controlling software, and the application of formal methods in software engineering. He is a System Architect with Asterion Inc., Marlboro, MA. His previous professional experience includes variety of technical positions with automated test equipment and networking companies.

Mr. Lechowicz is a member of the IEEE Computer Society and ACM.

Authorized licensed use limited to: Northeastern University. Downloaded on July 26, 2009 at 11:49 from IEEE Xplore. Restrictions apply.