ACHIEVING SELF-AWARENESS OF SDR NODES THROUGH ONTOLOGY-BASED REASONING AND REFLECTION

J. Wang (Department of Electrical and Computer Engineering, Northeastern University, Boston, jiawang@ece.neu.edu); M. M. Kokar (Department of Electrical and Computer Engineering, Northeastern University, Boston, kokar@coe.neu.edu); K. Baclawski (College of Computer and Information Science, Northeastern University, Boston kenb@ccs.neu.edu); D. Brady (Department of Electrical and Computer Engineering, Northeastern University, Boston, brady@ece.neu.edu).

ABSTRACT

Software defined radios add a degree of flexibility and versatility that is not possible with hardware based communication. However, software based on ad hoc data structures or database schemas is limited to the features explicitly supported by the software. Ontology-Based Radio (OBR) uses ontologies to add inferencing and reasoning capabilities which make radios *self-aware,* i.e., understand their own capabilities and the capabilities of other nodes. One important application is for radios to query each other and to interoperate in ways that are not explicitly provided by the software. We show how ontologies and rules, in combination with Java reflection, can be used to implement self-awareness and interoperability. We illustrate how such radios would interoperate by giving an example in which radios negotiate the length and structure of equalizer training sequences.

1. INTRODUCTION

Ontology-Based Radio (OBR) is a mechanism for software defined communication nodes to *interoperate*, i.e., understand other nodes and modify the processing of packets during a communication session both at the source and the destination. This mechanism uses an ontology to specify not only the structure of communication packets but also the processing of those packets according to the communication protocol. Nodes have the ability to query both their own capabilities and the capabilities of other nodes. The use of ontologies adds flexibility, inferencing and reasoning features that are not available with ad hoc data structures or database schemas.

In this paper we demonstrate the concept of OBR using a prototype in which two-way communication between two nodes is implemented using a bi-directional acoustic link. The query mechanism is based on the Web Ontology Language (OWL) [1]. However, OBR presents a number of challenges not faced by other ontology-based applications. (1) Real-time processing demands higher performance for inference and reasoning than an interactive application. (2) The "knowledge base" of a node includes state information that is continually varying, in contrast with the static knowledge bases required by most ontology-based reasoning systems. (3) The "facts" are not stored in a knowledge base but rather are embedded in the software that implements the communication protocol.

We solve the first two problems by using a Prolog-based OWL reasoner which provides not only very fast reasoning, but also considerably smaller memory requirements than other OWL based theorem provers. Furthermore, it allows the radio to update intermediate derivations dynamically in sync with an incoming stream of facts, as required by the second problem above. The third problem is solved by using a feature of programming languages called *reflection*. This feature is built into languages such as Java and C++, and it can be added to languages such as C. Reflection provides a powerful mechanism whereby a running program can observe its own data structure types and the values of particular variables. The advantage of reflection over an ad hoc approach is that the code implementing the communication protocol need not have any monitoring or retrieval code for providing this feature.

In our prototype we show how the communication between two OBR nodes can be controlled to improve communication performance. In the example presented in this paper, two OBR nodes tailor the training sequence length in each packet, according to the channel dynamics and noise level. In static conditions, this permits a 60% reduction in the packet overhead (training sequence length), while equalizer training is still improved in situations of high noise or channel dynamics. Most importantly, these examples demonstrate how ontology based reasoning can be used to achieve these performance gains. We also demonstrate a chain of interactions between two OBR nodes in the process of negotiating the protocol parameters. The real goal of these experiments was to show that opportunities for negotiating communication protocols do exist and can be achieved using the OBR concept.

This paper presents an update on our progress in the efforts to fully implement the concept of Ontology-Based Radios as discussed in [2]. We have implemented a number of features that put us closer to the ultimate goal – radios that have sufficient self-awareness for them to negotiate their communication protocols. In particular, we extended our ontology by adding a number of rules that specify the behavior of the OBR nodes. The behaviors are controlled by a reasoner that derives behavior that satisfies a top-level *goal.* The goal is specified in terms of communication performance, and behavior is derived by means of rules.

The architecture of our OBR consists of a collection of producer-consumer queues that are used for particular functions, such as sending and receiving packets, checking communication performance and logical inference. This architecture is presented in Section 2.

To implement such a concurrent architecture we needed a reasoner that is appropriate for this kind of architecture. The JTP theorem prover [3] that we used to demonstrate the proof-of-concept, worked very well for offline reasoning, but was not suitable for our requirements. A discussion of our inference engine is provided in Section 3.

The reasoning process makes use of both the ontology and a set of rules. The ontology defines the basic terms in which the nodes communicate – classes and properties. The rules specify, in declarative form, how to react to particular situations. Rules used in our demonstration are discussed in Section 4. Rules use both other rules and ground facts to derive conclusions. In the context of SDR, static facts that are known *a priori* are stored in a fact base. But some of the ground facts are only available as the radio is operating. These ground facts change over time, sometimes at a very high rate. These values are accessed through the mechanism called *reflection,* described in Section 5.

In Section 6 we discuss how the capability of interoperability can be used to improve communication. In particular, we discuss how two OBR nodes negotiate equalizer training sequences. Finally, in Section 7 we provide our conclusions and future work. In particular, we focus on providing interoperability of OBR nodes that use different communication protocols (waveforms).

2. OBR Architecture

The OBR architecture includes five services which are implemented using Java threads. Figure 1 shows these services: DSS (Data Source Service, which generates data), DO (Data Out Service, which sends out data), DI (Data In Service, which receives data), MS (Monitor Service, which monitors the received data and responses) and DS (Data Sink Service, which consumes data).

The node labeled SDR in the architecture represents the

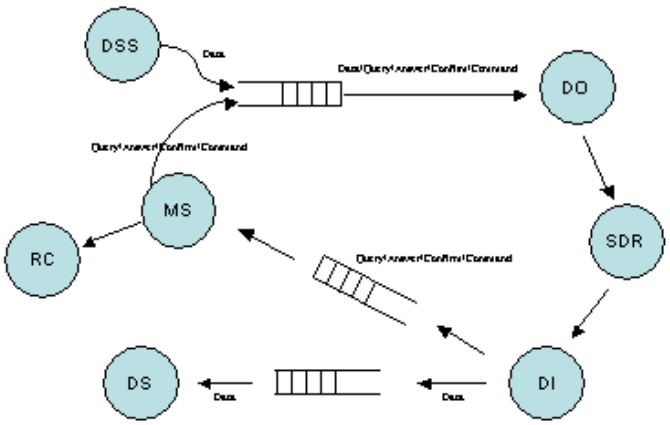

Figure 1: The Architecture of OBR

software defined radio. In the prototype system this is represented by a Java class which sends and receives messages. This class includes all the communication functions, such as compressing, filtering, modulation and equalization. The RC node is the reasoning component, which is implemented using a version of Prolog called Kernel Prolog. The Monitor Service uses RC to do reasoning. The five services communicate with one another by using producer-consumer queues.

Five types of message are used in communication: *data, confirm, query, answer* and *command*. A "data message" is a message whose content is a sequence of characters, which was generated by the Data Source Service. A "confirm message" is a message whose content is either "Continue", or "CommandReq." The Continue message is used when performance is acceptable, so that no changes are needed. The CommandReq message is used when performance is not adequate, so that the OBR should adjust its communication protocol. A "query message" queries both the channel condition and the structure of the communication node. An "answer message" carries the answer to a corresponding query. A "command message" is used to control the communication node, such as a command to change the communication protocol. Figure 2 shows the five types of message.

3. Inference in an OBR

OBR inference is provided by RC, the reasoning component, which is a goal-driven system. The facts are loaded into Prolog dynamically and used to achieve a goal, such as to generate a command that changes the communication protocol used by the communication nodes.

After reasoning, inferred facts are discarded. This prevents the knowledge base from increasing in size over time.

The RC was implemented using Kernel Prolog [4], a lightweight Java based Prolog interpreter. The axioms of

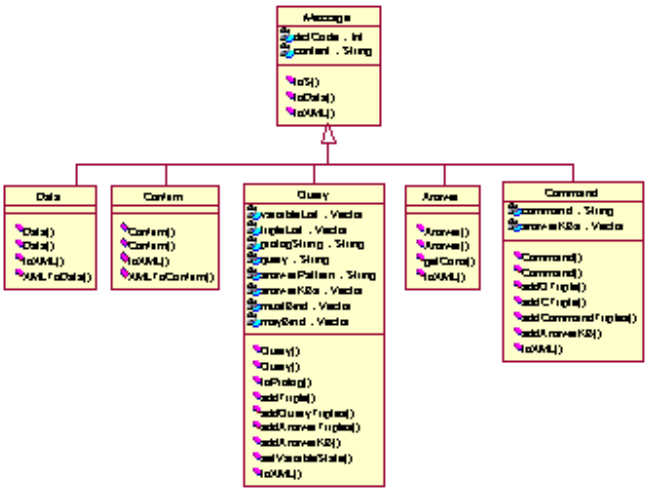

Figure 2. Message types

OWL have been written in Prolog, and our ontology for SDR, which was previously written in OWL, have also been translated into Prolog facts. We have also developed a set of rules to react to particular situations, all of which are asserted into the knowledge base of Kernel Prolog when the RC begins functioning.

An example of an OWL rule is the subPropertyRule. This rule states that if a property is a subproperty of another one, then any fact using the subproperty implies that the corresponding fact using the superproperty also holds. For example, in the OBR ontology, the `hasTransmitterPacket` and the `hasReceiverPacket` properties are subproperties of `hasPacket`. Consequently, a query for the packet for an object will return the packet even though a packet is always either a transmitted or a received packet. Although this is a very simple example, it illustrates the general principle of inference. Most uses of RC involve much deeper chains of inference. All the queries are processed by a Prolog interpreter so that all inferences will be properly performed. OWL has many built-in rules such as subClass, subProperty, disjointWith, unionOf. These rules represent the axioms of the OWL language.

4. Rules

The set of rules for communication processing includes rules to check performance of communication, and to generate queries and commands according to different channel conditions.

When data is received, the Monitor Service uses the RC to check performance. The rule to check performance includes, among others, a query for the equalizer error and a comparison between the equalizer error and predefined upper and lower thresholds. If the equalizer error is between the thresholds, performance is considered to be acceptable and a "confirm" message with content "continue" will be sent back. Otherwise, a confirm message with content "CommandReq" will be sent back, which requests a command to change the structure of the communication protocol.

The following is an example of one of the rules used for checking performance:

```
checkPerformance(X) :-
equalizerError(E),
pv(obr8upperPerformanceThreshold,
object8MonitorServiceDispatch, UPt),
pv(obr8lowerPerformanceThreshold,
object8MonitorServiceDispatch, LPt),
compare('>', E, LPt),
compare('<', E, UPt),
assign('Continue ', X).
```

The rule begins with the conclusion or *goal*, in this case whether to check performance. The ":-" symbol separates the goal from the hypothesis required to satisfy the goal. In this case, if the extracted equalizer error is larger than the predefined lowerPerformanceThreshold (`LPt`) and it is smaller than the predefined upperPerformanceThreshold (`UPt`), then a "Continue" message will be returned.

The command generation rules will generate commands according to the channel condition and the protocol currently being used. So a query for the channel condition and information about the current protocol will be issued first. The answers to those queries are facts which will be loaded into Prolog temporarily. After the commands are generated, the facts are discarded.

The following are examples of some command generation rules:

```
command(C) :- prevPacketExist,
queryNode(Answer), first(Answer,
VPrevEL), rest(Answer, Left1),
first(Left1, VRmsDelay), rest(Left1,
Left2), first(Left2, VER),
rest(Left2, Left3), first(Left3,
VPrevTL), dif(VRmsDelay, VPrevEL,
Dis), halfValue(VPrevEL, Half),
compare('<', Dis, Half), pv(obr8pf,
```

```
object8MonitorServiceDispatch, PF),
compare('<', VER, PF), pv(obr8value,
object8MonitorServiceDispatch,
Value), decrease(VPrevTL, Value,
Ltemp), compute(':', Ltemp, 1, TL),
trainingDataCommand(TL, C).

queryNode(Answer) :-
queryOtherNode([[VPrevEL, VRmsDelay,
VER, VPrevTL], [pv, rdf8type, VSDR,
obr8SDR], [pv, obr8hasPrevPacketInfo,
VSDR, VPPI], [pv, obr8hasPrevEL,
VPPI, VPrevEL], [pv, obr8hasPackets,
VSDR, VPacket], [pv,
obr8hasLastMultipath, VPacket,
VLastMultipath], [pv, obr8rmsdelay,
VLastMultipath, VRmsDelay], [pv,
obr8hasEqualizer, VPacket,
VEqualizer], [pv, obr8equalizerError,
VEqualizer, VER], [pv, obr8hasPrevTL,
VPPI, VPrevTL]], Answer).

trainingDataCommand(TL, Command) :-
initPath(InitPath),
pvexeString(obr8newTrainingData,
variable8SDRObject, TL,
TrainingDataCommand),
concat(InitPath, TrainingDataCommand,
Command).
```

These rules say that if the difference between the `rmsDelay` and the previous equalizer coefficient vector length is not very large (smaller than half of the previous equalizer length), and if the equalizer error is smaller than a predefined performance threshold, then the length of the training data can be decreased by a predefined value.

The rules to check performance and generate commands are written in advance and loaded into Prolog interpreter when the communication node starts functioning.

5. Reflection

When a query or a command is received, the Monitor Service asks the RC to answer the query or execute the command. Information about the current communication parameters are obtained by using reflection.

Java reflection is a built-in feature of Java. It allows a Java program to examine itself introspectively during run time. In order to use Java reflection with Kernel Prolog, a new built-in class called "pvref" was written and added to Kernel Prolog's built-in collection.

The following shows how to link Java reflection with rule engine queries. First we define patterns and link them with Jave reflection. A *pattern* is an expression having the form: (property slot1 slot2 ...), where a slot can be filled with either an object or a variable. As a result of our modification to Kernel Prolog, certain patterns are recognized by the Prolog interpreter as being reflective. For example, a pattern of the form `(function object Variable)`, where `function` is Java method name, `object` is a Java object and `Variable` is a variable, is evaluated by reflectively computing the value of the Java expression `object.function()` and setting `Variable` to the result.

In our experiment, a query is composed with a sequence of patterns with variables. When each pattern in the query is satisfied by Prolog, all the variables in the query will be assigned to values. In those variables, some of them are specified as "must bind" variables. Similarly, a command is composed of a sequence of patterns with variables, just like a query, except that at least one pattern is an executable pattern. For example, a pattern of the form `(function, object, value)` is evaluated by reflectively executing the Java expression `object.function(value)`. Commands are distinguished from queries by using a different built-in function. Queries use `pvref`, while commands use `pvexe`.

6. An Example: Negotiation of Equalizer Training Sequences

In the following example, we will show how the interaction between two OBR nodes can be controlled to improve communication performance.

6.1. Packet Structure

We first describe the structure of packets transmitted by the SDR radio component. In our experiment, each transmitted packet includes a header, training data and ordinary data. We use DSSS as the alphabet for the header and for the training data. For example, DSSS(2, 7) is an instance of class DSSS, the "2" meaning that each chip is a BPSK chip, and the "7" meaning that the length of the bit vector is 2^7-1 = 127. The bit vector is generated as an m-sequence. So using alphabet DSSS(2, 7), each header or training symbol will be mapped into 127 chips, where each chip is a BPSK chip.

In summary, each transmitted packet includes: a header symbol (127 chips); several training data symbols (the number of training data symbols depending on the channel characteristics, with each symbol represented by 127 chips if we use the DSSS(2, 7) alphabet, or 31 chips if we use the

DSSS(2, 5) alphabet); and a sequence of data symbols, each being a BPSK symbol.

The RLS algorithm is used to calculate the multipath structure of the fading channel, and an equalizer based on the RLS algorithm is used by the receiver to process the received data [5]. The equalizer coefficient vector is divided into a feedback coefficient vector and a feed forward coefficient vector. The length of the feedback coefficient vector of our equalizer is one third of the length of the equalizer.

6.2. Establishing the communication channel

In this example, negotiation of the length of the training data was accomplished by six transmissions. Suppose we initialize one transmission node (call it node A) as the transmitter, another node (node B) as the receiver. Then after node A sends data to node B, node B will check performance. If performance is satisfactory, then node B return a "confirm" message with content "Continue", and node A will continue to send data to B. If the performance is not satisfactory, a "confirm" message with content "CommandReq" (request a command from the other node) will be returned. Node A will then generate a command to change the communication protocol. The command generation rules first send a query from node A to node B requesting the channel condition and the current protocol parameters. When node B receives this query, it will infer the answer and send the answer back to node A. When node A receives the answer, it will generate the command. After the command is generated, it is sent to node B, and executed on node A, thus changing the protocol at node A. When node B receives the command, it will execute the command, thus changing the protocol at node B. A "confirm" message with content "Continue" is then sent to node A.

6.3. Example: Negotiation of equalizer training length

In this example, we will show that by negotiating the length of the training data according to the channel dynamics and noise level, we can reduce the packet overhead, while equalizer training is improved in situations of high noise or channel dynamics.

To initialize the experiment, we set the transmitted packet to use a DSSS(2, 7) symbol (mapped to 127 chips) as the header, and a sequence of 12 DSSS(2, 5) symbols (31 chips for each symbol) as the training data.

The predefined `upperPerformanceThreshold` and `lowerPerformanceThreshold` of the performance checking rules are set to the same value to make sure a command request will be issued each time that the performance is checked. The command rules are defined in the following way:

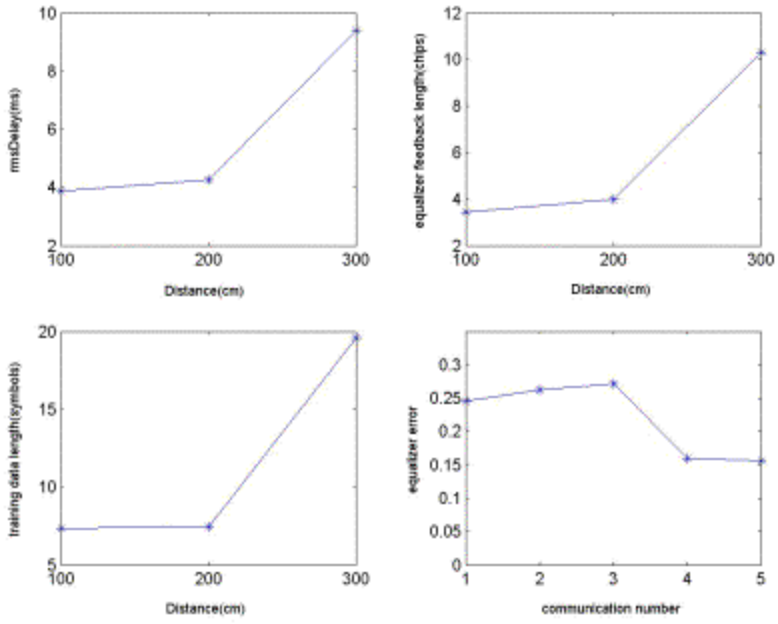

Figure 3. Some example data

1). If the difference between the `rmsDelay` and the previous equalizer feedback coefficient vector length is large (larger than half of the previous equalizer feedback coefficient vector length), then a new equalizer must be constructed, the length of equalizer being 3*(integer part of `rmsDelay` + 1). A new training data is also constructed, with the length of (3*`rmsDelay`*20) mod (symbol length), which is about 20 times the length of the equalizer.

2). If the difference between the `rmsDelay` and the previous equalizer feedback coefficient vector length is not very large (smaller than half of the previous equalizer feedback coefficient vector length), and if the equalizer error is smaller than a predefined performance threshold, then the length of the training data can be decreased by a predefined value. The old equalizer will be used.

3). If the difference between the `rmsDelay` and the previous equalizer feedback coefficient vector length is not very large (smaller than half of the previous equalizer feedback coefficient vector length), and if the equalizer error is larger than a predefined performance threshold, then the length of the training data can be increased by a predefined value. The old equalizer will be used.

The predefined performance threshold was set to 0.122. The increase or decrease value was set to 1, which means that one symbol will be added or removed from training data sequence each time.

Figure 3 shows the results of experiments using two acoustic nodes. The four plots in Figure 3 show the average rmsDelay, the length of the equalizer coefficient vector, the length of the training data sequence and the length of the ordinary data sequence. We tried three inter-node ranges: 1m, 2m and 3m. Some objects were placed around the speaker and microphone so that there would be some multipath effects.

From Figure 3 we can see that for the worst condition (in our case, 3 meters distance with reflection), about twenty symbols were selected according to the negotiation rules, while in other cases, fewer symbols were needed. From these figures we see this approach can result in a 60% reduction in packet overhead.

The last plot shows the change of the equalizer error when the communication nodes change their equalizer and training data. In this experiment, the distance between two nodes was 3 meters, and since we initially used only twelve symbols as training data, the first three transmissions had high equalizer errors. In fact, this situation requires at least twenty training symbols. The equalizer size of the two nodes was increased to twenty in the third packet, and consequently, the equalizer error was reduced greatly.

7. Conclusions and future work

We have presented a mechanism for software defined communication nodes to *interoperate*, i.e., understand other nodes and modify the processing of packets during a communication session both at the source and the destination. A negotiation of two OBR nodes to tailor the training sequence length according to the channel dynamics and noise level was used as an example, and the result shows that this negotiation results in a 60% reduction in the packet overhead, and equalizer training is improved in situations of high noise or channel dynamics. The main goal for these experiments was to show that opportunities for negotiation of protocol parameters do exist and can be achieved using the OBR concept presented in this paper. In the future we will continue to investigate the interoperability of the communication nodes, in particular, we will focus on using different communication protocols (waveforms).

At this time we are not aware of any other implementation of an ontology-based radio. The closest to our approach is the concept of the XG program [6] in which ontologies and negotiation are supposed to be used for dynamic spectrum management. This concept is described in [7, 8]. It is our belief that our approach is appropriate for the implementation of this concept.

8. REFERENCES

[1] DAML. DARPA Agent Markup Language web site, 2001. www.daml.org.

[2] J. Wang, D. Brady, K. Baclawski, M. M. Kokar and L. Lechowicz. "The Use of Ontologies for the Self-Awareness of the Communication Nodes." In Proceedings of the Software Defined Radio Technical Conference, SDR'03, Orlando, FL, 2003.

[3] R. Fikes, J. Jessica, and F. Gleb, "JTP: A System Architecture and Component Library for Hybrid Reasoning.", Proceedings of the Seventh World Multiconference on Systemics, Cybernetics, and Informatics. Orlando, Florida, USA. July 27 - 30, 2003.

[4] Paul Tarau, "Kernel Prolog: a Lightweight Prolog-in-Java Interpreter with Fluent based Built-ins USER GUIDE", 1999. http://www.binnetcorp.com/kprolog/KernelPrologUserGuide.html

[5] John G. Proakis, *Digital Communications*. McGraw-Hill, New York, 1995.

[6] Request for Comments Documents. DARPA. http://www.darpa.mil/ato/programs/xg/rfcs.htm. 2003.

[7] The XG Vision. Request for Comments. Version 2.0. BBN. http://www.darpa.mil/ato/programs/xg/rfc_vision.pdf. 2003.

[8] XG Policy Language Framework. Request for Comments. http://www.darpa.mil/ato/programs/xg/rfc_policylang.pdf. BBN. Version 1.0. April 16, 2004.

Acknowledgments

This research has been partially supported by the NSF grant 0225442.