Waveform Reconstruction from Ontological Description

Leszek Lechowicz (Northeastern University, Boston, MA; llechowi@ece.neu.edu)
Mieczyslaw M. Kokar (Northeastern University, Boston, MA; m.kokar@neu.edu)

ABSTRACT

We present the details of a method for ontology-based waveform reconfigurability. In this method cognitive radios share the same base SDR ontology, which allows the radios to understand the concepts in a uniform way thus enabling transfer of more complex concepts from one node to another. In the process of reconfiguration, nodes can receive descriptions of waveforms expressed in Web Ontology Language (OWL) and Rules and then automatically configure their processing according to the specification. Such specifications would contain both structural descriptions of software components and finite state machines (FSM) necessary to compose the waveform from simpler software modules. The waveform configuration process encompasses generating state machines, building a model of the waveform by generating OWL individuals and relationships between them using the inference engine and the specified rules. The constructed model is then used to instantiate state machines and other software components and to connect them in the prescribed way. The result of the overall process is such that a cognitive radio is able to learn and construct a waveform it did not know before.

A proof-of-concept system has been built confirming the feasibility of the proposed method. In the process of this system's evaluation three different waveforms (BPSK31, QPSK31 and RTTY) have been described in OWL and Rules, the descriptions were successfully transferred from one node to another and then used by the receiving node to construct fully functional software modules implementing the waveforms.

1. INTRODUCTION

Most of the SDR architectures offer a set of adjustable and observable parameters of a waveform (also known as *knobs and meters*). These parameters can be used to implement the *Set* and *Get* operations, which allow making changes in the waveform parameters to improve the communications. The *Set* and *Get* approach to reconfigurability has been investigated by Wang et al. [1,2] in their Ontology-Based Radio (OBR) architecture. It was further refined by Moskal in his Cognitive Radio Framework (CRF) [3].

In the previous work ([4,5]), an interoperability scenario was proposed in which not only could Cognitive Radios (CRs) change the parameters of the waveform (i.e. perform *Set/Get* operations) but could also negotiate their *Reconfiguration* i.e. the use of a different waveform. In the current scenario when a CR receives a request for a specific software component it does not know, it can query the sender for a description of that component as a composition of simpler components. If any of the simpler components are also not known to the node, the querying can iteratively continue until at some level of decomposition the receiving node knows all the components. The method assumes the existence of a *base SDR ontology* shared by all CR nodes. It also assumes that all components not in the base ontology can be decomposed into simpler ones and that the decomposition process can be repeated at each level, until the components are represented by components from the base ontology. These two fundamental assumptions guarantee that two CR nodes will understand each other.

The method for waveform reconfiguration presented in this paper is based on the interoperability scenario introduced in [4,5] and relies on the idea that the knowledge can be transferred from one CR to another and used to construct software components the CR node did not previously know. Just like in [4,5], the CRs share the base SDR ontology and can transfer descriptions of software components as compositions of simpler components expressed in Web Ontology Language (OWL) and rules. The scenario in [4,5] did not, however, provide a way to specify a behavior. The waveform reconfigurability method described here includes the ability to transfer descriptions of behaviors in the form of finite state machine (FSMs) models. Such descriptions – also expressed in OWL and rules – are used by the receiving node to generate a component that can be added to other components to provide the behavior for the composition.

2. OVERVIEW OF THE RECONFIGURATION METHOD

The waveform reconfigurability method depends on the following four elements: an SDR ontology representing components and their properties, a formal language in which waveform descriptions are expressed, an inference engine that provides formal reasoning capability and the application of FSM models for automatic generation of behaviors.

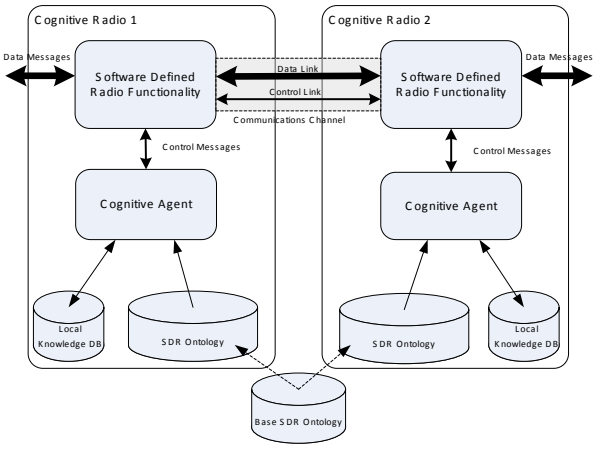

Figure 1. Ontology-Based reconfigurable Cognitive Radios.

The general system architecture of a reconfigurable CR is shown in Figure 1. The SDR part of the radio's software provides the communications services to the user. The bulk

of data sent across the communications channel consists of data messages from higher layers of protocol (data link). In addition to that, CRs exchange signaling messages (control link).

The process of reconfiguration involves the following steps:

1. A CR node sends a *Reconfigure* request to another node
2. If the other node knows the requested waveform it executes the request and switches to that waveform. If it doesn't know the waveform it responds with a request for its description.
3. A node that initiated the request for reconfiguration sends the description of the waveform, which is represented in OWL/Rules and may contain either a description of the component structure or a specification of the state machine.
4. The receiving node verifies whether it knows all the components in the description. If not, it sends a request for description for all components it doesn't know. This process is repeated until all unknown components are decomposed to simpler components the node knows.
5. A model of the composite component is built out of OWL individuals.
6. Based on the model, appropriate software components are instantiated from the local component library. State machines are generated from their descriptions.
7. The waveform component is assembled by connecting the components as described in the OWL model.
8. The newly constructed waveform is put into service.

3. PREREQUISITES

Some conditions have to be satisfied in order for the waveform reconfiguration method to work:

- CRs have to be able to communicate with each other.
- CRs have to share the same base SDR ontology. All other concepts in their SDR ontologies have to be decomposable into the concepts in the base ontology.
- CRs have to be able to send and respond to queries about any arbitrary element not in the base ontology
- CRs have to be able to incorporate the new facts they learned from other nodes into their local knowledge bases
- CRs have to be able to reason about the facts in their knowledge base, the facts they learned through queries and their internal status. They have to be able to use these facts to reconfigure themselves as necessary

4. ONTOLOGY AND CHOICE OF THE LANGUAGE.

The *base SDR ontology* is a standardized set of SDR-related concepts and their relationships. At the moment of this writing there is a Cognitive Radio Ontology (CRO) developed by the Wireless Innovation Forum [6], which has been selected as a starting point for the proof-of-concept system developed in this research.

The base ontology is the nucleus of all instances of SDR ontology. A particular instance in addition to base ontology may contain some additional local or vendor specific concepts – all of them have to be decomposable to concepts from the base ontology to ensure interoperability.

SDR ontology together with the node specific knowledge (e.g. node's configuration, waveform parameters, communication channel's QoS parameters etc.) constitute that node's knowledge base.

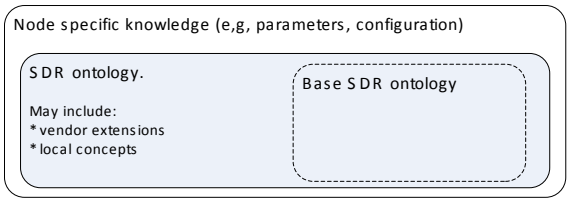

Figure 2. Cognitive Radio's knowledge base.

Although many languages have been developed to represent ontologies, in recent years the family languages based on RDF and RDFS gained prominence due to its involvement with the Semantic Web applications. Web Ontology Language (OWL) is the latest member of the family and it has been adopted as the official ontology language for the Semantic Web [7]. We selected OWL for the ontology development primarily due to the availability of development tools and inference engines supporting it.

As it was discussed in [4,5], OWL alone is not expressive enough to describe relationships in composite components and state machines. It has to be augmented with a rule language. Our choice is the rule language of BaseVISor – a forward-chaining inference engine developed by VIStology, Inc [8]. BaseVISor is based on the Rete network optimized for the processing of RDF triples, and it incorporates axioms and consistency checks for R-entailment, which supports all of the RDF/RDFS and a part of OWL-DL semantics.

5. COMPONENTS

Waveform reconfigurability has been developed around the idea of building complex software components out of simpler ones. The selection of particular software component architecture is not critical provided that all elements required by the proposed method can be implemented in it. In the proof-of-concept built during our investigation we implemented a lightweight framework based on the well known Observer design pattern [9], but other frameworks (e.g. CORBA) could have been used as well.

In the ontology all components are descendants of the class *Component*. The class *Component* has three subclasses: *BasicComponent*, *CompositeComponent* and *StateMachineComponent,* representing components from the base ontology, components not in the base ontology that are not state machines and state machine components respectively.

It should be emphasized that although the concept of Software Defined Radio has been developed around the idea

of replacing majority of the radio's hardware with the software routines, nothing in the waveform reconfiguration method restricts the components only to ones implemented in software. Specific method's implementations may allow mixed hardware/firmware and software configurations and it is the responsibility of those implementations' middleware to provide means for instantiation and composition with mixed types of components.

Components interact with other components through ports and signals and can be driven by an external clock (or clocks) (Figure 3).

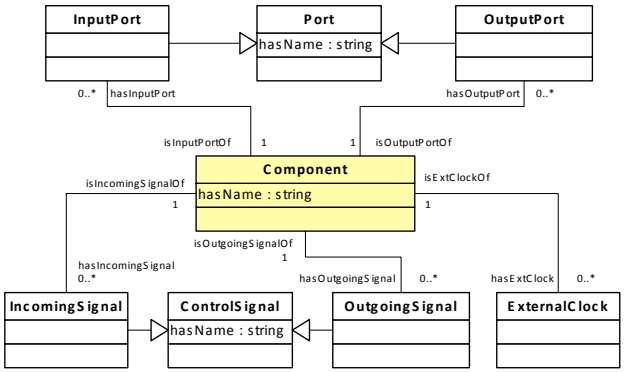

Figure 3. External connections of a Component.

Ports are interfaces through which data flows from one component to another. A port is characterized by two features – its direction and the data type it carries. Two ports are connected if they are related through the *isConnectedTo* relationship. An output port can drive more then one input ports. Any restrictions on how many input ports can be driven by a single output have to be imposed by local rules, the reconfigurability method itself does not set any limits.

Signals are binary messages sent from one component to another to notify about asynchronous events. Signals are fundamentally different from ports. Ports handle major flows of data so the throughput is the primary consideration. Signals on the other hand have to be able to handle large number of receivers. For example a reset signal in a very complex component might need to be received by all subcomponents (potentially a large number). This characteristic of signals might have significant impact on how they are practically implemented (particularly in hardware or firmware).

An external clock is a source of periodical events that are used to pace the flow of data through synchronous components. Components may require more than one external clock (not common) or may not require any clock signals at all – in this case the inputs are processed when they are asserted on the input ports.

Two sets of parameters may be associated with a component: *instantiation parameters* and *run-time parameters*.

The instantiation parameters are used during the construction of an instance of the component. In case of components implemented in an object oriented language the instantiation parameters might simply be passed to the constructor routine as input parameters.

Certain components might also have observable and/or adjustable run-time parameters (*knobs and meters*), which can be read and/or adjusted through *Get* and *Set* operations. A run-time parameter can be adjustable or can be read-only. Each of these parameters has a tag (a name) that is used to identify it during *Get*/*Set* operations

6. COMPONENT INSTANTIATION

Components can be implemented not only in software but also in firmware or hardware. A system can support more than one kind of component instantiation - for example it can provide a number of hardware resources implementing the functionality and additionally a software version of the same component in case the number of provided hardware resources is not sufficient.

A descendant of *CompositeComponent* type may be instantiated either as a single discreet component (if available) or as a composition of simpler components. Similarly a descendant of *StateMachineComponent* type may be instantiated as a discreet component but if such a component is not available the associated description of the state machine is used to generate an executable component instance.

The different types of instantiation are managed by a set of classes derived from the class *Instantiation*. In the proof-of-concept system, three instantiation classes have been defined in the ontology:

- *JavaClassInstance* – for instantiations from Java classes. An individual of this class contains the name of the JAR file and the name of the java class that implements the component.
- *ComponentComposition* – for instantiations that create instances of components as composition of simpler components.
- *StateMachineCodeGen* – for state machine components which are created through on-the-fly generation of executable state machines form their ontological descriptions.

The instantiation knowledge is node specific; it is not a part of the standardized ontology. The selection of an appropriate instantiation class for a particular component is made by the inference engine. A set of rules governing the selection is specific to the CR node and is defined in its local knowledge base. The reasoner associates a specific instantiation class with each component class based on the rules and data available in the knowledge base. Later on, when an OWL individual representing an instance of a component is asserted to be *rdf:type* of that component class, the rules assert that this individual is also *rdf:type* of the instantiation class associated with that component type and that in turn fires another rule that is specific for that component type and instantiation type, which asserts for this individual the facts required for the creation of the instance.

7. STATE MACHINES

State machines are frequently used in the definition and implementation of communication protocols. Traditionally state machines are hardcoded and are the integral part of the overall protocol implementation thus they cannot easily be changed or extended if new functionality is to be added. Reconfigurable systems that are to be able to learn new, unfamiliar protocols have to be able to support the creation and instantiation of state machines from their descriptions.

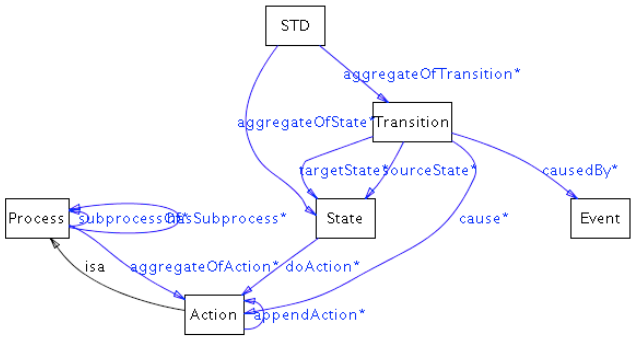

Figure 4. WIF's prototype of STD ontology [6].

Wireless Innovation Forum (WINNF) proposed that state transition diagrams (STDs) be used to describe finite state machine [6]. We used their prototype of STD ontology as a starting point for defining the state machine description ontology. Certain concepts in the ontology we developed have been inspired by similar constructs in the UML StateMachine package [10], but it should be understood that this ontology is not a mapping of the UML standard.

State machine components are externally similar to any other components. They communicate with other components through ports and signals and can be driven by an external clock.

In order to better illustrate the state machine description ontology, consider the following example of a simple state transition diagram of a decimator by factor 4 (Figure 5).

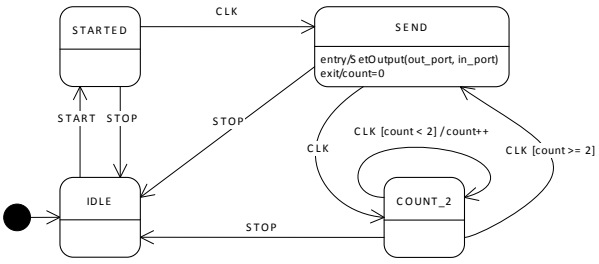

Figure 5. An state transition diagram of a decimator by 4.

The external view of the decimator is shown in Figure 6.

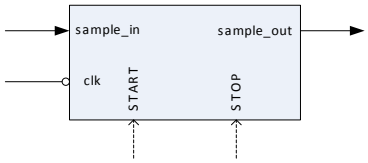

Figure 6. External inputs and output of the decimator component.

The decimator component is activated when it receives *START* signal. The reception of *STOP* deactivates it by making it to transition back to the IDLE state. When active, the decimator stays in the loop between the SEND and the COUNT_2 state. It sends to the output a sample read from the input port every time it enters state SEND, which happens every fourth clock event (CLK).

The following entities are involved in the definition of a state machine:

- *States*
- *Transitions* between states
- *Clocks*
- *Input* and *output ports*
- *Incoming* and *outgoing signals*
- Definitions of *events*, which can be generated on:
  - a reception of a signal
  - a change of value on an input port
  - a reception of a clock signal
- *Properties* that store auxiliary data necessary for the state machine to work. They are equivalent to local variables in programming languages.
- *Parameters* which are equivalent run-time parameters of regular components and which support *Get* and *Set* operations.

States are fundamental elements of the state machine and are represented by individuals of type *fsm.State*. A state may define *OnEnter* and *OnExit* action sequences which are executed when the state machine transitions into and out of the state respectively. A definition of the state also references the individuals representing the transitions that originate in that state.

```
<obr:fsm.hasState variable="_Ind.STARTED">
    <rdf:type resource="owl:NamedIndividual"/>
    <rdf:type resource="obr:fsm.State"/>
    <obr:hasName datatype="xsd:string">STARTED
    </obr:hasName>
    <obr:fsm.hasTransition variable=
      "_Ind.STARTED.Transition.0">
        <rdf:type resource="owl:NamedIndividual"/>
        <rdf:type resource="obr:fsm.Transition"/>
        <obr:fsm.triggeredBy variable="_Ind.Event.CLK"/>
        <obr:fsm.hasTarget variable="_Ind.SEND"/>
    </obr:fsm.hasTransition>
    <obr:fsm.hasTransition
      variable="_Ind.STARTED.Transition.1">
        <rdf:type resource="owl:NamedIndividual"/>
        <rdf:type resource="obr:fsm.Transition"/>
        <obr:fsm.triggeredBy variable="_Ind.Event.STOP"/>
        <obr:fsm.hasTarget variable="_Ind.IDLE"/>
    </obr:fsm.hasTransition>
</obr:fsm.hasState>
```

Figure 7. Definition of the state STARTED in the example decimator state machine.

Transitions are represented by individuals of the class *fsm.Transition*. A description of transition includes the target state and the event that triggers it. A transition may be guarded by a constraint (*fsm.Constraint*), which is an expression that evaluates to a logical true/false value. In addition to a guard the transition may also define a sequence of actions to be taken when the transition is executed. The transition actions are executed after the *OnExit* action

sequence of the originating state and before the *OnEnter* sequence of the target state.

An *expression* (*fsm.Expression*) is an arbitrary statement that evaluates to a single value. In addition to their use as guards expressions are used as parameters to actions and API functions. The other types of expressions include API function calls, arithmetical operations, bit operations, values read from input ports and constant values.

Events trigger transitions between states. Three subclasses of *fsm.Event* correspond to three sources of events – clocks, signals and changes of input port values.

An *action* (*fsm.Action*) represents an executable statement. A collection of actions executed together is an *activity*. The simplest form of an activity is an *action sequence* which simply is an ordered list of individual actions. The type of actions supported in the state machine definitions include emitting a signal, setting output port to a value of an expression, setting a property (local variable) to a value of an expression. In addition to these there are two actions used for flow control (IF and WHILE statements) and there's also an action returning a value that is used in the definition of the *Get* operation handler for a parameter.

Properties are similar to local variables in the programming languages and may be used as data buffers, counters, flags etc.

Parameters may be used to access and possibly adjust the internal data values in the state machine. The definition of the parameter includes an action sequence for *Get* operation. It might also include an action sequence for *Set* operation if the parameter can be adjusted. If the *Set* action sequence is defined, the parameter definition may also include a validation constraint.

8. RULES IN COMPONENT DESCRIPTIONS

A complete description of a component consists of two parts:

- An external description that lists all the ports, signals, clock inputs and instantiation parameters. This description contains enough information to be able to instantiate the component and connect to the others but it does not say how such component can be composed. For *BasicComponents* (i.e. components in the base ontology that are not decomposable any further) this is the only description that is required.
- An internal description that augments the external description with the details how to build such component. The internal description is a part of the component's instantiation definition – and it contains either the description of the component composition (for components derived from *CompositeComponent*) or the description of the state machine (in case of descendants of *StateMachineComponent*).

An internal description of a *CompositeComponent* establishes relationships among OWL individuals representing different types of objects – components, ports, signals etc. involved in the composition. The easiest way to reference these individuals would be to use their URIs (*Uniform Resource Identifiers*). Unfortunately, when the component description is created, the concrete URIs of the individuals are not known. Indeed, they cannot be known, otherwise no OWL model could contain more than one composition of particular type. Still for the purpose of the composite component description one needs to be able to reference a particular individual without knowing its URI.

Consider a composite component consisting of a cascade of two multiply-accumulate (MAC) components as shown in Figure 8. If the MAC component is not available in the CR, the CR can request and receive its description as a composition of an adder and a multiplier. Since in OWL all unique individuals have to have different URIs, the URI of the multiplier in the composition substituting for MAC1 has to be different from the URI of the multiplier in the composition MAC2. Similarly, the URIs of the adders and the ports all have to be different. That clearly creates a problem when one tries to describe connections between the ports of the components within a composition.

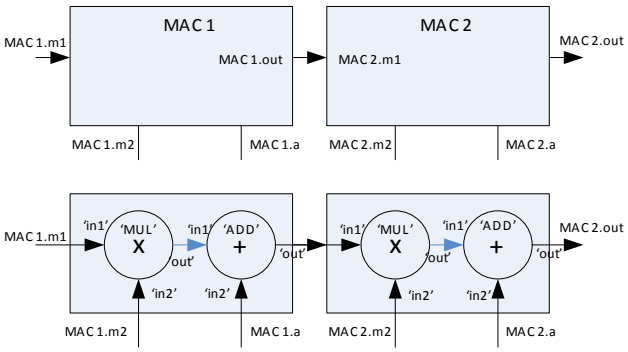

Figure 8. Individuals in component compositions.

We dealt with this problem by associating tags (labels) with all individuals in the description. The tags have to be unique at a given level of scope – e.g. all subcomponents have to have unique labels, all ports of the specific component have to be unique, etc. The labels are used to differentiate between individuals. Describing relationships in this approach however is no longer as simple as asserting a property between two individuals with known URIs. The descriptions of the relationships must necessarily be written as rules in which the labels are used to select the appropriate individuals and only then the relationship is asserted. Pseudo code of a rule describing a connection between the output of the multiplier and an input of the adder in MAC might look like the following:

If the specified individual is of type MAC and if:

- *it has a subcomponent with a tag 'MUL' and*
- *this subcomponent has an output port with a tag 'o' (assign the port's resource to the rule variable '_outPort')*
- *also if it has a subcomponent with a tag 'ADD' and*

- *this subcomponent has an input port with a tag 'a1' (assign the port's resource to '_inPort')*

then assert:

- *'_outPort' drives '_inPort'*

Similarly composed rules are used to describe other relationships in the component (e.g. signal connections, clock connections etc.).

9. COMPOSITE COMPONENTS

In order to fully define a composite component more data is required in addition to subcomponents and connections.

Consider the *PSK31CharEncoder* component used in the experiments in the proof-of-concept system (Figure 9).

The external ports and signals of *PSK31CharEncoder* are defined in its external description (see section 8). The internal description contains the definitions of subcomponents, connections between them and the connections between external ports and signals of the ports and signals of subcomponents. The tags of the external ports and signals can be different than the tags of ports/signals they are internally connected to.

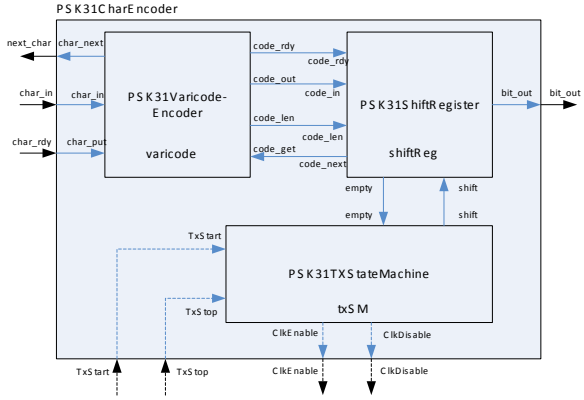

Figure 9. The internal structure of PSK31CharEncoder.

Two additional elements of the composite component description are not shown in Figure 9. One of them is the mapping of the external clock inputs of the composite component to clock inputs of the internal components. In this particular example the mapping is trivial as all internal components have only one clock input each and they all map into the singular clock input of the composite component.

The second type of information not shown in the above diagram is the mapping of run-time parameters of the composite component into run-time parameters of internal components. In the proof-of-concept system we use simple pass-through type of mapping but in a real system that might not be sufficient. A solution similar to the one used for state machine run-time parameters might also be implemented for composite components.

10. COMPONENT DESCRIPTION TRANSFER

The process of reconfiguration starts when one of the CRs sends a request to another node to use a particular waveform (i.e. to instantiate and put online an instance of the specific software component implementing such waveform). If the receiving node knows that component then if other conditions are met the node can create the instance of the component and start using it. If the node is unfamiliar with the requested component it sends a query to the other node requesting relevant information about the component.

What exactly triggers the query is the assertion that an individual is of the specific *rdf:type* that is not present in the local knowledge base or that an individual is of type that is defined in the knowledge base but it does not have any instantiation class associated with it. Both situations are caught automatically in the BaseVISor rules and special procedural attachments designed to handle them are called.

After the descriptions of the component are transferred from the other node, the new facts are asserted in the knowledge database and the new rules are added to the local rule set. Then the inference engine is called again and this time around when it gets to the point where it processes the original assertion that triggered the component query, the knowledge base does have both the external and internal description of the component. When the individual is asserted to be of that specific type, the description rules kick in and automatically generate OWL individuals for all internal and external elements of the composite component. The presence of these individuals triggers in turn all other rules that define connections and other relationships among them effectively defining the whole composite component.

The process of building a state machine from its ontological description is very similar. The OWL model of the state machine is also built automatically through asserted facts and rules. The waveform reconfiguration method does not impose any specific way for the translation of the OWL model of the state machine into a component. In the proof-of-concept system we implemented a two-stage process for that. In the first stage the OWL constructs are translated into related Java constructs buffered in memory. In the second stage the buffered Java source code is compiled in memory into a binary class that is used to create an instance of the state machine component.

The process of constructing an OWL model of a composite component or a state machine is completely automatic. It utilizes the power of formal reasoning in the forward-chaining inference engine. As long as the description rules and facts are correct, the resulting model is correct too. Since the actual working instance of the composite component or state machine is a simple one-to-one mapping of OWL individuals and relationships into components and connections (or java constructs in case of the state machine descriptions), the opportunities to make any errors due to ambiguities in interpretation are virtually eliminated and barring any errors in the mapping process itself, the building of the component from its description is provably correct.

Once the description of a component is transferred and integrated in the local knowledge base, it is automatically

applied to all other instances of that component type. The CR effectively learned the facts about the previously unknown component and is able to apply the acquired knowledge as needed.

11. PROOF-OF-CONCEPT SYSTEM

During our investigations we built a proof-of-concept system. The system has been implemented in Java and consists of two processes communicating with each other over the network. One of the processes functions as a master node that initiates the reconfiguration request and responds to queries for component descriptions. The other one acts as a slave node and it receives reconfiguration requests, sends component queries to the master as needed and then integrates in the local knowledge base the received component description. The slave node implements the middleware responsible for generating state machines and assembling composite components from their ontological descriptions.

Both nodes use BaseVISor as their inference engines. The integration of BaseVISor with the rest of the cognitive agent is done through BaseVISor's java API and through procedural attachments. The java API is used primarily to interact with the fact database i.e. to assert facts and to make queries. The API is also called when the inference process needs to be (re)triggered. Procedural attachments are used to define new constructs in the BaseVISor rule language and in the experimental system they are used for two purposes – to generate new resources when the rules are building an OWL model (of a composite component or a state machine) and to initiate the component query to the master node when the rule detects that an unknown type of an individual has been asserted.

We selected transmit sides of three digital waveforms very popular in the amateur radio service as the subjects of our experiments. These are BPSK31, QPSK31 and RTTY. The reason for this selection is that all of them are of non-trivial complexity but at the same time they don't require great computational resources and their complexity is not so big as to impede the ability to diagnose problems that were bound to appear in the process of building the experimental system. All these waveforms are narrowband so they can be generated and monitored by a computer with a soundcard.

For the purpose of testing the selected waveforms a test harness has been built in the slave process (Figure 10).

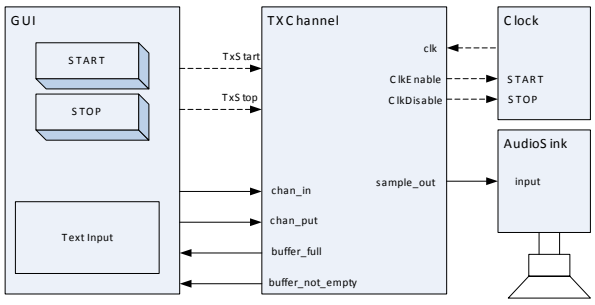

Figure 10. Waveform test harness.

The test harness consists of a simple GUI panel that allows to generate the events *TxStart* and *TxStop* that start and stop the waveform under test (*TXChannel*). A text input panel in the GUI allows entering of textual messages that are passed onto the waveform as the payload data. The waveform generates *ClkEnable* and *ClkDisable* signals that control the software clock. The output of the waveform is connected to the input of the AudioSink component that drives the computer's soundcard. The audio output from the soundcard is monitored by a freeware application *fldigi* [11] that is able to receive and decode all three waveforms.

We implemented all waveforms with the same set of external connections as shown in the test harness diagram - they are all derived from the same class *TXChannel*. That simplifies the development of the test harness and it also resembles an actual CR radio where the waveforms change as needed but sources and sinks of samples as well as the control inputs remain the same.

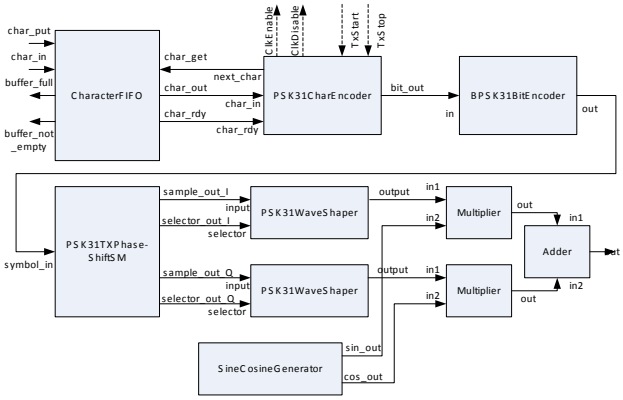

Figure 11. Decomposition of BPSK31TXChannel.

All waveforms have been implemented as discreet software components ready to be instantiated. We also created composite component descriptions for each of them. The decomposition of *BPSK31TXChannel* is shown in Figure 11. *BPSK31TXChannel* and *QPSK31TXChannel* both use *PSK31CharEncoder* which is also implemented as a discreet software component and it also has a component composition (Figure 9). *PSK31TxStateMachine* is implemented as a discreet software component but it also has a description of its state machine. With only some components decomposed to simpler ones in the proof-of-concept system we were able to test the following scenarios for *BPSK31TXChannel* waveform alone:

- the whole waveform available as a discreet software component.
- the waveform component is not available; its description is transferred and the composite component is composed as in Figure 11.
- not only the waveform is not available, but also *PSK31CharEncoder* is not available as well. In this scenario a composite *PSK31CharEncoder* is nested

within the composite component implementing the waveform.

- in this scenario also *PSK31TxStateMachine* is missing so at the end of the reconfiguration we get an auto-generated state machine embedded inside a composition for *PSK31CharEncoder* that in turn is embedded in the waveform composition.

Similar scenarios were investigated for *QPSK31TXChannel* and *RTTYTXChannel*. All of the test scenarios were successfully executed and we were able to observe proper encoding of the textual data for all three waveforms (using the *fldigi* program).

12. CONCLUSIONS

We proposed a method for waveform reconfiguration based on the transfer of facts and rules expressed in OWL and BaseVISor rule language. The method relies on the existence of a standardized base SDR ontology to ensure interoperability between nodes. The method assumes that all concepts not in base ontology can be decomposed into concepts that are defined there. When a reconfiguration request is received by a node and the node is not familiar with the requested software component, it sends a query to the node that originated the request in order to obtain the description of the unknown component. There are two types of descriptions – one that describes a component as a composition of other, simpler components and another one that describes the component as a state machine, which is particularly useful for describing behaviors. Both types of components are externally similar i.e. they use the same external interface of ports, signals etc. to connect with other components. We discussed different ways of instantiating components and the process of transferring and integrating component descriptions in the local knowledge base. We discussed how the power of formal reasoning assures that given correct input the resulting composite component or state machine is also correct. We described the proof-of-concept system we built for our experiments and described some test scenarios we investigated while experimenting with the implementation of transmit sides of three popular waveforms used in amateur radio (BPSK31, QPSK31, RTTY). The experiments were successful and the proposed method has been proven to work for the limited sample of waveforms we experimented with.

Future work should include issues we did not have the opportunity to explore. For example, versioning of the software is very important from the compatibility point of view. In our method there are a few places that may be sensitive to the versioning issues – e.g. the middleware that implements the infrastructure of the components (ports, signals etc.), the OWL model to a composite component or state machine mapping software, etc. All potential problems with versioning should be identified and remedial procedures for them determined. Additionally, although we did some preliminary analysis and we did gather some experimental data for some generated composite components and state machines of different complexity, the bounds of usability of this method for real-life waveforms should be established.

13. REFERENCES

[1] J. Wang, D. Brady, K. Baclawski, M. Kokar, 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, 2003.

[2] J. Wang, M. Kokar, K. Baclawski, and D. Brady, *Achieving self-awareness of SDR nodes through ontology-based reasoning and reflection*., in SDR Technical Conference, Proceedings of the Software De_ned Radio Technical Conference, 2004.

[3] J. Moskal, *Interfacing a Reasoner with Heterogeneous Self-Controlling Software*., PhD Dissertation, Northeastern University, April 2011.

[4] L. Lechowicz, M. Kokar. *Achieving Dynamic Interoperability of Communication: Transfer of Ontology and Rules Between Nodes.* In Proceedings of the Software Defined Radio Technical Conference SDR'06, 2006.

[5] L. Lechowicz, M. Kokar. *Composition, Equivalence and Interoperability: An Example.* In Proceedings of the Software Defined Radio Technical Conference SDR'07, 2007.

[6] Wireless Innovation Forum, *Description of the Cognitive Radio Ontology.* Working Document WINNF-10-S-0007, August 29, 2010.

[7] *OWL 2 Web Ontology Language. Document Overview.* W3C Recommendation 27 October 2009.
http://www.w3.org/TR/2009/REC-owl2-overview-20091027/

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

[9] E. Gamma, R. Helm, R. Johnson, J. Vlissides, *Design Patterns. Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1995.

[10] *OMG Unified Modeling Language (OMG UML), Superstructure, Version 2.4.1*, Object Management Group, 2011
http://www.omg.org/spec/UML/2.4.1/Superstructure/PDF/

[11] Fldigi (Fast Light Digital Modem Application)
http://www.w1hkj.com/Fldigi.html