Ontology-based Reconfigurability of Cognitive Radio

A Dissertation Presented

by

Leszek Janusz Lechowicz

to

The Department of Electrical and Computer Engineering

in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

in the field of

Computer Engineering

Northeastern University
Boston, Massachusetts

November, 2012
Dissertation Title: Ontology-based Reconfigurability of Cognitive Radio.

Author: Leszek Janusz Lechowicz

Department: Electrical and Computer Engineering

Approved for Dissertation Requirement for the Doctor of Philosophy Degree

Dissertation Adviser: Prof. Mieczyslaw Kokar Date

Dissertation Reader: Prof. Kenneth P. Baclawski Date

Dissertation Reader: Prof. Kaushik R. Chowdhury Date

Dissertation Reader: Prof. Gunar Schirner Date

Department Chair: Prof. Ali Abur Date

Graduate School Notified of Acceptance:

Department Chair: Prof. Ali Abur Date
Abstract

Software Defined Radio (SDR) is a radio in which all or majority of the functionality of the physical layer is implemented in software. One of the biggest benefits of SDR is the relative ease in which the functionality can be changed. Such flexibility could potentially be used in achieving interoperability of the communications systems. Cognitive Radio (CR) – a paradigm that combines SDR technology with a cognitive agent - allows radios to change their operational behaviors in order to achieve a variety of goals, e.g. performance optimization, power saving, opportunistic use of resources, etc. Cognitive Radios are expected to play the leading role in the communications systems interoperability.

In this work a method for ontology-based waveform reconfigurability is proposed. In this method all interoperating 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 overall result is such that a cognitive radio is able to acquire, construct and use 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 and these 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.
## Contents

1 Introduction

1.1 Software Defined Radio (SDR) ................................................................. 1

1.2 Cognitive Radio (CR) ............................................................................. 2

2 Problem formulation

2.1 Interoperability of communications systems ............................................. 5

2.1.1 Definition of interoperability ............................................................. 5

2.1.2 Interoperability measurement models ............................................... 6

2.1.3 The need for interoperability in communications systems ................. 7

2.2 Shortcomings of static reconfiguration schemes ....................................... 8

2.3 Reconfigurability of the physical layer as the key for interoperability ....... 9

2.4 Requirements ..................................................................................... 11

3 Related research

3.1 Cognitive radio as a self-controlling software system .............................. 12

3.2 Language issues .................................................................................. 14

3.2.1 Imperative vs. declarative ............................................................... 15

3.2.2 Web Ontology Language (OWL) ..................................................... 16

3.2.3 Rules .............................................................................................. 21

3.2.4 Negation and open vs. closed world reasoning .................................. 23

3.2.5 Functional equivalency ................................................................. 24

3.2.6 Behavioral aspects ........................................................................ 31

3.3 Component-based software systems .................................................... 35

3.3.1 Process algebras ............................................................................ 35

3.3.2 Software Communications Architecture (SCA) ................................. 37

3.3.3 The SpecC methodology and language ............................................ 39

3.3.4 Functional Description Language ................................................... 44

3.3.5 The use of OWL in the composition of software services .................. 46

3.3.6 Service-Oriented Architecture ....................................................... 46

4 A Method for Ontology-Based Cognitive Radio Reconfigurability ......... 50

4.1 Software Defined Radio reconfiguration .............................................. 50

4.2 Prerequisites ...................................................................................... 52

4.3 Base SDR ontology ............................................................................ 53

4.3.1 The contents and granularity .......................................................... 53

4.3.2 Base ontology as the basis of the cognitive radio’s knowledge base ..... 56

4.3.3 The choice of the language ............................................................. 57

4.3.4 The choice of Base SDR ontology .................................................. 57

4.4 General system architecture .................................................................. 58

4.5 An overview of waveform reconfiguration .......................................... 59

4.5.1 Waveform description ................................................................... 60

4.5.2 Correctness ..................................................................................... 62

4.5.3 A simplified overview of the waveform construction process .......... 63

4.6 Components ...................................................................................... 66

4.6.1 Component types ........................................................................... 66

4.6.2 The external interfaces of the Component ...................................... 67

4.6.3 Ports ............................................................................................... 68
1 Introduction

When Guglielmo Marconi got interested in the research done by Heinrich Hertz he could hardly grasp the profound importance of the work he was about to accomplish. The idea that a signal can be sent through an empty space changed our world in ways unthinkable at that time. Today one cannot imagine a world without wireless communications. Radio, TV, cellular phones, wireless computer networks are all integral parts of our life even though some of those technologies were developed very recently.

Rapid development of communications systems theory brought about an incredibly fast pace of development of new wireless applications for which solutions based on custom designed hardware simply could not be created fast enough.

The unrelenting advances in modern microprocessors and reconfigurable logical devices (FPGAs) made software-based wireless design a reality. In fact most of the modern communications systems devices like cellular phones, wireless computer cards, cable modems etc. are implemented as software defined radios (SDRs) in which most of their functionality is implemented in software running on specialized Digital Signal Processors (DSPs).

Software Defined Radios process the bits of data into samples driving DACs for the transmit channel and receive samples from ADCs and turn them into the bits of data using very complex digital communications algorithms. In all their sophistication they do however remain ignorant about themselves – their internal structures and parameters. Mitola discussed this problem in his paper in 1999 [Mitola 1999a] and he postulated that if SDRs were made aware of their internal structures and the environment they operate in – i.e. if the SDRs were turned into Cognitive Radios, then a completely new level of functional possibilities would open. One of the biggest promises cognitive radios bring to the table - the ability to modify their structures based on the results of gathered data and reasoning - could be applied to variety of problems, from optimizing the speed of the data transmission over particular channel, to self-organizing wireless networks, to dynamic spectrum management, to protocol discovery and transfer, etc. This work concentrates on using the reasoning capabilities of cognitive radios for solving the problems related to reconfigurability.

1.1 Software Defined Radio (SDR)

The origins of Software Defined Radio can be traced back to certain defense-sector projects in the 1980’s. The actual term “software defined radio” is credited to Joseph Mitola who introduced it in his conference paper in 1991 [Mitola 1991]. One of the first publicly known software defined radio initiatives was a military project named SpeakEasy (phase I), which was run from 1992 to 1995 [Torre 1992]. SpeakEasy I was a rack full of equipment, hardly portable by today’s standards. It served however as a proof that fully functional software-based radio can be implemented. SpeakEasy I was soon followed by SpeakEasy II, which was a radio of much smaller, practical size and which had programmable vocoder and sufficient processing power to support multiple waveforms [CRT Ch.1]. SpeakEasy II was followed by US Navy’s Digital Modular Radio (DMR) and other defense-related projects. The SDR technology also spread
beyond military applications and became one of the technologies driving enormous success of wireless communications in the late 90’s and 2000’s.

SDR Forum defines a Software Defined Radio as “radio in which some or all of the physical layer functions are Software Defined” [SDRF Defs]. A simplified block diagram of a software defined radio is shown in Figure 1-1.

![Figure 1-1  Simplified block diagram of a Software Defined Radio.](image)

The basic idea of the Software Defined Radio is to sample the input signal from the antenna using ADC and then process the samples in a DSP processor. Provided that the sampling can be done with high enough frequency and that the DSP processor is fast enough to process those samples in real time, the majority of the hardware circuitry of a traditional analog receiver is replaced by software algorithms in DSP processor leading to significant savings in the cost of hardware and power consumption. Similarly, the hardware circuitry required for the transmit path in an analog device is replaced by a set of algorithms that produce samples passed on to the digital-to-analog converter (DAC), the output of which drives the RF power amplifier. In practical implementations the RF portion of the SDR might still include substantial amount of analog hardware. First of all the signal level from the antenna is usually orders of magnitude too small to drive the analog-to-digital (ADC) converter to full range. Second, high resolution ADCs and DACs available today might not have enough bandwidth to work with the required frequencies without shifting them down using the frequency conversion. Additionally, the output of the power amplifier usually needs analog filters that suppress spurious emission and a circuitry they matches the amplifier’s output impedance to the impedance of the antenna.

### 1.2 Cognitive Radio (CR)

Mitola posed a hypothetical question a GSM wireless network could ask one of its handsets: “How many distinguishable multipath components are you seeing?” [Mitola 1999a]. This kind of information can be extracted from the time domain values of
coefficients of the equalizer filter running in the GSM handset. The author pointed out that there are at least two fundamental problems with this kind of question. First, the network has no standard language in which to ask such a question. Second, even though the data is there, embedded in the taps of the equalizer, the handset cannot access it, as it doesn’t have the computational description of its own structure. As the author stated, the handset doesn’t “know what it knows”. In his paper Mitola addressed the first problem by introducing Radio Knowledge Representation Language (RKRL) – a language designed to represent radio domain knowledge. The work on RKRL is extended to significant detail in his PhD dissertation [Mitola 2000] where he also proposed 9 levels of radio cognition capabilities (see Table 1-1). Cognition level 0 corresponds to a traditional radio whether implemented as SDR or in hardware. All levels of cognitive capabilities higher than 0 require that the radio has some reasoning capabilities.

Table 1-1 Characteristics of Radio Cognition Task [Mitola 2000].

<table>
<thead>
<tr>
<th>Level</th>
<th>Capability</th>
<th>Task Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>Pre-programmed</td>
<td>The radio has no model-based reasoning capability</td>
</tr>
<tr>
<td>1</td>
<td>Goal-driven</td>
<td>Goal-driven choice of RF band, air interface, and protocol</td>
</tr>
<tr>
<td>2</td>
<td>Context</td>
<td>Infers external communications context (minimum user</td>
</tr>
<tr>
<td></td>
<td>Awareness</td>
<td>involvement)</td>
</tr>
<tr>
<td>3</td>
<td>Radio Aware</td>
<td>Flexible reasoning about internal and network architectures</td>
</tr>
<tr>
<td>4</td>
<td>Capable of</td>
<td>Reasons over goals as a function of time, space, and context</td>
</tr>
<tr>
<td></td>
<td>Planning</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>Conducts</td>
<td>Expresses arguments for plans/ alternatives to user, peers,</td>
</tr>
<tr>
<td></td>
<td>Negotiations</td>
<td>networks</td>
</tr>
<tr>
<td>6</td>
<td>Learns Fluents</td>
<td>Autonomously determines the structure of the environment</td>
</tr>
<tr>
<td>7</td>
<td>Adapts Plans</td>
<td>Autonomously modifies plans as learned fluents change</td>
</tr>
<tr>
<td>8</td>
<td>Adapts</td>
<td>Autonomously proposes and negotiates new protocols</td>
</tr>
</tbody>
</table>

Although there’s no mandate that says that a Cognitive Radio has to be implemented based on Software Defined Radio, in practice the relative ease with which radio’s parameters can be accessed makes such solution a preferred one. This tight association between CR and SDR is so widely accepted that SDR Forum added the definition of the CR to its glossary of definitions [SDRF Defs].

According to [SDRF Defs], a Cognitive Radio is defined 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. [CRT Ch.13] base their definition of a Cognitive Radio on the definition of a cognitive agent i.e. “a system that can:

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

All of the above definitions emphasize Cognitive Radio’s capability to express their knowledge of themselves and their ability to reason about themselves and the environment they operate in.
2 Problem formulation

2.1 Interoperability of communications systems

The large variety of communication systems used by the state and federal public safety agencies can pose significant interoperability problems as it was clearly shown during the tragic events of September 11, 2001. The initial response to the terrorist attack involved several emergency services i.e. New York Fire Department units, New York police, Port Authority police, Emergency Medical Services etc. Those services could not directly talk to each other; they had to pass messages through the command center and/or dispatch. The lack of direct line of communication aggravated the already chaotic situation as the information flow and coordination efforts between the responders were seriously affected. The 9/11 Commission Report specifically points out the inability of New York Fire Department chiefs to directly communicate with NYPD helicopters’ crews who reported instability of the WTC North Tower following the South Tower collapse [911]. Had their reports immediately reached the firefighters in the North Tower it is quite possible that those who were on the lower floors of the building could have evacuated before the tower collapsed. The 9/11 Commission recognized the importance of the inter-agency communications in the first response to the catastrophic event and included appropriate recommendations in that respect.

It should be emphasized that the government and particularly Department of Defense was aware of the challenges posed by the interoperability well before September 11, 2001. Interoperability of communications systems between units from different branches of the US Armed Forces has been recognized as one of the critical areas and the research activity in that matter can be traced back to mid-1960s with the first DoD directive concerned with the interoperability published in 1965 (DoD Directive 4630.5). Defense-related institutions have remained the leading interoperability research centers till the present times.

2.1.1 Definition of interoperability

According to the survey by Ford et al. [Ford 2007a], 34 definitions of interoperability have been proposed in research papers, standards and government documents. The definition included in Department of Defense Directive 2010.6 published first in 1977 is the oldest and most frequently referred to. The interoperability is defined there as:

“The ability of systems, units, or forces to provide services to and accept services from other systems, units, or forces and to use the services so exchanged to enable them to operate effectively together.” [DoDD 2010.6]

Even though some other definitions might be more specific in the particular field of interest, the appeal of the above is its applicability to a variety of problems. The use of the word “system” may suggest technical context (communications, data etc.), while “units” and “forces” allow interpreting this definition within the context of logistics, business organizational structures, operations etc. Note that depending on the particular understanding of the above definition, the term “services” might refer to completely
different concepts. In the technical contexts “services” might mean protocols, packets, APIs (Application Programming Interfaces), electrical standards, telephony standards etc. while in other contexts they might refer to procedures, orders, deliveries etc. In this thesis, service refers to a protocol, or more specifically to a waveform.

### 2.1.2 Interoperability measurement models

In order to be able to gauge to what degree given two systems are compatible, an interoperability measure is necessary. According to Ford et al., fourteen measurement models had been published in the literature related to interoperability before the authors published their own model [Ford 2007a]. One should refer to their publication for an extensive summary of all those methods. A short summary included in the authors’ another paper [Ford 2009] is presented in the following table.

<table>
<thead>
<tr>
<th>Method</th>
<th>Author</th>
<th>Type</th>
<th>Main contribution</th>
</tr>
</thead>
<tbody>
<tr>
<td>SoIM</td>
<td>LaVean</td>
<td>NL</td>
<td>Interoperability can be measured in levels</td>
</tr>
<tr>
<td>QoIM</td>
<td>Mensh et al.</td>
<td>NL</td>
<td>Interoperability is correlated to measures of effectiveness</td>
</tr>
<tr>
<td>MCISI</td>
<td>Amanowicz and Gajewski</td>
<td>NL</td>
<td>The distance between systems modeled as points in space indicates their interoperability</td>
</tr>
<tr>
<td>LISI</td>
<td>DoD</td>
<td>L</td>
<td>Systems possess interoperability attributes</td>
</tr>
<tr>
<td>IAM</td>
<td>Leite</td>
<td>NL</td>
<td>Same as LISI</td>
</tr>
<tr>
<td>OIM</td>
<td>Clark and Jones</td>
<td>L</td>
<td>Organizations interoperate, but have different interoperability attributes than technical</td>
</tr>
<tr>
<td>Stoplight</td>
<td>Hamilton et al.</td>
<td>NL</td>
<td>Operations and acquisitions both have interoperability requirements</td>
</tr>
<tr>
<td>LCI</td>
<td>Tolk</td>
<td>L</td>
<td>Operational interoperability is an extension of technical interoperability</td>
</tr>
<tr>
<td>LCIM</td>
<td>Tolk and Muguira</td>
<td>L</td>
<td>Conceptual interoperability bridges system interoperability</td>
</tr>
<tr>
<td>NMI</td>
<td>NATO</td>
<td>L</td>
<td>Same as LISI</td>
</tr>
<tr>
<td>NCW</td>
<td>Alberts and Hayes</td>
<td>L</td>
<td>Interoperability occurs in the physical, information, cognitive and social domains; lack of interoperability increases difficulty in accomplishing the mission</td>
</tr>
<tr>
<td>NTI</td>
<td>Stewart et al.</td>
<td>L</td>
<td>Social, personnel and process interoperability are valid types of non-technical interoperability</td>
</tr>
<tr>
<td>OIAM</td>
<td>Kingston et al.</td>
<td>L</td>
<td>There are levels of ability of organizations to be agile in their interoperations</td>
</tr>
<tr>
<td>NID</td>
<td>Schade</td>
<td>L</td>
<td>Levels of interoperability can be described in linguistic terms</td>
</tr>
<tr>
<td>i-Score</td>
<td>Ford et al.</td>
<td>NL</td>
<td>Interoperability measurements are operational process-specific and have a theoretical maximum value</td>
</tr>
</tbody>
</table>

*NL = non-leveling, L = leveling

The measurement methods are divided in the table into two broad categories – leveling (L) and non-leveling (NL). Leveling measures are conceptually based upon a maturity model developed by the Carnegie-Mellon University’s Software Engineering Institute [Humphrey 1997]. In the leveling models different aspects of interoperability are assigned numerical integer values.
depending on the set of threshold values representing increasing interoperability capabilities. The major weakness of any leveling model is very limited precision of the measurement (one of only a few integer values). All existing leveling models also use fixed sets of only one or a few interoperability attributes. The fixed set of attributes precludes any new attributes to be added in the future making those methods prone to become outdated.

Non-leveling interoperability measurement methods use other ways of assigning numerical values to particular elements of the model. For example MCISI [Amanowicz 1996] measures interoperability of two systems A and B as a normalized distance $d(A,B)$ defined as an average distance between particular features $a_i, b_i$ of both systems.

$$d(A,B) = \frac{1}{n} \sum_{i=1}^{n} |a_i - b_i|$$

Systems A and B are fully compatible if $d(A,B)$ is equal to 0.

The iAScore methodology [Ford 2007b] is based on the concept of the operational thread for which an interoperability matrix is calculated based on the multiplicity matrix (dependent on how many times a particular system has to support the thread) and on the so called spin matrix in which each element is assigned one of the three possible values \{-1, 0, 1\} with 1 meaning no human or machine translation necessary, 0 meaning only machine translation necessary and value -1 indicating that human translation is necessary. The i-Score methodology is different than the others as not only does it allow one to calculate the current interoperability score but also an optimal score for a given operational thread.

Only a few of the existing interoperability measurement models have been followed up with publications beyond the initial papers they were proposed in. None of the models has gained a widespread use so far.

2.1.3 The need for interoperability in communications systems

The 911 Commission Report emphasized the importance of the interoperability of communications systems used by emergency services and government agencies. The efficiency and speed in which a communications network integrating first responders can be established is probably the most important feature in this scenario.

Public safety is perhaps the most visible but not the only area where interoperability of communications systems is very important. Efficient flow of information, reports and orders among the units from different branches of the US armed forces (or between the US forces and units from allied countries) is critical for any joint military operation. The problem of the interoperability of military communication networks is in some respects similar to public safety but carries additional challenges like increased requirements for security (authentication and encryption), robustness in the presence of interference (jamming) and greater redundancy.

Interoperability problems also affect our daily lives. For example Verizon - arguably the most popular cellular network in the US - uses the CDMA standard, while GSM seems to
be the dominant standard used around the world. This makes most cellular phones sold by Verizon unusable in majority of other countries. The incompatibility of coding standards is additionally aggravated by the fact that there are at least four separate frequency bands used for cellular communications in the world. So even if a particular mobile handset supports given cellular network’s coding, it will still be incompatible if it cannot work in the required frequency band. There are specialized “international” phones that support all four frequency bands and most if not all cellular network protocols, but they are significantly more expensive than handsets designed for use with a single carrier.

2.2 Shortcomings of static reconfiguration schemes

Traditional reconfigurability schemes are based on the assumption that there are a finite number of broadly understood variables in the configuration of a particular system. The term “broadly understood” means the variables were introduced at the design time and are shared by all the communicating nodes. When such an assumption is true, then all choices of values for these variables can be accounted for algorithmically, although it might be difficult to do so if the number of variables is high. In many practical applications, though, this approach works reasonably well.

A very common example of such a (static) reconfigurable system is a modern network protocol stack. The very idea of the protocol stack is that software modules operating at a particular level of the OSI reference model can be selected in the protocol stack depending on the current state of the configuration and particular protocol. For example removing the Ethernet cable from a notebook’s network card socket and turning on the wireless network interface will deactivate a physical layer driver for the Ethernet interface and activate one for the wireless card. A request for a file from an FTP site will select the IP, TCP and FTP protocol modules [RFC959], while a similar request for a file from a Windows server would activate Server Message Block (SMB) Protocol, which depending on the local configuration, interacts directly with the TCP/IP protocol stack or with the NETBIOS transport module, which in turn works on any of the three transport/network stacks: NETBUI, Netware/IPX and also TCP/IP [SMB].

The network protocol stack reconfigurability based on a limited number of software modules responsible for implementing particular protocols works reasonably well for two reasons:

- The structures of the protocol stacks are well defined and although there are differences between the ones based on TCP/IP vs. those based on the OSI reference model, the functionalities of their respective layers are similar and can be mapped with relative ease.
- In recent years computing systems converged on only a handful of networking protocols. The reason behind this convergence was the desire to simplify the management of the network-related software, which used to be much more complex and much more prone to programming errors when a number of now defunct protocols (e.g. DECnet, Banyan, Novell) were still widely used.

The changes in networking standards don’t happen very often, but when they do they require major changes in the software. At a minimum, new drivers and configuration files
for the new protocols have to be installed. Frequently, other software modules also have to be updated to take advantage of the new capabilities. These kinds of changes are time consuming, they require taking the system offline and they always present an opportunity for introducing new errors in the software.

In addition to the problems with the upgrade itself, the newly upgraded system cannot take advantage of the new features unless the other systems it has to cooperate with have also been updated. For network-wide features it means that all nodes on the network have to be updated before one can take advantage of the new capabilities.

### 2.3 Reconfigurability of the physical layer as the key for interoperability

The functionality of modern communications systems is usually delineated into several layers of protocols. Such structure is used in TCP/IP [RFC1180] - arguably the most ubiquitous protocol stack and in the Open Systems Interconnection protocols recommended by ITU [X.200]. Each of the protocols at the particular layer use services of the lower layer protocols and provide services to the protocols at the higher layer.

In both TCP/IP and OSI protocol stacks the connectivity between two nodes is established in the physical layer, which is responsible for sending/receiving bits of data to/from physical medium, which in case of wireless devices is a radio frequency channel (i.e. a fragment of electromagnetic spectrum). It should be noted that even though the concept of protocol stack applies only to systems using digital communications, all the functionalities of purely analog communications system as signal modulation/detection, power amplification, radiating/receiving electromagnetic waves are also included in the physical layer of the digital system. One could argue that analog communications is done entirely in the physical layer.

Since physical layer is responsible for end-to-end communications in analog systems and for the connectivity for all other protocol layers in the digital systems, no interoperability problems can be solved without solving physical layer incompatibility first. In other words – without providing physical layer connectivity all other interoperability considerations are moot. The physical layer connectivity can be facilitated if the related hardware and software can be reconfigured. In fact the ability to reconfigure any protocol layer is so important from the interoperability point of view that one of the research projects led by Motorola Labs was named End-To-End Reconfigurability (E²R) [Bourse 2005]. This work thus concentrates mostly on the issues related to the reconfiguration of the physical layer.

National Task Force for Interoperability (NTFI) identified five key reasons why communications infrastructures of public safety agencies have problems with interoperability [NTIF 2003]:

1. Incompatible and aging communications equipment.
2. Limited and fragmented funding.
3. Limited and fragmented planning
4. Lack of coordination and cooperation
5. Limited and fragmented radio spectrum.

Any interoperability issues arising from limited and fragmented radio spectrum are highly relevant to the physical layer. The scarcity and fragmentation of radio spectrum complicates the design and deployment of the communications networks in densely populated areas where many services compete for limited amount of available bandwidth. In those circumstances the coordination of the frequency channel assignments is hard and services might require more complex radios to make use of the channels scattered across multiple bands. The channels assigned in the crowded bands might be more susceptible to interference from other services located in the neighboring frequencies.

Two wireless systems can be incompatible simply by design if they don’t share any protocols at one or more layers of the protocol stack. The aging can be a problem too - the newer generations of the communications equipment tend to implement more efficient state-of-the-art standards while dropping the old and inefficient ones. If a newer generation system is not specifically designed to be backwards compatible it might not be able to communicate with older radios.

The reconfiguration of the physical layer can be solved in many ways, depending on the design of the communications systems involved. In some situations the required solutions might not be viable from the technical and/or economical point of view. For example – if two legacy analog systems use different types of modulation (e.g. one uses FM and the other one uses AM), then theoretically the modulator modules in the radios of one of those systems could be replaced to make those radios compatible. In reality such hardware reconfiguration would hardly be practical as the cost of design, hardware and labor could easily exceed the residual value of those two legacy systems.

One of the most important features of Software Defined Radios is the relative ease of their reconfiguration. If the previously mentioned two analog systems used software defined radios, it would be a matter of reprogramming one of the sets of radios to use the kind of modulation used in the other one to make them compatible. Such reprogramming can be done for example by downloading a new binary file containing compiled code and rebooting the radio to force its processor to use the new code.

SDRs can be reconfigured in even more sophisticated ways than just a simple software download. For example if the SDR has enough internal storage a variety of versions of the software could be stored and booted on demand. Or an SDR could request and download a software module containing a particular protocol over the network as needed. Or in an even more sophisticated scenario an SDR could implement a variety of fundamental software blocks used in radio communications and use them to compose a particular waveform. If the need to adapt to another waveform arose, all the SDR would have to do would be reconfiguring its software by instantiating and connecting a new set of fundamental functional blocks to implement the new waveform.
All previously mentioned software-based methods for physical layer reconfigurability assume the SDR has a priori knowledge of the required waveforms. Such an assumption cannot be true in a general case; as it was said before, the aging of the equipment is one of the reasons for the incompatibility.

The key for the interoperability of the communications systems therefore is the reconfigurability that doesn’t require a complete set of a priori knowledge, but which can adapt and learn as it operates. A regular Software Defined Radio is not able to support such functionality for a simple reason that an SDR does not know its structure – the radio therefore must include a cognitive agent.

2.4 Requirements

The following set of features has been identified as required for a method for communications system reconfigurability:

- **Platform independence.** The reconfigurability is driven by the need for the interoperability among heterogeneous systems so any method adopted for that purpose has to be platform-independent.

- **Common language.** As in the case of two human beings, two communications systems can talk to each other only when they share the same language. Therefore any method for a truly interoperable reconfigurability has to include the selection of a common language.

- **Shared knowledge.** Even if two radios do share the same language (syntax), they might not be able to communicate unless they can interpret the conveyed messages in exactly the same way. In other words the semantics of any message passed between the nodes has to be uniformly interpreted among interoperating systems. This requires that an appropriate body of knowledge for particular domain of interest is identified, standardized and shared between all systems.

- **Knowledge extensibility.** A method that assumes a static body of knowledge cannot be fully interoperable as it cannot properly react as new, previously unknown concepts are introduced. Systems built upon such a method would require a system-wide upgrade every time a new waveform, modulation, coding etc. is to be supported. A truly interoperable system should be able to acquire new concepts from other systems, integrate these concepts into its knowledge base and use so gained knowledge in order to construct new, previously unknown to it, functionality.
3 Related research

3.1 Cognitive radio as a self-controlling software system

Cognitive Radio can be seen as one of the possible implementations of self-controlling software architecture. In a self-controlling software system part of the software constitutes the controlled system itself (called a plant in control theory), while the other elements of the software implement a controller which changes values of the control inputs of the plant and a quality of service subsystem, which computes values for the feedback loop (Figure 3-1) [Kokar 1999].

![Figure 3-1 A self-controlling software system. [Kokar 1999]](image)

In a Cognitive Radio the waveform application is the object of the control. The cognitive agent plays major roles in both - the controller and the QoS functionalities. It uses data gathered from the waveform and from the environment and makes control decisions based on the specified goal(s).

The self-controlling software model uses three levels of control: feedback, adaptation and reconfiguration. The first of the three levels is very similar to the feedback loop in a classical control system – the control software uses the values of both – inputs and outputs as well as the internal state to determine how to stimulate the plant in the next cycle. The feedback loop works well if the changes in the environment are small.

In case when the feedback alone is no longer sufficient to achieve the goals, the system might have to adapt the plant’s software to the new circumstances. The adaptation loop involves two software modules: a model estimator that maintains the model of the plant and a controller designer, which adjusts the controller’s parameters based on the plant’s model.

The third level of control – reconfiguration – is used when neither the adaptation nor the feedback can provide satisfactory results. The Reconfigurer module may use a specification database and/or a component database in the reconfiguration process in order to make structural (non-parametric) changes in the plant, the controller software and in the software responsible for the adaptation loop.

The cognitive radio might have one or more goals active at any given moment. The goals can be as simple as just maintaining communications with the peer node or as complicated as optimizing certain communications parameters (e.g. maximizing bitrate,
minimizing power consumption etc.) or adhere to some complex policies. Some of the goals contradict the others (e.g. the goal of maintaining the highest possible bitrate is usually detrimental to minimal power consumption) and it is the cognitive agent’s responsibility to arbitrate between them.

The cognitive agent calculates control inputs for the plant based on data gathered from the waveform as well as from the environment. The environment in this context encompasses everything external to the cognitive radio that is relevant to any of the goals, e.g. geographical coordinates, the level of noise in the communications channel, the multipath structure of the channel, characteristics of the other CR nodes the radio is communicating with, etc.

Any goals related to communications with other radios might require that the controller issue requests and/or queries to the other radios. For example if the node’s cognitive engine concludes that in order to optimize bit rate with the other radio, both nodes should switch to a different type of waveform, an appropriate reconfiguration request will be sent to the other radio and the local waveform will be reconfigured only after receiving confirmation that the other radio accepted the request.

This thesis deals only with the reconfiguration loop of self-controlling software. The feedback loop, and possibly the adaptation loop, within each radio is incorporated in the waveforms that the radios execute. The reconfiguration loop of any CR is shown symbolically in Figure 3-2. The intent of this figure is to stress the fact that CR nodes may need to include reconfiguration as part of their signaling, i.e., in the control of their functionality.

It should be noted that the run-time reconfiguration may lead to unwanted transient behaviors of the communication nodes. In particular, the reconfiguration can lead to problems with the stability of the whole system, as was identified in [Kokar 1999]. This issue, however, was not addressed in this thesis.

Mitola described the cognition cycle using a model based on the Observe-Orient-Decide-Act (OODA) feedback loop introduced by USAF Col. John Boyd [Boyd 1995]. Mitola
extended OODA by introducing two more processes to the control loop – Plan and Learn [Mitola 1999b]. This model, also known as ideal CR architecture (iCRA), is characterized by the Observe-Orient-Plan-Decide-Learn-Act (OOPDLA) loop (see Figure 3-3). The processes inherited from the original OODA model have the following interpretation in the context of cognitive radio:

- **Observe** – collect data about the environment using an array of sensors
- **Orient** – integrate the observations with policies, goals and preferences, hardware and software limitations and past experience
- **Decide** – identify the changes that need to be made in the SDR configuration in order to address the new situation
- **Act** – reconfigure the SDR.

The added processes are:

- **Plan** - responsible for planning decision for the long run
- **Learn** - supports machine learning based on past experience.

![Figure 3-3 The cognition cycle proposed by Mitola [Mitola 1999b].](image)

Although Mitola never made such a direct connection, it is quite obvious that iCRA with its OOPDLA loop fits very well into the self-controlling software system architecture. iCRA represents a very advanced vision of a Cognitive Radio. To the best of our knowledge, so far no system has implemented the full cognition cycle; this shows the magnitude of difficulties in the practical realization of this vision.

### 3.2 Language issues

The definition of cognitive radio (see section 1.2) implies that some kind of a computer language has to be used to express the knowledge a cognitive radio node possesses about itself and its communications environment. Additionally the radio has to be able to
prepare and send queries to other radios and respond to similar queries received from other nodes. The choice of the language has such a great impact on the overall design of the system that it has to be carefully considered not only from its functionality and expressiveness point of view but also from the point of view of the choice of available inference engines, required computational resources and the ability to interface with other programming languages.

This section highlights the most important issues related to the choice of language. One should refer to the article by Kokar and Lechowicz [Kokar 2009] for more details on this subject.

3.2.1 Imperative vs. declarative

The most basic criteria for classifications of computer languages is whether a language is imperative (procedural) or declarative.

A program written in an imperative language consists of a list of statements which are executed in the order determined by the order of the statements in the source code and the presence of flow control structures such as if-then-else statements, conditional and unconditional loops, etc. A program written in an imperative language is said to implement an algorithm. The examples of imperative languages include C/C++, Pascal, Basic, Fortran, Java and other.

Declarative languages use a different approach. Instead of instructing the computer how to achieve the desired result, a declarative language specifies what to achieve. It is the generic inference engine processing the declarative language program that decides how to execute a program to achieve the goal. A program written in a declarative language is thus a collection of facts (clauses) with a goal (or query). The inference engine tries to find the solution to satisfy the goal.

While from the input/output point of view both types of languages behave similarly i.e. they take some data on input and produce some data on output, their properties are quite different when it comes to the ways they can be modified.

Changes to the program written in an imperative language usually involve the replacement of the program and have to be made offline at the source code level. The source code has to be recompiled, the old version of the program taken offline, new version loaded and the system restarted.

In case of the program written in a declarative language usually only the rule base has to be changed, which can be done by adding or removing clauses to/from the rule base while the program is running. Online modification is possible because the inference engine never changes and changes to the clauses can be treated almost like changes to input data.

The above difference in how program updates are made is not definitive, as there are many exceptions, which blur the line. For example Java is an object-oriented, imperative language, but it is possible to modify its compiled code at runtime. On the other hand, a compiled Prolog program cannot be modified at runtime despite being a (primarily)
declarative, logic language. In general however programs written in declarative languages tend to be easier to modify at runtime than ones written in imperative languages.

In order to consider which type of language is better suited for cognitive radio, one has to recall previously stated characteristics of a cognitive radio (section 1.2) i.e. self-awareness, the ability to tell what it knows and what it wants, the ability to draw conclusions from the facts it learned and the ability to react to situations it has not seen before.

It appears that a declarative language should be able to satisfy all of those requirements. In particular it is definitely easier for the radio to express its knowledge in a declarative language rather than procedural as it is very easy to express ‘what’ without having to specify ‘how’.

The ability to react to a surprise also favors declarative languages. Programming in an imperative language implicitly assumes that all possible valid combinations of input parameters are accounted for in the algorithm. If they are not, the behavior of the program is undefined, which might lead to situations where the program fails to react at all or suffers a catastrophic failure (crash).

A system written in a declarative language can adapt by incorporating the knowledge it gained during the program execution with the rule base effectively modifying its program. It can also find answers to unexpected goals or queries. It should be emphasized that the assumption that the program can add to its rule base facts expressed in the declarative language implies that the language has computer processable semantics, i.e. that the language belongs to the group of formal languages (i.e. languages with formal syntax) with formal semantics. Such a language together with inference rules and the inference engine constitutes a formal system.

An inference engine of the formal system can apply inference rules to the set of sentences in the language to derive new sentences. A formal system should be sound which means that when given a consistent set of true sentences it can only derive true sentences from that set.

Even though it appears that declarative languages are better suited for cognitive radio applications, one should not forget that the great flexibility of such languages comes with a price. The worst case time complexity of inference in most of the declarative languages falls into the undecidable complexity class [Enderton 2001]. The consequence of this is that given such a physical restriction as the available memory or time allotted for the computation, the inference engine may return an “unknown” answer if any of the restrictions have been violated.

### 3.2.2 Web Ontology Language (OWL)

Ontology is a term that originally used to refer to a branch of philosophy concerned with the concept of being. Researchers in the Artificial Intelligence field started to use this term to describe an approach to knowledge representation.

In this context many definitions of what ontology is have been proposed. Gruber defined it as “an explicit specification of conceptualization” [Gruber 1993]. While this definition gained some popularity due to its conciseness it has been criticized as being vague especially with respect to the interpretation of the term conceptualization [Guarino 1995].
Cocchiarella defined *formal ontology* as “the systematic, formal, axiomatic development of the logic of all forms and modes of being” [Cocchiarella 1991]. While as Guarino noted, the interpretation of the term *formal ontology* can be debated [Guarino 1995], this definition is important for two reasons – it specifies that the ontology has to be formal (i.e. developed in a rigorous way) and that the ontology is concerned with all forms and modes of being. The latter implies that an ontology may include not only entities related to the world of interest, but also abstract, meta-level concepts that can be used to model this world.

Uschold and Jasper proposed a much more detailed definition of ontology, which is perhaps the best description of how the term *ontology* is understood in this work:

“An ontology may take a variety of forms, but it will necessarily include a vocabulary of terms and some specification of their meaning. This includes definitions and an indication of how concepts are inter-related which collectively impose a structure on the domain and constrain the possible interpretation of terms” [Uschold 1999].

In the light of the above definitions, the ontology of a particular domain of interest is just knowledge of this domain represented in some formal way. It specifies the concepts of the domain, attributes of those concepts and relationships among them.

In recent years the AI community has reached a consensus that a common language for representing ontologies is desirable. The Web Ontology Language (OWL) seems to be the one most widely used for that purpose - particularly in projects related to Semantic Web.

As it was mentioned earlier, an ontology for a domain specifies the concepts of the domain, their attributes and relationships between them. A very simple ontology related to the cognitive radio domain is shown in Figure 3-4.

![Figure 3-4. A simple ontology – Component and Ports.](image)

In OWL concepts are specified as *classes*, and are interpreted as sets of *individuals*. Typically, classes are represented graphically by rectangles. For instance, in this 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 this 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 Figure 3A4. 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) with specifications of domains and ranges of all the properties. It also states that some of the classes satisfy the *disjointWith* property, i.e., they have no individuals in common. All the classes, sub-classes and properties in this listing can be directly traced to Figure 3-4.

```xml
<?xml version="1.0"?>
<rdf:RDF
  xmlns="http://www.LeszekLechowicz.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#"
  <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>
</rdf:RDF>
```
To illustrate the use of OWL in the software radio domain, consider a simple component - Quadrature Modulator (QM) shown in Figure 3-5.

![Figure 3-5. A simple component (Quadrature Modulator).](image)

In order to describe the QM component, the ontology needs to have additional concepts. In particular the notion of a Component – which might be either a simple building block (BasicComponent) or an entity composed off many BasicComponents (Module) has to be formalized. Thus in addition to concepts introduced in Figure 3-4 the concepts shown in Figure 3-6 have to be added to the ontology.
The QM module shown in Figure 3-5 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 this figure, each of the basic components has at least one input port and an output port of its own.

In Figure 3-7, a graphical representation of a partial description of an individual of the QuadratureModulator class is shown. This representation is the result of the “export” operation of the Ontoviz plugin to the Protégé ontology editing tool. The arrows annotated with ‘io’ (short for ‘individual of’) represent the rdf:type relationships 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 [Preuveneers 2005].
When the cognitive radio doesn’t know a particular component (e.g., QuadratureModulator) it may send a query to the other radio for the description. It should be able to prove using its reasoner that the individual it constructed (let’s call it “MyComponent”) is indeed of the requested type. In other words it should be able to prove that the following OWL statement is true:

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

The radio has to be able to describe a module as a composition of other modules or basic components without being specific about particular individuals. For instance, a component type might be specified as

```
MyComponentType(?X) :- hasComponent(?X,?Y) ∧ hasComponent(?X,?Z) ∧ connectedTo(?Y, ?Z)
```

Unfortunately, the expressive capabilities of OWL may not be sufficient to express all such specifications. As shown in [Krötzsch 2011], even with property chains introduced in OWL2 [OWL2 2009c], OWL2 is still unable to express certain classes of relations – e.g. relations between individuals referenced by properties. To show this, [Krötzsch 2011] uses the example of the concept of a child of married parents, which cannot be expressed in OWL.

```
childOfMarriedParents(?X) :- hasParent(?X,?Y) ∧ hasParent(?X,?Z) ∧ married(?Y, ?Z)
```

### 3.2.3 Rules

The expressive limitations of OWL can be resolved by using other, more expressive languages and more powerful reasoning mechanisms.

Other languages, (e.g. Prolog) allow 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:

```
¬A₁ ∨ ¬A₂ ∨ …. ∨ ¬A_n ∨ B
```

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

```
A₁ ∧ A₂ ∧ …. ∧ A_n ⇒ B
```

The symbols used in these two formulas represent negation (¬), logical OR operation (∨), logical AND operation (∧) and logical implication (⇒). In operational terms, this rule states that whenever A₁, A₂, …., A_n hold, so does B. Using Prolog notation, this formula can be represented as:

```
B :- A₁ , A₂ , .... , 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
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 the previous example, one could create a description of the \textit{QuadratureModulator} class by using rules (with variables). The definition could include the predicate \textit{hasQMConnections} as shown below ("?" as the first character in a name indicates that it is the name of a variable):

\[
\text{hasQMConnections}(Q) :-
\begin{align*}
\text{Module}(Q), & \ \text{hasSubComponent}(Q, M1), \\
\text{type}(M1, \text{Multiplier}), & \ \text{hasInputPort}(M1, \text{InPortM1}), \\
\text{hasInputPort}(Q, I), & \ \text{isConnected}(I, \text{InPortM1}), ..., \\
\text{hasSubComponent}(Q, M2), & \ \text{not}(M1 == M2), ...
\end{align*}
\]

This is not a complete rule. The intent 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).

Once the ontology is extended with this kind of rules, the inference engine can verify that all the necessary connections are in place and thus a given structure is in fact a \textit{QuadratureModulator}. In the experiments described in [Lechowicz 2006] the BaseVISor inference engine was used. BaseVISor is a forward-chaining inference engine developed by VIStology, Inc.\(^1\) It is based on the Rete network algorithm optimized for the processing of RDF triples, it also incorporates axioms and consistency checks for R-entailment which supports all of the RDF/RDFS and a part of OWL-DL semantics. In exchange for the price of not supporting all of the OWL-DL elements, BaseVISor provides P-SPACE performance for ground RDF graphs [Matheus 2005].

The need to extend the expressivity of OWL by adding rules has also been recognized in [Poston 2005], [Ginsberg 2006] and [Ginsberg 2007], where both OWL and rules were used to express spectrum access policies. In these experiments, Jena\(^2\) was used as an inference engine for reasoning over both the ontology and the rules.

Describing certain relationships in a rule language can also be more natural and more convenient than trying to stay within the confines of OWL at all cost. For example one could use the minimum cardinality restriction to express the fact that there are two distinct individuals of type \textit{Multiplier} associated with the class \textit{QuadratureModulator}. The same effect is achieved by adding a clause in the \textit{hasQMConnections} rule that states:

---

\(^1\) http://www.vistology.com/

not(?M1 == ?M2). The latter is arguably easier to understand when a person who did not write the rule is trying to understand how it’s supposed to work.

### 3.2.4 Negation and open vs. closed world reasoning

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) independently of the 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*.

It should be mentioned that OWL2 made asserting negative facts easier by providing syntactic sugar in the form of negative object and data property assertions. The same effect was achievable in OWL1 by adding in the class constructor a complement of the class defined by the positive assertion of the property. For example the OWL2 statement (in OWL functional syntax rendering):

\[
\text{NegativeObjectPropertyAssertion(:isMarriedTo :Bob :Debbie)}
\]

is equivalent to the OWL1 construct:

\[
\text{ClassAssertion(ObjectComplementOf(ObjectHasValue(:isMarriedTo :Debbie)) :Bob).}
\]

Some of the rule languages 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 *non-monotonic*. 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. E.g., 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 [Kifer 2007]. If only one type of negation is admitted in a formal language, there can be difficulties related to the modeling of the various situations and to the reasoning about them using automatic inference engines. For instance, if one accepts only logical negation, one 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 the CWA is accepted, 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 knowledgebase that it does, which is rather unlikely that a node would have a complete knowledge about all the nodes it communicates with).
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 underway to achieve such a goal [Motik 2006]. 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 the structure of radio components. A complete knowledge of a component must include both what sub-components 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 Figure 3-5. The description of the QuadratureModulator component must express the fact that an individual of InputPort is connected to exactly one multiplier. It can be captured by two rules that define the isExclusiveInputPortOf(?P,?C) and isNonExclusiveInputPortOf(?P,?C) predicates.

\[
isExclusiveInputPortOf(?P,?C) :-
\quad \text{inputPortOf}(?P,?C), \text{not}(\text{isNonExclusiveInputPortOf}(?P,?C)).
\]

\[
isNonExclusiveInputPortOf(?P,?C) :-
\quad \text{inputPortOf}(?P,?C), \text{inputPortOf}(?P,?D), \text{not}(?C=?D).
\]

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 such a deduction could not be made, 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 the closed world model had to be assumed. 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 exclusive) may be negated, leading to an inconsistency (both the facts that the port is exclusive and is not exclusive in the same knowledgebase). The reasoner must ensure that such a situation does not take place. One way to avoid this could be achieved by a consistency checker (a part of the reasoner) monitoring for such non-monotonic changes. In case the consistency check failed the reasoner would have to retract the facts that have been derived through the use of NAF.

### 3.2.5 Functional equivalency

Augmenting OWL with a rule language enables Cognitive Radio nodes to exchange information, e.g. 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 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 [Lechowicz 2007]:

- 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.
- The same functionality using two different data types is seen as two different structures as the structure-based approach doesn’t allow for easy abstraction of the functionality from the data type.
- The structure-based approach doesn’t 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 (Figure 3-5) expressed in terms basic components and rules might not be able to realize that an alternative, more efficient implementation of such structure might exist, e.g. one that uses an available specialized Multiply-Add hardware unit (Figure 3-8).

Figure 3-8. A composite module functionally equivalent to QuadratureModulator in Figure 3-5.

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 the ones 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, they do not provide
capabilities of expressing equivalence of functions. In order to address those issues a more expressive language has to be selected.

Quantification over functions lies in the realm of second-order logic [Shapiro 2000], which in turn is extended by higher-order logic [Brown2007]. While functions are at least partially covered by a number of declarative languages, for this experiment a higher-order logic language Metaslang was chosen. Metaslang is the language of the Specware tool [McDonald 2001]. The main reason for the selection of this language was that it supports composition, using constructs of category theory like morphism and colimit [Smith 2006]. Specware also integrates two theorem provers (Snark [Stickel 1994] and Isabelle [Nipkow 2002]) 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. Pavlovic and Smith [Pavlovic 2003] 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, first some basic elements and concepts of the language need to be introduced. Additionally, the following example shows how the composition of specifications works in Metaslang. It also shows 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.

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 = true
  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 PartialOrder 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 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. Finally, it presents some of the axioms that define these three operations.

```plaintext
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.

```plaintext
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 Metaslagn, make 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.

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 intention of the above discussion was to show an example of the capability of reasoning about functions. While Metaslang was chosen to carry out the experiments, this goal could be achieved in a number of other declarative languages if reasoners for them were available. One of the possible candidates is Common Logic (CL), which recently has become a standard of ISO/IEC [ISO24707]. According to [ISO24707], 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 intentional 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” [NISTIR7310], [Hayes 2005]. 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. At the time of writing however no inference engine supporting CL was found.
3.2.6 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 [Padulo 1974]. This limitation has been long recognized and as a result the concept of monads [Wadler 1990] has been used to address this problem. The main idea is to use monads 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 it 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.

Note that by chaining the various predicates indicated above (marked by the italic font), 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, or LTL for short [Manna 1991]. 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 $X \varphi$ ($\varphi$ must hold at the next time instant); $G \varphi$ ($\varphi$ must hold on the entire subsequent time); $F \varphi$ ($\varphi$ will hold eventually, in the future); $\psi U \varphi$ ($\varphi$ holds at the current or a future position, and $\psi$ has to hold until that position. At that position $\psi$ does not have to hold any more.); $\psi R \varphi$ ($\varphi$ is true until the first position in which $\psi$ is true, or forever if such a position does not exist. I.e., $\psi$ releases $\varphi$.) 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 [Manna 1995], where properties of systems such as reachability, safety, deadlock are proven in a formal way. The approaches used in this process include deduction (theorem proving), model checking and proof-carrying code. Since deductive inference for this domain is undecidable, model checking has been used more often to verify that a property holds in a given model [Holzmann 2004]. Some approaches use a combination of the two methods, e.g., see [Bjorner 2000].

Proof-carrying code (PCC) [Necula 1997] is a mechanism by which a host system can verify with certainty that a program received from an untrusted code producer exhibits behavior that complies with the previously established set of safety rules. These rules (also known as safety policy) are chosen in such a way that they guarantee the safe behavior of programs. The code producer is required to create a formal safety proof that confirms that the code abides by the safety rules. The safety proof is attached to the program and the receiving host uses a simple and fast proof validator to check that the proof is valid and that the code is safe to execute. There’s no requirement for establishing trust between the nodes as the positive validation of the safety policy proof guarantees that the code is safe to execute.

The exact verification scenario is not discussed here but it is assumed that the creation of the proof would be done off line 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, only one aspect of temporal reasoning is discussed here - the one that is captured by the \( X \) operator of LTL. However, instead of implementing a temporal logic, an approach that is typical in engineering is followed. Towards this aim, the concept of unit delay is formalized and then properties of systems composed of components that include delays proven. In particular, unit delay allows 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, first the function UnitDelay.Func whose domain and range is the type Sample is defined. Then two commutativity axioms for this operation are stated - 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 the quantifying is done over all functions with domain Sample and range Sample. 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 \to Sample

axiom UnitDelay_commutativity is
   fa( f:( Sample\toSample ), x:Sample )
     UnitDelay.Func( f(x) ) = f( UnitDelay.Func(x) )

axiom UnitDelay_commutativity2 is
   fa( f:( Sample*Sample\toSample ), x:Sample, y:Sample )
     UnitDelay.Func( f(x,y) ) = f( UnitDelay.Func(x), UnitDelay.Func(y) )
endspec

The “ideal” Adder and Multiplier components (i.e., components without any delay) were specified in section 3.2.5. In this section the delays introduced by these two components are defined. As one can see below, the specs define that they introduce a one step delay.

AdderDelay = spec
import UnitDelaySpec

op Adder.Func: Sample*Sample \to Sample
def Adder.Func(x,y) = UnitDelay.Func( Sample.add( x, y ) )
endspec

MultiplierDelay = spec
import UnitDelaySpec

op Multiplier.Func: Sample*Sample \to 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 FPGA). 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 (Snark) 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 one is able to express time dependencies between different components of the system without introducing time explicitly. Second, 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 [McDonald 2007]. 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. Increased computational efficiency that can sometimes be achieved through imperative programming as well as more direct connection to common programming languages were additional motivations.

In Accord the behavior is encapsulated in a module. Module 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-spects). 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.

### 3.3 Component-based software systems

Component-based software engineering is a well-known software development paradigm, which emphasizes the separation of concerns through modularity and encapsulation. Such a modular approach seems to fit the needs of SDR systems very well. This section highlights some of the research efforts in this area.

#### 3.3.1 Process algebras

Process algebras, also known as process calculi, are a family of related formalisms used for modeling concurrent systems. They provide tools for high-level description of interactions, communications and synchronizations among the processes. Through algebraic laws they allow for the manipulation of the processes’ descriptions and formal reasoning about their equivalence.

Among the many formalisms proposed, two gained widespread recognition – Robin Milner’s Calculus of Communicating Systems (CCS) [Milner 1980] and C. A. R. Hoare’s Communicating Sequential Processes (CSP) [Hoare 1984]. Both systems were initially developed independently, but since they were being developed during the same time, they cross-pollinated each other with the ideas, making them very similar in some respects.

In CCS, interactions between processes are represented as passing messages through communication channels. Each process has a well-defined process interface, i.e. a set of input and output channels through which the process communicates with other processes. The following drawing shows an example of a simple interactive system consisting of two processes: CS – modeling a computer scientist and CM – modeling a coffee machine. [Aceto 2007].

![Interacting processes CM and CS](Aceto 2007)

In the above diagram, the underlined labels are used for outputs and not underlined for the input of the process. Using CCS notation, the CM process is defined as:

$$CM = \text{coin.coffee.CM}$$
which means that the CM process waits for the input \textit{coin}, then outputs \textit{coffee} and then returns to the initial state to wait for another \textit{coin} event.

Similarly, the CS process in the diagram may be defined as:

\[
CS \overset{\text{def}}{=} pub.coin.coffee.CS
\]

which in this example has an interpretation of an academic who publishes a paper, then drops a coin to the coffee machine, drinks the coffee and then starts working on another publication.

CCS defines the following set of fundamental expressions:

\[
P, Q ::= K \mid \alpha.P \mid \sum_{i \in I} P_i \mid (P | Q) \mid P[f] \mid (P \setminus L)
\]

where

- K is a process name
- \(\alpha\) is an action
- I is an index set
- \(\sum P_i\) is a choice of process among \(P_i\) processes; only one of the processes is selected
- \(P|Q\) is two processes \(P\) and \(Q\) running in parallel
- \(P[f]\) is a process in which the action labels have been renamed by some relabeling function \(f\)
- \(P\setminus L\) is a process without a set of labels \(L\)

One of the most interesting applications of CCS is proving the behavioral equivalence of two processes. For this purpose the processes are modeled as labeled transition systems (LTSSs) by explicitly describing their states and their transitions from state to state. A \textit{trace} of a process \(P\) is a sequence of actions \(\alpha_1\ldots\alpha_k\) such that there is a sequence of transitions \(P_0\ldots P_k\) such that

\[
P = P_0 \xrightarrow{\alpha_1} P_1 \xrightarrow{\alpha_2} \ldots \xrightarrow{\alpha_k} P_k
\]

for some \(P_1\ldots P_k\).

Let \(\text{Traces}(P)\) denote a collection of traces of \(P\) i.e. all possible finite sequences of interactions with the process \(P\). If process \(Q\) is behaviorally equivalent to process \(P\) then \(\text{Traces}(Q) = \text{Traces}(P)\). The equality of traces does not however guarantee the behavioral equivalence of the processes. The behavioral equivalence can only be determined by proving that the processes are related by a \textit{strong bisimulation} relationship and that some additional conditions are met [Aceto 2007].

Process algebras found applications in software for formal specification, modeling and verification of multi-process, parallel systems. These tools are used in mission critical
software systems where failure of any component would have disastrous results (e.g. nuclear plant control systems, spacecraft flight control, avionics). CCS based tools are particularly useful in evaluating the software with respect to possible deadlocks or livelocks.

### 3.3.2 Software Communications Architecture (SCA)

SCA is an open architecture framework designed to fit the specific requirements of software defined radios in which hardware and software components have to cooperate in harmony. SCA was initially developed to support the needs of Joint Tactical Radio Systems (JTRS) program run by the US military. Wireless Innovation Forum is currently working on an international standard based on the SCA.

SCA has been designed to provide a specification for the “deployment, management, interconnection and intercommunication of software in embedded, distributed-computing communications systems” [SCA 2006].

SCA uses Common Object Request Broker Architecture (CORBA) framework [CORBA 2008] for software composition. SCA is built upon the Core Framework (CF) – a set of application layer CORBA interfaces, which provide an abstraction layer for the underlying system software and hardware.

The following table contains the list of interfaces the Core Framework provides: [SCA 2006]

<table>
<thead>
<tr>
<th>Class</th>
<th>Interface</th>
<th>Provided Functionality</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Application Interface</td>
<td>Port</td>
<td>The management and control interfaces for all system components.</td>
</tr>
<tr>
<td></td>
<td>Life Cycle</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Testable Object</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PropertySet</td>
<td></td>
</tr>
<tr>
<td></td>
<td>PortSupplier</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ResourceFactory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Resource</td>
<td></td>
</tr>
<tr>
<td>Base Device Interface</td>
<td>Device</td>
<td>The management and control of hardware devices within the system through their software interfaces.</td>
</tr>
<tr>
<td></td>
<td>Loadable Device</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ExecutableDevice</td>
<td></td>
</tr>
<tr>
<td></td>
<td>AggregateDevice</td>
<td></td>
</tr>
<tr>
<td>Framework Control Interface</td>
<td>Application</td>
<td>The instantiation, management and destruction/removal of software from the system.</td>
</tr>
<tr>
<td></td>
<td>ApplicationFactory</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DomainManager</td>
<td></td>
</tr>
<tr>
<td></td>
<td>DeviceManager</td>
<td></td>
</tr>
<tr>
<td>Framework Services Interface</td>
<td>File</td>
<td>Storage related support functions and services.</td>
</tr>
<tr>
<td></td>
<td>FileSystem</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FileManager</td>
<td></td>
</tr>
<tr>
<td></td>
<td>FileManager</td>
<td></td>
</tr>
</tbody>
</table>

The communications software (i.e. the waveforms) is the “application” software in the SCA architecture and it implements Base Application Interfaces. The software components that provide access to the system hardware resources are referred to as
“devices” and they implement Base Device Interfaces. Non-hardware-related (software only) resources provided by the system for applications are called “services” - SCA does not define any interfaces a service have to implement.

The structure of software layers in the SCA architecture is shown in Figure 3-10. An “application” (a waveform) usually consists of multiple software components that are loaded on the top of the CORBA middleware providing Framework Control (FC) and Framework Service (FS) interfaces. The application software may access a subset of the operating system services that is enumerated in the Application Environment Profile (AEP). Typically AEP is a subset of Portable Operating System Interface (POSIX) specification.

![Figure 3-10 SCA Architecture Layer Diagram](SCA 2006).

SCA architecture defines several software components that always have to be present in the system in order to provide platform-wide services. The DomainManager contains knowledge of all software components installed in the system, including references to their locations in the file system, device drivers and all applications. The DeviceManager has knowledge of all devices and services instantiated through it. At least one DeviceManager has to be present; the system however may have more than one of them. They all have to be registered so that the DomainManager always has a full picture of the available devices and services.
Figure 3-11  SCA Management Hierarchy [SCA 2006].

SCA defines a set of XML files collectively referred to as the Domain Profile in order to describe the attributes of the services, devices and applications installed on the system. All CORBA elements are described by Software Package Descriptors (SPD) and Software Components Descriptor (SCD) files. The SPD provides a software package level metadata such as the identification, the name of the binary files, dependencies on other packages, etc. The SCD file defines CORBA interfaces supported by a component.

Since applications are usually composed of multiple software components, there’s a Software Assembly Description (SAD) file associated with each application. The SAD references all packages required for the application, defines all the connections between application components including the connections to the devices and services and identifies a single component within the application as the assembly controller.

It should be emphasized that SCA is a framework designed for making it easy to “plug-in” the software components into the SCA-compliant system. It is however neither system specification nor implementation architecture. A particular SCA implementation architecture is the result of combining the requirements of SCA-compatible platform with the needs of particular domain and the project technical specification.

3.3.3 The SpecC methodology and language

The SpecC is a design methodology and a System Design Language developed in the Center for Embedded Computer Systems at University of California, Irvine. It is a tool for designing embedded systems, where functionality is spread among hardware and software components. In contrast with traditional design methodologies, where frequently the hardware and the software parts of the embedded system are designed separately, SpecC allows to design and model all components together.

Although the purpose and goals of the SpecC language and methodology are quite different from the communications systems reconfiguration, some of the ideas used in their design could be used in the design of any component-based software system. The
hierarchical structure of behaviors, communications channels and their inlining, behavior synchronization, pipelining and the representation of state machines are the examples of features that could be used in other systems based on software composition.

### 3.3.3.1 The SpecC model

One of the major sources of problems in the traditional embedded system design methods is the lack of separation between the computational part of the software and the part responsible for communications. The computational part of the software is where the actual “business” logic of the system resides. The communications software, although vital for the system’s operation, should play the subservient role to the computation layer and should be easily replaceable as the system evolves.

The SpecC model makes a clear distinction between the communication and computation. The computation is encapsulated in one or more behaviors. The behaviors are connected through communications channels. The diagram in Figure 3-12 shows the difference between the traditional and SpecC models. In the traditional model processes P1 and P2 communicate with each other through three shared variables s1 ... s3. By assigning/reading values to/from those variables the communication software within the process P1 communicates with its counterpart in the process P2. P1 and P2 necessarily have to include both computational and communication software.

![Figure 3-12 Traditional vs. SpecC model [Dömer 2002a].](image)

The SpecC separates the communications mechanism and encapsulates it in the communications channel C1 with send and receive routines. The internal implementation of the protocol (variables v1 ... v3) is no longer visible to any of the communicating behaviors (B1, B2) so if the need arises the communications channel can easily be replaced with another one with a compatible interface.

The behavior and channel model is maintained throughout the design process in order to make the changes easy. In the final stage of implementation however the channels are inlined back to variables wherever possible to make the code as efficient as possible (Figure 3-13).
In order to make the reuse of existing intellectual property (IP) modules, SpecC provides an adapter model in which the legacy IP module is connected through a legacy communications means to an adapter module, which from the opposite side presents the interface of a communications channel. An additional component called transducer is necessary for the combined adapter and IP modules in order to present the same interface as a standalone behavior B. The need for the transducer come from the fact that channels are passive components in SpecC model and as such cannot be connected directly.

The additional features supported by SpecC include IP channel encapsulation, hierarchical channels, inlining of hierarchical channels etc. One should refer to the publications listed on CECS’s web site for more details on SpecC model.

The SpecC methodology uses a 4-level design cycle. Each of the levels of concentrates on different aspects of the system design [Gerstlauer 2001].

1. **Specification Model** is the highest level model of the system. This model is used for verifying system’s algorithmic functionality with no timing constraints.
2. **Architecture Model** – in this model the top level of behavior hierarchy is defined and the time of execution estimated.
3. **Communication Model** – this model is concentrated on bus allocation, protocol selection, channel partitioning, protocol, transducer insertion and inlining. At this level of the design the delays of the components are also estimated.
4. **Implementation Model** – this model includes exact timing information and is the basis for the actual implementation of the system including hardware interface and hardware FSM synthesis, software interface synthesis and code generation.

### 3.3.3.2 The SpecC language

The SpecC language has been designed with the following goals in mind [Dömer 2002a]:

- **Executability** – the ability to compile and execute the model at any level of the system design is crucial from the simulation point of view.
- **Synthesizability** – only constructs that can be translated into both software and hardware implementations can be supported.
- **Modularity** – for ability to decompose behaviors hierarchically.
- **Completeness** – the language need to support all idioms found in embedded systems such as concurrency, hierarchy, synchronization, exception handling, timing and explicit state transitions.
- **Orthogonality** – is desirable so that orthogonal concepts found in the embedded systems can be described by independent language constructs. Orthogonality together with simplicity makes the language easy to learn and understand.

The syntax of the SpecC language has been derived from ANSI C. It is in fact a superset of ANSI C so each program written in ANSI C has valid syntax in SpecC. The SpecC requires that at least a single behavior is defined in the system, therefore the SpecC version of the famous “Hello, world!” program looks as below:

```c
#include <stdio.h>
behavior Main
{
    void main(void)
    {
        printf( "Hello, world!" );
    }
}
```

**Figure 3-15 "Hello, world!" program in SpecC.**

In order to support concepts not present in ANSI C, additional keywords have been introduced. Some of those keywords are listed below. The reader should refer to the SpecC language specification [Dömer 2002b] for full description.

- **behavior** – defined a behavior component
- **interface** – defines the communications channel interface
- **channel** – defines a communications channel that implements the interface
- **implements** – see above
- **out** – denotes an output parameter of the behavior
- **in** – denoted an input parameter of the behavior.
- **par** – defines a block in which child behaviors are run in parallel
- **fsm** – defines a state machine
- **pipe** – defines a pipelined execution of child behaviors
• event – defines a synchronization primitive
• wait – makes the behavior wait for an event
• notify – sends an event
• try – defines a block in which the execution of the child behavior is terminated if any of the events specified in the trap or interrupt block that follows it happens
• trap – defines a block that is executed on the termination of the behavior specified in the preceding try block.
• interrupt – defines a block that is executed when the execution of the behavior specified in the preceding try block is interrupted due to the arrival of the event. After the execution of the interrupt behavior is finished, the behavior in the try block resumes.

The following contains a partial listing of a sample project available at CECS’ web site [Shin 2001].
This example implements a simple parity checker (behavior parity), which internally runs two child behaviors in parallel – even and ones. The child behaviors use notify and wait statements to synchronize the flow of data between them.

```c
behavior parity(in unsigned int Inport, out unsigned int Outport,
               in event Start, out event Done)
{
    unsigned int data, ocount;
    event istart, idone;

    even U00(Inport, Outport, Start, Done, data, ocount, istart, idone);
    ones U01(data, ocount, istart, idone);

    void main(void) {
        par {
            U00.main();
            U01.main();
        }
    }
}

behavior even(in unsigned int Inport, out unsigned int Outport,
               in event Start, out event Done, out unsigned int data,
               in unsigned int ocount, out event istart, in event idone)
{
    void main(void) {
        unsigned int mask;

        while (1) {
            wait(Start);

            data = Inport;
            mask = 0x0001;

            notify(istart); // start one's counter
            wait(idone);    // wait for the result of one's counter
            Outport = ocount & mask; // even parity checker
            notify(Done);
        }
    }
}
```
behavior ones(in unsigned int inport, out unsigned int outport,  
in event start, out event done)
{
    void main(void) {
        unsigned int data;
        unsigned int ocount;
        unsigned int mask;
        unsigned int temp;

        while (1) {
            wait(start);
            data = inport;
            ocount = 0;
            mask = 1;

            while (data != 0) {
                temp = data & mask;
                ocount = ocount + temp;
                data = data >> 1;
            }

            outport = ocount;
            notify(done);
        }
    }
}

Figure 3-16 Partial listing of parity checker SpecC code.

3.3.4 Functional Description Language

Functional Description Language (FDL) is an XML-based language designed for specifying signal processing functionalities. The development of FDL was one of the projects within End-To-End Reconfigurability (E^2R) initiative led by Motorola. When introducing the FDL in their paper, Dolwin and Zhong [Dolwin 2007] proposed a scenario in which a user is trying to establish a VOIP phone call on his laptop through his wireless LAN (WLAN) provider’s hot spot at the airport but in this particular location the WLAN access point doesn’t support WCDMA waveform. The network operator can then send a query to the laptop whether the wireless card can support a dual mode configuration i.e. WLAN and GPRS waveforms at the same time. The query is in the format of FDL document that is interpreted by the laptop’s Configuration Control module (CCM). The CCM detects that some extra software objects (Signal Processing Modules – SPMs) are required to support the requested functionality. The network operator then evaluates the overhead related to downloading the software modules and then instructs the CCM to download and install them in the local database. The CCM downloads the SPMs from the manufacturer’s database, installs them on the hardware without stopping the current WLAN configuration and then once the hardware programming is complete it is instructed by the network operator to start executing the new configuration i.e. WLAN + GPRS. At this point the user should be able to establish the VOIP session over the GPRS link.

FDL allows for the description of signal processing algorithms in a form of hierarchical flow of signals (data and control) between functional blocks (processes) communicating via one-to-one and one-to-many channels. The top level language element is Alg_Algorithm which is used to describe the signal processing chain. Alg_Algorithm is broken down into subcomponents subProcess and subChannel. A process and a channel
are linked via *inputPort* or *outputPort* objects. Each port specifies the type of data that passes through it. Data types include both application and control types. Each of the processes can be optionally constrained by the deadline value, which specifies the maximum point of time that process can finish its processing relative to the start of the algorithm.

The following diagram presents a graphical representation of the FDL description of a sample protocol [Dolwin 2007].

![Figure 3-17 FDL description of IEEE 802.11a OFDM waveform [Dolwin 2007].](image)

The FDL is probably the first real effort towards transferring waveform descriptions through the network and using these descriptions to compose waveforms locally from simpler signal processing modules. FDL has some really weak points though. One of them is the requirement for a network operator and the fact that it is network operator’s responsibility to maintain dialog with all the nodes on its network and to decide whether a node requires a reconfiguration or not. Another weak point is the requirement that SPMs must be downloaded from the manufacturer’s database if they are not already present in the node’s local database. The available papers related to FDL don’t mention anything about how the SPMs would be indentified and versioned and how to guarantee that the functionality of a particular SPM is implemented in the uniform on the many different hardware and software platforms.

Unfortunately with the end of E²R initiative all research activities in the FDL project stopped and it is unclear whether it might resume in the future.
3.3.5 The use of OWL in the composition of software services

Preuveneers and Berbers in their paper related to alleviating certain non-functional concerns with pervasive services [Preuveneers 2005] propose a method for composing software services out of simpler components using Web Ontology Language (OWL) for the service description.

For that purpose they created an ontology of concepts related to service composition. Elements such as components, ports, messages, parameters, connectors etc. are defined as OWL classes. Since the meta-model concepts were encoded as OWL classes, the actual description of a concrete service could only be expressed as a collection of OWL individuals (i.e. unique instances of OWL classes) related to each other through a set of instances of OWL properties.

The described approach allows for the structure of a service composed of simpler software modules to be described. It also allows the use of an automatic reasoner (e.g. Pellet) to check for restrictions the meta-level object might impose.

In the described scenario the use of the OWL individuals to describe a software service appears to be adequate as the nature of the software service is such that a given host usually has only one instance of such service. For example it is unusual to have two or more separate Web servers or telnet servers running on a single machine. Usually only one web server or telnet sever are running on the host – they might be able to serve multiple requests at the same time by using multithreading etc. but they are still a single server for each type of service.

In the case of cognitive radio the situation is different – there might be several instances of the same component (e.g. an FIR filter) running at the same time therefore the approach taken by the authors cannot be applied directly in the describing the structures of the components.

In the cognitive radio case one could build models of multiple instances of a particular component out of OWL individuals but the description of the component class (i.e. the recipe for all instances of the class) cannot be tied to any particular set of OWL individuals.

Despite not being directly applicable to the description of composite components in the cognitive radio, the described system is an important contribution as it demonstrates the use of OWL ontologies in software composition.

3.3.6 Service-Oriented Architecture

Service-Oriented Architecture (SOA) is a set of software design principles and methodologies aimed at designing applications based on interoperable and reusable services. OASIS\(^1\) defines SOA in their reference model specification [OASIS 2006] as: “A paradigm for organizing and utilizing distributed capabilities that may be under the control of different ownership domains.” The perceived strength of SOA lies in that it

---

\(^1\) Organization for the Advancement of Structured Information Standards
provides a framework for matching needs with capabilities and for combining the capabilities to address these needs.

The key concepts describing the SOA are visibility, interaction and effect [OASIS 2006]. Visibility is the ability to locate the services required to address the needs, it includes descriptions of functions and technical requirements, related constraints and policies, and mechanisms for access or response. Interaction is the activity of using the located capabilities. Using messages as the basic form of communication, interaction typically involves a series of information exchanges and invoked actions. The result of interaction is an effect (or a set/series of effects). This effect has a form of the information returned to the caller and/or the change in the state of entities involved in the interaction.

A SOA-based software system provides a set of loosely-integrated services that can be used within multiple domains. SOA abstraction defines properties oriented towards the reuse of services and the integration of heterogeneous environments; therefore it is not constrained by any specific implementation and/or technology.

Branca and Atzori [Branca 2012] list several architectural principles that must be followed during the design and development of a SOA-based service (Table 3-2).

<table>
<thead>
<tr>
<th>Principle</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>Encapsulation</td>
<td>Each service has well defined capabilities and is able to communicate with other services using standardized interfaces</td>
</tr>
<tr>
<td>Loose coupling</td>
<td>Services maintain a relationship with others that minimize dependencies and only require they maintain an awareness of each other</td>
</tr>
<tr>
<td>Contract</td>
<td>Services subscribe to a communication agreement with other services</td>
</tr>
<tr>
<td>Abstraction</td>
<td>Services hide their own internal logic from the outside world</td>
</tr>
<tr>
<td>Autonomy</td>
<td>Services have to be well-defined, complete and independent from the context or the state of other services</td>
</tr>
<tr>
<td>Reusability</td>
<td>The logic is divided into services with the intention of promoting reuse</td>
</tr>
<tr>
<td>Composability</td>
<td>Many services can be coordinated and assembled to form complex services</td>
</tr>
<tr>
<td>Statelessness</td>
<td>Services minimize retaining information specific to an activity</td>
</tr>
<tr>
<td>Discoverability</td>
<td>Services are defined to be externally descriptive so that they can be found and assessed via an available discovery mechanism</td>
</tr>
</tbody>
</table>

The composability principle emphasizes the ability of SOA-based systems to compose more complex services out of the simpler ones. In this respect SOA is similar to other component-based architectures.

In SOA-based systems three fundamental types of actors can be recognized – Service Provider, Service Customer and Service registry. The interaction among them is shown in
Service Provider is the component that makes the service available to other nodes. Service Provider publishes the description of its service to Service Registry. The Service Registry maintains a database of available services together with their descriptions and additional information as URLs, how the service can be invoked etc. When a Service Customer wants to use the service, it obtains its description from the Service Registry, looks up the Service Provider and binds to the service and then invokes the service as needed.

![SOA collaborations diagram](image)

SOA-based systems are commonly built with the use of Web Services (WS). The use of WS technologies have gained industry-wide acceptance and provide greater interoperability than it would otherwise be possible with proprietary protocols. SOA-based systems built upon Web Services typically use Simple Object Access Protocol (SOAP) for the transport protocol of the messages they send and receive. SOAP is based on the Extensible Markup Language (XML) and its messages are sent over the network, typically using Hypertext Transfer Protocol (HTTP). Web Service Description Language (WSDL) is an XML-based standard for describing services. Both Service Providers and Service Customers use WSDL to publish/retrieve the details of the services. The Service Registry itself is usually implemented according to the Universal Description Discovery and Integration (UDDI) standard.

SOA gained the most widespread use in the enterprise-level systems. All major software vendors use this paradigm in their business products and there are also open source projects in this space (with Red Hat’s JBoss being arguably the most popular).

SOA-based systems slowly start to find use in wireless networks; they are primarily used for high-level network management functionalities (e.g. [Duan 2009]), or as end-user applications (e.g. [Yee 2009]).

At the low level, which is the domain of Software Defined Radio, the SOA has not gained any acceptance so far. There is some anecdotal evidence that the major reason for this situation is the perceived poor performance of SOA-based systems compared to traditional SDR implementations. While the magnitude of this perceived performance degradation is a subject for debate, convincing the industry insiders to give up some performance in exchange for benefits they don’t see might not be easy.
In summary, the review of the literature related to the problem being undertaken in this thesis led to the identification of a number of research efforts whose results may be useful for the accomplishment of our goals. However, none of the identified solutions could be readily used to resolve our problem.
4 A Method for Ontology-Based Cognitive Radio Reconfigurability

This chapter presents an overview of a method for ontology-based reconfigurability of the cognitive radio. The details of the proof-of-concept system implementing this method are described in the next chapter.

4.1 Software Defined Radio reconfiguration

Consider a hypothetical situation where the cognitive agent of one of the nodes decides that the quality of the data link with another node would be improved if both radios changed the parameters of the current waveform or switched to a different one. This node could send a reconfiguration request to the other one and if the other radio acknowledged, both radios could make the necessary changes and reestablish communication using the new waveform.

Most SDR architectures offer a set of adjustable and observable parameters of a waveform (also known as knobs and meters). These parameters are equivalent to the control inputs of the plant and the QoS data in the self-controlling software model presented in 3.1.

Mitola noticed that SDRs are not aware of the meaning of these parameters [Mitola 2000]. They are also unaware of the data that is available in their own data structures - for example a GSM SDR receiver would not be able to answer the question “how many distinguishable multipath components are in your location” even though the answer to this question is available in the coefficients of its equalizer. Mitola said that SDRs “don’t know that they know”.

The problem of the SDR node self-awareness was investigated by Wang et al. [Wang 2003] [Wang 2004]. They introduced Ontology-Based Radio (OBR) architecture designed specifically for the exchange of knowledge among the nodes (Figure 4-1).
Figure 4-1  Ontology Based Radio (OBR) architecture.

The OBR architecture uses two fundamental types of data traffic between radios. Data messages are generated/consumed by data sources/sinks in the higher layers of the protocol stack in support of the services provided to the user (voice, internet etc.). The control messages are generated and consumed by the cognitive agent, which consists of Monitor Service and Reasoning Component. The messages are sent and received in order to facilitate the exchange of the knowledge among the nodes. The Monitor Service forwards the received control messages to the Reasoning Component, which in response may generate updates to the node’s SDR software using Java Reflection mechanism and which also may generate new control messages sent to other radios.

Wang et al. implemented an experimental system in which two nodes exchanged information about the communications channel’s multipath structure as seen by each of them. This information was then used by the other station to optimize the training sequence in the packets sent to the node depending on the channel dynamics and the noise. This allowed for up to 60% reduction in the length of the training sequence reducing the overall communications overhead [Wang 2004].

This particular implementation of the OBR architecture relied on a specific technology (Java Reflection) and SDR-specific information in the cognitive agent in order to gain access to the SDR’s internal data structures. It also depended on a specific API between the Reasoning Component and Monitoring Service. Such dependencies would certainly create problems if the SDR software was modified or replaced or if there was a need for a different type of Reasoning Component.

These concerns were addressed by Moskal in the Cognitive Radio Framework (CRF) [Moskal 2011]. CRF extends the original OBR architecture and provides a thin generic API for connecting the inference engine with appropriate knobs and meters without the need for storing any SDR-specific information in the reasoner allowing for maintaining the interface with the SDR even if the SDR’s API changes.
The reconfiguration capability of both OBR and CRF is similar and is limited to these types of changes that can be achieved by a combination of *Set* and *Get* operations – e.g., changing waveform parameters, changing frequency, changing the output power etc. Although the publications related to OBR or CRF never mention this kind of ability explicitly, it could potentially be possible to use the *Set* operation to switch from one waveform to another provided that both nodes implement such a waveform, that both nodes have means to negotiate switching to this waveform and most importantly that the waveform is understood by both nodes in exactly the same way. If one of the nodes does not understand what the target waveform is there’s nothing it can do except for signaling an erroneous condition. Additionally, if two radios were developed by two different vendors, there’s no guarantee that despite bearing the same name the two waveforms are actually the same.

RTRP (Real-Time Research Platform) – an experimental platform created within E2R project tried to solve the problem of unknown waveforms by designating the manufacturer’s database server as the ultimate authority a node could defer to if it didn’t know the required component [Zhong 2006]. The component would be then downloaded in binary form from the server and instantiated by the node. In addition to obvious problems with having to maintain multiple versions of binary files for multiple hardware platforms, the biggest weakness of this approach is the necessity for the presence of the server on the network, which makes this kind of solution unsuitable for portable/mobile radios that are supposed to interoperate with radios made by other vendors in ad-hoc, temporary networks, i.e. in situations typical for emergencies when different services have to establish effective communication in the field.

### 4.2 Prerequisites

Just like there are prerequisites for a human to be able to learn new concepts (e.g. ability to speak and understand spoken language, having right amount of fundamental knowledge etc.), equivalent prerequisites exist for the ability to transfer knowledge from one Cognitive Radio to another.

First of all the radios have to have a way to communicate with each other. If the sets of frequency ranges they support are disjoint or if they implement totally different waveforms, they will not be able to communicate at all and obviously in such a case no transfer of knowledge is possible.

The second fundamental requirement is that the radios have to share the same *base SDR ontology*. It is important to emphasize that particular instances of SDR ontology don’t necessarily have to be the same. Since ontology captures knowledge in the domain at a particular moment of time, such knowledge doesn’t have to be identical in all cases. For example an instance of the SDR ontology in the company developing a new waveform may include elements associated with this waveform. These elements will not be present in the instances of SDR ontology developed by other companies.

*The Base SDR ontology* is a common subset of all instances of SDR ontology such that all other concepts in the various instances of the SDR ontology can be conveyed in terms
of the elements of this subset. In other words an arbitrary concept in an instance of the SDR ontology has to be either a concept in the base ontology or has to be decomposable to such concepts.

Base ontology is a standardized minimal level of knowledge a Cognitive Radio has to possess in order to be able to interoperate with other radios. The requirement for shared base ontology guarantees that the radios will be able to understand each other even if their instances of SDR ontology differ.

It should be noted that the ontology standardization does not have to stop at the base SDR ontology level. It is desirable to include more complex concepts as well – ideally up to the state-of-the-art of SDR technology at the time such standard is created.

Additionally, it should be made clear that even if two communicating radios can understand each other as a result of using the same SDR ontology, it does not imply that a component that is describable by the ontology is also implementable. The implementation of a component may require some components that the radios still would not know how to construct.

There are additional prerequisites for the transfer of knowledge:

1. The CRs have to be able to send queries and respond to queries about any arbitrary elements not in the base ontology.
2. They have to be able to incorporate new facts they acquired from other radios into their knowledge base.
3. They have to be able to reason about the facts in their knowledge base, the facts they acquired through queries and their internal status and use those results to reconfigure themselves.
4. The CR has to support recurrent query-response sequences. If a CR node receives a fragment of an ontology from another node and there are concepts appearing in the response for which it does not have descriptions in its knowledge base, it will send out queries about each of those concepts. The query-response sequence will continue until all unknown concepts are conveyed in terms of simpler concepts all the way down to elements from the base ontology. Each of the intermediate unknown concepts is added to the knowledge base as the query-response process continues.

### 4.3 Base SDR ontology

*Base SDR ontology* is a standardized subset of SDR ontology such that the concepts and relationships in this subset are sufficient for all other concepts and relationships in any instance of SDR ontology to be derived from.

#### 4.3.1 The contents and granularity

Given that most concepts in software defined radio domain can be decomposed into basic signal processing operations like adding, multiplying, delaying by a single cycle, mapping etc., the base SDR ontology could simply include only those very basic operations. Such an approach however would put unnecessarily heavy burden on the reasoner as all more advanced concepts would have to be derived from the simple ones.
In order to illustrate this point let’s consider an FIR filter – one of the simplest building blocks that is used in about every software defined radio implementation.

The output of the filter is a weighted sum of the current and a finite number of previous values of the sequence \( x[n] \). If the filter coefficients \( h[n] \) are constant in time, the output sequence is the result of convolution of the filter’s coefficients \( h[n] \) with the input \( x[n] \).

The above filter can be decomposed down to three fundamental operations i.e. multiplication, addition and a unit delay.

As it is shown in the above figure, an FIR filter of \( N \)-th order can be expressed as a set of \( N \) unit delays, \( N+1 \) multipliers and \( N \) adders with appropriate relationships between their inputs and outputs.

Now let’s suppose that the base ontology include Multiply-Accumulate (MAC) unit shown below.
We could substitute a MAC unit for each pair of a multiplier and adder as shown below.

This decomposition would require \( N \) objects fewer than the previous one, making the description of the structure of the FIR filter less complex.

There are two types of the CPU load that are affected by the reduction of complexity of the structure. The first type, which is important only during the reconfiguration is the CPU load incurred by the inference engine. In this case the effects are not obvious as the lower complexity of the structure comes at the expense of having a larger number of components in the knowledge base, which negatively affects the matching process performance during the inference.

The other type of the CPU load is the burden that the whole structure puts on the CPU when the component is in service. Even though it is impossible to state categorically that the more complex components are more efficient with the use of the CPU than the equivalent collection of less complex components, it is reasonable to expect that this is
the case. Additionally the fact that less complex structure requires fewer connections, which translates to fewer calls to the component middleware also indicates that lowering the complexity of the structure most likely translates into lower CPU load at runtime.

The above example shows that although it is possible to express the structure of the FIR filter in terms of processing primitives, it is more beneficial to use more complex constructs in its description. In fact it would be reasonable to include in the standardized SDR ontology all signal processing functional units that are widely used in practical implementations of software defined radios e.g. FIR filters, IIR filters, coders, decoders, modulators, demodulators etc.

Since the base ontology has to be standardized in order to facilitate the interoperability between different vendors, such a standard could include functional units of arbitrary complexity, as the standardizing body would deem appropriate.

The cognitive radio ontology should include not only such concepts as waveforms, algorithms, functional modules, state machines etc. but also concepts describing the state of the communications channel, the status of the radio, its configuration and setup. This type of data can also be useful in optimizing the communications between two nodes.

For example, if the level of interference on a particular channel increases above a certain level, one node might ask the other one for a list of unoccupied channels at the other node’s locations together with their noise level measurements. Based on this data and the measurements in its own location, the radio might decide to request a change to another channel with a lower level of interference.

The request to change a channel to improve the communications parameters may have consequences that go beyond the two cognitive radios communicating with each other. For example switching to a “better” channel might prevent another radio from communicating as that one particular channel might be the only one available in this radio’s location. In a different case, if many radios in a certain area simultaneously decide to change channels, this might lead to a system-wide instability. The problem with the instability of self-controlled software systems has been recognized by Kokar et al., [Kokar 1999] as a serious issue. The channel switching scenario therefore is a heavily simplified example that in the real-world implementation would require safeguards against the instability.

4.3.2 Base ontology as the basis of the cognitive radio’s knowledge base

Vendor-specific instances of SDR ontology could include proprietary elements not present in the standard. They would have to be decomposable into (expressible in terms of) the elements from the base ontology if they were to be used in communications with CRs from other vendors.

The SDR ontology together with vendor-specific extensions constitutes the nucleus of the knowledge base of a Cognitive Radio node. In addition to the ontology the knowledge
base also includes the facts and parameters specific to the operation of the particular CR node, the network (or networks) it participates in and the environment.

![Figure 4-6 Cognitive Radio node's knowledge base.](image)

With the progress in research and development of algorithms and waveforms, the SDR ontology would have to be periodically updated with new concepts that have been developed and widely accepted in the industry since the last revision. The future versions of the SDR ontology would have to be fully backwards compatible with the older versions.

The usage of vendor-specific ontology extensions (i.e., including concepts that are not directly defined in terms of the base ontology) to augment the SDR ontology has a potential for a situation where the same concept (a waveform, a functional module etc.) has been independently developed by two vendors. In this case despite the fact that both radios implement effectively the same functionality, this functionality cannot be used for communications between these radios. The reason is that since these two concepts are not included in the base SDR ontology, their semantics is confined to the ontology extensions therefore is unknown to the other vendor’s nodes.

### 4.3.3 The choice of the language

As the survey by Ding et al. [Ding 2007] shows, many languages have been developed in the past years to represent ontologies. In recent years the family of languages based on RDF and RDFS gained a prominent position as these languages have been heavily used for the development of the Semantic Web applications. Web Ontology Language (OWL) is the latest member of this family and has been adopted as the official ontology language for the Semantic Web [OWL2 2009a].

Although the base SDR ontology could be encoded in a variety of existing ontology languages, OWL has been selected as the language of choice primarily due to the larger number of tools and inference engines supporting it comparing to other languages. For the reasons discussed earlier in the document, OWL has to be augmented with rules – in case of the prototype system described in chapter 5 OWL is augmented with BaseVisor rules and the BaseVisor inference engine is the basis for the cognitive agent implemented in our work. See section 3.2.2 for the discussion on Web Ontology Language. Refer to section 3.2.3 for more information on rules.

### 4.3.4 The choice of Base SDR ontology

The requirement for a standardized Base SDR ontology is the key for CR interoperability because - as it was emphasized before – a common standardized understanding of the
SDR concepts is a prerequisite for a uniform interpretation of more advance concepts built upon simpler ones. Although the exact contents and the format of any base ontology are not critical, all the elements necessary for describing components have to be included. Unfortunately at the moment of this writing, no SDR-related ontology is recognized as a standard. There is however an ongoing effort within the Wireless Innovation Forum pushing towards this goal. An existing prototype of the SDR ontology, called the Cognitive Radio Ontology (CRO) [WIF2010] appears to be the most complete effort up to date and has been chosen as the starting point for the prototype system implementing the proposed method.

4.4 General system architecture

A high level architectural view of a system consisting of two communicating cognitive radio nodes is presented in Figure 4-7. Two types of messages – data messages and control messages – are exchanged between the nodes constituting a data and a control (signaling) link, respectively.

Each of the nodes contains a cognitive engine with an associated knowledge base, which includes a local copy of the base SDR ontology, vendor specific and/or local extension to SDR ontology and local facts.

In the situation where the used waveform and the quality of the communications link between the nodes satisfy the nodes’ requirements, the cognitive agents of the nodes only monitor the states of their respective software defined radio functionalities and the environment. They interact only when there is a change in the environment and/or requirements that warrants a reconfiguration.

When there is no hierarchy in the network, which is a common occurrence in ad-hoc wireless networks typical for radios deployed in the field, any node can initiate a reconfiguration process by sending a reconfiguration request to its peer. The other node can agree to the request and execute it or it can reject it if it is unable to comply due to the lack of resources (e.g. memory, CPU power), policy restrictions etc.
4.5 An overview of waveform reconfiguration

Fixed reconfiguration algorithms may not be well suited for environments where the number of possible configuration choices is significant and where the individual systems are deployed in the field and cannot be recalled and upgraded all at the same time. The latter property is particularly significant for applications where introducing changes in the field is of utmost importance. For example if the military developed a new modulation or encoding scheme that would make radios more resistant to a particular type of jamming used by the enemy forces, the desire would be to deploy this change to the theater as soon as possible. Traditionally this would require reprogramming the radios in the field, which is time consuming and might not be possible if some of the units operated in the forward bases with no contact with the main forces for an extended period of time.

It has already been shown that CR nodes can exchange knowledge that can be used to modify the parameters of a waveform in order to optimize the communications. The next set of questions one should ask is whether it is possible to describe a waveform in a way that would be uniformly interpreted across different architectures, whether such a description can be transferred from one CR node to another and finally whether such a description can be used to build the waveform locally.
The reconfiguration method proposed in this work is a positive answer to all these questions. It extends the reconfiguration functionality of the OBR (or CRF) and in addition to Set/Get operations introduces a Reconfigure request, whose role is similar to that of the reconfiguration loop in the self-controlling software model proposed by Kokar et al. [Kokar 1999] (see 3.1).

The Reconfigure request can be seen as a special case of much more generic Invoke mechanism which allows a software entity (i.e. a process, a network node, a cognitive radio etc.) to call an arbitrary public method in another entity. The advantage of Invoke lies in its generality – invoking any public method doesn’t require any method-specific message processing in the remote entity, all methods use the same handling of the input/output parameters and the same dispatch mechanism. Remote invocation is the basis for many inter-process communication systems and it traces its origins back to the original Remote Procedure Call system developed by Sun Microsystems for their Network File System (NFS). In recent times remote invocation gained widespread use as it is also the basis for modern service-oriented architectures (SOA).

In practical systems where the types of remote calls are well defined, their number is limited and doesn’t change, the full implementation of Invoke mechanism is unnecessary as the required functionality can be implemented in a number of other, simpler ways. The interoperability method proposed here does not require full Invoke mechanism and the experimental system built as a proof-of-concept does not implement it.

If both nodes have the target waveform in their component libraries, on the reception of the Reconfigure request the node simply instantiates the waveform and puts it in the service. In the more interesting case, when a node doesn’t know the requested waveform, it sends a query to the node that originated the Reconfigure requesting a description of the waveform. A waveform description expresses its structure as a composition of simpler components with interconnections among them. If on the reception of the description, any of the simpler components are still unknown to the receiving node, it sends additional requests for description of these unknown components. Such recursive process continues until the waveform is decomposed into components known to the node and implementable using its own resources. The knowledge gained in this process i.e. the transferred description of the waveform is then used to build a copy of the waveform from the components.

The waveform reconfiguration method as outlined in the previous paragraph depends on the ability to describe a waveform in a way that would be uniformly understood by CRs developed by different vendors. It also depends on the ability to transfer the description from one node to another and use this description in the receiving node to build a local implementation of the waveform.

4.5.1 Waveform description

According to the SDR Forum definition, a waveform is a “set of transformations applied to information to be transmitted and the corresponding set of transformations to convert received signals back to their information content” [SDRDefs]. Waveforms are usually
implemented as “applications” (compare 3.3.2) that can be loaded on the top of the middleware layer of the SDR.

The overall waveform functionality can be divided into two parts:

- signal processing
- the waveform behavior.

Signal processing is directly involved in the process of transforming the transmit data into the sampled signal on the transmit side and in transforming the received samples into the data on the receive side of the waveform. Since signal processing usually involves standardized operations as filtering, modulation, demodulating etc., a significant portion of the signal path of the waveform can be represented as a composition of simpler signal processing components.

Waveform behavior encompasses all the waveform functionality that is not directly involved in the signal processing – for example managing the internal state of the waveform, responding to asynchronous events etc. Since the behavior of a practically implementable waveform necessarily has to have a finite number of internal states, it can be represented by one or several finite state machines. Finite state machines can also be used to describe any non-standardized processing in the signal path of the waveform.

State machines implementing the waveform’s behavior or non-standard processing have to interact directly with other signal processing components therefore they have to be implemented as components as well.

A complete description of a waveform consists thus of two parts:

1. The description of state machines that encode the waveform’s behavior and non-standard signal processing.
2. The structural description of a composition consisting of the signal processing components as well as the state machine components created based on their descriptions (1). This structural description specifies all the components, the interconnections between them and the connections between the waveform’s external interfaces and the external interfaces of the specific internal components.

Both parts of the waveform description have a form of facts and rules that are used at the appropriate time to generate a model of the waveform. Such model consists of OWL individuals connected through properties, which in case of the state machine descriptions (1) represent states, transitions, guards etc., and in case of the structural description (2) represent components, ports, connections among them etc.

The constructed OWL model of the waveform is used to build an executable composite component out of the other simpler components. If any state machines are defined in the waveform, their components are generated from their descriptions and used in the composition as other components. The process of building the composition is done by a one-to-one mapping of the OWL individuals and properties into components and connections.
4.5.2 Correctness

Transferring waveform descriptions between radios from different vendors or based on different software architectures presents a challenge - how can one assure that the receiving node interprets the information in a correct way i.e. that the local model of the waveform built by the receiving node is correct.

Two alternatives in which this problem can be tackled have been considered.

In the first approach (*correctness by a proof*), the OWL model of the waveform is constructed in some arbitrary way and then a theorem prover is used to prove that the model satisfies all constraints imposed by the waveform definition.

In the second approach (*correctness by construction*), the OWL model of the waveform is constructed in a formal way using a forward-chaining inference engine. Assuming the description of the waveform (i.e. its descriptive and constructive rules) is correct and that the inference engine is sound, the result is correct by construction.

It should be mentioned that a third approach – one based on the *proof-carrying code* (PCC) method, could be used for this purpose as well. One of the reasons it has not been considered is the fact that even though this method might result in less burden put on the CPU of the receiving node, the construction of the proof would need to be the responsibility of either a tool for proof generation, or of the developer of the waveform. The former solution would require an additional study since proof generation is not a simple matter. The latter solution would be rather difficult to sell to the radio engineers who, in general, are not familiar with the advanced technology that would be required to generate such proofs. On the other hand, the waveform construction rules, as described in this thesis, are close to the expertise of the radio engineers, and thus this approach would be easier to accept by this community.

Correctness by construction aligns very well with the *constructivism* paradigm (see Appendix 8.1). It is a well established methodology that has been proposed for use in the development of digital circuits (e.g. [Leinwand 1982]) and software (e.g. [Hall 2002][Sifakis 2005]).

Correctness by construction also has some advantages over the other approach:

1. The process of building the OWL model of the waveform is done in a single phase. There’s no need for separate building and satisfiability proving stages.
2. There’s no need for a separate algorithm for building the OWL model from the description – building of the model is done automatically through the application of rules in the inference engine.
3. Descriptions required for proving the waveform satisfiability are usually more complex than the equivalent constructive ones. The reason is that not only do we have to prove that all the facts and relationships defining the waveform are asserted but we also have to prove that no extra facts and relationships are asserted. For example - it is not sufficient to prove that a component that is supposed to have two inputs actually has two distinct input ports. One also has to
prove that these two ports are the only input ports of this component. The equivalent constructive description would just include the rules for constructing these two ports.

4.5.3 A simplified overview of the waveform construction process

This section contains a simplified overview of the waveform (or any other composite component) construction process in order to lay out the necessary steps involved before they are described in more detail later in this chapter. Note that some of the steps described below do not explain at this point the conditions required for the associated rule or rules to be triggered. Please also note that in this simplified explanation the generation of state machines for state machine components is omitted for clarity.

Assume that a CR received a reconfiguration request to a waveform that it did not have in its knowledge base. Assume further that it responded with a query for the description, it received the response and it added the description in a form of facts and rules to the local KB.

In order to illustrate the steps involved in the waveform building process, consider a MultiplyAccumulate component, which is a composition of a Multiplier and an Adder (Figure 4-8).

Figure 4-8 MultiplyAccumulate component.

The following steps are involved in constructing the model of MultiplyAccumulate.
1. The construction process starts when the CR creates an OWL individual (e.g. \( m \)) and asserts that this individual is of type MultiplyAccumulate (Figure 4-9).
2. Asserting that an individual is of a specific component type triggers the external interface construction rule for this type that creates and asserts OWL individuals for external interfaces (ports, signals etc.) (Figure 4-10).

3. The component realization rule is triggered next – in case of a composite component it creates and asserts OWL individuals for subcomponents.
4. External interface construction rules of the subcomponents are triggered – they create OWL individuals representing external interfaces of subcomponents.

![Subcomponent's external construction rules](image1)

**Figure 4-12 Construction of the external interfaces of subcomponents.**

5. Connection rules for MultiplyAccumulate are triggered for the interface pairs that are supposed to be connected.

![MultiplyAccumulate.Composition connection rules](image2)

**Figure 4-13 Composition connection rules.**

6. At this point the realization rules for subcomponents are triggered and if any of the subcomponents need to be composed, these rules create OWL individuals for their subcomponents and the whole process in steps 3-6 repeats recursively at deeper and deeper levels until all subcomponents are available without further composition.

After the model of the waveform has been constructed, the software maps the OWL individuals representing components in the model into actual instances of software and/or hardware components and then makes the connections between them according to the object properties representing connections in the model. After the mapping and connection process is done the newly created waveform is put into service.
4.6 Components

The OBR reconfigurability method depends on the ability to compose software
components out of less complex components. As it was mentioned earlier in section 3.3
component-based software engineering allows for modularization and encapsulation,
which in turn allows for separating the framework-related functionality from the SDR-
related functionality of each component.
The selection of particular software component architecture is not critical provided that
all elements required by the proposed reconfiguration method can be implemented within
it. In the proof-of-concept system a lightweight framework designed specifically for this
purpose has been implemented but it should be emphasized that other frameworks (e.g.
CORBA) could have been used in its place.

4.6.1 Component types

In the ontology the class Component is a descendent of the class Object and has three
subclasses – BasicComponent, CompositeComponent and StateMachineComponent.

BasicComponents are Components, which are not further decomposable. All
BasicComponents are included in the Base SDR ontology and all of them have to be
implemented by all interoperating CR nodes. The examples of BasicComponents include
arithmetic operations, fundamental signal generators (step, ramp, sinewave), unit delay,
etc. It should be noted that the assignment of particular types of components to
BasicComponents is somewhat arbitrary.

For example, the modulo operation can be expressed as a composition of simpler
arithmetic operations:

\[ a \mod n = a - \lfloor \frac{a}{n} \rfloor \]

Since the modulo operation is one of the fundamental machine instructions in virtually all
modern processors it might be more efficient to include modulo in BasicComponents so
that it can be instantiated as a single component in all CR nodes rather than a
composition. The modulo operation is therefore a prime candidate for inclusion in
BasicComponents even though technically it could be decomposed.

StateMachineComponent is a special case of a Component that can be generated on-the-fly from a state machine description. More on state machine components in section 4.7.

CompositeComponents are components that can be decomposed into simpler
components.
4.6.2 The external interfaces of the Component

Components interact with other components through ports and signals and can be driven by an external clock (or on rare occasions multiple clocks).

Input ports are used to pass data into the component. After data has been processed appropriate values are asserted on the output ports. Components can have either both types of ports or only outputs or only inputs. It is also possible to have a component that doesn’t have either – in such a case the component would have to interact with other components exclusively through signals.

Signals are used to pass control events among the components. As proposed here signals are binary – i.e. they assert that an event has happened but they do not carry any payload as their intended use is similar to control inputs/outputs used in hardware/firmware components. While it is possible to extend the idea of signals to include payload data, the similar effect can be achieved by using a signal and a pair of ports for the payload data.

An external clock is a source of periodical events that the component uses to pace and synchronize the processing of the data. The role of the clock is comparable to the role of a clock signal in synchronous digital circuits.
Figure 4-15  External interfaces of the Component.

It should be noted that the selection of these three types of external interfaces was done to make interfacing between software and hardware/firmware components easier.

4.6.3 Ports

Ports are interfaces through which data flows from one component to another. Ports handle all data streams related to the SDR functionality. They can also be used for some other auxiliary purposes – i.e. diagnostics, control etc. A port is characterized by two features – its direction and the data type it carries.

There are two subclasses of the Port class in the ontology – InputPort and OutputPort, each of them having several subclasses for specific fundamental data types (Boolean, Character, Double, Float, Integer, String).

Two ports are connected if there exists a isConnectedTo relationship between them. Two properties drivesPort and isDrivenByPort are descendants of isConnectedTo and they determine the direction of the data flow in the connection.

More than one input port can be connected to an output port. Any restrictions as to the number of input port driven by a single output can be handled in the local rules - this issue however is not investigated in this work.

The interoperability method proposed here does not impose any restrictions on how the ports are implemented in hardware and software and how they interface with each other.
A component might have many ports, sometimes many ports of the same type. The rules that describe such components have to be able to differentiate between the ports but the specific URIs of these ports for a particular instance of the component are not known when these rules are developed. Because of this, their data properties are used to tag them so that the rules are able to select them as needed. More details on this in section 4.6.6.

### 4.6.4 Signals

Signals are binary communications sent or received by components to notify one another about asynchronous, infrequent events. As in the case of ports, the interoperability method proposed here does not impose any restrictions on how signals are implemented.

The requirements for signals are quite different from the requirements for ports. Ports handle the major flow of data (in case of the physical layer they process the received or transmitted signals’ samples) so the throughput is the primary consideration. Since majority of the port connections are one-to-one or one-to-a-few, handling connections to a large number of input ports is not a big concern.

Signals on the other hand have to be able to handle large numbers of receivers. For example – a reset signal might need to be connected to all subcomponents of the composite component. While connecting a signal to a large number of inputs doesn’t necessarily have to have an impact on the way signals are implemented in software, it might have a significant impact on hardware or firmware implementations, particularly when a large number of signals and large number of components are involved.
4.6.5 Component realization

During the process of reconfiguration, for each OWL individual representing a component in the model, a real instance of the component has to be created.

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

A CompositeComponent can be instantiated not only as a discrete software component but also as a composition of simpler components. A StateMachineComponent can be instantiated as a discrete software module, but it can also be generated on the fly from its state machine description.

Table 4-1 Component Realization Types.

<table>
<thead>
<tr>
<th>Component instantiation type</th>
<th>BasicComponent</th>
<th>CompositeComponent</th>
<th>StateMachineComponent</th>
</tr>
</thead>
<tbody>
<tr>
<td>discrete software module</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>hardware or firmware resource</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
</tr>
<tr>
<td>component composition</td>
<td></td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>state machine generation</td>
<td></td>
<td></td>
<td>Y</td>
</tr>
</tbody>
</table>

As it is shown in the table above a component can potentially have up to three different types of realization.

The different types of realization are managed by a set of classes derived from the class Realization. In the proof-of-concept system presented in this thesis three realization classes have been defined in the ontology:

- *JavaClassInstance* – for components created as an instance of a Java class.
- *ComponentComposition* – for components assembled from simpler components.
- *StateMachineCodeGen* – for state machine components which are created through generating Java source code from the state machine description on the fly and then compiling this code and creating an instance of such a created class.

Individuals of these classes contain information necessary to create instances of their associated components. In case of *JavaClassInstance*, it is the name of the JAR file and the name of the Java class in that file that implements the component, while *ComponentComposition* and *StateMachineCodeGen* contain facts and rules describing the OWL models for respectively: the component composition and the state machine.

A component implementation is not limited to a single realization of a particular type. For example a software component might have two implementations – one that is fast but uses a lot of memory and another that uses less memory but puts more burden on the CPU. In case of firmware and/or hardware implementations it is possible to have a
number of realizations optimized for different requirements – e.g. speed v.s. required hardware resources (e.g. memory, flip-flops, LUTs).

The selection of an appropriate realization class for a particular component is made by the reasoner. A set of rules governing the selection is specific to the CR node and is defined in its local knowledge base. The rules can be as complex as the needs and the expressivity of the rule language allow. This enables introducing changes in the component’s realization on the fly, without having to modify the CR’s run-time system. This is particularly useful when significant changes are being deployed in the field – in this case the rules could be modified to ignore any added/modified components before a certain point of time, or until the node’s rules are modified again after the updates are completed.

Based on the contents of the local knowledge base the reasoner associates a specific realization class with an OWL individual of the component type. This in turn fires the rule that is specific for this component type and realization type, which asserts for this individual the facts required for the creation of the instance.

Consider the following example from the proof-of-concept system. The class *obr:Adder* is a BasicComponent that has only one type of realization – i.e. the Java class instantiation. The local knowledge base contains the following rules and facts for the *obr:Adder* component (BaseVISor syntax).

![Figure 4-18 An example of an instantiation class assertion and rule.](image)

The first statement asserts that the class *obr:Adder.Java* is a subclass of both *obr:Adder* (the component) and of *obr:JavaClassInstance* (the Java class instantiation). Since *obr:Adder.Java* is the only realization class for *obr:Adder*, this class is selected by the realization selection rules (not shown here), which is done by the assertion of the triple (*obr:Adder, obr:hasRealizationClass, obr:Adder.Java*). This last fact is used in another rule (Figure 4-18), which fires when an OWL individual is asserted to be of *obr:Adder* class.
In the case presented here an individual of type obr:Adder is also asserted to be of type obra:Adder.Java. This fact in turn fires the Adder.Java Rule in Figure 4-18, which defines two data properties for this individual that are used for the creation of the instance (i.e. the system has to instantiate a java class components.Comp_Adder from the OBR.jar class).

4.6.6 Component description

The description of a component consists of one or two parts depending on which of the three major sub-classes of the class Component the component is type of.

The first part is the same for all types of component and it is a construction rule for the external interfaces.

BasicComponents are not decomposable into simpler components. They are included in the base SDR ontology and they have to be implemented as discrete components (software and possibly hardware or firmware) in all interoperating radios. Since they cannot be assembled from other components, the external interface generation rule is the only description such component has to have. Note that CRs have to provide appropriate realization rules for all BasicComponents. These rules however are local for each node and are not a part of the component description.

The second part of the description of a component of the type CompositeComponent orStateMachineComponent contains a set of facts and rules describing an OWL model of respectively – the component composition or the state machine:

- For CompositeComponent it is a construction rule that creates its subcomponents plus a set of rules defining connections among subcomponents’ interfaces and between subcomponents’ interfaces and the component’s external interfaces. All these rules are included in the definition of the component’s realization class, which is a subclass of both the component and the ComponentComposition class. The component composition realization is the only one required for CompositeComponent but additional realizations (e.g. as a discrete software component) might be provided as well.
For StateMachineComponent it is a set of rules that construct an OWL model of the state machine. These rules are included in the state machine realization class, which is a subclass of the StateMachineComponent and StateMachineCodeGen. The state machine component class might also have additional realizations.

Table 4-2 Component type vs. description.

<table>
<thead>
<tr>
<th>Component Type</th>
<th>External interface construction rules</th>
<th>Discrete component realization</th>
<th>Component composition realization</th>
<th>State machine code generation realization</th>
</tr>
</thead>
<tbody>
<tr>
<td>BasicComponent</td>
<td>required/ontology</td>
<td>required/</td>
<td>n/a</td>
<td>n/a</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local KB</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Composite-Component</td>
<td>required/ontology</td>
<td>optional/</td>
<td>required/ontology</td>
<td>optional/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local KB</td>
<td></td>
<td>local KB</td>
</tr>
<tr>
<td>StateMachine-Component</td>
<td>required/ontology</td>
<td>optional/</td>
<td>optional/ontology</td>
<td>required/</td>
</tr>
<tr>
<td></td>
<td></td>
<td>local KB</td>
<td></td>
<td>local KB</td>
</tr>
</tbody>
</table>

Table 4-2 contains the summary of description elements for different component classes. The external interface construction rules are required by all types of components. In addition to this, BasicComponent requires at least one discrete component realization, CompositeComponent requires a component composition realization that contains facts and rules that construct subcomponents and connects them. StateMachineComponent requires a state machine code generation realization that contains facts and rules that construct a model of the state machine.

4.6.7 OWL individuals in rules

The selection of specific OWL individuals representing interfaces in the connection and construction rules presents a challenge as their URIs (Uniform Resource Identifiers) are not known at the time the rules are developed. Consider an example of a simple composition consisting of two MultiplyAccumulate components as shown in Figure 4-20.

![Figure 4-20 Selecting OWL individuals in construction rules.](image)

The problem becomes immediately obvious if one considers a connection between the output ‘out’ of the multiplier and the input ‘in1’ of the adder inside each of the
MultiplyAccumulate components. The specific URIs of these two ports are not known when the connection rule is developed.

The problem of addressing specific individuals in rules has been solved by associating a data property (a tag) with each OWL individual representing an element of a component i.e. with each port, signal, subcomponent etc. The tags have to be unique only within a single instance of a component, which means that all corresponding OWL individuals in all instances of the component can have the same tag. These tags are used in the rules to select appropriate individuals.

In the above example in order to connect the ‘out’ output of the multiplier ‘MUL’ to the ‘in1’ input of the added ‘ADD’ in all instances of the component the connection construction rule has to select a port tagged ‘out’ in the subcomponent tagged ‘MUL’ and a port tagged ‘in1’ in the subcomponent tagged ‘ADD’ and then connect them together.

**IF**

\[
\text{<?ind\ rdf:type\ MAC> AND} \\
\text{<?ind\ hasSubComponent\ ?sub1> AND <?sub1\ hasTag\ 'MUL'> AND} \\
\text{<?sub1\ hasPort\ ?p1> AND <?p1\ hasTag\ 'out'> AND} \\
\text{<?ind\ hasSubComponent\ ?sub2> AND <?sub1\ hasTag\ 'ADD'> AND} \\
\text{<?sub2\ hasPort\ ?p2> AND <?p2\ hasTag\ 'in1'> AND}
\]

**THEN**

\[
\text{<assert\ ?p1\ drivesPort\ ?p2>}
\]

**END IF**

Figure 4-21 An example of a connection rule.

### 4.6.8 Component instantiation parameters

Some components require parameters for proper instantiation. For example a sine wave generator requires at least two parameters – the sampling frequency and the tone frequency.

The way the instantiation parameters are passed to the component is implementation specific for the particular CR platform. For software components the most obvious method is passing the parameters as the parameters of the object constructor. For hardware parameters, the platform must have an application specific method of passing the relevant information to the hardware when a hardware resource is requested.

The ontology for a specific component contains an ordered list of the formal parameters of the component. The actual values of the parameters are defined and associated with an OWL individual that represents an instance of the component.
An example of a formal parameter definition is shown in Figure 4.22 (an excerpt from the rules defining obr:SineCosineGenerator). The formal parameters are defined in obr:MethodParameterArray, which is an ordered sequence of obr:MethodParameterArrayElement. Each element of this array has an explicit index that denotes the position of the parameter on the list and an object property to the actual parameter, which in this particular case is a parameter of the double type, which has a name fsample (for sampling frequency).

4.6.9 Component run-time parameters

A component may have observable and/or adjustable run-time parameters (knobs and meters) that are used to implement Get/Set operations.

![ComponentParameter class diagram](image)

Run-time parameters are represented here in OWL by using individuals of ComponentParameter class. There are three data properties associated with this class:

- `hasName` - specifies the name of the parameter
- `hasType` - which specifies the type of the parameter
- `isReadOnly` - which specifies whether the parameter can be adjusted or it can only be read

The reconfiguration method does not assume or restrict the way in which run-time parameters are read and/or set. For example in the proof-of-concept system a pair of methods `getParameter()` and `setParameter()` has been defined in the abstract Component class, which is a parent for all classes implementing actual components. In systems implementing components in hardware/firmware, the system’s middleware has to provide ways for accessing the hardware resources associated with each parameter and provide translation to/from software type to their hardware/firmware equivalents.
4.7 State machines

Most non-trivial communications protocols use state machines of some sort. Some protocols are actually defined using state machines. For example all interface functions of General Purpose Interface Bus – the most popular standard for connecting measurement equipment – are specified with the use of state machine diagrams [IEEE488.1].

In traditional implementations state machines are usually hardcoded and are the integral part of the overall algorithm implementing a particular protocol. In this approach the functionality cannot be easily extended as any new functionality to be added to the protocol requires updating the existing state machines or adding new ones to the software module, and this in turn requires that the overall algorithm of the module has to be modified and the whole software module replaced with a new version.

It is quite obvious that any reconfigurable system that is to be able to learn new, unfamiliar protocols has to be able to support the creation and instantiation of state machines from their descriptions.

The base SDR ontology has to include concepts pertaining to the description of state machines such as state, transition, event, etc. A particular state machine will be specified as a subclass of the base OWL state machine class with relationships with other classes describing states, transitions, conditions, etc., expressed with the use of OWL relationships and rules.

Wireless Innovation Forum included some elements of state transition diagram (STD) ontology in their Cognitive Radio Ontology (CRO) prototype and proposed that STDs be used to define state machines in cognitive radios. A fragment of the CRO [WIF 2010] that is specifically related to the STD definition is shown below.

![State Transition Diagram ontology](image)

Figure 4-24 State Transition Diagram ontology [WIF 2010].

The elements of this ontology have been adopted as a starting point for the development of the state machine description ontology, which has been created in OWL with BaseVISor rules. Some concepts in this ontology have been inspired by similar constructs in the UML StateMachine package [UML 2011], but it should be understood that this is not an OWL/Rules mapping of the UML standard.
The cognitive engine of the node that is to learn a new protocol will have to be able to construct the state machine from its description, instantiate it and put it in the service. In the process of building the state machine, the ontology entities are mapped into appropriate language constructs in the source code.

In order to illustrate the state machine description and the process of mapping the description into the software constructs, consider the following state transition diagram (STD) of a component that implements decimation by factor 4.

![State Transition Diagram](image)

**Figure 4-25 An example of a state transition diagram for a decimator by factor 4.**

State machines are ‘packaged’ into components just like any other components in the interoperability method presented here. The state machine components can support all kinds of inputs/outputs that can be connected to the components (ports, signals, clocks), which implies that the state machine descriptions have to be able to describe all these input/outputs and the events related to them.

The external view of the component implementing the decimator state machine (Figure 4-25) is shown below.

![External View](image)

**Figure 4-26 External inputs and outputs of the decimator component.**
The component has an input port `sample_in` and an output port `sample_out`. Additionally, there are also two signal inputs – `START` and `STOP` and a clock input `clk`. It should be noted that internal representations of the ports and events in the state machine description don’t have to be named exactly the same way as the corresponding ports and signals at the component level. In the decimator example the events `START` and `STOP` have the same names as their respective signals but port names are different (e.g. `in_port` vs. `sample_in`) and so are the external clock (`clk`) and its event (`CLK`). The description of the state machine allows for mapping of the events into signals/clocks and external tags of the ports and their internal names.

The decimator enters the `IDLE` state on initialization or on the reception of the `STOP` signal. It remains in this state until it receives `START` signal. On the reception of `START` it enters the `STARTED` state. From there it transitions to `SEND` after receiving the clock event. While entering the `SEND` state it executes this state’s entry action sequence – specifically it sets the value of the output port to the value read from the input port. After receiving the clock event it transitions to the `COUNT_2` state. On leaving `SEND` the property `count` is set to zero. The state machine remains in `COUNT_2` only incrementing the `count` property on the consecutive clock events until the value of `count` is equal or greater than 2. In that case the state machine transitions back to the `SEND` state and copies another input sample to the output. If not interrupted by the `STOP` signal, the state machine remains in the `SEND<->COUNT_2` loop forever effectively writing every 4<sup>th</sup> input sample to the output.

The construction of the state machine component from the description can be done in a variety of ways depending on the platform on which this method is implemented and the language for the description of the state machines. In the proof-of-concept system a two stage process is used – in the first stage source code in Java is generated from the description into a memory buffer. In the second stage, this source code is compiled in memory and the resulting binary Java class is instantiated.

It should be emphasized that OWL with Rules is not the only potential choice for state transition diagram descriptions. An example of a possible alternative is State Chart XML (SCXML) proposed by World Wide Web Consortium [SCXML 2012]. This draft standard defines an XML-based language for the description of event-based state machines. Since just like UML it is based on Harel’s statecharts [Harel 1987], this language contains elements very similar to the concepts in the OWL/Rules-based language developed in this work. The potential use of SCXML has one major disadvantage – it is not a formal language, therefore cannot be used directly with an inference engine. This makes its use infeasible in the constructivism paradigm.

### 4.7.1 State machine description

The STD ontology shown in Figure 4-24 was a starting point for the development of the state machine ontology. Some of the major classes and relationships in this ontology are presented in a simplified and abridged class diagram in Figure 4-27. The top level class in
the description of a state machine is the *FiniteStateMachine* class, which is a child of the class *BehaviorModel*, which in turn is a child of *Process*.

![Figure 4-27 Selected classes and relationships in the state machine ontology.](image)

The following elements may be included in the description of the state machine:

- States with associated actions on entering and exiting the state. At least one state is required in the description.
- Transitions between states with optional guards and actions.
- Input and output ports that function as gateways for data flowing to/from the state machine. These ports are externally identical to component ports so that the state machine components can be freely connected to other types of components.
- Incoming and outgoing signals that are used to pass asynchronous events to/from the state machine. As in the case of ports, signals are externally identical to component signals.
- Definitions of events, which can be generated on:
  - a reception of a signal
  - a change of a value on an input port
  - a reception of a clock tick
• Properties that are used to store auxiliary data necessary for the state machine to work. Properties are equivalent to local variables in programming languages and can be used as counters, flags, data buffers, etc.
• Parameters – a special case of properties that can be accessed from the outside of the state machine component. State machine parameters act exactly as component parameters described in 4.6.9. Each parameter might define get and/or set actions that are to be executed on the Get and Set operations, respectively. Parameters might also specify validation expressions that are used in the Set operation to ensure that the values set are correct.

4.7.2 States
A state is the fundamental element of the state machine. A state machine must have at least one state. In OWL states are represented by individuals of the type fsm.State. A state might define OnEnter and OnExit action sequences, which are executed when the state machine respectively enters or exits it. The transitions are referenced in the states they originate from.

```xml
<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 4-28 Definition of the state STARTED for the example decimator state machine.

The example of a state definition is shown in Figure 4-28. The state STARTED originates two transitions - one to the state SEND that is triggered by the clock event and another one to the state IDLE, triggered by the STOP event.

4.7.3 Transitions
Transitions are represented by individuals of the OWL class fsm.Transition. A transition is referenced by a state it originates from and it contains a reference to the target state (through fsm.hasTarget property). It also contains a reference to an event that triggers the transition (fsm.triggeredBy).

A transition may be guarded by a constraint (fsm.Constraint). In such a case the transition is triggered only when the associated event happens and the constraint evaluates to logical true value. Constraints are built out of conditional statements and Boolean operations. Constraints can be nested so they can be used to describe logical expressions of arbitrary complexity. See 4.7.5 for more details on constraints.
More than one transition originating in the same state can be triggered by the same event. In this case the transition guards are responsible for deciding which transition is to be executed. It is very important to understand, that there’s no specific order in which guards are evaluated so the guards should be constructed carefully in order to avoid the situation where more than one transition is eligible for execution.

A transition may also define a sequence of actions to be taken when the transition is executed. The transition actions are executed after the originating state’s OnExit actions are completed and before the OnEnter actions of the target state. See 4.7.7 for more details on actions.

![Figure 4-29 An example of a transition with a guard and an action sequence.](image)

An example of a transition with a guard and an action sequence is shown in Figure 4-29. This is a BaseVISor encoding of the transition that originates and ends in state COUNT_2 of the example state machine (Figure 4-25).

The guard is referenced through the property `fsm.hasGuard` and is of type `fsm.EvalCmp.LT`, which represents a ‘<’ (less than) comparison operation. The left side of the comparison (`fsm.hasLeftExpr`) references the `count` property and the right side (`fsm.hasRightExpr`) represents an integer value of 2.

The optional action sequence of the transition are referenced through the property `fsm.hasTransitionActions`. Action sequences are arrays (i.e. ordered sequences) of individual array elements each of which references a single action (through
In the example above the action sequence consists of a single action `fsm.Action.IncrementProperty`. The data property `fsm.hasValue` defines the value of the increment (in this case 1) and the object property `fsm.setProperty` defines the target property to be incremented (in this case `count`).

### 4.7.4 Expressions

In the state machine description an individual of the class `fsm.Expression` represents an entity that can be evaluated to a value i.e. an expression. The `fsm.Expression` is a generic class and the actual expressions are built out of individuals of its descendent classes.

There are several subclasses of the `fsm.Expression`:

- `fsm.APIFunctionCall` – represents an instance of a call to an API function that returns a value. An individual of this type references a target API to be called as well as an array of expressions to be passed to the API as parameters.
- `fsm.ArithmOp` – represents an arithmetical operation. The following subclasses are its descendants:
  - `fsm.ArithmOp.ADD`
  - `fsm.ArithmOp.DIV`
  - `fsm.ArithmOp.MOD`
  - `fsm.ArithmOp.MUL`
  - `fsm.ArithmOp.SUB`
- `fsm.BitOp` – represents a bitwise operation on an integer argument. The following operations have been defined:
  - `fsm.BitOp.AND`
  - `fsm.BitOp.NOT`
  - `fsm.BitOp.OR`
  - `fsm.BitOp.SHL`
  - `fsm.BitOp.SHR`
  - `fsm.BitOp.XOR`
- `fsm.Constraint` – represents an expression that evaluates to logical true or false value. Constraints are used as guards and conditions (see 4.7.5)
- `fsm.InputPort` – represents a value read from an input port
- `fsm.Property` – represents a value of the property (local variable)
- `fsm.ValueExpr` – represents an immediate (constant) value.

### 4.7.5 Constraints

Constraints – the descendants of the class `fsm.Constraint` – are expressions that evaluate to a Boolean value (i.e. true or false). Constraints are used as

- transition guards
- conditions in flow control actions (IF, WHILE)
- validation conditions for the state machine parameter `Set` operations
- conditions for input change events

Constraints include unary and binary conditionals and unary and binary Boolean expressions (see Table 4-3).
### Table 4-3  Constraints

<table>
<thead>
<tr>
<th>Conditional</th>
<th>Unary</th>
<th>Binary</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td><code>fsm.EvalBoolFalse</code></td>
<td><code>fsm.EvalCmp.EQ</code></td>
</tr>
<tr>
<td></td>
<td><code>fsm.EvalBoolFalse</code></td>
<td><code>fsm.EvalCmp.GE</code></td>
</tr>
<tr>
<td></td>
<td><code>fsm.EvalCmp.GT</code></td>
<td><code>fsm.EvalCmp.LE</code></td>
</tr>
<tr>
<td></td>
<td><code>fsm.EvalCmp.LE</code></td>
<td><code>fsm.EvalCmp.LT</code></td>
</tr>
<tr>
<td></td>
<td><code>fsm.EvalCmp.LT</code></td>
<td><code>fsm.EvalCmp.NEQ</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fsm.LogOp.OR</code></td>
</tr>
<tr>
<td></td>
<td></td>
<td><code>fsm.LogOp.XOR</code></td>
</tr>
</tbody>
</table>

### 4.7.6 Events

Events trigger transitions between states. Events are represented by individuals of the descendants of the class `fsm.Event`:

- `fsm.ClockEvent` – represents a periodical event generated by a clock
- `fsm.SignalEvent` – represents a reception of a signal
- `fsmInputChangeEvent` – generated when a new value satisfying the associated condition has been asserted on the specified input port. Only one event for a single change of port value is generated even if there are more than one input change events referencing this and more than one of these events’ conditions are satisfied. The order in which the events’ conditions are evaluated is unspecified.

The following fragment of BaseVISor code shows how a clock and a signal event from the example state machine are defined.

```xml
<obr:fsm.hasEvent variable="_Ind.Event.CLK">
  <rdf:type resource="obr:fsm.ClockEvent"/>
  <rdf:type resource="owl:NamedIndividual"/>
  <obr:hasName datatype="xsd:string">CLK</obr:hasName>
  <obr:fsm.hasClock variable="_Ind.Clock.clk"/>
</obr:fsm.hasEvent>

<obr:fsm.hasEvent variable="_Ind.Event.START">
  <rdf:type resource="obr:fsm.SignalEvent"/>
  <rdf:type resource="owl:NamedIndividual"/>
  <obr:hasName datatype="xsd:string">START</obr:hasName>
  <obr:fsm.hasInSignal variable="_Ind.InSignal.START"/>
</obr:fsm.hasEvent>
```

**Figure 4-30**  A clock and a signal event definition for the example decimator state machine.

An example of an input change value shown below was taken from the PSK31TXStateMachine component definition prepared for the proof-of-concept system.
The property `fsm.hasInputPort` references the input port being watched. The property `fsm.hasChangeExpression` references the constraint that must be satisfied for the event to be generated. In this particular example the new value asserted on the specified port has to evaluate to Boolean true.

### 4.7.7 Actions

An action (`fsm.Action`) represents an executable statement. An activity (`fsm.Activity`) is a collection of actions executed together. An action sequence (`fsm.ActionSequence`) is a descendant of this class used for defining activities in the state machine definitions. An action sequence is an ordered sequence of actions that are executed one after another. An activity is also an action which implies that activities can be nested i.e. an activity can be called from another activity just as if it were a single action.

The following classes are descendants of `fsm.Action`:

- `fsm.Action.FlowCtrl` – represents a flow control statement. Two types of flow control statements have been defined:
  - `fsm.Action.FlowCtrl.IF`
  - `fsm.Action.FlowCtrl.WHILE`
  Flow control statements facilitate branching (IF) and looping (WHILE) in the action sequences. They reference a constraint through the property `fsm.hasCondExpr` which is a branch or loop condition respectively. The IF action references at least one activity for true value of the constraint and optionally another activity for false value. The WHILE action’s constraint is the loop guard.
- `fsm.Action.SetOutput` – which represents setting an output port to the value of the associated expression.
- `fsm.Action.SetProperty` – which represents setting a state machine’s property (local variable) to a value.
- `fsm.Action.Return` – which translates into a statement returning the value of the associated expression to the caller. This action is used in the definition of state machine parameter’s Get action sequence.

### 4.7.8 Properties

A property (`fsm.Property`) is an equivalent to a variable in a computer program and is used to store data necessary for the operation of the state machine. Properties are typically used for counters or as buffers for values read from the input ports.
A property might be defined as read-write or read-only. Read only properties cannot be modified within the state machine.

```
<obr:fsm:hasProperty variable="_Ind.Property.FSAMPLE">
  <rdf:type resource="owl:NamedIndividual"/>
  <rdf:type resource="obr:fsm:Property"/>
  <obr:fsm:hasValue datatype="xsd:double">8000.0</obr:fsm:hasValue>
  <obr:hasName datatype="xsd:string">FSAMPLE</obr:hasName>
  <obr:hasType datatype="xsd:string">double</obr:hasType>
  <obr:fsm:isReadOnly datatype="xsd:boolean">false</obr:fsm:isReadOnly>
</obr:fsm:hasProperty>
```

Figure 4-32 An example of a property definition.

### 4.7.9 Parameters

State machine parameters (*fsm.Parameter*) are used to implement *Get* and *Set* operations. They function as run-time parameters for the state machine component. Parameters themselves do not define any storage for the values being read/set. They provide an interface to a property or properties and allow for an arbitrary set of actions (i.e. *fsm.Activity*) to be executed on *Get* or *Set* operation.

The get activity is associated with the parameter through *fsm.hasGetActions* property. In the simplest case this activity would just return a value of one of the properties, in more complicated case returning a value of a parameter might involve some calculations, which would be defined in the actions sequence.

The set activity is associated through *fsm.hasSetActions* and again in the simplest case such activity would contain an action to set a value of a property to the input value. The *Set* operation can optionally contain a constraint for valid values passed to the set operation. Such constraint (*fsm.Constraint*) if defined is associated with the parameter through *fsm.hasCondExpr* property.

### 4.7.10 State machine generation

The executable state machine component is built based on the constructed OWL model. The OWL individuals and properties are translated into appropriate executable constructs. The interoperability method does not impose any specific implementation of this mapping.

In the proof-of-concept system the mapping is done in two phases – in the first phase the OWL individuals and properties are mapped into appropriate Java source code constructs. In the second phase the generated source is compiled in the memory and the resulting binary Java class is used to create an instance of the state machine component. The source code mapping used in the proof-of-concept system is presented in Table 4-4.
4.8 Details of the waveform construction process

A simplified UML activity diagram for the waveform reconfiguration can be found in Appendix 8.1 (Figure 8-1). The node that initiates the reconfiguration is called the Master and the node that responds to the reconfiguration request is called the Slave. From the reconfiguration request point of view the Master node is a “client” and the Slave node is the “server”. These two roles however are reversed when the Slave node sends a query for the component description. For the sake of clarity the names Master and Slave are used throughout the text to refer to the node that initiated the reconfiguration request and to the node that responds to this request. The particular assignment of these two designations is valid only for the duration of the single reconfiguration request.

Both the Master and the Slave nodes are involved in the process and as it can be seen from the diagram the activities in the Slave node can be further partitioned into the ones executed by the inference engine and the ones executed by the procedural code.

The Master node initializes the reconfiguration request and during its execution it responds to the queries from the Slave node as needed.

The procedural code in the Slave node is necessary for providing communications services, packing/unpacking messages, dispatching the received requests/responses to right subroutines and finally mapping the OWL model into software and/or hardware components and putting the waveform into the service.

The inference engine with the associated knowledge base and rules is the part of the Slave node’s software where the actual construction of the OWL model takes place. Note that once an OWL individual is asserted to be of the waveform type, the process is controlled solely by the inference engine and the procedural code in the Slave is used only for querying the Master for unknown component types.

- The process of waveform reconfiguration starts with the Master (M) sending a reconfiguration request to the Slave (S) to use a specific type of waveform (C).
• In response S creates an individual \( w \) and asserts its type in the knowledge base.

\( \langle \text{generate } w \rangle, \langle \text{assert } w \ \text{rdf:type } C \rangle \)

• S invokes the reasoner.

• In case the waveform class C is unknown, the UnknownType rule is triggered and it executes a procedural attachment requestClass that sends a query to M for the description of C. After receiving the description, S integrates the description’s facts and rules in the local KB and re-invokes the inference engine.

\[
\text{IF} \quad \langle w \ \text{type } C \rangle \ \text{AND} \\
\quad \text{NOT}< C \ \text{subclassOf owl:Class} > \ \text{AND} \\
\quad \text{NOT}< C \ \text{subclassOf InstantiableObject} > \\
\text{THEN} \\
\quad \text{execute requestClass( } C \text{ )} \\
\text{END_IF}
\]

requestClass( C ): 
\[
\langle S \ \text{sendQuery } C \rangle \\
\langle M \ \text{respondQuery description}(C) \rangle \\
\langle S \ \text{integrate description}(C) \ \text{in the local KB} \rangle
\]

• The reasoner finds a match for the external interface construction rule, which generates OWL individuals for all external interfaces.

\[
\text{IF} \quad \langle w \ \text{type } C \rangle \\
\text{THEN} \\
\quad \text{FOR all interfaces required} \\
\quad \quad \langle \text{generate OWL individual } ?\text{ind} \rangle \\
\quad \quad \langle \text{assert } ?\text{ind rdf:type } ?\text{interfaceType} \rangle \\
\quad \quad \langle \text{assert } ?\text{ind is}[\text{Interface}]\text{Of } ?\text{ind} \rangle \\
\quad \quad \text{END_FOR} \\
\text{END_IF}
\]

• The component realization rules select a realization type (?RealType) for the waveform individual and assert this fact.

\( \langle \text{assert } w \ \text{rdf:type } ?\text{RealType} \rangle \)

The proof-of-concept system described in chapter 5 implements the following set of rules for the selection of the realization class:

- Select a realization class derived from JavaClassInstance if one exists (/* 2 */).
- If no realization class derived from JavaClassInstance exists then select one derived from ComponentComposition, if it exists (/* 3 */).
If no realization class derived from `JavaClassInstance` or from `ComponentComposition` exists, then select one derived from `StateMachineCodeGen`, if it exists (/* 4 */).

IF /* 1 */
  `<?x subClassOf InstantiableObject> AND
  `<?y subClassOf ?x> AND `<?y subClassOf Realization>
THEN
  <assert ?y realizationOf ?x>
END_IF

IF /* 2 */
  `<?comp type ?x> AND `<?y realizationOf ?x> AND
  `<?y subClassOf JavaClassInstance>
THEN
  <assert ?comp rdf:type ?y>
END_IF

IF /* 3 */
  `<?comp type ?x> AND `<?y realizationOf ?x> AND
  NOT ( `<?z realizationOf ?x> AND `<?z neq ?y> AND
  `<?z subClassOf JavaClassInstance> )AND
  `<?y subClassOf ComponentComposition>
THEN
  <assert ?comp type ?y>
END_IF

IF /* 4 */
  `<?comp type ?x> AND `<?y realizationOf ?x> AND
  NOT ( `<?z realizationOf ?x> AND `<?z neq ?y> AND
  ( `<?z subClassOf JavaClassInstance> OR
  `<?z subClassOf ComponentComposition> ) ) AND
  `<?y subClassOf StateMachineCodeGen>
THEN
  <assert ?comp type ?y>
END_IF

After the realization type for the individual w has been asserted, the reasoner matches the realization rule for this type.

IF
  `<w type ?RealType>
THEN
  <execute ?RealType's realization rule>
END_IF

For the realization class derived from `JavaClassInstance` two data properties are asserted – one for the name of the Java class and another for the name of the JAR file.

IF
  `< ??RealType subClassOf JavaClassInstance>
THEN
In case the realization class is derived from `ComponentComposition`, OWL individuals for all subcomponents are generated and asserted to be of appropriate types.

IF
  <?RealType subClassOf ComponentComposition>
THEN
  FOR all subcomponents
    <generate OWL individual ?ind>
      <assert ?ind type subcomponentType>
  END_FOR
END_IF

If the realization class is derived from `StateMachineCodeGen` then its realization rule generates all OWL individuals and their properties that together form a model of a state machine.

IF
  <InstType subClassOf StateMachineCodeGen>
THEN
  <generate and assert all OWL individuals and properties defining the state machine>
END_IF

• In case the component realization class is derived from `ComponentComposition`, after asserting the types of the OWL individuals representing subcomponents, the inference engine matches their external interface construction rules which create OWL individuals representing their external interfaces.

• The creation of the external interfaces of the subcomponents makes the inference engine match the connection rules for the component.

An example of a connection rule:

IF
  <?x subComponentOf w> AND <?x hasTag “sub1”> AND
  <?p1 outputPortOf ?x>AND<?p1 hasTag “port1”> AND
  <?y subcomponentOf w> AND <?y hasTag “sub2”>AND
  <?p2 inputPortOf ?y> AND <?p2 hasTag “port2”>
THEN
  <assert ?p1 drivesPort ?p2>
END_IF

If the waveform model generation rule includes a subcomponent that also is not in the knowledge base, the UnknownType rule for this subcomponent will be triggered and the
process of acquiring the knowledge and building the model of the subcomponent will be started.

This recursive behavior doesn’t have any limits on the number of nesting levels other than the usual memory size and/or time restrictions.

The inference engine plays the central role in the knowledge acquisition process:

- it controls the process
- it infers all standard relationships consequent to the defined OWL constructs (e.g., indirect subclasses, inverse object properties etc.) that have not been asserted explicitly
- it infers majority of the facts locally through construction rules (no need to transfer knowledge that can be found in the local KB)
- it triggers the query process when it infers that the specified component type is not present in the local KB
- it infers the component realization type for an individual representing a component, based on the rules for component realization selection.

It should also be noted that the process of acquiring new knowledge is controlled entirely by the rules and that there is no arbitrary code involved (except for the code that allows the exchange of messages between the nodes). Since the knowledge acquisition process is fully rule driven, it can be modified without having to upgrade the run-time code of the Master or Slave node.

### 4.9 The summary of the ontology-based waveform reconfiguration

#### 4.9.1 The principles

- All radios include cognitive agents.
- The method assumes that the radios have to have some basic means of establishing communications between them, which implies they share at least one communications channel and at least one waveform.
- A base SDR ontology – a body of SDR-specific knowledge - is defined and shared among all nodes using the method. The base ontology includes concepts related to the domain of interest – in this case the domain of Software Defined Radios. The choice of concepts included is the base ontology is somewhat arbitrary and has to be agreed upon by a standardization process. It is desirable that all functional modules used in the modern communications systems are included in the base ontology. In addition to communications-related concepts, the base ontology also includes concepts necessary for describing Finite State Machines.
- The nodes use a standardized language for exchanging messages and a standardized language (OWL with Rules) to encode knowledge.
• The nodes can negotiate the configuration and/or parameters of their communications link.
• If a request includes a waveform (component) unknown to the node, the node can send a query for the missing piece of knowledge.
• After receiving the description of the new component the node constructs a model of the component out of the OWL individuals. Such a model is constructed automatically through the construction and connection rules triggered when an OWL individual is asserted to be of this component type.
• The model construction process is controlled solely by the inference engine and related rules.
• The node then instantiates actual software modules and connects them as represented in the model.
• The composed waveform is put online and the radios start using it for communications.

4.9.2 Limitations of the method and its applicability to other domains
There are several factors that limit the applicability of the proposed method.
• The waveforms have to be decomposable into the software modules from the base SDR ontology and finite state machines.
• Due to finite memory and CPU resources, the number of modules, ports and connections in the composition is limited.
• The size and complexity of the state machines is also bound by memory/CPU.
• The size and complexity of state machines and the structure of the module is limited by the resource and time requirements of the inference engine.
• Algorithms that cannot be converted into reasonably sized state machines cannot be constructed in this method. They could however be implemented as “black box” components and added to the ontology as regular discrete software components.

4.9.3 Applicability to other domains
The reconfiguration method proposed here has been discussed within the context of the radio communications domain and more specifically within the domain of software defined radios. It should be emphasized however, that only the ontology is specific to the SDR domain and thus the method could also be applied to solve reconfigurability problems in other domains in which knowledge can be captured by a domain ontology and algorithms can be implemented as reasonable size finite state machines. The examples of such other domains include networking protocols, control systems, advanced consumer electronics products, etc.
5 The proof-of-concept system

During the investigation of the ontology-based reconfigurability of the cognitive radio a proof-of-concept system has been built. The system has been implemented in Java and consists of two processes communicating with each other over the network connection. One of the processes acts as a master node that initiates the Set, Get or Reconfigure requests and responds to queries for component descriptions. The other process acts as a slave node that receives and executes these requests. During the execution of the Reconfigure requests, the slave node may send queries to the master as needed. Depending on local realization rules and the contents of the local knowledge base the slave node instantiates the requested waveform either as a discrete software component or as a composition that may also include state machines the slave node is able to construct on-the-fly.

5.1 Overview of the architecture of the proof-of-concept system

![The proof-of-concept system architecture.](image)

The architecture of the proof-of-concept system is shown in Figure 5-1. An acoustic link has been selected for the communications channel in order to simplify the experimental setup. The Master and the Slave processes are running on two separate computers. The acoustic link connecting them is used to carry the data traffic. A separate TCP/IP connection is used to carry the signaling messages. Normally these two kinds of traffic are carried over the same wireless connection (compare Figure 4-1), but since the use of an acoustic link limits the selection of possible waveforms to narrowband ones (with low effective bitrates), the signaling traffic had to be offloaded to a separate network connection for performance reasons. This move allowed also for further simplification of
the experimental setup as it allowed for constructing the waveform only in the slave node as the bidirectional signaling traffic no longer needed to be carried over the acoustic link.

Since the TCP/IP protocol stack can also be used to connect processes on a single host, it is also possible to run both processes on a single computer provided that the available soundcard supports full duplex mode.

A program called fldigi [Fldigi] is run on the master node to monitor the acoustic signals produced by the constructed waveforms. Fldigi is a popular freeware application developed by David Freese (amateur radio callsign W1HKJ). It is used in amateur radio service to generate and decode digital waveforms.

Both Master and Slave processes use the BaseVISor forward chaining inference engine [BaseVISor]. The integration of BaseVISor with the rest of the cognitive agent is done through BaseVISor’s public Java API and through procedural attachments. The Java API is used mainly to interact with the fact base - assert facts and make queries. The API is also called when the inference process needs to be (re)triggered. Procedural attachments, which are used to define new constructs in the BaseVISor rule language, are used for two purposes – to generate new resources when creating OWL individuals while building a model (of a composite component or a state machine) and to initiate a component query to the master node when the rule detects that an unknown type of an individual has been asserted.

The SDR ontology is in the form of OWL and BaseVISor source files that are read by BaseVISor during its initialization. The initial contents of the ontology (i.e. what set of components is available) are defined in configuration files that are manually loaded in both processes. Any number of configuration files can be created for different tests. The Slave node additionally allows the modification of the contents of its ontology through a GUI interface, which is particularly convenient during the waveform description transfer testing.

The Slave node’s software includes the elements necessary for building executable waveforms. The reconfiguration module contains the code that performs the mapping of the OWL waveform model into actual software components. The components are instantiated from the component library which contains java classes for each of the components that has a JavaClassInstance realization. All components are Java classes and they are derived from a Component class which is a part of the component middleware. The component middleware defines a set of generic classes (e.g. Component, Port, Signal, etc.) that define interfaces necessary for assembling composite components and generating state machine components.

The Slave node implements a test harness that provides a set of external connections necessary for the constructed waveform to function. These connections include a port to the soundcard, an input connection to the text input box in GUI, signal connections to GUI used for starting and stopping the waveform and an external clock.
5.2 SDR Ontology

The SDR ontology used in the experimental system has been developed based on the prototype proposed by Wireless Innovation Forum [WIF 2010].

The following areas of the prototype had to be added or significantly expanded:

- classes and properties related to the state machine definition
- classes and properties related to the component realization
- miscellaneous classes and properties supporting component definitions
- components required by the set of waveforms selected for experiments

It should be noted that the proof-of-concept ontology contains only a small subset of the components a full standardized SDR ontology would contain if created.

The SDR ontology is defined in several source files that include:

- an OWL file containing the definition of all OWL classes and properties (OBROntology.owl)
- a BaseVISor source file containing generic rules involved in the component realization selection (AuxRules.bvr)
- sets of two or three BaseVISor files per each component in which the facts and rules defining these components are located

The naming convention and the contents of the different BaseVISor files defining components are presented in Table 5-1.

<table>
<thead>
<tr>
<th>File Name</th>
<th>Contents</th>
</tr>
</thead>
<tbody>
<tr>
<td>Comp_&lt;component_name&gt;.bvr</td>
<td>The external interface construction rule for the component</td>
</tr>
<tr>
<td>Comp_&lt;component_name&gt;_Java.bvr</td>
<td>The java class instance realization class definition for the component</td>
</tr>
<tr>
<td>Comp_&lt;component_name&gt;_Composition.bvr</td>
<td>The composition realization class definition for the component</td>
</tr>
<tr>
<td>Comp_&lt;component_name&gt;_SM.bvr</td>
<td>The state machine realization class for the component</td>
</tr>
</tbody>
</table>

Only the first of the four files in the table is defined for every known component. Although not required for components derived from CompositeComponent or StateMachineComponent, all components in the experimental system define their Java instance realization classes. In addition to these, the components that are being decomposed in the experiments have their composition realizations defined and the state machine components include their state machine realization definitions.

It should be emphasized again that even though a component might be fully defined and available in the node that initiates the reconfiguration (Master), it might not be available at all in the node that receives the reconfiguration requests (Slave). In this case the Slave has to construct the component based on the description received from the Master.
The Slave node’s GUI can manipulate the list of the rule files passed to the inference engine during its initialization, which is important for setting up the initial contents of the knowledge base for the experiments without having to edit the configuration files.

5.3 Interfacing with BaseVISor

The cognitive engines of both the Master and the Slave nodes are based on the BaseVISor inference engine. Their code is written in Java interfaces with BaseVISor engine using two mechanisms:

- BaseVISor Java API
- Procedural attachments added to BaseVisor.

BaseVISor Java API is used for the following purposes:

- starting/restarting the inference process
- asserting facts in the reasoner’s fact base
- querying the fact base

The last of the three functionalities is used most frequently - particularly in the process of mapping an OWL model of a waveform into an executable composite component or generating Java source code for an OWL mode of a state machine.

Procedural attachments allow for BaseVISor rule language to be augmented with new functionality. They are created in a form of Java classes derived from `com.vistology.bvr.function.Function`. When BaseVISor encounters a procedural attachment statement in the rule code, it invokes it providing a way for inserting procedural code into the reasoner execution path.

In the proof-of-concept system two procedural attachments have been created:

- `makeAsset` – which creates a new instance of the BaseVISor Asset class (i.e. an entity with an associated resource) with its resource string given as the attachment’s parameter. This procedural attachment is used in the construction rules to generate OWL individuals.
- `requestClass` – which is called when the reasoner encounters an unknown component in the reconfiguration request. This procedural attachment sends a query to the Master node requesting a description of the class whose name has been passed to the attachment as a parameter.

5.4 Component middleware

Component middleware software provides a framework for connecting software components. The proof-of-concept system implements its own lightweight set of Java classes providing these services. It should be noted that although it would be possible to implement the component middleware using one of the existing component technologies (e.g. CORBA), the decision to develop a custom solution made it possible to make the middleware layer of software as minimal as possible.
The component middleware consists of several Java classes, which roughly correspond to equivalent classes defined in the ontology.

The generic functionality of a component is defined in the `Component` class. This class defines API that allows the definition of and access to component’s ports, allows for the definition of incoming and outgoing signals and provides methods for `Set` and `Get` operations on run-time parameters.

`SynchronousComponent` is a class derived from `Component` that augments it with API for clock definition, clock connectivity and synchronous event handling.

Specific type ports are all derived from either `InputPort` or `OutputPort`, which both in turn derive from the `Port` class. `Port` in turn derives from `OBRObservable`, which is the class that provides the basic connectivity based on the Observer design pattern [Gamma 1995] and which is also used to provide connectivity for signals and clocks.

### 5.5 Reconfiguration module

The reconfiguration module consists of several classes responsible for mapping the OWL models of the waveforms/components or state machines into executable components.

#### 5.5.1 State machine generation

The process of translating the OWL model of the state machine into an executable component consists of three phases:

- First the reconfiguration software queries the inference engine, gathers the necessary data and puts it into auxiliary data structures.
- Based on the retrieved information, Java source code for the state machine component is generated in memory.
- The source code is then compiled in the memory and an instance of the component class is created.

In order to illustrate the mapping of a state machine component description into Java source code, consider an example of a hypothetical component implementing downsampling by factor 4. This component copies the value of its input to its output every 4\(^{th}\) clock event.
The above state machine has 4 states, it responds to clock events CLK and asynchronous events START and STOP. It has one input port \texttt{(in\_port)} and one output port \texttt{(out\_port)}. The external view of a component implementing such a state machine might look like in Figure 5-3.

A full set of rules defining the component together with the resulting Java source code are available in Appendix 8.5.

The rule defined in 8.5.3 constructs the external interfaces of the component i.e. the ports \texttt{sample\_in}, \texttt{sample\_out}, the signals \texttt{START} and \texttt{STOP} and the external clock \texttt{clk}. Section 8.5.4 contains the facts and rules that define the class \texttt{DwnSampler.SM}, which is the state machine realization class for the component \texttt{DwnSampler}. The complete Java source code for this example generated by the reconfiguration module is listed in 8.5.5.

The following table contains the examples of different types of mappings between the state machine definition and the Java source code for the downsampler component. Refer to marks on the margin in sections 8.5.4 and 8.5.5 for the locations in the code for particular mappings.
Table 5-2  Downsample example mappings.

<table>
<thead>
<tr>
<th>Element</th>
<th>Margin mark</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input port</td>
<td><em>1</em></td>
<td>An input port in_port with the external port tag sample_in. Note that input port definition also automatically creates a variable to which the port value is automatically assigned every time an input value of the port changes.</td>
</tr>
<tr>
<td>Output port</td>
<td><em>2</em></td>
<td>out_port with the external port tag sample_out</td>
</tr>
<tr>
<td>Clock</td>
<td><em>3</em></td>
<td>An external clock clk</td>
</tr>
<tr>
<td>Clock Event</td>
<td><em>4</em></td>
<td>CLK event</td>
</tr>
<tr>
<td>Incoming Signal</td>
<td><em>5</em></td>
<td>START event</td>
</tr>
<tr>
<td>Signal Event</td>
<td><em>6</em></td>
<td>START signal</td>
</tr>
<tr>
<td>Property</td>
<td><em>7</em></td>
<td>count</td>
</tr>
<tr>
<td>State</td>
<td><em>8</em></td>
<td>IDLE</td>
</tr>
<tr>
<td>Transition</td>
<td><em>9</em></td>
<td>from IDLE to STARTED</td>
</tr>
<tr>
<td>ActionSequence</td>
<td><em>10</em></td>
<td>OnEnter handler in the state SEND</td>
</tr>
<tr>
<td>Action</td>
<td><em>11</em></td>
<td>set value of out_port to the value of in_port</td>
</tr>
<tr>
<td>Guard</td>
<td><em>12</em></td>
<td>a guard for the transition from COUNT_2 looping back to the same state</td>
</tr>
</tbody>
</table>

5.5.2 Assembling composite components

The reconfiguration module creates composite components in two phases:

- First it queries the inference engine to gather the data about the OWL model of the composite component and puts the data in auxiliary data structures.
- In the second phase a container object for the composite component is created. Then using the data retrieved from the inference engine the reconfiguration module creates external interfaces of the composite component and inserts them into the container. In the next step the instances of constituent components are created using the realization data selected for these components by the inference engine. Finally the instances of the components are inserted into the container and all appropriate connections among components’ interfaces and between components’ interfaces and external interfaces are made.

The container objects are instantiated either from CompositeComponent or SynchronousCompositeComponent classes depending on whether the specific composite component is synchronous (i.e. it requires a clock signal(s)) or not. The container classes derive either from Component or SynchronousComponent so they present external interfaces indistinguishable from the components implemented as discrete software modules.

An example of a complete description of a composite component (PSK31CharEncoder) can be found in Appendix 8.6.
5.6 Signaling communications

Because of low bandwidth of the acoustic channel simulating the communications channel, the signaling traffic between the Master and the Slave processes has been offloaded to a TCP/IP connection. The two processes can be located on two separate computers or on the same computer – in both cases they communicate with each other in exactly the same way, the only difference is that in the single computer configuration TCP/IP protocol stack internally uses a loopback interface so no packets are actually sent on any network.

The communications software used in the proof-of-concept system consists of three separate Java classes. The communications server (CommServer) runs as a part of the Master process. When it starts it opens a TCP/IP socket and listens for an incoming connection request from the Slave process. The client software implemented in the class CommClient runs in the Slave process. The Master process initiates all requests. The Slave responds to requests but it can also initiate queries (e.g. unknown class queries).

There are several types of messages that can be exchanged between the nodes. They are all implemented as objects of the class CommMessage. CommMessage has a field of the associated enumeration type (CommMessage.MsgType) that determines the type of the message. CommMessage implements the Serializable interface, which makes preparing and sending messages very straightforward. In order to send a message, an instance of the class has to be created with appropriate MsgType and a Serializable message data passed to the constructor as parameters. The message data depends on the type of message and it is either a string or a class. The message is then passed to the writeObject() method of the ObjectOutputStream associated with the socket.

Among the many types of messages implemented (some for debugging purposes) the ones directly related to reconfiguration are shown in the table below.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Response Type</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>TxChanReconfig</td>
<td>TxChanReconfigResp</td>
<td>The main reconfiguration request</td>
</tr>
<tr>
<td>GetParam</td>
<td>GetParamResp</td>
<td>Get operation on a run-time parameter of the waveform under test</td>
</tr>
<tr>
<td>SetParam</td>
<td>SetParamResp</td>
<td>Set operation on the run-time parameter of the waveform under test</td>
</tr>
<tr>
<td>ReqClass</td>
<td>ReqClassResp</td>
<td>A class query sent by the Slave to the Master</td>
</tr>
</tbody>
</table>

The payload data for these messages is shown in the next table.
Table 5-4  Payload data of the messages.

<table>
<thead>
<tr>
<th>Message Type</th>
<th>Message data</th>
</tr>
</thead>
<tbody>
<tr>
<td>TxChanReconfig</td>
<td>String with the requested waveform type</td>
</tr>
<tr>
<td>TxChanReconfigResp</td>
<td>Status response</td>
</tr>
<tr>
<td>GetParam</td>
<td>A name of the run-time parameter</td>
</tr>
<tr>
<td>GetParamResp</td>
<td>Status response</td>
</tr>
<tr>
<td>SetParam</td>
<td>A name of the parameter and its target value</td>
</tr>
<tr>
<td>SetParamResp</td>
<td>Status response</td>
</tr>
<tr>
<td>ReqClass</td>
<td>A name of the class requested</td>
</tr>
<tr>
<td>ReqClassResp</td>
<td>The set of rules and facts describing the requested class</td>
</tr>
</tbody>
</table>

The Slave sends a status response for each of the requests. The message data in each of these responses contains a class with a status of the request execution, which contains a status code (Ok or Error) and in case the status code is not Ok it also contains a string with an error message.

5.7 Test harness

A waveform in a software defined radio needs to be connected to some additional external components (e.g. a microphone, a speaker, a PTT button, an RF power amplifier etc.) in order to function properly. These external components are provided by the SDR platform and the waveforms are designed with the assumption that these external elements are available for them.

The proof-of-concept system also has to provide interfaces to some external components in order to make the waveforms fully functional. This is achieved by providing a test harness for the constructed waveforms. The test harness requires that all waveforms derive from TXChannel and it provides a uniform set of external interfaces for all of them.
Figure 5-4 Waveform test harness.

The test harness consists of some GUI controls in the Slave’s process main window. Two buttons generate `TxStart` and `TxStop` events, which are used to start and stop the waveform under test. The text input panel allows one to enter text messages that are passed onto the waveform as payload data. The waveform generates `ClkEnable` and `ClkDisable` signals, which are used to control a software clock through its `START` and `STOP` signals respectively. The output of the waveform is connected to the AudioSink component, which provides an interface with the soundcard. The audio output from the soundcard is monitored by `fldigi` program running in the Master node.
6 Experiments and results

6.1 Selection of the test subjects

To test the proof-of-concept system, a number of waveforms had to be selected. These waveforms were defined in OWL/Rules and then composed by the Slave node on the fly. In the selection of the waveforms for the experiments several criteria were taken into consideration:

- The waveforms had to be narrowband in order to fit into the range of audio frequencies so that the communications channel could be simulated by an acoustic link.
- The waveforms for obvious reasons have to be of non-trivial complexity.
- The complexity of the waveforms cannot however be so great as to impose great difficulty in debugging issues that were bound to be found in the experimental system.
- There should be a way to verify that the composed waveforms are correct.
- The selected waveforms should have low or moderate requirements for memory and CPU as not to impede the possibility of running the experimental software, including the BaseVISor inference machine.

Based on the above conditions, the transmit sides of the following three digital waveforms used in the amateur radio service have been selected as the subjects of the experiments:

- BPSK31 (Binary Phase Shift Keying, 31 baud)
- QPSK31 (Quadrature Phase Shift Keying, 31 baud)
- RTTY (Radioteletype)

All of the above waveforms require bandwidth significantly smaller than the bandwidth of single side band transmitters used in amateur service on HF bands. They require modest amounts of memory and processing power. What’s equally important, the verification of the composed waveforms does not pose any problems as programs able to decode and encode these waveforms are freely available for download. One of such programs (arguably the most popular one) – *fldigi* – has been used in the proof-of-concept system for verification.

6.2 PSK31

BPSK31 and QPSK31 are two variants of the PSK31 waveform developed in late 1990’s by Peter Martinez G3PLX [Martinez 1998]. Refer to Appendix 8.3 for a detailed description of PSK31.

The transmit side of the PSK31 waveform is shown in Figure 6-1.
The text characters are being buffered in a FIFO buffer as they are typed. Next they are transcoded into Varicode codes of variable length. The code’s bits are then serialized by either BPSK31 or QPSK31 serializer into 2-bit symbols. BPSK uses only two of 4 possible bit combinations – ‘10’ for varicode bit ‘0’ and ‘00’ for varicode bit ‘1’. QPSK serializer uses convolutional coder of length 5 to calculate the symbol. The differential phase state machine determines the sign of both I and Q signals (+1 or -1) based on the current symbol. It also determines the wave shape selector based on the current and the previous symbol. Components from the character FIFO to the differential phase state machine are clocked at 31.25 Hz, which is 8000/256. All elements from the wave shapers are clocked at 8000 Hz. The wave shapers produce 31.25 Hz envelopes clocked at 8000 Hz that are multiplied with the quadrature sinusoidal signals generated by the sine/cosine generator. The output signal is created by adding the two orthogonal waveforms together.

The external view of the BPSK31TXChannel and QPSK31TXChannel waveforms is shown in Figure 6-2. As expected, the waveforms define TXChannel interfaces to be compatible with the test harness.

The internal structure of BPSK31TXChannel is shown in Figure 6-3. QPSK31TXChannel has a similar structure with QPSK31BitEncoder component in place of BPSK31BitEncoder.
The PSK31CharEncoder is a composite transcoder and serializer that encodes ASCII characters into varicode words and then serializes them. This component can be further decomposed as shown in Figure 6-4.

The ASCII character read from input FIFO is translated into a Varicode code. The code together with its length in bits is put on the input of the shift register, which reads these two values as soon as it’s empty. PSK31TXStateMachine is a state machine component that controls the process of serializing the Varicode words. It inserts separation bits in
between the consecutive words by writing false value onto the shift register’s shift port, which makes the shift register output bit 0. When the separation bits are finished and the shift register is not empty, the state machine assert true value on the shift register’s shift port which allows the shift register to shift out the bits of the varicode word. The state machine also inserts a number of zero bits at the beginning of transmission before any data bits are transmitted – it is the so called lead-in sequence or preamble. It inserts zero bits when the character fifo is empty but the transmit function is activated (e.g. a pause in typing characters). It finally also inserts zero bits at the end of transmitted data – it is the so called lead-out sequence or post-amble. The length of the lead-in or lead-out sequence is not standardized but typically it is 31-62 bits long (i.e. 1 to 2 seconds worth of transmitting). The purpose of these sequences is to allow the receivers to synchronize (lead-in) and/or flush their receive buffers (lead-out). Since every sequence of 2 or more consecutive zero bits is a separator, these sequences are ignored by the character decoders and can be of any arbitrary length.

**PSK31TXStateMachine** is a state machine component. Its state transition diagram is shown in Figure 6-5.

![Figure 6-5 State transition diagram for PSK31TXStateMachine.](image)

The state machine sits in the **IDLE** state when the transmitter is inactive. When the transmitter starts, the state machine enters **LEAD_IN** state in which zero bits are being shifted out. It stays in **LEAD_IN** state until the number of zero bits shifted out is equal or greater than **LEAD_IN_BITS**. It then enters either **SEND_IDLE_BITS** or **SEND_DATA_BITS** depending on whether there is data in the shift register. If the shift
register is not empty, the state machine remains in \textit{SEND\_DATA\_BITS} until all data bits have been shifted out and then enters the \textit{SEND\_SEPARATION\_BITS} state for \textit{NUM\_SEP\_BITS} cycles. Then it either goes back to \textit{SEND\_DATA\_BITS} or \textit{SEND\_IDLE\_BITS} depending on whether the shift register has data or not. The reception of the transmitter stopping event causes the state machine enter the state \textit{LEAD\_OUT} in which a lead-out sequence of zero bits is shifted out.

The above state transition diagram is the basis of the state machine, which is described in the state machine realization class for \textit{PSK31TXStateMachine}.

\textit{BPSK31TXChannel} and \textit{QPSK31TXChannel} implement a set of run-time parameters. Their names, types and default values are listed in Table 6-1. For a full list of all run-time parameters refer to Table 8-2 in Appendix 8.4.

\begin{table}[h]
\centering
\begin{tabular}{|l|l|l|l|}
\hline
\textbf{Name} & \textbf{Type} & \textbf{Default Value} & \textbf{Notes} \\
\hline
fCenter & double & 800.0 & Center frequency \\
\hline
nrSeparationBits & integer & 2 & Number of zero bits transmitted between characters \\
\hline
nrLeadInBits & integer & 62 & Number of bits in the lead-in (preamble) sequence \\
\hline
nrLeadOutBits & integer & 62 & Number of bits in the lead-out (post-amble) sequence \\
\hline
\end{tabular}
\caption{Run-time parameters of BPSK31TXChannel and QPSK31TXChannel.}
\end{table}

\subsection*{6.3 RTTY}
Radioteletype (RTTY) is a communications protocol that evolved from earlier landline teletype printers, which had their beginnings in mid-1800s. The beginnings of RTTY can be traced back to the end of 1920’s when the US Navy Department started experimenting with connecting teletype printers through the radio. RTTY gained popularity in the following years as it provided reliable communications over HF (High Frequency) links, which were the primary means for wireless connectivity over long distances before the advent of geostationary satellites. Although the satellites and Internet made RTTY obsolete in many applications, it is still used in some maritime services and is also very popular in the amateur radio service.

Since RTTY evolved from electromechanical landline-based teletype printers, it retained some of its predecessor properties and uses the same terminology though not necessarily in the same meaning. For example – electromechanical teletype printers use two voltage levels for two binary values – typically +80V for logical ‘1’ (or \textit{mark} level) and -80V for logical ‘0’ (or \textit{space} level). RTTY also uses \textit{mark} and \textit{space} concepts but they no longer apply to voltage levels – they actually apply to two frequencies.

RTTY is an asynchronous mode protocol based on Frequency Shift Keying. It uses two frequencies for \textit{mark} and \textit{space} levels. The difference between these frequencies is called \textit{shift} and its value depends on many factors – the baud rate, the transmitter frequency etc. In amateur radio service RTTY on HF bands uses 45.45 baud rate and 170 Hz frequency shift. The baud rates used on higher frequencies (VHF, UHF) are often higher than 45.45 and higher baud rates also require wider frequency shifts to minimize errors.
In RTTY the transmitted text is encoded in the 5-bit International Telegraph Alphabet No. 2 (ITA) [CCITT S.1 1998].

Table 6-2  International Telegraph Alphabet No. 2 [CCITT S.1 1998].

<table>
<thead>
<tr>
<th>Code (lsb first)</th>
<th>Letter shift</th>
<th>Figure shift</th>
</tr>
</thead>
<tbody>
<tr>
<td>00000</td>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>00001</td>
<td>E</td>
<td>3</td>
</tr>
<tr>
<td>00010</td>
<td>LF (line feed)</td>
<td>LF (line feed)</td>
</tr>
<tr>
<td>00011</td>
<td>A</td>
<td>–</td>
</tr>
<tr>
<td>00100</td>
<td>SPACE</td>
<td>SPACE</td>
</tr>
<tr>
<td>00101</td>
<td>S</td>
<td>BELL</td>
</tr>
<tr>
<td>00110</td>
<td>I</td>
<td>8</td>
</tr>
<tr>
<td>00111</td>
<td>U</td>
<td>7</td>
</tr>
<tr>
<td>01000</td>
<td>CR (carriage return)</td>
<td>CR (carriage return)</td>
</tr>
<tr>
<td>01001</td>
<td>D</td>
<td>$</td>
</tr>
<tr>
<td>01010</td>
<td>R</td>
<td>4</td>
</tr>
<tr>
<td>01011</td>
<td>J</td>
<td>'</td>
</tr>
<tr>
<td>01100</td>
<td>N</td>
<td>.</td>
</tr>
<tr>
<td>01101</td>
<td>F</td>
<td>!</td>
</tr>
<tr>
<td>01110</td>
<td>C</td>
<td>:</td>
</tr>
<tr>
<td>01111</td>
<td>K</td>
<td>(</td>
</tr>
<tr>
<td>10000</td>
<td>T</td>
<td>5</td>
</tr>
<tr>
<td>10001</td>
<td>Z</td>
<td>&quot;</td>
</tr>
<tr>
<td>10010</td>
<td>L</td>
<td>)</td>
</tr>
<tr>
<td>10011</td>
<td>W</td>
<td>2</td>
</tr>
<tr>
<td>10100</td>
<td>H</td>
<td>#</td>
</tr>
<tr>
<td>10101</td>
<td>Y</td>
<td>6</td>
</tr>
<tr>
<td>10110</td>
<td>P</td>
<td>0</td>
</tr>
<tr>
<td>10111</td>
<td>Q</td>
<td>1</td>
</tr>
<tr>
<td>11000</td>
<td>O</td>
<td>9</td>
</tr>
<tr>
<td>11001</td>
<td>B</td>
<td>?</td>
</tr>
<tr>
<td>11010</td>
<td>G</td>
<td>&amp;</td>
</tr>
<tr>
<td>11011</td>
<td>SHIFT TO FIGURES</td>
<td></td>
</tr>
<tr>
<td>11100</td>
<td>M</td>
<td>.</td>
</tr>
<tr>
<td>11101</td>
<td>X</td>
<td>/</td>
</tr>
<tr>
<td>11110</td>
<td>V</td>
<td>:</td>
</tr>
<tr>
<td>11111</td>
<td>SHIFT TO LETTERS</td>
<td></td>
</tr>
</tbody>
</table>
Since the 32 possible combinations of 5 bits are not sufficient to encode all characters and digits, ITA2 uses two modes to shift the meaning of the codes. In the letters mode most codes are interpreted as letters of the alphabet, in the figures mode they are interpreted as digits and some special characters. The receiver is shifted into figures mode on the reception of code ‘11011’ while the code ‘11111’ shifts the receiver back to letters. The interpretation of certain codes (LF, CR, BELL, NULL) is identical in both modes.

The set of characters in ITA2 constitutes only a small subset of ASCII characters. ITA2 doesn’t distinguish between lowercase and uppercase letters so each upper and lowercase pair of ASCII characters is matched into a single code in ITA2. In addition to this only digits and a small subset of control codes, major punctuation symbols and selected few special characters are supported in ITA2. When translating text from ASCII to ITA2, characters not supported by ITA2 are usually omitted from the message so it is very important that the messages prepared for transmission over RTTY use only the basic set of characters.

Since the transmission of characters in RTTY is asynchronous, the characters have to be delineated by additional symbols.

![Diagram of asynchronous transmission of characters in RTTY](image)

**Figure 6-6  Asynchronous transmission of characters in RTTY.**

The RTTY transmitter remains at the *mark* frequency in between the characters. The transmission starts with the shift to *space* frequency for the duration of 1 bit. This is so-called *start bit*. After the start bit, the 5 bits of ITA2 code are sent by shifting the transmitter’s frequency to *mark* or *space* depending on the value of particular bit ( 1 = *mark*, 0 = *space*). The end of the character is delineated by shifting to *mark* frequency for the duration of 1, 1.5 or 2 bits depending on the accepted format. This is so called *stop bit* – the name is obviously a misnomer as the duration of *stop bit* can be longer than 1 bit. The typical length of a stop bit in amateur radio service is 1.5 bit.

If the input characters were coming at a slow and/or irregular rate, the transmitter could potentially remain at the *mark* frequency for long periods of time, which would have a negative impact on the state of synchronization of the receivers. In order to avoid such situation the SHIFT TO LETTERS character is being sent as an idle character when there’s no data. Because of the characteristic sound the shift character produces when
RTTY is generated in audio frequencies, the shift character used as an idle character is also called a *diddle* or a *diddle character*.

A block diagram of the transmit section of RTTY is shown in Figure 6-7.

![Figure 6-7 Transmit section of RTTY.](image)

The input characters are buffered in a FIFO. They are encoded into ITA2 characters by the Baudot encoder. This encoder inserts the necessary shift codes when transitions from letters to figures or vice versa are required. It also inserts a *diddle character* when no data is available in the FIFO. The 5-bit ITA2 codes are serialized in the next step. The serializer prepends the code word with a start bit and appends the stop bit. The output of the serializer controls the input of the FSK modulator, which generates one of the two mark and space frequencies depending on the value on its input. The output of the modulator is the actual RTTY signal.

As in the case of PSK31 waveforms, the RTTY waveform (*RTTYTXChannel*) has been defined as a component with *TXChannel* interfaces in order to be compatible with the test harness (Figure 6-8).

![Figure 6-8 The external view of RTTYTXChannel.](image)

Its internal structure is shown in Figure 6-9.
The waveform is activated on the reception of TxStart event. TxStart causes the RTTYCharEncoder to initialize and then send ShiftStart signal to RTTYTxShiftSM. The state machine initializes itself, and then generates the event ShiftOn, which starts the FSKModulator and signals RTTYCharEncoder that the waveform is ready to start sending out bits.

RTTYCharEncoder is responsible for transcoding ASCII characters into ITA2 code words, inserting shift codes as necessary and inserting diddle characters into the character stream when there’s no data in the FIFO. RTTYTxShiftSM is responsible for serializing data, inserting start and stop bits and controlling the FSKModulator.

RTTYTxShiftSM is a state machine component. Its state transition diagram is shown in Figure 6-10.
Figure 6-10 State transition diagram for RTTYTxShiftSM.

The state machine remains in the IDLE state until it receives Start event on which it transitions to LEAD_IN state. On entering LEAD_IN state ShiftOn event is sent out to turn on the FSKModulator and acknowledge to RTTYCharEncoder that the state machine is ready. In LEAD_IN the waveform sends out a preamble (a lead-in signal) at mark frequency, the length of which is determined by the parameter leadInSamples. After the preamble depending on availability of the data from RTTYCharEncoder the state machine enters either SEND_MARK state, which is a state in which the waveform sits idly at the mark frequency, or SEND_START_BIT state, in which the start bit of the new character is sent out. Note that in normal circumstances SEND_MARK state should never be entered because as it was earlier discussed the character encoder is supposed to insert diddle characters when there’s no data in the buffer. After the start bits, the five bits of the code word (SEND_DATA_BITS) and the stop bit (SEND_STOP_BITS) are sent and if the next character is available (as it always should be with diddles) the state machine transition backs to SEND_START_BIT initiating the next character. The reception of Stop signal in any state except for IDLE and LEAD_OUT causes the property stopPending to be set to true value. At the conclusion of sending the current character, if the stopPending flag is set the state machine transitions to LEAD_OUT state in which a post-amble sequence at the mark frequency is generated before the transition to IDLE.

RTTYTXChannel also uses a set of run-time parameters (Table 6-3).
Table 6-3  Run-time parameters of RTTTYTXChannel.

<table>
<thead>
<tr>
<th>Name</th>
<th>Type</th>
<th>Default Value</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>fCenter</td>
<td>double</td>
<td>800.0</td>
<td>Center frequency = ( \frac{f_{\text{mark}} + f_{\text{space}}}{2} )</td>
</tr>
<tr>
<td>fShift</td>
<td>double</td>
<td>170.0</td>
<td>The difference between mark and space frequencies</td>
</tr>
<tr>
<td>baudRate</td>
<td>double</td>
<td>45.45</td>
<td>The baud rate of the transmission</td>
</tr>
<tr>
<td>stopBits</td>
<td>double</td>
<td>1.5</td>
<td>The length of the character stop sequence (stop bit) in the baud rate</td>
</tr>
<tr>
<td>markHigher</td>
<td>boolean</td>
<td>true</td>
<td>If true then mark frequency is higher than space frequency</td>
</tr>
<tr>
<td>nrStartDiddle</td>
<td>integer</td>
<td>10</td>
<td>Number of diddle characters inserted before the data</td>
</tr>
<tr>
<td>nrStopDiddle</td>
<td>integer</td>
<td>10</td>
<td>Number of diddle characters appended after the data</td>
</tr>
<tr>
<td>nrLeadInMarkBits</td>
<td>integer</td>
<td>20</td>
<td>The length of the mark frequency preamble (lead-in) in bits.</td>
</tr>
<tr>
<td>nrLeadOutMarkBits</td>
<td>integer</td>
<td>20</td>
<td>The length of the mark frequency post-amble (lead-out) in bits.</td>
</tr>
</tbody>
</table>

6.4 Reconfiguration scenarios

In the process of experimenting with the proof-of-concept system several experiments for different reconfiguration scenarios have been executed.

6.4.1 A known waveform

In this scenario the Master requests a waveform that is known by Slave and is available immediately for instantiation as a Java class. The table below includes the asserted, the inferred and the total number of facts before and after the execution of the scenario.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>Nr facts before execution</th>
<th>Nr facts after execution</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assert</td>
<td>Infer</td>
</tr>
<tr>
<td>BPSK31TXChannel</td>
<td>5293</td>
<td>26165</td>
</tr>
<tr>
<td>QPSK31TXChannel</td>
<td>5293</td>
<td>26165</td>
</tr>
<tr>
<td>RTTTYTXChannel</td>
<td>5293</td>
<td>26165</td>
</tr>
</tbody>
</table>

Facts in BaseVISor are triples (subject, predicate, object). An example of a few facts related to obr:Adder, retrieved from BaseVISor’s fact base is shown below:

subject predicate object

<table>
<thead>
<tr>
<th>obr:Adder rdf:type</th>
<th>owl:Thing</th>
</tr>
</thead>
<tbody>
<tr>
<td>obr:Adder rdf:type</td>
<td>owl:Class</td>
</tr>
<tr>
<td>obr:Adder rdfs:subClassOf</td>
<td>obr:BasicComponent</td>
</tr>
<tr>
<td>obr:Adder rdfs:subClassOf</td>
<td>obr:Component</td>
</tr>
<tr>
<td>obr:Adder rdfs:subClassOf</td>
<td>obr:Adder</td>
</tr>
<tr>
<td>obr:Adder owl:equivalentClass</td>
<td>obr:Adder</td>
</tr>
<tr>
<td>obr:Adder owl:sameAs</td>
<td>obr:Adder</td>
</tr>
</tbody>
</table>

Figure 6-11  An example of facts in BaseVISor's fact base.
6.4.2 An unknown waveform

In this scenario, the Master process requests a waveform but the waveform is not known to the Slave. The Slave executes a query for the waveform description and it has all the necessary subcomponents ready to be instantiated from Java. The rightmost column in the table holds the names of components for which the Slave had to send a query.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>Nr facts before exec.</th>
<th>Nr facts after exec.</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assert</td>
<td>Infer</td>
<td>Total</td>
</tr>
<tr>
<td>BPSK31TXChannel</td>
<td>5289</td>
<td>26068</td>
<td>31357</td>
</tr>
<tr>
<td>QPSK31TXChannel</td>
<td>5289</td>
<td>26068</td>
<td>31357</td>
</tr>
<tr>
<td>RTTYTXChannel</td>
<td>5289</td>
<td>26068</td>
<td>31357</td>
</tr>
</tbody>
</table>

6.4.3 An unknown waveform, unknown subcomponent composition

This scenario tests a situation, where a subcomponent of the unknown waveform is also unknown and it has to be composed from its own subcomponents before it can be used to compose the waveform. Note that the columns holding the number of facts after the execution hold two numbers for each waveform. The first two of these numbers were captured after the facts and rules from the first query (i.e. for BPSK31TXChannel) were incorporated in the fact base and processed by the inference engine. The second set of numbers was captured at the end of the reconfiguration process, when all facts and rules pertaining to the second query (i.e. for PSK31CharEncoder) were integrated in the knowledge base and processed by the inference engine.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>Nr facts before exec.</th>
<th>Nr facts after exec.</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assert</td>
<td>Infer</td>
<td>Total</td>
</tr>
<tr>
<td>BPSK31TXChannel</td>
<td>5285</td>
<td>25971</td>
<td>31256</td>
</tr>
<tr>
<td></td>
<td>5293</td>
<td></td>
<td></td>
</tr>
<tr>
<td>QPSK31TXChannel</td>
<td>5285</td>
<td>25971</td>
<td>31256</td>
</tr>
<tr>
<td></td>
<td>5293</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.4.4 An unknown waveform, unknown subcomponent state machine

In this scenario, a subcomponent of the unknown waveform is also unknown and it is a state machine component so it has to be generated from its description before it can be used to compose the waveform.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>Nr facts before exec.</th>
<th>Nr facts after exec.</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assert</td>
<td>Infer</td>
<td>Total</td>
</tr>
<tr>
<td>RTTYTXChannel</td>
<td>5285</td>
<td>25983</td>
<td>31268</td>
</tr>
<tr>
<td></td>
<td>5293</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
6.4.5 An unknown waveform, an unknown composite subcomponent, unknown state machine sub-subcomponent

In this scenario, the Slave node does not know a waveform, does not know a subcomponent of the waveform that is also a composition, finally it does not know the state machine component, which is a subcomponent of the unknown subcomponent of the waveform.

<table>
<thead>
<tr>
<th>Waveform</th>
<th>Nr facts before exec.</th>
<th>Nr facts after exec.</th>
<th>Queries</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Assert</td>
<td>Infer</td>
<td>Total</td>
</tr>
<tr>
<td>BPSK31TXChannel</td>
<td>5281</td>
<td>25886</td>
<td>31167</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>QPSK31TXChannel</td>
<td>5281</td>
<td>25886</td>
<td>31167</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

6.4.6 Reconfiguration through Set operation

The parameters of the communications can be modified through the use of the Set operation on the run-time parameters. All run-time parameters listed in Table 6-1 and Table 6-3 have been tested with their respective waveforms. Both types of waveform realization were tested – Java class instance and a component composition. All tests were completed successfully.

6.5 Verification of the platform independence

One of the requirements for the reconfiguration method is its independence from any specific hardware/software platform. This is one of two major reasons why Java was selected for the system’s software development (the other reason being the ease of interfacing with BaseVISor).

In order to verify the platform independence the Master and Slave processes were executed on computers running different operating systems as shown in the table below. In some of these tests both processes were run on the same host. All tests were successful.

<table>
<thead>
<tr>
<th>Master host</th>
<th>Slave host</th>
</tr>
</thead>
<tbody>
<tr>
<td>Windows 7 64-bit</td>
<td>same host as Master</td>
</tr>
<tr>
<td>Windows 7 64-bit</td>
<td>Windows XP</td>
</tr>
<tr>
<td>Windows XP</td>
<td>same host as Master</td>
</tr>
<tr>
<td>Windows 7 64-bit</td>
<td>LUbuntu Linux 12.04</td>
</tr>
<tr>
<td>LUbuntu Linux 12.04</td>
<td>same host as Master</td>
</tr>
</tbody>
</table>
6.6 Testing the boundaries of the applicability of the method

The use of a forward-chaining inference engine makes the scalability of the proposed method a concern. In order to gauge the behavior of the method for waveform and state machine models of much bigger size two series of experiments have been devised.

6.6.1 Testing for the limits of the size of a composition

In this series of tests a number of TXChannel-compatible components with increasing numbers of internal components have been generated and tested.

The general structure of such a test component is shown in Figure 6-12.

The component consists of a varying length cascade of multipliers, which is fed with a sinusoidal signal from SineCosineGenerator. The in1 inputs of all multipliers are connected to the ConstantOutput component with the constant value 1.0. The output of each multiplier, except for the last one, is connected to the input in2 of the next multiplier in the chain. The output of the last multiplier is connected to the external port sample_out. A special stub component TestRulesSM has been created for the purpose of satisfying the requirements of TXChannel interfaces so that these components could be used in the test harness. TestRulesSM works by simply looping back signals TxStart and TxStop to ClkEnable and ClkDisable.

Components of different sizes of the multiplier chain were tested in the test harness. A variety of parameters related to both the reconfiguration process and the online execution of the simulated ‘waveform’ were gathered using JConsole – a Java SDK utility allowing monitoring an execution of Java processes.
The Slave process which hosts the test harness has been assigned 2048MB of heap memory for these tests. At least 10 measurements were taken for each size of the component. The results after averaging and processing are presented in Table 6-5 and in the following graphs.

Table 6-5 Composition size testing results.

<table>
<thead>
<tr>
<th>Nr Components</th>
<th>Avg Incremental Memory Usage [MB]</th>
<th>Incremental Nr of Facts</th>
<th>Avg reconfiguration time [s]</th>
<th>Online CPU load</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>7.9</td>
<td>3542</td>
<td>4.56</td>
<td>&lt; 0.5%</td>
</tr>
<tr>
<td>5</td>
<td>10.0</td>
<td>3915</td>
<td>3.62</td>
<td>&lt; 0.5%</td>
</tr>
<tr>
<td>8</td>
<td>12.3</td>
<td>5034</td>
<td>4.20</td>
<td>&lt; 0.5%</td>
</tr>
<tr>
<td>13</td>
<td>16.6</td>
<td>6899</td>
<td>4.04</td>
<td>&lt; 0.5%</td>
</tr>
<tr>
<td>23</td>
<td>20.5</td>
<td>10629</td>
<td>5.01</td>
<td>&lt; 0.5%</td>
</tr>
<tr>
<td>53</td>
<td>55.6</td>
<td>21819</td>
<td>6.71</td>
<td>3.00%</td>
</tr>
<tr>
<td>103</td>
<td>161.2</td>
<td>40469</td>
<td>11.94</td>
<td>5.50%</td>
</tr>
<tr>
<td>203</td>
<td>656.7</td>
<td>77769</td>
<td>24.54</td>
<td>8.90%</td>
</tr>
<tr>
<td>303</td>
<td>1770.7</td>
<td>115069</td>
<td>52.69</td>
<td>13.40%</td>
</tr>
</tbody>
</table>

Figure 6-13 Average incremental memory use vs. number of components.
Figure 6-14  Incremental number of facts vs. number of components.

Figure 6-15  Average reconfiguration time vs. number of components.
As expected, the memory use becomes a problem with compositions containing large numbers of components. For a hypothetical embedded system with 1 GB of memory the maximum size of a composition would be around 200 components. This number seems to be adequate for representing a waveform if the SDR ontology contains functional blocks at a high level complexity. It would not be sufficient if the waveform had to be decomposed all the way to the very fundamental operations. For example, if a reasonably long, 70-tap FIR filter were to be composed from Multiplier, Adder and UnitDelay elements, it would require 210 components, which is the very verge of what a system with 1 GB of memory could do. The CPU load imposed by the composed component on the system, while running, scales linearly with the number of components. This result is in line with our expectations.

**6.6.2 Testing for the limits of the size of a state machine**

In this series of tests, TXChannel-compatible state machine components with increasing number of states and transitions were generated and tested using the same methodology as component size testing (see 6.6.1). The general structure of the generated state machine is shown in Figure 6-17.
Figure 6-17 State transition diagram for a test state machine component.

The state machine consists of a ring arrangement of several \( RUN_x \) states. When the state machine is activated, on the reception of each clock event, it transitions from one state in the ring to the next. On each transition between \( RUN_x \) states the action sequence \( \text{nextSample}() \) is executed – this action sequence calculates and sends to the output the next sample of a sinusoidal signal. The state machine is activated when it receives the \( TxStart \) event. The reception of the \( TxStop \) event makes the state machine transition from any \( RUN_x \) state back to \( IDLE \).

The results of measurements for different sizes of the state machine are presented in Table 6-6. The online CPU burden was below 0.5% in all cases; therefore it has been excluded from the table. Note that for the number of states greater than 161 the compilation in memory was failing due to exceptions from Java libraries. A possible workaround for this problem would be to store the source code for the state machine in an external file that would be externally compiled to a class file and then loaded.

<table>
<thead>
<tr>
<th>Nr States</th>
<th>Avg Incremental Memory Usage [MB]</th>
<th>Incremental Nr of Facts</th>
<th>Avg reconfiguration time [s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>5.7</td>
<td>3414</td>
<td>4.16</td>
</tr>
<tr>
<td>3</td>
<td>6.3</td>
<td>3570</td>
<td>3.83</td>
</tr>
<tr>
<td>6</td>
<td>9.2</td>
<td>4038</td>
<td>3.83</td>
</tr>
<tr>
<td>11</td>
<td>11.6</td>
<td>4818</td>
<td>3.70</td>
</tr>
<tr>
<td>21</td>
<td>21.1</td>
<td>6378</td>
<td>4.64</td>
</tr>
<tr>
<td>51</td>
<td>78.2</td>
<td>11058</td>
<td>5.23</td>
</tr>
<tr>
<td>101</td>
<td>333.7</td>
<td>18858</td>
<td>7.45</td>
</tr>
<tr>
<td>161</td>
<td>1058.0</td>
<td>28218</td>
<td>11.85</td>
</tr>
</tbody>
</table>
Figure 6-18  Average incremental memory use vs. number of components.

Figure 6-19  Incremental number of facts vs. number of states.
Figure 6-20  Average reconfiguration time vs. number of states

The number of states in excess of 100 for 1 GB of memory seems reasonable for even complex state machines. This number would probably increase if the state machine’s auto-generated code were stored and compiled externally, not in the memory buffer.
7 Conclusions

7.1 Contributions

This research made the following contributions:

- Identified the problem with interoperability of currently used communications systems.
- Proposed that dynamic reconfiguration is the key for achieving interoperability.
- Identified general requirements for the communication systems reconfigurability solution.
- Proposed an outline of the reconfiguration method based on the transfer of waveform description and common ontology.
- Researched and analyzed language issues in Cognitive Radio, co-authored a paper in Proceedings of the IEEE.
- Researched and analyzed software component technologies and self-controlling software.
- Proposed the use of constructivism approach to knowledge transfer.
- Proposed the use of correct-by-construction approach to building waveform models from their descriptions.
- Proposed the use of state machines to capture the behavioral parts of the waveform functionality.
- Developed a general architecture of a system in which waveform descriptions can be transferred.
- Developed ontology for waveform description.
- Developed ontology for state machine description.
- Developed concept of component realization, developed details of the process in which the type of realization for a component is selected.
- Developed the details of the overall waveform transfer and construction process.
- Built a proof-of-concept system to test the proposed method
  - Developed component composition middleware
  - Developed reconfiguration software, which includes state machine generation and assembly of composite components
  - Integrated BaseVISor inference engine with the proof-of-concept system, augmented BaseVISor’s capabilities with procedural attachments
  - Developed software for control communications over TCP/IP connections
  - Selected a set of waveforms for experiments, defined a set of necessary components, developed their descriptions including component compositions and state machine definitions and demonstrated that the waveforms can be either selected or generated from the descriptions
  - Wrote approx 20,000 lines of Java code and 18,400 lines of BaseVISor rules to support the experimentation.
7.2 Future research

The proof-of-concept system built for the proposed method provides a starting point for further research in the area of an ontology-based approach to knowledge transfer and the cognitive radio reconfiguration.

Future research may include:

- The security implications of the waveform reconfiguration. The security-related topics are so broad that they were not considered in this work. Some research areas might include:
  - different levels of trust between cognitive radios and their impact on waveform description transfer
  - temporary vs. permanent knowledge augmentation
  - different levels of sensitivity of waveforms.
- Fall back procedures to regain connectivity if the reconfiguration process fails.
- Compatibility issues related to potential versioning of:
  - Base SDR ontology
  - Component middleware and reconfiguration software
  - SDR components
- Performance improvements, particularly in the memory use area.
- Further experimentation, particularly with implementing more complex waveforms to better gauge the boundaries of the usefulness of the system.
- The replacement of BaseVISor with other inference engines, preferably ones that use different principles for reasoning (i.e. not Rete network based).
- The application of the method in a completely different domain.
8 Appendices

8.1 Constructivism - a paradigm in cognitive science.

When a human being encounters a concept he or she is not familiar with, the usual reaction is the question “What is it?” addressed to the person who used this concept in the conversation. Such a question usually starts a process (sometimes involving additional questions and answers) in which a fragment of someone’s knowledge is transferred to another person.

The mechanism in which a person acquires knowledge is of great interest for researchers in cognitive science. Due to difficulties in explaining how the human brain works internally there are many theories that try to explain the process of learning. Three of such theories gained wider recognition:

- **Behaviorism** – is focused on the objectively observable behavior and defines learning as the acquisition of new behaviors
- **Cognitivism** – tries to explain the learning as a sequence of elementary cognitive abilities (recognize, recall, analyze, reflect, apply, create, evaluate, etc.)
- **Constructivism** – considers knowledge as a structure that is constructed in the memory of the learner rather than a static (absolute) entity that is common to all agents [Marton 1997].

The constructivism paradigm is particularly interesting, as it seems to fit best the process of transferring knowledge from one cognitive radio to another. Its consequence is that knowledge cannot be just copied from one cognitive agent to another but rather needs to be first communicated and then constructed using mental capabilities of the learner. In this way new concepts can be communicated efficiently as the cognitive agent uses previously learned concepts in the process of acquiring new knowledge.

In order to illustrate this point let’s consider a forward chaining inference engine with an associated knowledge base KB. Since a forward chaining inference engine generates all facts that can be deduced given a particular set of rules and initial facts, a materialization of the knowledge base at the time the inference engine is not actively processing facts always yields the full set of facts (asserted and inferred).

Let’s assume that at a certain point of time the materialization M1 of the KB has cardinality |M1|. Assume that a fact f has been asserted and the inference engine processed it. At this point the materialization M2 of the KB has cardinality |M2|.

There are three possible situations:

- |M2| = |M1|  
  The fact f is redundant i.e. it has already been asserted or inferred
- |M2| = |M1| + 1  
  The fact f has been added to KB but no further inferences were made
\[|M_2| > |M_1| + 1\]

The fact \(f\) has been added to KB and additional facts have been inferred based on it.

The third case in which the incremental number of facts in the knowledge base exceeds the number of facts asserted can be interpreted as the manifestation of the constructivism – additional knowledge has been constructed based on the newly asserted facts and the knowledge acquired before.

It should be noted that the constructivism approach fits well into the requirements for interoperability. Cognitive radios may be built based on different hardware and/or software platforms therefore their internal knowledge representations might be different. Because of this, the direct transfer of the contents of the knowledge base from one radio to another is not feasible. In the constructivism the internal representation of knowledge is always constructed individually by each of the cognitive radios based on the facts and rules it acquired.

### 8.2 Waveform reconfiguration UML activity diagram
Figure 8-1  UML activity diagram for reconfiguration process.
8.3 PSK31

PSK31 was developed in late 1990’s by Peter Martinez G3PLX [Martinez 1998]. It was developed with several goals in mind:

- Excellent weak signal properties for long distance communications through ionospheric propagation modes on HF bands (1.8-30 MHz).
- Low inherent latency and reasonable bit rate enabling two-way interactive communications by typing on the keyboard in the real-time.
- No specialized hardware or modifications to existing amateur radio communications equipment required to support it.

The designed protocol fulfills all those requirements and has become a tremendous success in the amateur radio community.

PSK31 uses differential phase shift keying (PSK) modulation at 31.25 bit/s rate of transfer. The low bit rate, together with cosine waveform shaping of the phase-shifted carrier, make the PSK31 signal’s spectrum very narrow (ideal signal requires only 31.25 Hz of bandwidth). Concentrating the output power of the transmitter in such a narrow band, together with the ability to use very narrow filters on the receive side, makes PSK31 an excellent choice for intercontinental contacts where signals are usually weak, frequently barely above atmospheric noise level.

PSK31 uses the Varicode alphabet to represent ASCII characters [Wheatley 2008]. Characters in Varicode alphabet have different lengths with characters used more frequently having codes shorter than standard 8 bits (e.g. ‘e’ = 11, ‘o’ = 111, ‘t’ = 101) and characters used rarely having codes longer than 8 bits. Two consecutive zero bits ‘00’ are used to delineate consecutive characters. Because of this type of symbol synchronization no Varicode code contains two zero bits in two consecutive positions within the word.

8.3.1 Operating PSK31 station

A typical setup for an amateur radio station working in PSK31 mode is shown in Figure 8-2. PSK31 software is running on the computer connected to the HF transceiver. At the minimum, two connections are required – one that connects the audio output of the HF transceiver to the input of the computer sound card and another one that connects the output of the soundcard to the microphone input of the transceiver (or to an auxiliary audio input frequently found in modern transceivers for connecting external modulation sources like computers).

In the minimum setup with only audio connections, the operation requires a lot of manual control – frequency tuning, switching the transceiver between transmit (TX) and receive (RX) modes, switching the PSK31 program between RX and TX modes, manual setting of the received frequency in the PSK31 program’s contact log, etc. For these reasons additional connections are usually made between the computer and the transceiver. The PTT connection is controlling the transceiver state – it remains in the receive mode if the PTT input is open and switched to TX mode if PTT input is shorted to the ground. PTT control can be achieved through any of the computer’s IO ports (serial or parallel).
The CAT interface connection is used to control the transceiver from the computer. It usually is implemented as a serial connection (or USB link in some modern transceivers) and in most implementations is bidirectional (some old transceivers used unidirectional CAT interfaces – the computer was able to set parameters in the radio but wasn’t able to read anything from the radio). In majority of the modern radios it is possible to set and read all parameters of the radio through the CAT interface making it possible to have full control of the communications process without even touching the radio.

Figure 8-2  PSK31 station setup.

Thanks to freely available software packages that support PSK31, operating this mode is very easy and intuitive. Most applications are organized in a way similar to fldigi, which is shown in Figure 8-3. The bottom part of the window is a “waterfall” display that shows the spectrum of the audio signal received from the transceiver. As it can easily be seen from the picture, there’s a strong signal at the audio frequency 800Hz. The receive cursor (two red lines) has been aligned with the signal and it is being decoded and displayed in the top part of the window. If the program and the transceiver were switched into the transmit mode, the program would start generating PSK31 signal at the same audio frequency as determined by the receive cursor (i.e. 800 Hz). The middle part of the window, which is empty in the picture, is used to input characters from the keyboard. The input window works as a FIFO so an arbitrarily long sequence of characters can be typed into the window at any arbitrarily fast rate. If the rate of the character input is very low, the PSK31 transmitter inserts idle characters to keep sending at 31.21 bit rate so that the receivers can maintain bit clock synchronization even when no data is being sent at a moment.
8.3.2 PSK31 signal generation

A block diagram of the transmit section of the PSK31 module is shown below.

There are two variants of the PSK31 signal – BPSK and QPSK.

1. BPSK (Binary Phase Shift Keying) uses only two phase shifts – 0 deg and 180 deg. Bit “1” of the input data is represented as no shift in phase (00 bit code at the output of the serializer), bit “0” is represented as 180 deg phase shift (code 10).
2. QPSK (Quadrature Phase Shift Keying) uses four possible phase shifts. This potentially would allow to encode two bits of data in a single symbol, effectively doubling the bit rate. In PSK31 however the additional bit capacity is used for error correction through convolutional coding. The two-bit symbols at the output of the serializer are calculated using two polynomials: $s_0 = x^4 + x^2 + x^1 + x^0$, $s_1 = x^4 + x^3 + x^0$, where $x$ is an inverted bit of input data. The inversion of the input data bits is done so that if the transmitter is idle (i.e. the input data is a stream of zeros), the output of the serializer $s_1s_0$ is equal 10, which produces 180 phase changes exactly like in BPSK.

The characters are buffered in the FIFO buffer of an arbitrary length. In practical implementations the character buffer’s length tends to be significant – in hundreds and thousands of characters to allow for copy and paste operations between other applications and the transmit buffer.

The 8-bit ASCII characters from the buffer are transcoded into variable length Varicode codes (2-15 bit long). The varicodes are next serialized into 2 bit symbols. In the BPSK mode only two symbols are used – 00 and 10 - representing 0 and 180 deg shift respectively. In the QPSK mode all four possible 2 bit symbols are used. The state machine calculates next phase based on the current state and the incoming 2-bit symbol. Additionally, in the QPSK, the serializer is using the convolutional coder to introduce redundancy that allows for error correction on the receiver side. The orthogonal components of the output signal (I and Q) are next multiplied by the wave shaping function and then combined into a single signal in the quadrature modulator built off two multipliers, an adder and a quadrature carrier generator that produces sine and cosine signals of the specified frequency. The samples of such created signal are sent to the soundcard to be converted to audio signal.

8.3.3 PSK31 receiver

The PSK31 receiver is much more complex than the transmit section. A simplified diagram is shown in Figure 8-5.

![Figure 8-5 Simplified block diagram of the receive section of PSK31 waveform](Wheatley 2008).
The received audio signal from the HF transceiver is sampled by the sound card at 8 kHz sampling frequency. The first stage of the PSK31 receive path is a complex mixer, which converts the PSK31 signal at the specific receiving audio frequency into a complex baseband signal, i.e. the pair of signals I and Q. The I and Q signals are next decimated 16 times to the sampling frequency of 500Hz. At that point the signals are fed into two types of filters. The Frequency Error Filter is a 65-tap low pass FIR with the bandwidth a bit larger that the width of the PSK31 baseband signal. The output of this filter is used by the Automatic Gain Control (AGC) and Automatic Frequency Control (AFC) circuitry. The second type of the FIR filter – Matched Data Bit filter – has a narrower bandwidth (unsuitable for AFC and AGC purposes) and is designed to provide the best SNR while keeping the inter-symbol interference minimal.

The signal at the output of the AGC block is used to control the level of the signal that is fed into the AFC module and into the decoding. The frequency error signal from the AFC circuitry is used to correct the frequency of the sine/cosine generator.

The actual decoding starts with the clock signal recovery and associated sample selection module. The current and the previous values of the data bits in the I and Q signal paths are fed into the phase difference detector. The output of the phase difference detector is a two-bit symbol. In case of BPSK31 it is either 00 or 10 for no or 180 deg change in phase, respectively. Since in BPSK31 this is equivalent to 1 or 0 bit of input data, the inverted higher bit of the symbol is fed into the shift register and is used to lookup an appropriate varicode code to be sent to the output of the receiver. In case of QPSK31, the stream of symbols from the phase detector is sent to the Viterbi decoder, which provides error correction and the decoding of the convolutional code. The stream of the output bits from the Viterbi decoder is sent to the shift register as in the BPSK31 case.

8.3.4 The Varicode alphabet

PSK31 uses a variable-length alphabet to represent ASCII characters [Wheatley 2008]. The length of the code for a particular character depends on how frequently it is used in English with more commonly used characters having short codes (e.g. ‘ ‘ (space) = 1, ‘e’ = 11, ‘o’ = 111) and rarely used having codes with lengths significantly exceeding 8 bits. Since two consecutive zero bits delineate consecutive characters in PSK31, no Varicode contains two zeros in consecutive positions.

<table>
<thead>
<tr>
<th>ASCII</th>
<th>Varicode</th>
<th>ASCII</th>
<th>Varicode</th>
<th>ASCII</th>
<th>Varicode</th>
<th>ASCII</th>
<th>Varicode</th>
</tr>
</thead>
<tbody>
<tr>
<td>NULL</td>
<td>1010101011</td>
<td>@</td>
<td>1010111101</td>
<td>128</td>
<td>1110111101</td>
<td>192</td>
<td>1101110111</td>
</tr>
<tr>
<td>SOH</td>
<td>1011011011</td>
<td>A</td>
<td>1111101</td>
<td>129</td>
<td>1110111111</td>
<td>193</td>
<td>1101111011</td>
</tr>
<tr>
<td>STX</td>
<td>1011101101</td>
<td>B</td>
<td>11101101</td>
<td>130</td>
<td>1110110101</td>
<td>194</td>
<td>1101111111</td>
</tr>
<tr>
<td>ETX</td>
<td>1101110111</td>
<td>C</td>
<td>101010101</td>
<td>131</td>
<td>1111010111</td>
<td>195</td>
<td>1101111011</td>
</tr>
<tr>
<td>EOT</td>
<td>1011101011</td>
<td>D</td>
<td>10110101</td>
<td>132</td>
<td>1111011011</td>
<td>196</td>
<td>1101111111</td>
</tr>
<tr>
<td>ENQ</td>
<td>1101011111</td>
<td>E</td>
<td>11101111</td>
<td>133</td>
<td>1111011101</td>
<td>197</td>
<td>1101111111</td>
</tr>
<tr>
<td>ACK</td>
<td>1101110111</td>
<td>F</td>
<td>11011011</td>
<td>134</td>
<td>1111011111</td>
<td>198</td>
<td>1110101011</td>
</tr>
<tr>
<td>BEL</td>
<td>1011111111</td>
<td>G</td>
<td>11111101</td>
<td>135</td>
<td>1111101011</td>
<td>199</td>
<td>1110101011</td>
</tr>
<tr>
<td>BS</td>
<td>1011111111</td>
<td>H</td>
<td>101010101</td>
<td>136</td>
<td>1111101011</td>
<td>200</td>
<td>1110101011</td>
</tr>
<tr>
<td>HT</td>
<td>11101111</td>
<td>I</td>
<td>11111111</td>
<td>137</td>
<td>1111101111</td>
<td>201</td>
<td>1110101101</td>
</tr>
<tr>
<td>LF</td>
<td>11101</td>
<td>J</td>
<td>1111111101</td>
<td>138</td>
<td>1111101011</td>
<td>202</td>
<td>1110101111</td>
</tr>
<tr>
<td>VT</td>
<td>1101101111</td>
<td>K</td>
<td>101111101</td>
<td>139</td>
<td>1111101111</td>
<td>203</td>
<td>1110110101</td>
</tr>
</tbody>
</table>

Table 8-1 The Varicode alphabet [Wheatley 2008].
FF 1011011101 L 11010111 140 1111111011 204 11101101101
CR 11111 M 10111011 141 1111111111 205 11101101111
SO 1101110101 N 11011101 142 1111111111 206 11101101101
SI 1110101011 O 10101011 143 10101010101 207 11101101111
DLE 1011110111 P 11010101 144 10101010111 208 111011 10111
DC1 1110101101 Q 111011101 145 10101011011 209 11101101101
DC2 1110101111 R 10101111 146 10101011101 210 11101101111
DC3 1110101111 S 1101111 147 10101011111 211 111101 01011
DC4 1101011011 T 1101101 148 10101101011 212 111101 01101
NAK 1101101011 U 101010111 149 10101101101 213 1111 0101111
SYN 1101101101 V 110110101 150 10101101111 214 1111 0110101
ETB 1101010111 W 1101101 151 10101110101 215 1111 0110111
CAN 1101111011 X 111011011 152 10101110111 216 1111 0111011
ESC 1101010101 \ 111110111 153 10101111111 217 1111 1010101
FS 1101011101 \ 111101111 154 10110101011 218 11111 010111
GS 1110111011 \] 111111011 155 10110101101 219 11111 011011
RS 1101110111 ^ 101101101 156 10110110101 220 11111 011111
US 1101111111 _ 1101101 157 10110111011 221 1111110101 1
SPACE 1 ` 111111111 a 1011 161 10110111011 225 11111101101
" 101011111 b 1011111 162 10110111101 226 11111101111
# 111110101 c 101111 163 10110111111 227 1111111010 1
$ 111011011 d 10101 164 10110101011 228 1111111011 1
% 1011010101 e 11 165 11010101111 229 11111111011
& 1010110111 f 111101 166 11011010111 230 111111111 11
' 101111111 g 1011011 167 11011011101 231 11111111111
( 11110111 h 101011 168 11011011111 232 10101010101
) 1110111 i 1101 169 11011101011 233 1010101011
* 101011111 j 1011111 170 11011110101 234 10101010111
+ 111011111 k 1011111 171 11011110111 235 1010101011
, 1110101 l 1101 172 11011110101 236 1010101011
- 110101 m 111011 173 11011110111 237 1010101011
. 101011 n 1111 174 11011111011 238 10101011101
/ 110101111 o 111 175 11011111101 239 10101011111
0 101011111 p 11111 176 11011111111 240 1010101101
1 1011111 q 110111111 177 11010101101 241 1010101101
2 110101111 r 10101 178 11010101101 242 1010101101
3 1111111 s 10111 179 11010101111 243 1010101101
4 101110111 t 101 180 11010110101 244 1010101111
5 101010111 u 11011 181 11010110111 245 10101011011
6 101010111 v 111011 182 11010111011 246 1010101101
7 110101111 w 110101 183 11010111101 247 10101011011
8 110101111 x 1101111 184 11010111111 248 1010101101
9 110110111 y 1011101 185 11011010101 249 10101011111
: 11101011 z 111010101 186 11011010111 250 10101111101
; 110111011 { 1010110111 187 11011011101 251 10101111101
< 111101101 | 11011011 188 11011011101 252 10101111111
= 1010101 } 1010110101 189 11011011111 253 10101010101

132
<table>
<thead>
<tr>
<th>Operation</th>
<th>Value 1</th>
<th>Value 2</th>
<th>Value 3</th>
<th>Value 4</th>
<th>Value 5</th>
<th>Value 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>&gt;</td>
<td>11010111</td>
<td>~</td>
<td>1011010111</td>
<td>190</td>
<td>11011101011</td>
<td>254</td>
</tr>
<tr>
<td>?</td>
<td>1010101111</td>
<td>DEL</td>
<td>1110110101</td>
<td>191</td>
<td>11011101101</td>
<td>255</td>
</tr>
</tbody>
</table>
### 8.4 Component run-time parameters

<table>
<thead>
<tr>
<th>Component</th>
<th>Parameter</th>
<th>Operation</th>
<th>Type</th>
<th>Legal values</th>
</tr>
</thead>
<tbody>
<tr>
<td>AudioSink</td>
<td>volume</td>
<td>get/set</td>
<td>double</td>
<td>[0, 1.0]</td>
</tr>
<tr>
<td></td>
<td>fSample</td>
<td>get</td>
<td>double</td>
<td>n/a</td>
</tr>
<tr>
<td>BPSK31TXChannel</td>
<td>fCenter</td>
<td>get/set</td>
<td>double</td>
<td>(0, 4000]</td>
</tr>
<tr>
<td></td>
<td>nrSeparationBits</td>
<td>get/set</td>
<td>integer</td>
<td>[2, 100]</td>
</tr>
<tr>
<td></td>
<td>nrLeadInBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td></td>
<td>nrLeadOutBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td>QPSK31TXChannel</td>
<td>fCenter</td>
<td>get/set</td>
<td>double</td>
<td>(0, 4000]</td>
</tr>
<tr>
<td></td>
<td>nrSeparationBits</td>
<td>get/set</td>
<td>integer</td>
<td>[2, 100]</td>
</tr>
<tr>
<td></td>
<td>nrLeadInBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td></td>
<td>nrLeadOutBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td>FSKModulator</td>
<td>fCenter</td>
<td>get/set</td>
<td>double</td>
<td>(0, 4000]</td>
</tr>
<tr>
<td></td>
<td>fShift</td>
<td>get/set</td>
<td>double</td>
<td>(0, 2000]</td>
</tr>
<tr>
<td></td>
<td>markHigher</td>
<td>get/set</td>
<td>boolean</td>
<td>true/false</td>
</tr>
<tr>
<td>Multiplier</td>
<td>scaler</td>
<td>get/set</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>Adder</td>
<td>scaler</td>
<td>get/set</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td>PSK31CharEncoder</td>
<td>nrSeparationBits</td>
<td>get/set</td>
<td>integer</td>
<td>[2, 100]</td>
</tr>
<tr>
<td></td>
<td>nrLeadInBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td></td>
<td>nrLeadOutBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td>PSK31TXStateMachine</td>
<td>nrSeparationBits</td>
<td>get/set</td>
<td>integer</td>
<td>[2, 100]</td>
</tr>
<tr>
<td></td>
<td>nrLeadInBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td></td>
<td>nrLeadOutBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td>RTTYCharEncoder</td>
<td>nrStartDiddle</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 100]</td>
</tr>
<tr>
<td></td>
<td>nrStopDiddle</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 100]</td>
</tr>
<tr>
<td>RTTYTxChannel</td>
<td>fCenter</td>
<td>get/set</td>
<td>double</td>
<td>(0, 4000]</td>
</tr>
<tr>
<td></td>
<td>fShift</td>
<td>get/set</td>
<td>double</td>
<td>(0, 2000]</td>
</tr>
<tr>
<td></td>
<td>baudRate</td>
<td>get/set</td>
<td>double</td>
<td>[10, 150]</td>
</tr>
<tr>
<td></td>
<td>stopBits</td>
<td>get/set</td>
<td>double</td>
<td>[1, 100]</td>
</tr>
<tr>
<td></td>
<td>markHigher</td>
<td>get/set</td>
<td>boolean</td>
<td>true/false</td>
</tr>
<tr>
<td></td>
<td>nrStartDiddle</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 100]</td>
</tr>
<tr>
<td></td>
<td>nrStopDiddle</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 100]</td>
</tr>
<tr>
<td></td>
<td>nrLeadInMarkBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td></td>
<td>nrLeadOutMarkBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td>RTTYTxShiftSM</td>
<td>baudRate</td>
<td>get/set</td>
<td>double</td>
<td>[10, 150]</td>
</tr>
<tr>
<td></td>
<td>stopBits</td>
<td>get/set</td>
<td>double</td>
<td>[1, 100]</td>
</tr>
<tr>
<td></td>
<td>nrLeadInMarkBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td></td>
<td>nrLeadOutMarkBits</td>
<td>get/set</td>
<td>integer</td>
<td>[0, 500]</td>
</tr>
<tr>
<td>SineCosineGenerator</td>
<td>fSample</td>
<td>get/set</td>
<td>double</td>
<td></td>
</tr>
<tr>
<td></td>
<td>fTone</td>
<td>get/set</td>
<td>double</td>
<td>(0, fSample/2]</td>
</tr>
<tr>
<td></td>
<td>amplitude</td>
<td>get/set</td>
<td>double</td>
<td></td>
</tr>
</tbody>
</table>
8.5 An example of state machine mapping

This appendix contains an example of a complete description of a state machine component. The described component is a downsampler by factor 4, which forwards the value on its input port to the output port every 4th clock event.

8.5.1 An external view of the component.

![External view of the downsampler](image)

Figure 8-6 An external view of the downsampler.

8.5.2 State transition diagram

![State transition diagram](image)

Figure 8-7 A state transition diagram for the downsampler.

8.5.3 External interface construction rule

```xml
<rule name="DwnSampler Rule">
  <body>
    <triple>
      <subject variable="_Ind"/>
      <predicate resource="rdf:type"/>
      <object resource="obr:DwnSampler"/>
    </triple>
  </body>
  <head>
    <bind variable="_Ind.params">
      <makeAsset>_Ind_.params<makeAsset>
    </bind>
    <bind variable="_Ind.params.0">
```

135
8.5.4 State machine component realization rules
<rule name="DwnSampler.SM Rule">
  <body>
    <triple>
      <subject variable="_Ind"/>
      <predicate resource="rdf:type"/>
      <object resource="obra:DwnSampler.SM"/>
    </triple>
  </body>

  <head>
    <bind variable="_Ind.FSM">
      <makeAsset><_Ind/>.FSM</makeAsset>
    </bind>
    <assert>
      <Individual variable="_Ind" rdf:type="owl:NamedIndividual">
        <obra:hasFSMDefinition variable="_Ind.FSM"/>
        <rdf:type resource="owl:NamedIndividual"/>
        <rdf:type resource="obra:DwnSamplerFSM"/>
      </Individual>
    </assert>
  </head>
</rule>

<rule name="DwnSampler Definition">
  <body>
    <triple>
      <subject variable="_Ind"/>
      <predicate resource="rdf:type"/>
      <object resource="obra:DwnSamplerFSM"/>
    </triple>
  </body>

  <head>
    <bind variable="_Ind.InputPort.in_port">
      <makeAsset><_Ind/>.InputPort.in_port</makeAsset>
    </bind>
    <bind variable="_Ind.OutputPort.out_port">
      <makeAsset><_Ind/>.OutputPort.out_port</makeAsset>
    </bind>
    <bind variable="_Ind.Clock.clk">
      <makeAsset><_Ind/>.Clock.clk</makeAsset>
    </bind>
    <bind variable="_Ind.Event.CLK">
      <makeAsset><_Ind/>.Event.CLOCK</makeAsset>
    </bind>
    <bind variable="_Ind.Event.START">
      <makeAsset><_Ind/>.Event.START</makeAsset>
    </bind>
    <bind variable="_Ind.Event.STOP">
      <makeAsset><_Ind/>.Event.STOP</makeAsset>
    </bind>
    <bind variable="_Ind.InSignal.START">
      <makeAsset><_Ind/>.InSignal.START</makeAsset>
    </bind>
    <bind variable="_Ind.InSignal.STOP">
      <makeAsset><_Ind/>.InSignal.STOP</makeAsset>
    </bind>
    <bind variable="_Ind.Property.count">
      <makeAsset><_Ind/>.Property.count</makeAsset>
    </bind>
    <bind variable="_Ind.IDLE">
      <makeAsset><_Ind/>.IDLE</makeAsset>
    </bind>
    <bind variable="_Ind.IDLE.Transition.0">
      <makeAsset><_Ind/>.IDLE.Transition.0</makeAsset>
    </bind>
    <bind variable="_Ind.STARTED">
      <makeAsset><_Ind/>.STARTED</makeAsset>
    </bind>
    <bind variable="_Ind.STARTED.Transition.0">
      <makeAsset><_Ind/>.STARTED.Transition.0</makeAsset>
    </bind>
    <bind variable="_Ind.STARTED.Transition.1">
      <makeAsset><_Ind/>.STARTED.Transition.1</makeAsset>
    </bind>
    <bind variable="_Ind.SEND">
      <makeAsset><_Ind/>.SEND</makeAsset>
    </bind>
  </head>
</rule>
8.5.5 Java code generated for the state machine

```java
package autogen;
import interfaces.*;
import base.*;
import java.util.*;
import java.util.logging.*;
import java.util.concurrent.locks.ReentrantLock;
class DwnSampler
    extends SynchronousComponent
    implements IAsynchronous {
    // constant definitions
    private InputPortDouble in_port_port = new InputPortDouble("sample_in");
    private double in_port;
    private OutputPortDouble out_port_port = new OutputPortDouble("sample_out");
    private int count;

    private final int d_idx_COUNT_2 = 0;
    private final int d_idx_STARTED = 1;
    private final int d_idx_IDLE = 2;
    private final int d_idx_SEND = 3;
    private final State d_state_COUNT_2 = new State_COUNT_2(d_idx_COUNT_2,"COUNT_2");
    private final State d_state_STARTED = new State_STARTED(d_idx_STARTED,"STARTED");
    private final State d_state_IDLE = new State_IDLE(d_idx_IDLE,"IDLE");
    private final State d_state_SEND = new State_SEND(d_idx_SEND,"SEND");

    private State d_currentState = null;
    private ArrayList<State> d_stateArray = new ArrayList<State>();
    private HashMap<String, IFunction> d_eventHashMap = new HashMap<String, IFunction>();
    private HashMap<String, IFunction> d_inputEventHashMap = new HashMap<String, IFunction>();
    private HashMap<String, IFunction> d_clkEventHashMap = new HashMap<String, IFunction>();
    private final ReentrantLock d_lock = new ReentrantLock();

    @SuppressWarnings("LeakingThisInConstructor")
    public DwnSampler( String tag ) {
        super(tag);
    }

    synchronized public void processEvent(String eventName) {
        // handle event
    }
}
```


// initialize state array
  d_stateArray.add(d_state_COUNT_2);
  d_stateArray.add(d_state_STARTED);
  d_stateArray.add(d_state_IDLE);
  d_stateArray.add(d_state_SEND);

  // register ports
  this.addInputPort(in_port_port);
  this.addOutputPort(out_port_port);

  // register clock subscriptions
  addClockSubscription("clk");

  // add code to subscribe to async ports

  // subscribe to signal events
  d_subscribesEvents.add("STOP");
  d_subscribesEvents.add("START");

  // set event handler hash map
  d_eventHashMap.put("STOP", new Event_Handler_STOP());
  d_eventHashMap.put("START", new Event_Handler_START());

  // set clock event handler hash map
  d_clkEventHashMap.put("clk", new Sync_Event_Handler_CLK());

  // module specific signals
  initialize();
}

@Override
public void reset() {
  initialize();
}

protected void receiveEvent_impl(IncomingSignal inSig, Object arg) {
  try {
    d_lock.lock();
    String tag = inSig.getTag();
    d_eventHashMap.get(tag).execute(arg);
  }
  finally {
    d_lock.unlock();
  }
}

public void update(Observable o, Object arg) {
  try {
    d_lock.lock();
    if (o instanceof InputPort) {
      InputPort inp = (InputPort)o;
      d_inputEventHashMap.get(inp.getTag()).execute(arg);
    }
  }
  finally {
    d_lock.unlock();
  }
}

@Override
protected void synchFetchData_impl(IClockSource clk) {
  try {
    d_lock.lock();
    in_port = in_port_port.getValue();
  }
  finally {
    d_lock.unlock();
  }
}

@Override
protected void synchExecute_impl(IClockSource clk) {
  try {
    d_lock.lock();
    d_clkEventHashMap.get(clk.getTag()).execute(null);
  }
  finally {
    d_lock.unlock();
  }
}
private void initialize() {
    in_port = (double)0.0;
    count = (int)0;
    out_port_port.setValue(0.0);
    d_currentState = d_state_IDLE;
    d_currentState.onEnter();
}

private class Event_Handler_STOP implements IFunction {
    public void execute(Object arg) {
        d_currentState.notify_Event_STOP();
    }
}

private class Event_Handler_START implements IFunction {
    public void execute(Object arg) {
        d_currentState.notify_Event_START();
    }
}

private class Sync_Event_Handler_CLK implements IFunction {
    public void execute(Object arg) {
        d_currentState.notify_sync_CLK();
    }
}

private class Transition {
    public Transition( int srcIdx, int tgtIdx ) {
        d_sourceIdx = srcIdx;
        d_targetIdx = tgtIdx;
    }

    public void execute() {
        State src = d_stateArray.get(d_sourceIdx);
        State tgt = d_stateArray.get(d_targetIdx);
        src.onExit();
        this.onTransition();
        d_currentState = tgt;
        d_currentState.onEnter();
    }

    protected void onTransition() {
        // empty by default
    }

    protected int d_sourceIdx;
    protected int d_targetIdx;
}

private abstract class State {
    public State( int idx, String name ) {
        d_idx = idx;
        d_name = name;
    }

    public String getName() {
        return d_name;
    }

    public int getIndex() {
        return d_idx;
    }

    abstract public void onEnter();
    abstract public void onExit();

    abstract public void notify_sync_CLK();
    abstract public void notify_Event_STOP();
    abstract public void notify_Event_START();

    private String d_name;
    private int d_idx;
}

private class State_IDLE extends State {
    public State_IDLE( int idx, String name) {
        super(idx,name);
    }

}
@Override
public void onEnter() {
}

@Override
public void onExit() {
}

@Override
public void notify_sync_CLK() {
}

@Override
public void notify_Event_STOP() {
}

@Override
public void notify_Event_START() {
    d_trans_STARTED0.execute();
    return;
}

private Transition d_trans_STARTED0 = new Transition( this.getIndex(), d_idx_STARTED );

private class State_STARTED extends State {
    public State_STARTED( int idx, String name) {
        super(idx,name);
    }

    @Override
    public void onEnter() {
    }

    @Override
    public void onExit() {
    }

    @Override
    public void notify_sync_CLK() {
        d_trans_SEND0.execute();
        return;
    }

    @Override
    public void notify_Event_STOP() {
        d_trans_IDLE1.execute();
        return;
    }

    @Override
    public void notify_Event_START() {
    }

    private Transition d_trans_SEND0 = new Transition( this.getIndex(), d_idx_SEND );
    private Transition d_trans_IDLE1 = new Transition( this.getIndex(), d_idx_IDLE );
}

private class State_SEND extends State {
    public State_SEND( int idx, String name) {
        super(idx,name);
    }

    @Override
    public void onEnter() {
        out_port_port.setValue(in_port);
    }

    @Override
    public void onExit() {
        count = (int)(0);
    }

    @Override
    public void notify_sync_CLK() {
        d_trans_COUNT_20.execute();
        return;
    }

}
private class State_COUNT_2 extends State {
    public State_COUNT_2( int idx, String name) {
        super(idx,name);
    }
    @Override
    public void onEnter() {
    }
    @Override
    public void onExit() {
    }
    @Override
    public void notify_sync_CLK() {
        if( (count) < (2) ) {
            count += (1);
            return;
        }
        if( (count) >= (2) ) {
            d_trans_SEND2.execute();
            return;
        }
    }
    @Override
    public void notify_Event_STOP() {
        d_trans_IDLE1.execute();
        return;
    }
    @Override
    public void notify_Event_START() {
    }
}

private Transition d_trans_COUNT_20 = new Transition( this.getIndex(), d_idx_COUNT_2 );
private Transition d_trans_IDLE1 = new Transition( this.getIndex(), d_idx_IDLE );

8.6 An example of component composition

This appendix contains a complete description of PSK31CharEncoder component, including its composite component realization.

8.6.1 Block diagram

![Block diagram of PSK31CharEncoder]

Figure 8-8 PSK31CharEncoder block diagram.

8.6.2 External interface construction rule

```xml
<Rule name="PSK31CharEncoder rule">
  <body>
    <subject variable="_Ind"/>
    <predicate resource="rdfs:subClassOf"/>
    <object resource="obr:PSK31CharEncoder"/>
  </body>
</Rule>
```

```xml
<bind variable="_Ind.params">
  <makeAsset>_Ind/.params</makeAsset>
</bind>
<bind variable="_Ind.params.0">
  <makeAsset>_Ind/.params.0</makeAsset>
</bind>
<bind variable="_Ind.Clk.clk">
  <makeAsset>_Ind/.Clk.clk</makeAsset>
</bind>
<bind variable="_Ind.in.char_in">
  <makeAsset>_Ind/.in.char_in</makeAsset>
</bind>
```
8.6.3 Composite component realization definition

```xml
<rule name="PSK31CharEncoder.Composition rule">
  <body>
    <triple>
      <subject variable="_Ind"/>
      <predicate resource="rdf:type"/>
      <object resource="obra:PSK31CharEncoder.Composition"/>
    </triple>
  </body>
</rule>
```

<!-- txSM clock rule -->
<rule name="txSM clock rule">
<body>
<Individual variable="_Ind">
<rdf:type resource="obra:PSK31CharEncoder.Composition"/>
<obra:hasSubComponent anonVar="true" variable="_subCompClk">
<obra:hasName datatype="xsd:string">txSM</obra:hasName>
<obra:hasExtClock variable="_subCompClk">
<obra:hasName datatype="xsd:string">clk</obra:hasName>
</obra:hasExtClock>
</obra:hasSubComponent>
<obra:hasExtClock variable="_compClk">
<obra:hasName datatype="xsd:string">clk</obra:hasName>
</obra:hasExtClock>
</Individual>
</body>
</head>
<assert>
<Individual variable="_compClk">
<obra:drivesClock variable="_subCompClk"/>
</Individual>
</assert>
</rule>

<!-- shiftReg clock rule -->
<rule name="shiftReg clock rule">
<body>
<Individual variable="_Ind">
<rdf:type resource="obra:PSK31CharEncoder.Composition"/>
<obra:hasSubComponent anonVar="true" variable="_inputPort">
<obra:hasName datatype="xsd:string">empty</obra:hasName>
</obra:hasSubComponent>
<obra:hasOutputPort variable="_outputPort">
<obra:hasName datatype="xsd:string">empty</obra:hasName>
</obra:hasOutputPort>
</Individual>
</body>
</head>
<assert>
<Individual variable="_outPort">
<obra:drivesPort variable="_inPort"/>
</Individual>
</assert>
</rule>
<obr:hasSubComponent anonVar="true">
  <obr:hasName datatype="xsd:string">shiftReg</obr:hasName>
  <obr:hasInputPort variable="_inPort">
    <obr:hasName datatype="xsd:string">code_in</obr:hasName>
  </obr:hasInputPort>
</obr:hasSubComponent>

<obr:hasSubComponent anonVar="true">
  <obr:hasName datatype="xsd:string">varicode</obr:hasName>
  <obr:hasOutputPort variable="_outPort">
    <obr:hasName datatype="xsd:string">code_out</obr:hasName>
  </obr:hasOutputPort>
</obr:hasSubComponent>

<head><assert>
  <Individual variable="_outPort">
    <obr:drivesPort variable="_inPort"/>
  </Individual>
</assert></head>

<rule name="shiftReg port code_len rule">
  <body>
    <Individual variables="_Ind">
      <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
      <obr:hasSubComponent anonVar="true">
        <obr:hasName datatype="xsd:string">shiftReg</obr:hasName>
        <obr:hasInputPort variable="_inPort">
          <obr:hasName datatype="xsd:string">code_len</obr:hasName>
        </obr:hasInputPort>
      </obr:hasSubComponent>
      <obr:hasSubComponent anonVar="true">
        <obr:hasName datatype="xsd:string">varicode</obr:hasName>
        <obr:hasOutputPort variable="_outPort">
          <obr:hasName datatype="xsd:string">code_len</obr:hasName>
        </obr:hasOutputPort>
      </obr:hasSubComponent>
    </Individual>
  </body>
</rule>

<rule name="varicode clock rule">
  <body>
    <Individual variable="_Ind">
      <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
      <obr:hasSubComponent anonVar="true">
        <obr:hasName datatype="xsd:string">varicode</obr:hasName>
        <obr:hasExtClock variable="_subCompClk">
          <obr:hasName datatype="xsd:string">clk</obr:hasName>
        </obr:hasExtClock>
      </obr:hasSubComponent>
      <obr:hasExtClock variable="_compClk">
        <obr:hasName datatype="xsd:string">clk</obr:hasName>
      </obr:hasExtClock>
    </Individual>
  </body>
</rule>

<rule name="varicode port code_get rule">
  <body>
    <Individual variables="_Ind">
      <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
      <obr:hasSubComponent anonVar="true">
        <obr:hasName datatype="xsd:string">varicode</obr:hasName>
      </obr:hasSubComponent>
    </Individual>
  </body>
</rule>
<Individual variable="_inPort">
  <hasInputPort variable="_inPort">
    <hasName datatype="xsd:string">code_get</hasName>
  </hasInputPort>
</Individual>

<hasSubComponent anonVar="true">
  <SubComponent>
    <hasName datatype="xsd:string">shiftReg</hasName>
    <hasOutputPort variable="_outPort">
      <hasName datatype="xsd:string">code_next</hasName>
    </hasOutputPort>
  </SubComponent>
</hasSubComponent>

<Individual>
  <assert>
    <Individual variable="_outPort">
      <drivesPort variable="_inPort"/>
    </Individual>
  </assert>
</Individual>

<!-- External signal rules -->

<rule name="External signal TxStart rule">
  <body>
    <Individual variable="_Ind">
      <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
      <hasIncomingSignal variable="_inSig">
        <hasName datatype="xsd:string">TxStart</hasName>
      </hasIncomingSignal>
      <hasSubComponent anonVar="true">
        <hasName datatype="xsd:string">txSM</hasName>
        <hasIncomingSignal variable="_subCompSig">
          <hasName datatype="xsd:string">TxStart</hasName>
        </hasIncomingSignal>
      </hasSubComponent>
    </Individual>
  </body>
  <head>
    <assert>
      <Individual variable="_inSig">
        <drivesSignal variable="_subCompSig"/>
      </Individual>
    </assert>
  </head>
</rule>

<rule name="External signal TxStop rule">
  <body>
    <Individual variable="_Ind">
      <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
      <hasIncomingSignal variable="_inSig">
        <hasName datatype="xsd:string">TxStop</hasName>
      </hasIncomingSignal>
      <hasSubComponent anonVar="true">
        <hasName datatype="xsd:string">txSM</hasName>
        <hasIncomingSignal variable="_subCompSig">
          <hasName datatype="xsd:string">TxStop</hasName>
        </hasIncomingSignal>
      </hasSubComponent>
    </Individual>
  </body>
  <head>
    <assert>
      <Individual variable="_inSig">
        <drivesSignal variable="_subCompSig"/>
      </Individual>
    </assert>
  </head>
</rule>

<rule name="External signal ClkEnable rule">
  <body>
    <Individual variable="_Ind">
      <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
      <hasOutgoingSignal variable="_outSig">
        <hasName datatype="xsd:string">ClkEnable</hasName>
      </hasOutgoingSignal>
      <hasSubComponent anonVar="true">
        <hasName datatype="xsd:string">txSM</hasName>
        <hasOutgoingSignal variable="_subCompSig">
          <hasName datatype="xsd:string">ClkEnable</hasName>
        </hasOutgoingSignal>
      </hasSubComponent>
    </Individual>
  </body>
  <head>
    <assert>
      <Individual variable="_inSig">
        <drivesSignal variable="_subCompSig"/>
      </Individual>
    </assert>
  </head>
</rule>
<rule name="External signal ClkDisable rule">
<Individual variable="_Ind">
  <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
  <obr:hasOutgoingSignal variable="_outSig">
    <obr:hasName datatype="xsd:string">ClkDisable</obr:hasName>
  </obr:hasOutgoingSignal>
  <obr:hasSubComponent anonVar="true">
    <obr:hasName datatype="xsd:string">txSM</obr:hasName>
    <obr:hasOutgoingSignal variable="_subCompSig">
      <obr:hasName datatype="xsd:string">ClkDisable</obr:hasName>
    </obr:hasOutgoingSignal>
  </obr:hasSubComponent>
</Individual>
</head>
</assert>
</Individual>
</rule>

<rule name="External InputPort(char_in) rule">
<Individual variable="_Ind">
  <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
  <obr:hasInputPort variable="_inPort">
    <obr:hasName datatype="xsd:string">char_in</obr:hasName>
  </obr:hasInputPort>
  <obr:hasSubComponent anonVar="true">
    <obr:hasName datatype="xsd:string">varicode</obr:hasName>
    <obr:hasInputPort variable="_subCompPort">
      <obr:hasName datatype="xsd:string">char_in</obr:hasName>
    </obr:hasInputPort>
  </obr:hasSubComponent>
</Individual>
</head>
</assert>
</Individual>
</rule>

<rule name="External InputPort(char_rdy) rule">
<Individual variable="_Ind">
  <rdf:type resource="obra:PSK31CharEncoder.Composition"/>
  <obr:hasInputPort variable="_inPort">
    <obr:hasName datatype="xsd:string">char_rdy</obr:hasName>
  </obr:hasInputPort>
  <obr:hasSubComponent anonVar="true">
    <obr:hasName datatype="xsd:string">varicode</obr:hasName>
    <obr:hasInputPort variable="_subCompPort">
      <obr:hasName datatype="xsd:string">char_put</obr:hasName>
    </obr:hasInputPort>
  </obr:hasSubComponent>
</Individual>
</head>
</assert>
</Individual>
</rule>
<rule name="External OutputPort(next_char) rule">
<body>
<Individual variable="_Ind">
<rdf:type resource="obra:PSK31CharEncoder.Composition"/>
<obr:hasOutputPort variable="_outPort">
<obr:hasName datatype="xsd:string">next_char</obr:hasName>
</obr:hasOutputPort>
<obr:hasSubComponent anonVar="true">
<obr:hasName datatype="xsd:string">varicode</obr:hasName>
<obr:hasOutputPort variable="_subCompPort">
<obr:hasName datatype="xsd:string">char_next</obr:hasName>
</obr:hasOutputPort>
</obr:hasSubComponent>
</Individual>
</body>
<head>
<assert>
<Individual variable="_subCompPort">
<obr:drivesPort variable="_outPort"/>
</Individual>
</assert>
</head>
</rule>

<rule name="External OutputPort(bit_out) rule">
<body>
<Individual variable="_Ind">
<rdf:type resource="obra:PSK31CharEncoder.Composition"/>
<obr:hasOutputPort variable="_outPort">
<obr:hasName datatype="xsd:string">bit_out</obr:hasName>
</obr:hasOutputPort>
<obr:hasSubComponent anonVar="true">
<obr:hasName datatype="xsd:string">shiftReg</obr:hasName>
<obr:hasOutputPort variable="_subCompPort">
<obr:hasName datatype="xsd:string">bit_out</obr:hasName>
</obr:hasOutputPort>
</obr:hasSubComponent>
</Individual>
</body>
<head>
<assert>
<Individual variable="_subCompPort">
<obr:drivesPort variable="_outPort"/>
</Individual>
</assert>
</head>
</rule>

<rule name="Component parameter(nrSeparationBits) rule">
<body>
<Individual variable="_Ind">
<rdf:type resource="obra:PSK31CharEncoder.Composition"/>
<obr:hasComponentParameter variable="_param">
<obr:hasName datatype="xsd:string">nrSeparationBits</obr:hasName>
</obr:hasComponentParameter>
<obr:hasSubComponent variable="_comp">
<obr:hasName datatype="xsd:string">txSM</obr:hasName>
</obr:hasSubComponent>
</Individual>
</body>
<head>
<assert>
<Individual variable="_param">
<rdf:type resource="obr:PassthruComponentParameter"/>
<obr:targetComponent variable="_comp"/>
<obr:targetParameterName datatype="xsd:string">nrSeparationBits</obr:targetParameterName>
</Individual>
</assert>
</head>
</rule>

<rule name="Component parameter(nrLeadInBits) rule">
9 References


