Implementation of a Space Communications Cognitive Engine

Timothy M. Hackett*, Sven G. Bilén*, Paulo Victor R. Ferreira†, Alexander M. Wyglinski†, and Richard C. Reinhart‡
*School of Electrical Engineering and Computer Science
The Pennsylvania State University, University Park, Pennsylvania 16801
Email: tmh5344@psu.edu, sbilen@psu.edu
†Department of Electrical Engineering & Computer Engineering
Worcester Polytechnic Institute, Worcester, MA 01609, USA
Email: prferreira@wpi.edu, alexw@wpi.edu
‡ NASA John H. Glenn Research Center, Cleveland, Ohio 44135
Email: richard.c.reinhart@nasa.gov

***Abstract*—Although communications-based cognitive engines have been proposed, very few have been implemented in a full system, especially in a space communications system. In this paper, we detail the implementation of a multi-objective reinforcement-learning algorithm and deep artificial neural networks for the use as a radio-resource-allocation controller. The modular software architecture presented encourages re-use and easy modification for trying different algorithms. Various trade studies involved with the system implementation and integration are discussed. These include the choice of software libraries that provide platform flexibility and promote reusability, choices regarding the deployment of this cognitive engine within a system architecture using the DVB-S2 standard and commercial hardware, and constraints placed on the cognitive engine caused by real-world radio constraints. The implemented radio-resource-allocation-management controller was then integrated with the larger space–ground system developed by NASA Glenn Research Center (GRC).**

Keywords*—*cognitive engine; neural networks; reinforcement learning; SCaN Testbed; space communications; machine learning

I. Introduction

With the continual increased computing performance with each new generation of general purpose processors (GPPs) and field programmable gate arrays (FPGAs), the usage of software-defined radios (SDRs) has become increasingly feasible to use in real space systems. As a result, proposed advanced algorithms for communications, such as cognitive algorithms, can now be deployed and tested.

Onboard the International Space Station (ISS), NASA currently has three operational SDRs on the Space Communications and Navigation (SCaN) Testbed. NASA is interested in determining the applicability of SDRs for their future missions by considering their flexibility in changing signaling waveforms, new software development paradigms, and new operational aspects. We plan to use one of the SDRs on-orbit along with ground SDRs and commercial modems as part of a system to test the performance of a newly proposed multi-objective reinforcement learning algorithm using deep neural networks for the use as a radio-resource-allocation controller [1].

Currently, the experimental state-of-the-art for NASA's space communications is adaptive communications, which works based on a lookup-table built by experts *a priori*. The table tells the radio which modulation-coding pair (MODCOD) to use based on the measured E_s/N_0 (energy per symbol to noise power spectral density ratio). This table is built to maintain a quasi-error-free transmission [2]. In essence, this adaptive algorithm is a radio-resource-allocation optimization that takes into account bit error rate (BER) and throughput. Our proposed reinforcement-learning neural network (RLNN) cognitive engine [1] is to be a generalization of the adaptive algorithm's bi-objective optimization to a multi-objective optimization. In our proposed case, the objectives that the cognitive algorithm tries to optimize are bit error rate (BER), throughput, occupied bandwidth, spectral efficiency, transmit power efficiency, and DC power consumption. The different parameters that can be changed in our experiment (thanks to the flexibility of an SDR waveform) are the modulation and coding scheme (MODCOD), filter roll-off factor, symbol rate, and transmit power. Instead of creating a static table that maps E_s/N_0 values with transmission parameters (referred to as "action tuplets" throughout this paper), the cognitive engine *learns* the optimal actions given the channel condition through a reinforcement-learning process [1].

The goal of our experiment is to reuse the same system architecture as [2], but replace their adaptive algorithms workstation with our proposed multi-objective cognitive engine workstation. This significantly reduces development time, decreases project risk, and provides a fair comparison between the adaptive and cognitive algorithms. As a result, our implementation of the multi-objective cognitive engine will also be constrained to fit within the Digital Video Broadcasting–Satellite–Second Generation (DVB-S2) standard [3] as did the adaptive experiment in [2]. The DVB-S2 standard is

978-1-5386-3988-7/17/$31.00

commonly used in the commercial satellite broadcast market because of its granularity of operational modes achieved via several MODCOD pairs that allow scalable data rates when using variable coding modulation (VCM) and adaptive coding modulation (ACM).

This paper elaborates on the implementation details of porting the MATLAB algorithms in [1] to a real system. Section II discusses the cognitive applications to NASA's missions. Section III summarizes the structure of our cognitive engine. Section IV discusses the software libraries used in this implementation and justifications for their use. Section V discusses implementation trade studies and adaptations of the original proposed cognitive engine to meet the constraints imposed by the existing testing architecture. Section VI provides the implemented architecture to be used for ground and on-orbit testing. Finally, Section VII provides future work and conclusions based on the work presented in this paper.

II. Cognitive Applications to NASA's Missions

NASA's future space architecture will enable human, robotic, and science exploration of the solar system. The rise in science data volume requirements, greater inter-connectivity, autonomous navigation, and the use of small satellites (among other requirements) will all add greater complexity to the communications network. Cognitive algorithms offer potential solutions to improve the efficiency and reduce the complexity of the future communications systems by managing and operating the system without human operators. This approach enables more system automation, but must also preserve the high level of reliability that space missions have achieved.

The use of cognition in space communications is relatively new. Research underway at NASA Glenn Research Center is looking at various aspects for cognitive applications [4]. The first domain is between the radios of the science spacecraft and either the ground station (for direct-to-ground connection) or through a relay satellite to the gateway ground station (or to a relay if on-board processed). The radios may optimize the link between the two radio nodes striving to optimize the pass (e.g. limit power consumption) or maximize the data throughput. Cognition could be applied to trade operating parameters (e.g., modulation, coding, frequency, power, and bandwidth) with performance parameters (e.g., bit/frame error rate, spectral efficiency, and power consumption). Effects on the links might include range changes, scintillation and other propagation effects, multipath, and interference. Cognitive algorithms could learn system behavior and change the configuration to optimize the links from different locations and conditions.

The second application of cognitive algorithms might apply to the inter-connectivity and data flow from and among the science spacecraft, relay satellite, or ground station. As data is relayed or communicated through the network, cognition could update routing tables, move data through the network using a store and forward protocol, transferring data custody from node to node and optimally route data from science spacecraft back to mission operations centers through various paths.

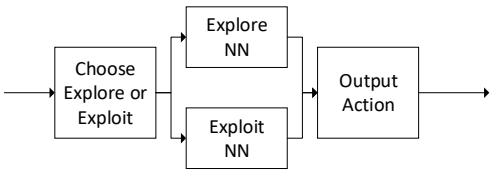

Fig. 1. General flow block diagram of the RLNN (adapted from [1])

.

Finally, higher level applications may benefit from cognition such as scheduling or configuration. For example, automating the scheduling of network assets among various space users based on previous patterns of use could help optimize the utilization of the network. The network might also change the initial link configuration for a scheduled connection based on past performance or identify anomalous asset performance through a "big data" analysis. In addition, functions that the user spacecraft could perform for itself might include orbit propagation to automate its antenna pointing, data storage monitoring to automate its request for services, and performance monitoring to report out-of-range telemetry or report quality-of-service assessments to the network manager.

For each of these areas—radio-to-radio optimization, assured data connectivity or internetworking, and system level applications—cognition may offer potential benefits to reduce complexity or improve operations. In addition, multi-objective cognition will work across each of these domains and globally optimize system behavior and performance and further provide overall benefit to NASA missions.

III. Overview of Cognitive Engine

The RLNN cognitive engine presented in [1] (an expansion on [5]) consists of a reinforcement-learning framework captured through the use of neural networks (NNs) to choose actions for both exploration and exploitation. An overview of the process flow of the RLNN is shown in Fig. 1. The exploration NN ensemble takes in an action tuple and the current E_s/N_0 and predicts the normalized multi-objective weighted-sum performance value. After predicting the performance for every permutation of actions, one can then threshold the actions that resulted in "good" and "bad" performances. Most of the time, the reinforcement learner will choose to explore an action tuple in the "good" performance space. This is referred to as virtual exploration [1], [6].

The set of exploitation NN ensembles takes in the normalized multi-objective performance metric values and the current E_s/N_0 and outputs the action parameter that should achieve the input objective performances. The exploitation NN ensembles hold the knowledge of the Q-table in traditional Q-learning. The exploitation NN ensembles are used to provide the action tuple that will give the best performance given the actions/performances the RLNN has already seen and the current E_s/N_0 value.

The measured performances for each of the objectives in the multi-objective function (bit error rate, throughput, occupied bandwidth, spectral efficiency, transmit power efficiency, and DC power consumption) are recorded. Periodically, the exploration and exploitation NNs are re-trained (using the Levenberg–Marquardt backpropagation algorithm) with the most recent action/performance knowledge.

IV. Software Libraries

The RLNN was written in the C++ language (we are using the C++11 standard [7]) with the intention of easily porting the engine between different machines and platforms (both ground-based radios and space-based radios)). Unlike the authors' original simulations [1] in MATLAB, C++ (being a lower-level language) does not come equipped with high level constructs, such as matrix algebra or NNs. In order to speed up development time, decrease development risk, and promote reuse, software libraries were leveraged for the matrix algebra operations, NN training and execution, and interfacing to external modems.

A. Licensing

There are a multitude of different open source software usage/redistribution licenses that an author can attribute to their work including (but not limited to) Berkeley Software Distribution (BSD) [8], GNU General Public License (GPL) [9], and GNU Lesser General Public License (LGPL) [10]. The GPL license requires that a work that uses a GPL-licensed work be released under GPL if redistributed [9]. As the name suggests, the LGPL license is a little more relaxed: a proprietary work that links to a LGPL-licensed library only needs to release the source code of the linked LGPL-licensed work, but not the full proprietary work [10]. These two types of licenses are known as "copyleft" licenses. Unlike (L)GPL, the BSD license is a permissive license that allows the use and redistribution of a BSD-licensed work without the release of the source code (but still needs to maintain copyright notices). This allows any proprietary application to use a BSD license without having to release its source code [8].

For our application, we chose to limit our software libraries to those that use permissive licenses, such as BSD. As the nature of SDR waveforms on space radios can be restricted by the International Traffic in Arms Regulations (ITAR) and other export control laws, the authors wanted to ensure their software could be added to the STRS Repository [11] without violating any software library licenses when redistributed. Unfortunately, limiting ourselves to only BSD-licensed, C++ software severely limited the scope of the libraries that could be used for this cognitive engine.

B. Neural Network Library

As the field of machine learning continues to rapidly expand, the number of software libraries continues to increase. Many of these libraries have application program interfaces (APIs) that use high level scripting languages (such as Python) to make it easy for the user to get started. Although some of these libraries were based on C/C++ cores (for performance), their C++ APIs were generally undocumented or nonexistent. For example, this was the case for the popular Torch library [12]. Another common issue was that some of the more popular libraries (such as Caffe [13]) that had C++ APIs also had a long list of dependencies and a complex build process. This issue would make it hard to port our cognitive engine to resource-constrained space radios. A third issue encountered was that some libraries (such as Darknet [14]) were designed to be used for very complex, deep convolutional NNs, which would be excessive for our cognitive engine.

MLPack [15] was chosen to be used for the NN library. Although MLPack's artificial neural network (ANN) is not as mature as other software libraries, it provided many advantages. First, the build process was relatively simple and the required dependency list was short. Second, its API is well-documented both with function definitions in Doxygen and code usage examples. Third, other machine learning algorithms in MLPack have been used by our cognitive communciations colleagues at NASA GRC, so using a common library would ease incorporation of our cognitive engine with their activities. For those interested in using MLPack's ANN library, currently, the ANN support is not yet included in any stable release—it is only included in the master GIT branch [15]. Additionally, MLPack does not support the Levenberg–Marquardt backpropagation algorithm for training. The authors wrote their own implementation of the algorithm and verified its performance using MATLAB's "trainlm" function in its Neural Network Toolbox.

C. Matrix Library

To handle the matrix and vector storage and operations, Armadillo [16], was chosen. Armadillo abstracts matrix algebra to an API similar to MATLAB, making it easier to port the algorithms in [1]. This library was chosen primarily because it was also used internally by MLPack making interfacing easier. Another useful feature is that it transparently handles multithreading for larger operations.

D. External Input-Output

The DVB-S2 receiver and the BPSK transmitter ground modems attached to our cognitive engine communicate using UDP. The Boost.Asio [17] library was leveraged to handle all UDP inputs and outputs for our cognitive engine. Boost is a very common and powerful C++ library and is also internally used by MLPack. To save and resume the state of the cognitive engine, the Boost.Serialization [18] library was used. This allows us to compare how the cognitive engine performs when using already trained weights on the next ground station pass versus having to relearn the weights at the beginning of a pass.

V. Trade Studies

Porting theoretical algorithms in MATLAB to a real system brings many challenges. One of the goals of our experiments is to leverage as much existing waveforms and technology as possible. This lowers risk, lowers development time, and

increases chances of near-term adoption. We chose to fit our algorithms within the DVB-S2 standard to reuse the already developed and extensively tested space transmitter waveform implemented by colleagues at NASA GRC [2]. Additionally, we chose to use the same test setup as used by [2], which uses commercial DVB-S2 receivers on the ground. Unlike in simulation, modems have physical limitations, such as how fast the automatic gain control (AGC) can handle transmit power changes and the rate at which frame error rate (FER) statistics are updated. The following sections discuss how our implementation accomodates these challenges.

A. Fitting within DVB-S2 Standard

In [1], a transmit action tuple consists of five parameters: modulation scheme, code rate, filter roll-off factor, transmit power, and symbol rate. Each time the cognitive engine receives a new frame, it needs to record both the performance of the frame (in terms of the multi-objective fitness values) and the action tuple that was used on that frame. Although the DVB-S2 protocol includes the MODCOD and filter roll-off factor within the physical layer and baseband headers, respectively, the symbol rate and transmit power are not included in any of the headers. To generalize this cognitive engine to any set of action parameters, the action tuple had to be specified outside of the DVB-S2 physical-layer headers.

The round trip time (RTT) for the adaptive algorithm in [2] at 1M baud was measured to be approximately 40 ms. This time includes both constant and variable delays. Constant delays include the transmission time of an AOS frame on the ML605 transmitter, the propagation delay from the ground station to the ISS (assuming the changing range is negligible), the frame decoding time on the space-based receiver, and the propagation delay from the ISS to the ground. Variable delays include the processing time on the ground (assuming the algorithm runs on a CPU), the time it takes between when the new action is decoded on the space radio to when the next frame can be updated with the new parameters, and the amount of time it takes for a DVB-S2 frame to be transmitted to the ground (higher symbol rates takes less time because the number of bits in a DVB-S2 frame is fixed). These delays make it more difficult to know on the ground which action tuple was used on each frame if we want to update the action tuple on a frame-by-frame basis.

In addition to the variable timing delays, the dynamic nature of the wireless channel causes another challenge. There is a (relatively high) chance that corrupted frames will be received on the ground when the cognitive engine makes "bad" action decisions before it is fully trained. For a simple solution in which a header is placed within the DVB-S2 payload with the action tuple used, the cognitive engine would never know the action tuples that caused poor performance. The uplink feedback channel could also get corrupted, which would cause the action tuple to not change on the next frame. This would cause a solution that relied solely on timing to record a received performance with the new action tuple, when in fact the action tuple from the previous frame was used again.

In our case, the scheme is very simple and does not require an additional protocol header. Instead of updating on a frame-by-frame basis, our cognitive engine updates at a period equal to or greater than the longest round trip time—in the 1M baud case, this would be a 40-ms period. In that 40-ms period, we will receive multiple frames with that action tuple, but our RLNN only uses the latest sample with that action tuple. We can solely base our protocol on timing without the need for any headers—the ground system sends the new action tuple and then waits for the worst case RTT. It then records the performance of a frame after the RTT timer elapses and chooses a new action. This makes the assumption that the uplink channel will not corrupt the feedback AOS frame. This is a valid assumption because, in our case, the feedback link is a more robust link than the most robust DVB-S2 downlink parameters. As a result, if the uplink gets corrupted and the action tuple never gets changed, then it does not matter which action is recorded on the ground because every action is going to result in poor performance.

For a cognitive engine that uses the knowledge of every frame received (regardless of whether or not it is the same action as the previous frame) a more complex timing architecture can be used. By leveraging the knowledge of the constant timing delays in the system, one can split the RTT time into periods of "certainty" and "uncertainty". In this architecture, a header is included inside the DVB-S2 payload that contains the action tuple. Essentially, if a DVB-S2 frame is received during the period of "certainty", the receiver already knows which action tuple was used for that frame (it does not even need to read the header inside of the payload). When a frame is received during the period of "uncertainty", it is ambiguous to the receiver which action tuple was used, so it needs to read the header inside of the frame. If the downlink is corrupted, then any frame that is received during the period of "certainty" can still be recorded (because the receiver is certain which action was taken without reading the frame). Any corrupted frame received during the period of "uncertainty" is just discarded and not recorded. It is guaranteed that, in every RTT, there is at least one frame in the window of "certainty". Fig. 2 shows a summary of how this timing architecture works.

Finally, the DVB-S2 standard is a constant-symbol-rate protocol. It does not handle variable symbol rates as the simulation in [1], so, in our testing, we will not be varying the symbol rate. The testing will only change the MODCOD, transmit power, and filter roll-off factor "on-the-fly".

B. Hardware Limitations

An implicit assumption used in the simulation in [1] is that the transmit power can be changed on a frame-by-frame basis. It is true that the space transmitter can change the transmit power on a frame-by-frame basis, but this would result in poor performance at the receiver because the AGC would not be able to keep up with the fast fluctuating power. As a result, a patch will be used for the implemented RLNN that limits the magnitude of the change in transmit power from one action tuple to another.

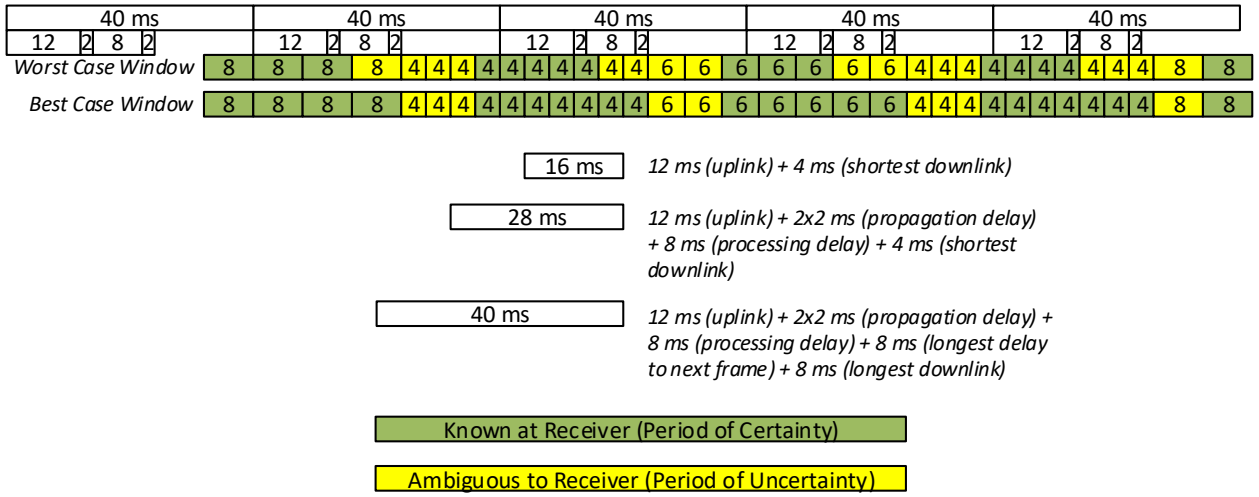

Fig. 2. Timing architecture showing the period of "certainty" and period of "uncertainty" for a RTT of 40 ms (1M baud symbol rate). The rectangles with "4", "6", and "8" represent the received 16-APSK, 8-PSK, and QPSK frames, respectively. These values represent example frame transmission durations (in ms) with these modulation orders.

Another hardware limitation is that the modem's calculated frame error rate is updated at 1 Hz. Although this is a sufficient update rate for typical applications, a 1-Hz rate is relatively slow for this experiment. Instead of directly polling the FER, we chose to use the E_s/N_0 measurements reported by the modem (at 100 Hz) and estimate the FER by using the FER-E_s/N_0 curves (measured *a priori*) for the received action tuple's MODCOD.

VI. Implemented Architecture

The implemented RLNN cognitive engine was written in object-oriented C++ using the libraries discussed in Section IV and trade study results in Section V. The RLNN is made up of an RLNN core, which is where the cognitive algorithms reside, and external drivers, which communicate with the ground modems. The object architecture for this implementation is shown in Fig. 3. The following sections provide details on each of these objects.

A. RLNN Core

The RLNN core is comprised of three modules: a training buffer, the NN predictors, and an application-specific object. The RLNN core provides the "glue" code between these modules. The RLNN core is designed to be a standalone object regardless of the physical interfaces. It has two main purposes: choose an action tuple based on exploration and exploitation and record the action tuple's corresponding performance.

1) Training Buffer: The training buffer is the main database holding the training samples for the explore and exploit NNs. The buffer holds the latest N unique actions and their corresponding performances. When training needs to occur, the user provides the training buffer with the fields needed for the input and output labels and any normalization parameters for both the explore and exploit NNs. The buffer then outputs the formatted input and output training samples to be directly used with the NN Predictor modules. The training buffer is constructed using Armadillo vector and matrix structures.

2) NN Predictor: The NN Predictor module abstracts ML-Pack's ANN architecture to a simple interface similar to MATLAB's Neural Network Toolbox. Each NN predictor module instantiates an ensemble of parallel NNs to be used for either exploration or exploitation prediction. When the "predict" class method is called, each of the parallel NNs compute a prediction and then the mean prediction is outputted back to the user.

We use two types of NN Predictor modules: a trainer module and an execution module. The trainer module is used whenever the NNs need to be retrained. Upon retraining, the weights are copied to the execution module, which is used for online predictions. This decoupling allows the training to occur simultaneously in a separate thread while the RLNN continues normal execution. This is important because NN training can take on the order of many seconds to complete, so we cannot block the main execution until the training is complete. For the exploration prediction, there is a single training module and a single execution module. For the exploitation prediction, we use a vector of trainer-module and execution-module pairs. Each trainer-module and execution-module pair corresponds to the prediction of one of the action parameters in an action tuple. It is important to note that, within each of these trainer/execution modules, there are multiple NNs running in parallel. The number of NNs that need to be run in parallel is a trade-off between the accuracy of the predicted action, the number of threads that can be executed in parallel, and training time, while maintaining real-time execution.

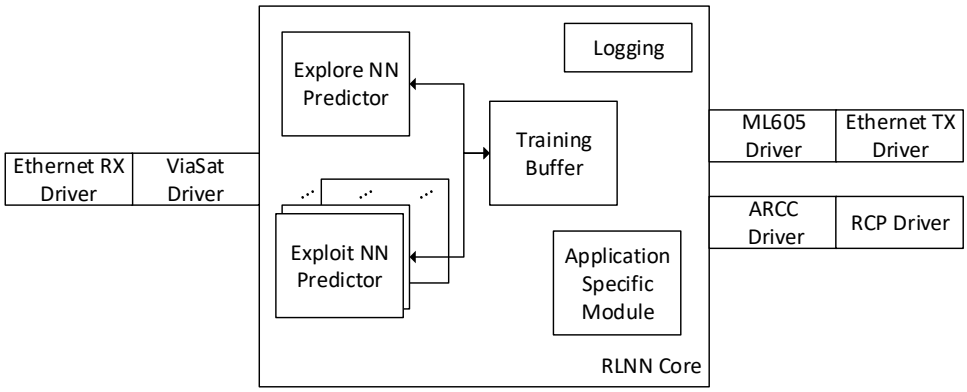

Fig. 3. Implemented software architecture of the RLNN.

3) Application Specific Object: The RLNN core "glue" code, training buffer, and NN predictor modules were designed to be completely generic, so that the RLNN could be reused easily. The application-specific object provides the "context" for our particular experiment. This object processes the raw measurements from the modems, calculates the normalized multi-objective performances, and helps the RLNN algorithms maintain generality. For those interested in reusing our cognitive engine for a different application (or different multi-objective performance metrics), only the Application Specific Object needs to swapped with a new object containing the new parameters and functions. For our experiment, this object is also where any hardware constraint patches are applied—estimating the FER from the FER-E_s/N_0 curves and limiting the magnitude between jumps of transmit power.

B. External Drivers

The external drivers communicate with the external modems. In the on-orbit experiments, the ViaSat modem is used for gathering E_s/N_0 statistics and the ML605 transmitter is used to transmit the new action tuplets to the space-based DVB-S2 transmitter [2]. For ground testing, GRC has ported the DVB-S2 waveform to their Advanced Radio for Cognitive Communications (ARCC) instead of having to use the JPL radio breadboard or engineering model.

1) ViaSat Modem Driver: The ViaSat modem sends its E_s/N_0/RSSI statistics over a UDP/IP/Ethernet stack. This driver parses incoming E_s/N_0/RSSI packets from the Boost.ASIO UDP receiver and returns them for usage within the RLNN core.

2) ML605 Transmitter Driver: The ML605 transmitter driver packetizes the chosen action tuple into a UDP/IP/Ethernet frame it requires for embedding into the uplink frame. We use Boost.ASIO's library to send raw Ethernet frames to the ML605 because it does not support the full IP stack.

3) ARCC Modem Driver: The ARCC modem driver converts the chosen action tuple into the format required for sending remote procedure calls (RPC) to the ARCC web server. We use the UNIX "curl" [19] command to build and send these RPC messages.

C. Logging

Logging of important information on each iteration (action tuple chosen, performance measured, exploitation/exploration, timestamps, etc.) are written into a global text file using a name/value pair. Logging is controlled using preprocessor directives. The log file is easily parsed by name using a script in MATLAB.

The Boost.Serialization library is used to archive and load the state of the objects in the RLNN. The user has the option to resume a session or start a new session and whether or not to save that session upon exit. For each object, the Boost.Serialization library writes its properties to a human-readable text file. When the RLNN is instantiated, the properties are initialized with the values saved in the log files.

VII. Conclusion

In this paper, the implementation of the algorithms described in [1] has been presented. The software libraries utilized, the modifications for real-world constraints, and the object-oriented software architecture were discussed. At the time of writing, this implemented system is currently in the ground-testing stage and will be used for on-orbit experiments in May 2017. The goal of this paper was to provide the reader with insight into building cognitive engines and integrating them into a real system. Upon review, it is planned that the RLNN core code will be released to the public. Another paper is planned to discuss the results and evaluation of this implementation. Additional future work includes adding a real-time received power predictor into the RLNN based on channel models [20], [21].

Acknowledgment

This work was supported by a NASA Space Technology Research Fellowship (grant number NNX15AQ41H) and a cooperative agreement with NASA John H. Glenn Research

Center (grant number NNC14AA01A). The authors would also like to thank Joseph Downey, Michael Evans, Dale Mortensen, and the rest of the Cognitive Communications Project team at NASA GRC for all of their help and support.

REFERENCES

[1] P. V. R. Ferreira, R. Paffenroth, A. M. Wyglinski, T. M. Hackett, S. G. Bilén, R. C. Reinhart, and D. J. Mortensen, "Multi-objective reinforcement learning for cognitive satellite communications using deep neural networks ensembles," *IEEE Journal on Selected Areas in Communications*, 2017, submitted.

[2] J. A. Downey, D. J. Mortensen, M. A. Evans, J. C. Briones, and N. Tollis, "Adaptive coding and modulation experiment with NASA's Space Communication and Navigation Testbed," in *34th AIAA International Communications Satellite Systems Conference*, October 2016.

[3] ETSI, *Digital Video Broadcasting-Satellite-Second Generation (DVB-S2) Standard*, ETSI EN 302 307, Std., Rev. 1.2.1, August 2009.

[4] R. C. Reinhart and B. K. Smith, "Using international space station for cognitive system research and technology with space-based reconfigurable software defined radios," in *International Astronautical Congress*, October 2015.

[5] P. V. R. Ferreira, R. Paffenroth, A. M. Wyglinski, T. M. Hackett, S. G. Bilén, R. C. Reinhart, and D. J. Mortensen, "Multi-objective reinforcement learning for cognitive radio–based satellite communications," in *34th AIAA International Communications Satellite Systems Conference*, October 2016.

[6] ——, "Multi-objective reinforcement learning-based deep neural networks for cognitive space communications," in *1st IEEE Cognitive Communications for Aerospace Applications Workshop*, June 2017.

[7] ISO, *Information technology Programming languages C++*, ISO/IEC 14882:2011, Std., 2011.

[8] U. of California Berkeley, *Berkeley Software Distribution*, Std., 1999.

[9] F. S. Foundation, *GNU General Purpose License*, Std., 2007.

[10] ——, *GNU Lesser General Purpose License*, Std., 2007.

[11] J. C. Briones. (2016) STRS repository. NASA Glenn Research Center. Available at https://strs.grc.nasa.gov/repository/.

[12] R. Collobert, C. Farabet, K. Kavukcuoglu, and S. Chintala. (2017) Torch. Available at http://torch.ch/.

[13] Y. Jia and E. Shelhamer. (2017) Caffe. Berkeley Vision and Learning Center. Available at http://caffe.berkeleyvision.org/.

[14] J. Redmon. (2013–2017) Darknet: Open source neural networks in c. Available at http://pjreddie.com/darknet/.

[15] R. R. Curtin, J. R. Cline, N. P. Slagle, W. B. March, P. Ram, N. A. Mehta, and A. G. Gray, "MLPACK: A scalable C++ machine learning library," *Journal of Machine Learning Research*, vol. 14, pp. 801–805, 2013.

[16] R. Curtin and C. Sanderson. (2017) Armadillo: C++ linear algebra library. Available at http://arma.sourceforge.net.

[17] C. Kohlhoff. (2016) Boost.asio. Available at http://www.boost.org/doc/libs/1_63_0/doc/html/boost_asio.html.

[18] R. Ramey. (2004) Boost.serialization. Available at http://www.boost.org/doc/libs/1_63_0/libs/serialization/doc/index.html.

[19] D. Stenberg. (2017) Curl. Available at https://curl.haxx.se/.

[20] P. V. R. Ferreira, R. Paffenroth, and A. M. Wyglinski, "Interactive multiple model filter for land-mobile satellite communications at Ka-band," *IEEE Access*, vol. PP, no. 99, pp. 1–1, 2017.

[21] T. M. Hackett, S. G. Bilén, P. V. R. Ferreira, A. M. Wyglinski, and R. C. Reinhart, "Implementation of a parameterized interacting multiple model filter on an FPGA for satellite communications," in *34th AIAA International Communications Satellite Systems Conference*, October 2016.