A Layered UVM Based Testbench Design for SpaceWire

Ahmet Çağrı Bağbaba[1], Buse Ustaoğlu[1], İnan Erdem[2], Berna Ors[1]

[1] Istanbul Technical University, Istanbul, Turkey
bagbaba@itu.edu.tr, ustaoglubu@itu.edu.tr, Siddika.Ors@itu.edu.tr
[2] Anka Microelectronic Systems, Istanbul, Turkey
inan.erdem@ankasys.com

Abstract

The Universal Verification Methodology is a standard which is designed to enable creation of reusable, robust and interoperable verification IP and testbench components. In this work, we implemented layered UVM testbench for SpaceWire which is a spacecraft communication network based in part on the IEEE 1355 standard of communications. This design helps further analyzes of SpaceWire by testing different SpaceWire layers such as exchange layer and character layer. Transactions were used at all layer of protocol and user can make analysis, coverage collecting and debugging through this design. In the conclusion, all simulator results and details about Verification IP design were given.

1. Introduction

The main aim of verification of System on Chip (SoC) designs is to provide that the design meets the functional requirements as defined in the functional specification. Also, another goal is to ensure that the result of some transformation is as expected. Randomized stimulus for the chip inputs under control of user-specified constraints or rules are generated by modern testbench-based verification environment and this environment also checks the results of each test automatically. There are several verification methodologies in order to develop random-constrained testbenches. Universal Verification Methodology (UVM) is the best known verification method.

UVM is based on SystemVerilog classes and is powerful OOP (Object Oriented Programming) [1]. Manual generation of test vectors is the method of conventional functional verification. However, manual generation of test vectors is not supported well in the verification. Hence, UVM-based verification is important due to the fact that it automates the functional verification process [2]. Therefore, we created test environment for SpaceWire [3] interface by using UVM. SpaceWire is a spacecraft communication network and based on the IEEE 1355 standard. It is point to point and full-duplex serial bus network. It is widely used in satellite systems. In this work, we created Verification Intellectual Property (VIP) for SpaceWire interface and simulated it in order to show that the interface works correctly.

In this work, a layered UVM based verification IP is designed with an emphasis of making the analysis and reporting easier for the verification of SpaceWire based designs. This implementation can also be used as the SpaceWire VIP to help further analyze the SpaceWire networks by looking into the data at different SpaceWire layers in [4]. The implementation is emitting transactions at all SpaceWire protocol layers to the user. At the signal layer, the VIP emits the bits that it captures by decoding the DS-encoded Spacewire signals. The VIP itself uses these transactions internally to capture the disconnect error conditions and forwards them to the character encoder component. The character encoder then takes the incoming bit streams and extracts the SpaceWire characters and codes out of it. After having a complete character or an error condition, then it emits these character transactions to both upper layer, which is the packet collection layer and also to the driver to help manage the exchange layer tasks(link initialization, error recovery and flow control). Once the packet collector gets the characters from the character encoder, then it creates SpaceWire packets out of them and emits them to the user. As a result, user has access to all the transactions from signal level, through character level and up to packet level. Therefore user can take transactions from any of the above mentioned layer and do reporting, analysis, coverage collection, debugging etc.

The implementation provides user also with the predefined protocol level coverage data by using SystemVerilog covergroup structures, which can be analyzed by the user to see if all the corner cases of a SpaceWire communication was hit.

The rest of this paper is organized as follows: in Chapter 2, we introduced our Design Under Test (DUT), SpaceWire. In Chapter 3, UVM is explained in detail. In Chapter 4, implementation and creating of test environment are given. In the final chapter, we concluded this paper by summarizing implementation and novelty of this paper.

2. SpaceWire

SpaceWire is a data-handling network and uncomplicated to design or implement. It also provides high-speed (2 Mbits/s to 200 Mbits/s) full-duplex data links and bi-directional [5]. Using point-to-point data-links and routing switches provides data-handling networks. SpaceWire helps reduce system integration costs and aims to encourage harmony between data-handling equipment and subsystems. Moreover, the construction of high-performance on-board data-handling systems is reduced by SpaceWire [6]. Spacwire's use began primarily in European Space Agency but it is currently used by NASA and many other organizations.

In Fig. 1 example SpaceWire architecture can be seen. In this architecture, there are many instruments which are responsible for different duties. Instrument 1 is for high data rates. From Instrument 1 to Mass Memory Module point-to-point link is used to stream data [6]. Point-to-point links in SpaceWire is to connect a high-data rate instrument to a memory. Thanks to point-to-point links (Instrument 1 in Fig. 1) packets can be sent directly to the memory. Moreover, SpaceWire is a router-based architecture. The router enables the implementation of more complicated SpaceWire architectures. After the connection of all SpaceWire units through router, any unit can send or receive data from another unit [6].

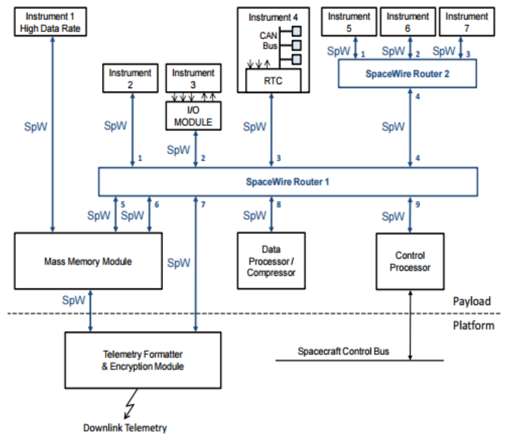


Figure 1. Example SpaceWire Architecture [6]

There are 6 layers in SpaceWire as Physical Layer, Signal Layer, Character Layer, Exchange Layer, Packet Layer, and Network Layer [3]. In Fig. 2, SpaceWire Open System Interconnection (OSI) Type Model can be seen. The Physical Layer explains cables and connectors. The Signal Layer defines voltage levels, encoding, and signaling rates. In the Character Layer, data is defined and control characters are used to manage data across link. In the Exchange Layer, flow control, link error detection, link error recovery, and link initialization are described. The Packet Layer handles transmission over SpaceWire link. In the last layer, Network Layer, source to destination node data transfers, and link errors are defined [7].

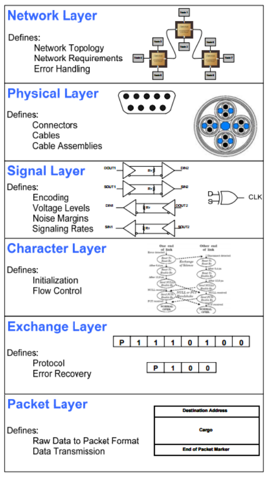


Figure 2. SpaceWire OSI Model [7]

In Physical and Signal Layers, Low Voltage Differential Signaling (LVDS) is used to execute communications such as packet and token passing accross a SpaceWire network. Also, Data-Strobe (DS) Encoded LVDS is used to communicate full-duplex, serial, and bi-directional data. In other words, DS encoding is used in SpaceWire in order to send information over LVDS. In Fig 3, data and strobe signals are XORed and synchronous clock is generated. The data values are transmitted directly and the strobe signal changes state whenever the data remains constant from one data bit interval to the next [6].

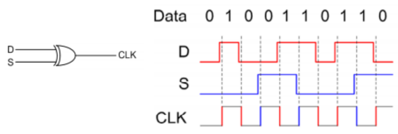
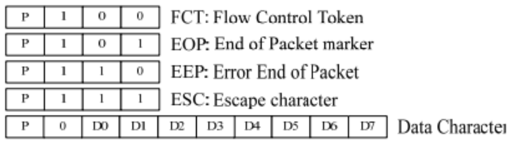


Figure 3. Data Strobe [7]

In the Character Layer, characters are defined as either link or normal characters. Normal characters pass through a SpaceWire network at a packet layer whereas link characters do not pass. As can be shown in Fig. 4, FCT and ESC are the link characters. ESC + FCT is the NULL control code and ESC + data character is the Time Code. Normal Characters, EOP and EEP, pass through a SpaceWire network at a packet layer. In the Exchange Layer, state diagram of the initialization and error recovery sequence, shown in Fig 5, is vital for this work. The flow starts or resets when an error is detected. These errors can be parity error, escape error etc.



P	1	0	0	FCT: Flow Control Token						
P	1	0	1	EOP: End of Packet marker						
P	1	1	0	EEP: Error End of Packet						
P	1	1	1	ESC: Escape character						
P	0	D0	D1	D2	D3	D4	D5	D6	D7	Data Characte

Figure 4. Character Layer [7]

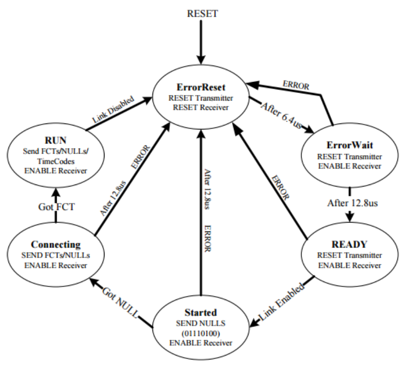


Figure 5. Exchange Layer State Diagram [7]

3. Universal Verification Methodology

UVM provides flexible and well established solution for complex system design verification. Since UVM consists of reusable components and is supported by tools of all major vendors of the industry, it is flexible [8]. In Fig 6, UVM testbench is shown. UVM verification elements are listed below.

- **Top Level:** It is the top level of VIP and includes Desing Under Test, uvm test, and interface.
- **Design Under Test:** In this work, SpaceWire is the DUT. DUT's funcionality is defined by using Hardware Description Language (HDL) such as Verilog or VHDL.
- **Interface:** It provides communication between DUT and verification environment. It also defines pin level activity of DUT.

- **Virtual Interface:** It connects ”virtual” world to ”real” world and also connects dynamic environment to RTL model.
- **uvm_test:** It is configurable and changeable depending on design.
- **uvm_env (Environment):** uvm env is used to create and connect the uvm components like driver, monitors , sequencers etc. A environment class can also be used as sub-environment in another environment.
- **Transactions:** A transaction is data item (packets, instructions etc.) which is eventually or directly processed by the DUT.
- **Agent:** Agent includes driver, sequencer, and monitor. If it is active, it should contain all three subtype. If it is passive, it should only include the monitor.
- **Monitor:** The monitor’s aim is to extract signal information and translate it into meaningful information to be evaluated by other components. Therefore, it does not drive any signals into the DUT.
- **Driver:** The driver pulls transactions from the sequencer and sends them repetitively to the signal-level interface. In other words, it is the block which interact with DUT.
- **Sequencer:** Sequencer sends the transaction to driver and gets the response from the driver.

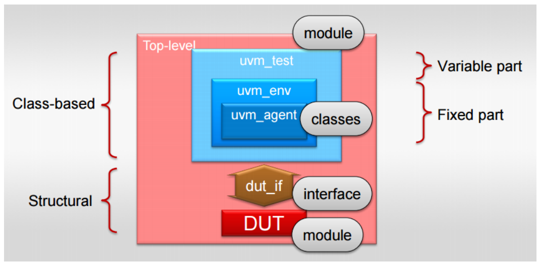


Figure 6. UVM Testbench [9]

Moreover, there are 4 phases in UVM as Build Phase, Connect Phase, Run Phase, and Report Phase. In Build Phase, child components, ports, and components are constructed and configured. In Connect Phase, ports and exports of components are connected. In Run Phase, main body of test is executed. In Report Phase, the pass and fail status are reported.

3.1. Coverage

Coverage is a metric that we use it in order to measure verification progress and completeness. Coverage metrics tell us what portion of the design has been activated during simulation (that is the controllability quality of a testbench).

4. Implementation

In this study, layered structure is proposed because it facilitates the organization of the implementation. Link, character and packet level information can be broadcasted the information to outside.

Our implementation can be shown in Fig. 7. The structure was explained in subsections.

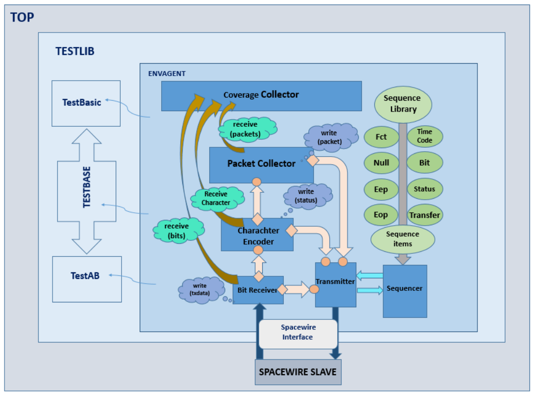


Figure 7. Overall Structure

4.1. Top Module

The module includes all of the components and determines the test code that is performed.

4.2. SpaceWire Slave

This unit is the DUT module which will be verified. Data and strobe signals produce the clock. The incoming data is sampled at the both positive and negative edges of the clock.

4.3. Test Library

Multiple tests can be used for observing different operations of the DUT and collecting variety of coverage results. We have defined 3 different tests in our test library. One of them is the base test and the other 2 are extended from it and add new properties. TestBasic sends Nulls and FCTs to see link initialization, TestAB sends N_Char, Nulls, EEP, EOP and TimeCode to generate a packet. Moreover, transmitter drives SpaceWire slave, which is based on defined characters, in these tests.

4.4. Environment and Agent

There is one agent in the environment. Thus, these are combined in this implementation and consist of driver, monitor and sequencer components and connect ports between them.

4.5. Line Encoder

Line encoder is the monitor type component and it collects the data bits from DUT via interface. It informs whether or not a link disconnection. Moreover, it writes the data to the analysis port and reach character encoder.

4.6. Character Encoder

Character encoder is also monitor type component. It writes states to the status analysis port in order to inform the driver about the status of the link before driving each single bits to the link. Based on the size of the received bits some actions are performed:

- **Size of received bits = 2:** The character type is determined and the parity of the previous character is checked. If there is a parity error, it is written to the status analysis port. If not, the first bit of the previous character is checked so as to determine it is control or data character.

- **Size of received bits = 4:** The bits other than parity and control flag are analysed. If the complete character is received and the character type is CONTROL, it is checked to determine whether or not it is an escape (ESC) character. If ESC is caught, the character type will be a CODE. Otherwise NON_CHAR type of characters that are FCT, EOP or EEP are determined based on the last 2 bits of the received data of this size and written to the status port (gotFCT or gotEOP or gotEEP flag is set to high).
- **Size of received bits = 5:** Character type is in CODE state and the code type is NULL. First, it is checked that if it is really a NULL character (ESC followed by FCT) otherwise an ESC error is generated. The status port is informed that a NULL code is received (gotNull flag is set to high). Also the new NULL code is written out of the character analysis port.
- **Size of received bits = 10:** In this situation, the character type is DATA. It is checked that whether the complete character is received or not and character type is changed to the NON_CHAR.
- **Size of received bits = 14:** Character type is in CODE state and the code type is TIME. Parity is calculated and the status port is informed that a TIME code is received (gotTime flag is set to high). Also the new TIME code is written out of the character analysis port.

4.7. Packet Collector

The another monitor type component catches the packets and write the port for broadcasting them outside.

- **Size of received bits = 4:** If the previous character type is DATA, it is followed by EOP and the current package is valid otherwise it is discarded. If the previous character type is CONTROL without a preceding DATA, it is also discarded.
- **Size of received bits = 8:** It is a NULL character and not part of the packet layer.
- **Size of received bits = 10:** DATA character is gotten.
- **Size of received bits = 14:** It is a TIME code. If it was gotten before the required EOP, it is discarded.

4.8. Transmitter

Transmitter is derived from UVM driver. It is connected to the line, character and packet encoder and depend on the information coming from the monitor type components. Line encoder informs the driver about the link status via received bits port, so the status of the link is checked, before driving each single bits to the link. At the same time character encoder notifies the error situations and gotten character types via a status port as mentioned in the previous sections. Moreover, credit error is handled in transmitter.

- **ERROR_RESET:** It is start point. If any error occurs in each state, they directly get back to this state.
- **ERROR_WAIT:** Rx is enabled. If there are any of the RX, FCT, N-Char or Time-Code error, it passes to the READY state after 12.8 us otherwise it returns RESET state.
- **READY:** Link is enabled. Unless any of the Rx, FCT, N-Char or Time-Code errors occur, it passes to the STARTED state otherwise it returns RESET state.
- **STARTED:** NULLs are sent. Unless one of the Rx, FCT, N-Char or Time-Code error information it returns RESET state. It passes to the CONNECTING state after gotNULL.
- **CONNECTING:** FCTs are started to be sent. If one of the Rx, N-Char or Time-Code error , it returns RESET state. It passes to the RUN state after gotFCT .
- **RUN:** N_Chars are started to be sent. It gets back to RESET state if RX or credit error occur. Credit error is handled in this state by checking number of N_Chars which can be sent when the FCT is received or the expected N_Chars which is transmitted based on the number of FCTs as in the protocol.

4.9. SpaceWire Interface

There are Spacewire DUT input and output signals in the interface which acts as a bridge and the DUT is became a blackbox. Driver component sends bits to the DUT and line encoder collects the bits via this bridge.

4.10. Sequence Library

Sequence library includes sequence items and sends them to the sequencer. In our implementation **Status**, **Bit**, **Transfer**, **EEP**, **EOP**, **FCT**, **Null**, **TimeCode** are used as sequence items. Then, these sequence items are randomized and sorted and get through to the driver.

4.11. Coverage Collector

Coverage collector is derived from UVM subscriber. It receives bits , characters, packets and status from related monitor type components and creates covergroups. These groups identify portions of the design that were never activated during simulation, which allows us to adjust our input stimulus to improve verification.

5. Simulation Results

The implementation has been simulated with Mentor Graphics QuestaSim 10.3 version.

5.1. Operational Results

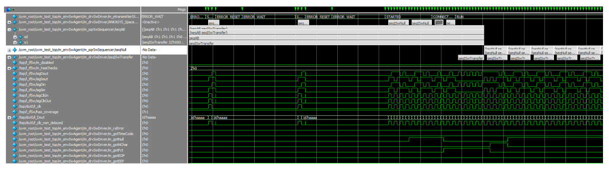


Figure 8. Clock Synchronization

Fig. 8 shows the signal level activity. As soon as Data and Strobe signals are sent within this period, they are synchronised and clock is generated. As a result, state transition is completed from beginning to the last state.

Link initialization transaction level activity can be seen in Fig. 9. STARTED state is returned to the RESET state a couple of times because the length of time since the last transition on Data or Strobe lines are sent longer than timeout period (850 ns).

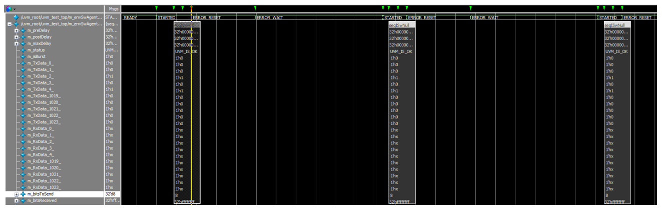


Figure 9. Link Initialization

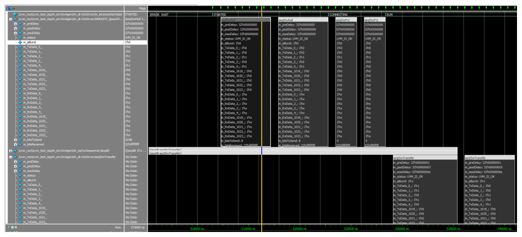


Figure 10. Started and Connect States

Fig. 10 shows the NULL and FCT character transactions in STARTED and CONNECTING state.

In Fig. 11, RUN state is showed. After N_Char is sent, it stays at the RUN state by keeping on sending NULLs.

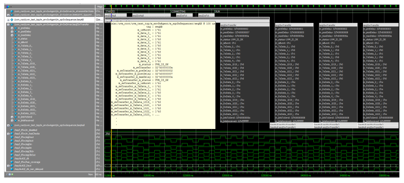


Figure 11. Run State

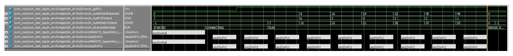


Figure 12. Credit Error

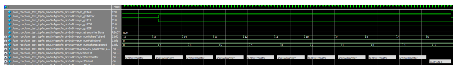


Figure 13. Credit Error

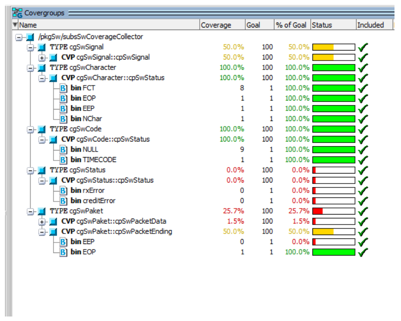
In Fig. 12, there are 9 FCTs which are sent to DUT but it causes credit error because the credit counter(N_Chars expacted) shall hold a maximum credit count of 56 due to the receive buffer capacity. Also, in Fig. 13, 9 N_Chars are transmitted after a FCT has sent which results in credit error, because for each FCT receive buffer can accommodate maximum 8 N_Chars.

5.2. Coverage Results

Fig. 14 shows coverage results. The declared covergroups can be seen explicitly in this graphic. For example, character and code covergroups have %100 coverage but Rx and credit errors have no coverage because these situation never occurs in the performed test. Multiple tests should be defined so as to take more valid results.

6. Conclusion

The layered UVM based VIP for SpaceWire protocol is implemented. The graphics give the VIP is the compatible with



Covergroups

Name	Coverage	Goal	% of Goal	Status	Included
/pkgSw/subsSwCoverageCollector					
TYPE cgSwSignal	50.0%	100	50.0%		✓
CVP cgSwSignal::cpSwSignal	50.0%	100	50.0%		✓
TYPE cgSwCharacter	100.0%	100	100.0%		✓
CVP cgSwCharacter::cpSwStatus	100.0%	100	100.0%		✓
bin FCT	8	1	100.0%		✓
bin EOP	1	1	100.0%		✓
bin EEP	1	1	100.0%		✓
bin NChar	1	1	100.0%		✓
TYPE cgSwCode	100.0%	100	100.0%		✓
CVP cgSwCode::cpSwStatus	100.0%	100	100.0%		✓
bin NULL	9	1	100.0%		✓
bin TIMECODE	1	1	100.0%		✓
TYPE cgSwStatus	0.0%	100	0.0%		✓
CVP cgSwStatus::cpSwStatus	0.0%	100	0.0%		✓
bin rxError	0	1	0.0%		✓
bin creditError	0	1	0.0%		✓
TYPE cgSwPaket	25.7%	100	25.7%		✓
CVP cgSwPaket::cpSwPacketData	1.5%	100	1.5%		✓
CVP cgSwPaket::cpSwPacketEnding	50.0%	100	50.0%		✓
bin EEP	0	1	0.0%		✓
bin EOP	1	1	100.0%		✓

Figure 14. Coverage

the the protocol specifications and errors are handled successfully. From bottom layer to top layer implementation provides a compact structure. Signals, characters and packets are received and written to the analysis ports, and transmitter takes status information from all layers. Users can observe the results and debug the codes in the transaction level more easily. Coverage results give more general idea and detailed coverage definitions and multiple tests are planned as the next work.

7. References

[1] Geng Zhong; Jian Zhou; Bei Xia, ”Parameter and UVM, making a layered testbench powerful,” *ASIC (ASICON), 2013 IEEE 10th International Conference on* , vol., no., pp.1,4, 28-31 Oct. 2013

[2] Neumann, F.; Sathyamurthy, M.; Kotynia, L.; Hennig, E.; Sommer, R., ”UVM-based verification of smart-sensor systems,” *Synthesis, Modeling, Analysis and Simulation Methods and Applications to Circuit Design (SMACD), 2012 International Conference on* , vol., no., pp.21,24, 19-21 Sept. 2012

[3] ECSS-E-50-12A(24 January 2003), SpaceWire, Links, Nodes, Routers and Networks

[4] Stohlmann, K.; Fey, G.; Ludtke, D., ”Automatic performance tracking of a SpaceWire network,” *in SpaceWire Conference (SpaceWire), 2014 International* , vol., no., pp.1-5, 22-26 Sept. 2014

[5] S.M. Parkes, ”SpaceWire: The Standard”, Proceedings, DASIA 99, Data Systems In Aerospace, 17-21 May 1999, Lisbon, Portugal, pp111-116, European Space Agency (ESA) publication no. SP-447, ISBN 92-9092-788-7.

[6] SpaceWire User's Guide (2012). [Online]. Available: https://www.star-dundee.com/knowledge-base/spacewire-users-guide

[7] Aeroflex Colorado Springs Application Note. *AN-SPW-004-002* [Online]. Available: http://ams.aeroflex.com/pagesproduct/appnotes /SpWin6PagesApNote.pdf

[8] Madan, R.; Kumar, N.; Deb, S., ”Pragmatic approaches to implement self-checking mechanism in UVM based TestBench,” *in Computer Engineering and Applications (ICACEA), 2015 International Conference on Advances in* , vol., no., pp.632-636, 19-20 March 2015

[9] Fitzpatrick, T. [Online]. Available: https://verificationacademy.com/cookbook