ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-WEBR01

RomLibEmu: NETWORK INTERFACE STRESS TESTS FOR THE CERN RADIATION MONITORING ELECTRONICS (CROME)

K. Ceesay-Seitz†, H. Boukabache*, M. Leveneur, D. Perrin, CERN, Geneva, Switzerland

Abstract

The CERN RadiatiOn Monitoring Electronics (CROME) are a modular safety system for radiation monitoring that is remotely configurable through a supervisory system via a custom protocol on top of a TCP/IP connection. The configuration parameters influence the safety decisions taken by the system. An independent test library has been developed in Python in order to test the system's reaction to misconfigurations. It is used to stress test the application's network interface and the robustness of the software. The library is capable of creating packets with default values, autocompleting packets according to the protocol and it allows the construction of packets from raw data. Malformed packets can be intentionally crafted and the response of the application under test is checked for protocol conformance. New test cases can be added to the test case dictionary. Each time before a new version of the communication library is released, the Python test library is used for regression testing. The current test suite consists of 251 automated test cases. Many application bugs could be found and solved, which improved the reliability and availability of the system.

INTRODUCTION

The Radiation Protection group within CERN is responsible for measuring levels of ionizing radiation at the CERN sites, experimental areas and in service caverns besides the LHC experiments in order to ensure the radiological safety of the persons on the CERN site as well as the people living in its neighbourhood. The CERN RadiatiOn Monitoring Electronics (CROME) have been developed in-house with the purpose of replacing the older radiation monitoring systems ARCON and RAMSES. In contrast to the old systems, CROME consists of fully independent units, called CROME Measurement and Processing Units (CMPUs), which perform their safety functions autonomously [1]. A CMPU consists of a radiation detector, which is an ionization chamber in most cases, a front-end board for the readout and the processing unit. The latter has at its core a Zynq-7000 System-on-Chip (SoC) with an embedded Linux running on the Processing System (PS) with integrated 32-bit ARM cores and a Programmable Logic (PL) section. Because the PL can operate autonomously and the design architecture is more immune to higher radiation than the PS [2], all the safety critical calculations and decision making are implemented inside the PL.

The CMPU's functionality is entirely configurable at runtime with roughly 150 parameters. This has the advantage that CROME can be deployed for very different usage scenarios – e.g. in service caverns or experimental areas with higher levels of radiation where it can be configured to trigger visible and audible alarms and with the possibility of being connected to machine interlocking systems, or as environmental monitors where the natural background radiation is monitored over long periods for informational purposes.

Figure 1 presents a system overview. During operation the CMPUs are connected to a SCADA supervisory system called REMUS – Radiation and Environment Monitoring Unified Supervision [3]. The CMPUs on the one communication end and the REMUS servers on the other end use the ROMULUS library [4] for communicating with each other. The library implements a custom protocol on top of TCP/IP that can be used to remotely configure the parameters on the CMPU at runtime, to read out its current and historical status, and to receive real time as well as historical measurement values.

The remote parametrization via REMUS is the only dedicated mechanism for users to configure the behaviour of the CROME system during operation. Radiation protection experts use REMUS to configure the CMPUs corresponding to the expected radiation conditions and safety requirements of a zone. Operators in the control room use REMUS to monitor the status and measurement results sent by the CMPUs.

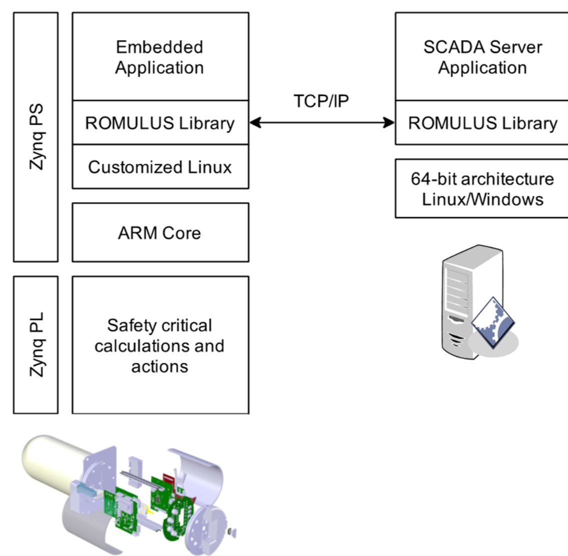

Figure 1: CROME Overview.

Since it influences the functionality as well as the safety function of the system, the communication mechanism needs to be robust and reliable. The RomLibEmu, the ROMULUS Library Emulation and Test Tool, has been developed to test the functionality and robustness of the ROMULUS library as well as of the CMPU's application.

† katharina.ceesay-seitz@cern.ch
* hamza.boukabache@cern.ch

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-WEBR01

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

The paper is structured as follows. First a definition for robust systems and robustness testing is provided. Then the RomLibEmu's concepts and functionalities are described in detail. Finally, the results and benefits obtained from using the tool are highlighted. A summary and an outlook conclude the paper.

BACKGROUND AND RELATED WORK

A system is considered as robust when it shows stability and acceptable behaviour in unforeseen operating conditions. Furthermore, it must not accept invalid input and certainly must not produce faulty output if presented with unexpected inputs. Rather it has to either reject the input or put itself into a safe or degraded state from which it can recover. Robustness is defined by the 610.12-1990 IEEE standard glossary of software engineering as "the degree to which a system or component can function correctly in the presence of invalid inputs or stressful environmental conditions". The latter could be network traffic overload, attacks or a misbehaving connected system [5]. It could also be physical conditions such as extreme temperatures or high radiation levels.

Robustness tests complement functional tests with test cases that particularly focus on sending boundary values or unexpected input values to a system with the goal of determining whether the system enters into an unwanted state such as an abort or a non-responsive state[6]. Common techniques for robustness testing are model-based techniques, fault injection, fuzzing, interception, code changes injection and mutation testing. Communication protocols are often tested with model-based techniques where a model is built from the specification which is then either directly evaluated or used to generate test cases. The most commonly used technique to evaluate the robustness of embedded systems is fault injection [7]. RomLibEmu currently focuses mainly on fault injection. In future it might be extended with model-based or fuzzing approaches.

The CRASH scale has been developed for classifying the severity of failures of operating systems when tested for robustness. CRASH stands for Catastrophic, Restart, Abort, Silent, Hindering [8]. It is widely used to classify robustness test results [7]. It was also adapted to classify failures of controllers in self-adaptive systems. Similar to our results the most common failure of the tested controller fell into the "Silent" category [9].

Besides aiding in the classification of test results it also provides a good guidance for scenarios that have to be tested for. We reinterpreted the scale as Catastrophic, Restart, Abort, Silent/Safety, Hindering/Harmless.

RomLibEmu: ROMULUS LIBRARY EMULATION AND TEST TOOL

Purpose and Motivation

As mentioned in the introduction, the network interface is the only dedicated mechanism for configuring the CROME Measurement and Processing Units (CMPUs) during operation. The first parametrization can also be done via the local file system, but this interface is only accessible to the development and maintenance team of CROME. Further utility tools exist that have been developed for testing and diagnosis. A physical test case, called CROME Case, that can read and manipulate external interfaces of the CMPU, has been developed as well. The tools and the test kit use the ROMULUS library for communication via the network. The library is intended to construct well-formed messages conforming to the ROMULUS-REMUS communication protocol. It only allows to send values of the expected datatypes for parameters and it is not capable of intentionally crafting malformed packets. While this feature strengthens the communication mechanism when the same library is used on both communication end points, it prevents the usage of the library itself for testing its own robustness and that of the CMPU application. The RomLibEmu, the ROMULUS Library Emulation and Test Tool, has been developed to overcome this limitation.

There are several scenarios in which it could happen that unexpected messages or malformed packets are sent to a CMPU:

- A fault in the ROMULUS library.
- A fault in the application that uses the ROMULUS library.
- The ROMULUS library versions of the CMPU application and the REMUS server are incompatible.
- Bad network connection or unexpected termination of the remote application causes ROMULUS packets to arrive only partially.
- Network or user-level misconfiguration:
  Packets that were not intended to be sent to the CMPU reach it because of a network misconfiguration. The application tries to interpret the irrelevant data according to the ROMULUS-REMUS communication protocol.
- Intentional attacks against CROME

Robustness testing of the CMPU's network interface is crucial to increase the availability and reliability of the radiation monitors. The availability would be affected by application crashes or aborts due to unexpected network traffic or heavy loads. The reliability would be affected if the CMPU would accept input values which it cannot process correctly. That could be e.g. input values that cause internal overflows or divisions by zero. The expected functionality of the CMPU would be affected if it would accept parameter values outside of the ranges defined in the requirements. These functional tests could be implemented by using the ROMULUS library itself. The already existing tools were mainly intended for manual tests and diagnosis. The RomLibEmu additionally facilitates the creation and generation of automated functional and robustness test suites. They can be used for automated regression testing as well.

Capabilities

The RomLibEmu has been developed independently from the ROMULUS library based on the protocol specifi-

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-WEBR01

cation. Figure 2 shows an overview of the test setup. The tool can be used to test any application software that uses the ROMULUS C library or any other library or application that implements the ROMULUS-REMUS communication protocol. The library provides functions for creating, sending and receiving packets via TCP/IP. Packets can be autocompleted to conform to the ROMULUS protocol or they can be intentionally crafted to not or only partially conform to the protocol. Further functions can interpret received packets and check for unexpected responses.

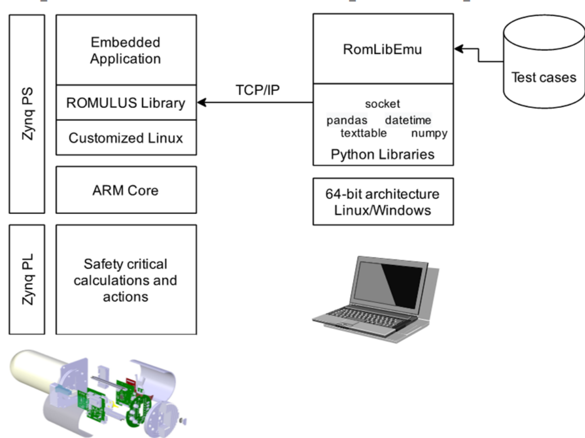

Figure 2: RomLibEmu Overview.

Metadata The protocol information is extracted from the protocol definition document in a semi-automated manner. The protocol consists of several different commands with similar structures. Each of them is identified by a unique command code and has a corresponding response code. Depending on the command type either the command or the response contains a list of parameters with or without added ID and of specified data types. Each packet further contains the data length, number of parameters, an ID and a checksum [4]. The command and response codes as well as the acknowledgement code for the messages have to be manually extracted from the documentation into a Python dictionary. The document contains several tables that define the parameter names, IDs and data types for the different messages. These need to be copied into a CSV (comma-separated values) file. The metadata generation scripts of the RomLibEmu reads the information and generates metadata dictionaries which are stored into dedicated files. These dictionaries can be used to access the parameters by name or ID and retrieve the corresponding data type byte size. The generated files are included into the RomLibEmu library. In case different protocol versions shall be tested, it is possible to use different input folders and include different dictionaries into the library.

Test case generation – conforming packets The library can be used to create test cases either manually or automatically. A test case can be specified as a dictionary with the ROMULUS packet field names as keys and the field's data as values. All values need to be provided as bytes in the correct byte order as described by the protocol. Values like the command codes can be retrieved from the generated dictionaries by their names. Several utility functions are available to facilitate the test case construction in a human readable form. The "DATA" field expects as value a dictionary that contains key-value pairs with the name of a parameter as key and its data as value. The parameter ID that will be placed into the protocol message is automatically determined from the metadata. Since the ID depends on the command type (there could be multiple parameters with the same name but different ID for different command types), a command code needs to be specified in the test case. In order to create a conforming packet, the ID, command code, number of parameters and the data need to be specified. The RomLibEmu then calculates the data length and the checksum and places them into the ROMULUS packet.

A test case also contains the expected response by the CMPU under test. The expected acknowledgement code depends on the test scenario and must therefore be added manually. In most cases the expected data values are only known for parameters and not for sensor measurements like the radiation dose or temperatures. In some test cases they can be added to the expected response packet. It is possible to add place holders for data values that cannot be automatically checked. Several fields of the expected response can be automatically determined by the RomLibEmu. The sender ID, the command's response code, the expected number of parameters and the data length can be generated. The checksum of the response packet can be calculated only if the expected data was known.

Test case generation – non-conforming packets Raw data can be inserted by adding a key that starts with the keyword "raw" and any data as value. These fields can be used to craft invalid packets. The default value of each field is an empty string. It is the responsibility of the test case writer to decide whether the lengths of the fields should match the ROMULUS-REMUS communication protocol for conforming packets or whether a non-conforming packet shall be created. In order to construct a packet manually it is possible to add the whole packet as a byte string to one of the fields of the packet and leave all other fields empty. Packets can also be modified after auto completion to facilitate the generation of malformed packets.

Use case The main purpose of the RomLibEmu is to facilitate the creation of automated test suits. Test cases can be defined in a test case dictionary. The RomLibEmu provides a function that traverses this dictionary and sends each packet over a TCP/IP socket to the target device. After sending a packet it waits for the response and compares it with the expected one that was defined in the test case. This verifies the ROMULUS packet structure which tests the ROMULUS library implementation as well as the content of the packet which tests the software application that uses the library. Automated test suits can be used fully or partially in regression test runs which are performed whenever a new software version is available.

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-WEBR01

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI

The functions of the RomLibEmu can also be used to freely create more complex functional test cases that consist of several ROMULUS commands. Scenarios that already lead to failures of the system can be reconstructed for analysis. Once the underlying fault has been removed, the same test case can be used in regression runs. The library can also be used by developers for test-driven development.

RESULTS

A test plan has been written containing around 80 main test cases which consist of several sub test cases. They are associated to the following categories:

1. Good cases – Packets that conform to the ROMULUS-REMUS communication protocol.
2. Application tests – Packets with correct ROMULUS packet structure, but bad parameter configuration.
3. Bad network connection – Simulating the loss of packets, long delay, etc.
4. ROMULUS library tests – Malformed ROMULUS packets that were generated due to bugs in the ROMULUS library; Network packets that were not targeted to the CMPU but arrive because of misconfiguration and are therefore treated by the application like legitimate traffic.
5. Denial-of-Service – Overload of the network interface by sending too many messages in a short time or in parallel, unexpectedly long packets.
6. Intentional attacks
7. Regression tests – Test cases that have already revealed a bug in some version of the application or the ROMULUS library

The responses of the CMPU's application can be classified according to the CRASH scale [8] as follows:

- Catastrophic – The OS crashes.
- Restart – Application hangs and requires a restart.
- Abort – The application crashes.
- Silent/Safety – An application error is expected but does not get triggered.
- Hindering/Harmless – The application returns a wrong error code.

Each time before a new software or firmware version is released, the test suite implemented with the RomLibEmu is executed. Table 1 lists the failures of the test runs per CRASH category. The robustness and in particular the availability of the CMPU was tested by trying to cause a "Catastrophic", "Restart" or "Abort" scenario. No test scenario fell into the "Catastrophic" category. One test case caused the application to hang and four test cases caused the application to crash, falling into the "Abort" category. "Silent" faults are very critical for safety, therefore we renamed the category into "Silent/Safety". These are scenarios in which the application would accept bad input that it cannot handle and it would not notify the user. In such scenarios one would rely on a faulty safety system in the belief that it is operating as specified, which could be very dangerous. The faults falling into the "Hindering/Harmless" category were not considered as critical. They are wrong behaviour from a functional perspective, but from a safety perspective in the case of the CMPU it is sufficient if any error code is returned and bad input is rejected. For another system, however, these faults may be critical. We also added a category "Functional" which contains test cases that failed because the response of the system did not fully conform to the protocol, but the error code was the expected one. From safety perspective they are not critical either.

The following test cases lead to "Abort" responses, meaning that the application crashed. (The numbers in brackets following each paragraph are the test case categories):

- A floating point parameter was set to quiet NaN (Not a Number).
  The ROMULUS library is not supposed to accept NaNs, therefore this tested the library implementation as well. Interestingly a test case with a signalling NaN passed. (2, 4, 6)
- An Ethernet frame with maximum data length was sent. The length of the IP header plus the length of the TCP header plus the length of the data equalled the Ethernet MTU of 1500. The TCP data was all 0. A similar test case where the TCP data contained arbitrary data, but the data length specified in the packet matched the maximum legal data length and the ROMULUS packet checksum was correct passed. (4, 6, 7)
- A packet with arbitrary data in the TCP data section with a packet length larger than the maximum ROMULUS packet length was sent. The data length inside the packet was set to 0 and the packet had a wrong checksum. (4, 6, 7)
- Many commands were sent in parallel. (2, 4, 5, 6)

Some examples of "Silent/Safety" failures were:

- Negative/positive infinites as well as quiet and signalling NaNs were sent to floating point parameters. The values were accepted and passed on to the PL that contains the safety critical code. (2, 4, 6)
- Invalid packet structures were accepted. The application therefore treated frame data like the checksum as parameter data and stored it silently. (2, 4, 6)

Table 2 summarizes the test runs. With each run some faults were removed and some more test cases were added. The number of passing test cases increased with each improved software version. In some intermediate test runs, however, previously passing test cases failed due to regressions that were introduced into the software or the firmware of the PL. Thanks to the test suite this could be spotted and fixed before releasing the new software and firmware versions.

CONCLUSION AND OUTLOOK

The RomLibEmu tool has been developed to test the robustness of the ROMULUS library as well as the CMPU's application that uses the library. It was successfully used

ISBN: 978-3-95450-221-9 ISSN: 2226-0358 doi:10.18429/JACoW-ICALEPCS2021-WEBR01

Table 1: Failures per CRASH Category

Test Run	Date of SW	Silent/ Safety	Abort	Restart	Hindering	Functional
1	04/09/19	19	2	1	3	2
2	11/12/19	10	4	0	3	2
3	20/12/19	6	0	0	3	2
4	26/05/20	6	0	0	3	3
5	02/03/21	2	0	0	3	2
6	30/09/21	0	0	0	3	0

Table 2: Test Run Summary

Test Run	Date of SW	Nr. Test Cases	Passed	Failed
1	04/09/19	57	30	27
2	11/12/19	58	39	19
3	20/12/19	64	53	11
4	26/05/20	65	53	12
5	02/03/21	81	74	7
6	30/09/21	83	80	3

Many of the main test cases consist of several sub cases. In total the current test suite contains 251 test cases.

for finding faults triggered by unexpected parameter values or malformed ROMULUS packets. It has been further used to test the input sanitizing module which performs parameter range checks once when receiving parameters over the network or reading them from the file system as well as once before writing derived values to the Programmable Logic section of the SoC. A few test cases have already been written that test functionalities of the full system covering PS and PL sections of the SoC.

This test bench clearly improved the quality, availability and safety of the CROME Measuring and Processing Units. So far the test bench focuses only on a small section of the functionality and on the communication interface. In future version we intend to extend the tool to further facilitate the test case generation for functional and safety tests. More benefits are expected from a more comprehensive test suite. There is also room for more automation. Model-based testing may be done by adding the capability of generating random input values and by adding an interface to a reference model of the system that predicts the expected outputs. Furthermore the tool may be combined with fuzz testing tools or modules with focus on network security.

REFERENCES

[1] H. Boukabache, M. Pangallo, G. Ducos, N. Cardines, A. Bellotta, C. Toner, D. Perrin, and D. Forkel-Wirth, "TOWARDS A NOVEL MODULAR ARCHITECTURE FOR CERN RADIATION MONITORING", *Radiation Protection Dosimetry*, vol. 173, April 2017, pp. 240–244. doi:10.1093/rpd/ncw308

[2] C. Toner, H. Boukabache, G. Ducos, M. Pangallo, S. Danzeca, M. Widorski, S. Roesler, and D. Perrin, "Fault resilient FPGA design for 28 nm ZYNQ system-on-chip based radiation monitoring system at CERN", in *Microelectronics Reliability*, vols. 100–101, p. 113492, 2019. doi:10.1016/j.microrel.2019.113492

[3] A. Ledeul, G. Segura Millan, A. Savulescu, B. Styczen, and D. Vasques Ribeira, "CERN Supervision, Control and Data Acquisition System for Radiation and Environmental Protection", in *Proc. 12th International Workshop on Personal Computers and Particle Accelerator Controls (PCaPAC'18)*, Hsinchu City, Taiwan, Rep. of China, 2018, pp. 248-252. doi:0.18429/JACoW-PCaPAC2018-FRCC3

[4] A. Yadav, H. Boukabache, K. Ceesay-Seitz, and D. Perrin, "ROMULUSlib: An autonomous, TCP/IP-based, multi-architecture C networking library for DAQ and Control applications", presented at 18th Int. Conf. on Accelerator and Large Experimental Physics Control Systems (ICALEPCS'21), Shanghai, China, Oct. 2021, paper MOBR01, this conference.

[5] S. Shah, D. Sundmark, B. Lindström, and S. Andler, "Robustness Testing of Embedded Software Systems: An Industrial Interview Study", in *IEEE Access*, vol. 4, pp. 1859-1871, 2016. doi:10.1109/ACCESS.2016.2544951

[6] C. Hutchison *et al.*, "Robustness Testing of Autonomy Software", *2018 IEEE/ACM 40th International Conference on Software Engineering: Software Engineering in Practice Track (ICSE-SEIP)*, 2018, pp. 276-285. doi:10.1145/3183519.3183534

[7] N. Laranjeiro, J. Agnelo, and J. Bernardino, "A Systematic Review on Software Robustness Assessment", *ACM Computing Surveys*, vol. 54, issue 4, nr. 89, pp. 1-65, 2021. doi:10.1145/3448977

[8] P. Koopman, J. Sung, C. Dingman, D. Siewiorek, and T. Marz, "Comparing operating systems using robustness benchmarks", *Proceedings of SRDS'97: 16th IEEE Symposium on Reliable Distributed Systems*, 1997, pp. 72-79. doi:10.1109/RELDIS.1997.632800

[9] J. Camara, R. de Lemos, N. Laranjeiro, R. Ventura, and M. Vieira, "Robustness Evaluation of Controllers in Self-Adaptive Software Systems", *2013 Sixth Latin-American Symposium on Dependable Computing*, 2013, pp. 1-10. doi:10.1109/LADC.2013.17

Content from this work may be used under the terms of the CC BY 3.0 licence (© 2022). Any distribution of this work must maintain attribution to the author(s), title of the work, publisher, and DOI