Runtime reconfigurable system for decommissioned satellite identification and capture

Raquel de Esteban
TEISA Dpto.
University of Cantabria
Santander, Spain

Fernando Manteca
TEISA Dpto.
University of Cantabria
Santander, Spain

Marcos Martinez de Alejandro
Thales Alenia Space
Madrid, Spain

Pablo Sanchez
TEISA Dpto.
University of Cantabria
Santander, Spain
sanchez@teisa.unican.es

***Abstract*—The increasing number of space missions has led to the accumulation of space debris, becoming a problem to be taken into account. A possible solution to eliminate rubble in space consists of launching satellites capable of detecting these obstacles and then destroy them. This paper presents a system for decomissioned satellite identification and capture. The system was developed with a methodology which provides support for component management as well as runtime system reconfiguration. The proposed solution is able to reconfigure itself at runtime with different configurations that provide different performance and energy consumption strategies to adapt to the environmental conditions during a space mission.**

***Index Terms*—runtime reconfiguration, space debris, satellites, object recognition, component-based**

I. Introduction

Man began his adventure in space with the launch of the Sputnik satellite in 1957 by the Russian Federal Space Agency. Since then, a large number of countries have launched satellites, probes and spacecraft, reaching more than 8,300 objects/vehicles in 2018.

Some of these objects have returned to Earth and disintegrated upon entering the atmosphere. But, many others have remained in orbit or disintegrated into small fragments that orbit around the Earth at speeds around 27,000 km/h. All these fragments, which vary from the size of a grain of rice to a truck, are called rubble or space debris.

The presence of fragments travelling at these speeds at altitudes where satellites usually orbit for navigation, communication or observation is of concern to space agencies. A small object at such high speeds could disable a subsystem of a satellite if impacting specific areas.

In recent years, concern about space debris has increased in space missions. Various types of space debris are found floating in space at different distances from Earth. Although it is unlikely for a spacecraft to hit any of this debris, the increasing number of mega-constellations of more than 100 satellites increases the probability of a collision.

This work was done as part of the FitOptiVis project, funded by the ECSEL Joint Undertaking (grant H2020-ECSEL-2017-2-783162), and the Platino project, funded by Spanish MINECO (Reference TEC2017-86722-C4-3-R)

978-1-6654-2116-4/21/$31.00 ©2021 IEEE

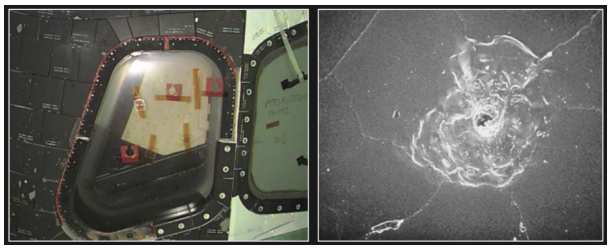

Fig. 1. Micro-impact found in a space shuttle window [1]

There is such a high density of space debris in some orbital planes that the probability of being hit is high. A clear example of this are the impacts received in each NASA space shuttle mission [1]. After each mission, elements such as the windows or the radiators placed on the doors were inspected in detail with microscopes, since a micro crack could lead to a catastrophe. Figure 1 reflects one of the micro-impacts found on one of the windows of a space shuttle.

Overall, NASA found more than 1,000 impacts on the shuttle over its years of operation [1]. This caused an average of 2 window changes as shown in Figure 1 after each mission.

The biggest contribution to space debris are satellites. Most satellites that are sent into space have a lifespan of 5 to 15 years. Once they stop being functional, they become part of the set of objects that make up space junk. In total, 24% of objects that are orbiting the earth are satellites, but only a third of these are in operation.

These satellites lose altitude over time, causing them to eventually disintegrate upon entering the atmosphere. This can take years from the moment it ceases to be functional until it finally falls to Earth. During this period of time, a series of events can occur that increase the number of objects that make up the space debris cloud.

Since 1961, more than 290 fragmentation events of orbiting objects have been recorded. In most cases these occur due to explosions caused by the satellites themselves or the upper stages of rockets. The main cause of these explosions is related to the residual fuel that remains in the tanks or in the pipes.

Over time, the internal and external mechanical integrity of

the satellite is reduced due to minor impacts or deterioration of materials. This can lead to leaks and/or the mixing of different fuel components causing an explosion. If the explosion is large enough it can destroy the satellite completely and scatter a large number of fragments of different sizes in many directions and at widely varying speeds [2].

II. Motivation

Different space agencies have started to think of various solutions to free the orbits of space debris. One possible approach is to launch satellites capable of capturing space debris and, at the end of the mission, burn with it in the atmosphere. In this work, a video unit capable of identifying satellite models has been designed and implemented. A satellite equipped with this system could be able to identify satellites that are no longer working and capture them to burn them when they fall to earth or redirect them to a "graveyard orbit".

The system presented in this work is made up of various software components with functionalities including video processing, image coding, recognition and positioning of objects, among others. To implement and connect the components, a component implementation methodology [3] has been used that allows the system to reconfigure itself at runtime with different configurations (set points) that allow different performances and energy consumption strategies to adapt to the environmental conditions that might be encountered during a space mission.

III. System Architecture

As shown in Figure 2, the implemented system is divided into two blocks: *Space Segment* and *Ground Segment*. The *Space Segment* encompasses all the components that will be integrated into the satellite, while the *Ground Segment* integrates the satellite's remote control system. Both subsystems are connected by a radio link.

The *Space Segment* includes two "chains" or paths for the video received from the cameras. The first path, at the top of the subsystem, simply compresses a high resolution image. Whereas, the lower path recognises a satellite in the image and provides an image including the result of the detection. The position of the detected satellite is used to guide the platform towards it.

On the other hand, the *Ground Segment* receives these two video streams, decompresses them and displays them on the screen so that the user on Earth can see what is happening in space during the mission at all times. It also receives the system's qualities and the directions to reach the target satellite, so they are displayed through a graphic user interface (GUI).

The *Camera* component models RGB and depth cameras. Two cameras are used in the implementation: a high resolution RGB camera (*Cam1*) and a depth camera (*Cam2*). This component is in charge of capturing and processing image frames from the cameras and sending them to the rest of the system.

The *Encoder* component provides image compression, making use of the recommended standard CCSDS122 [4]. On the other hand, the *Decoder* component decompresses the image that has been compressed by the *Encoder* and sends it to the *Display* component to be shown to the user. An example of the received images can be seen in Figure 3

The *Recogniser* component identifies satellites in the image using an algorithm based on convolutional neural networks. This component uses RGB and depth images to determine the position of the target satellite. If depth imaging is available, the component will also be able to determine how far away the target satellite is. Combining this information, the component defines the path that the mission satellite must follow in order to reach the target satellite. In turn, the *Satellite Pilot* component receives the commands from the *Recogniser* to move the satellite accordingly.

For the development of the *Recogniser* component, a study of the available algorithms for the location and detection of objects has been carried out, concluding that the use of Machine Learning techniques is the best way to perform satellite recognition. Within the field of Machine Learning, algorithms based on neural networks have been studied, and within these, convolutional neural networks, which are the most used for image recognition and processing.

The speed and precision of the most popular object recognition models have been compared and among the two fastest, choosing the one with the greatest precision. The chosen model has been the SSD MobileNet v2 [5]. This model has been trained using Tensorflow [6] for 12 hours with about 300 images of each satellite obtaining a total loss of 0.137 which

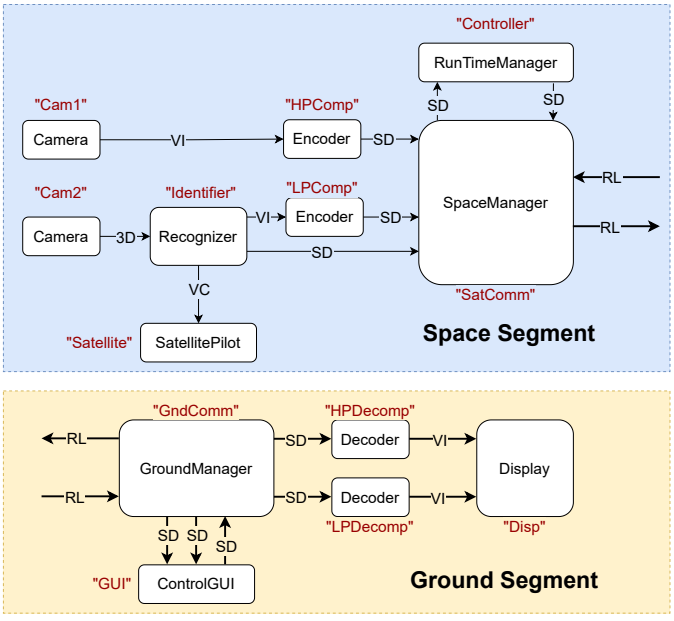

Fig. 2. System Architecture

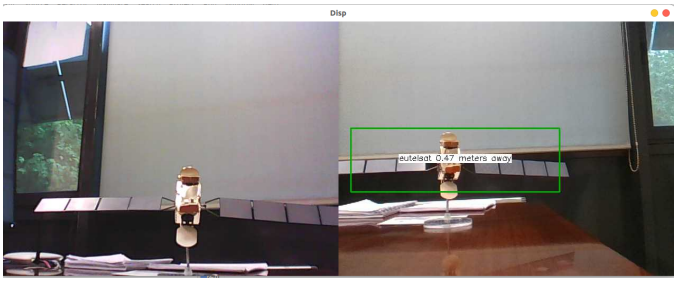

Fig. 3. Images received in the *Ground Segment*

is a good result, since the ideal is that the training loss falls between 0.2 and 0.15 for the selected model [7].

The *Space Manager* and *Ground Manager* components communicate the satellite with the Earth segment. These components act as multiplexers, sending various data streams from the satellite to Earth and vice versa.

The *Runtime Manager* component is in charge of evaluating the qualities of the system (power consumption, battery, etc.) and changing its configuration according to their values.

The *ControlGUI* component is responsible for remote control of the system. Therefore, it provides the user with the ability to change the system's set point. In addition, it monitors the qualities of the satellite that performs the mission.

A. Component connnection

Components are connected using their provided and required services. Thanks to these, the components receive and send data to each other. In the proposed design, there are five different service types.

- **VideoInterface (VI):** This service provides an RGB image. The components that provide this service are the *Camera 1*, which sends frames captured by the camera; the *Recogniser*, which provides the RGB frame coming from *Camera 2* with a printed rectangle locating the satellite if it has been identified; and both *Decoder* components, which send the decompressed RGB frames to the *Display*.
- **Image3D (3D):** This service provides an RGB frame and a depth map frame. The only component that provides this service is *Camera 2*, which is in charge of opening and controlling the streaming pipeline of the Intel Realsense [8] camera.
- **VehicleControl (VC)** is provided by the *Recogniser* consisting of the direction orders that the *Satellite Pilot* component must execute in order to reach the target satellite.
- **StreamData (SD)** provides generic data packets or blocks. The components that provide this service are both *Encoder* components, which send compressed image data; the *Runtime Manager*, which sends the system's qualities; the *Space Manager*, which sends the desired setpoint; and the *Ground Manager* which sends the direction orders and system's qualities to *ControlGUI* and the coded images to both *Decoder* components.
- **RFLink (RL)** service works exactly as *StreamData*. The difference between these services is that *RFLink* is only used to send and receive data packets between the *Space Manager* and the *Ground Manager*.

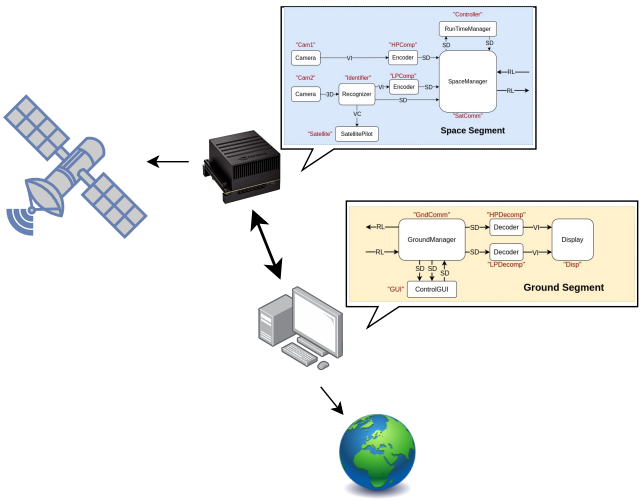

Fig. 4. Platform distribution of the proposed system

IV. Mapping to HW platforms

There are two main platforms, one for the *Space Segment*, which is an embedded system in the satellite that performs the mission, and one for the *Ground Segment*, which is a PC on Earth. The embedded system and the PC communicate via an RF link, as shown in Figure 4.

The *Space Segment* can be implemented on a single platform, which will be called the main platform, or have its components distributed in several boards. The main platform is the Jetson AGX Xavier [9] board coloured in red in Figure 5. All platforms in the *Space Segment* are connected to the same switch via ethernet, as shown in Figure 5. Also, the main platform is connected with the PC on Earth through an RF link. Since this platform is the one that communicates the satellite with Earth, the *Runtime Manager* and *Space Manager* components will always be mapped to it.

The allocation of the remaining components of the *Space Segment* can be modified at runtime through reconfiguration. Components *Cam1* and *Cam2* can be implemented on two Jetson Nano [10] platforms when they are not on the main platform. In addition, *Recogniser* component can be implemented on the other Jetson AGX Xavier [9] and the *Satellite Pilot* on a Jetson TX2 [11]. The FPGAs are used for the HW implementation of the *Encoder* components.

A. Component implementations

Each component has one or more implementations, which can be changed at runtime through reconfiguration. Implementations may vary the used algorithm to provide the compo-

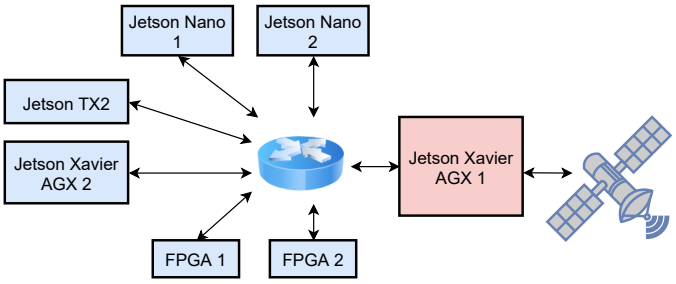

Fig. 5. Platforms and embedded systems used in the *Space Segment*

nent's functionality or simply change the platform on which such component is implemented.

Every component in the *Ground Segment* has only one implementation, while most components in the *Space Segment* have more than one implementation.

The *Camera* component has five possible implementations, with the first one being the default implementation:

1) **Turned off.** No camera is available, so the *Camera* component will send a blank image.
2) **RGB.** This implementation manages opening an RGB camera and capturing its video frames. Only *Cam1* uses this implementation.
3) **RGB remote.** This implementation behaves exactly as the previous one, but it implements the *Camera* component on a platform other than the main one. Only *Cam1* uses this implementation.
4) **3D.** This implementation manages opening an Intel Realsense [8] camera and capturing its video and depth frames. Only *Cam2* uses this implementation.
5) **3D remote.** This implementation behaves exactly as the previous one, but it implements the *Camera* component on a platform other than the main one. Only *Cam2* uses this implementation.

The *Encoder* component has three possible implementations, with the first one being the default implementation:

1) **Software.** This consists of a software implementation of the CCSDS122 recommended standard for Space Data Systems [4].
2) **Software remote.** This implementation behaves exactly as the previous one, but it implements the *Encoder* component on a platform other than the main one.
3) **FPGA.** Implements the hardware version of CCSDS122 standard [4] in a FPGA. An HLS process from specific C code has been used to the this configuration.

The *Recogniser* and *Satellite Pilot* components both have two implementations. A local implementation in the "main platform" and a remote implementation to run these components on another platform.

The *Runtime Manager* and *Space Manager* components only have one implementation since they are always mapped to the main platform and should not be implemented on another platform.

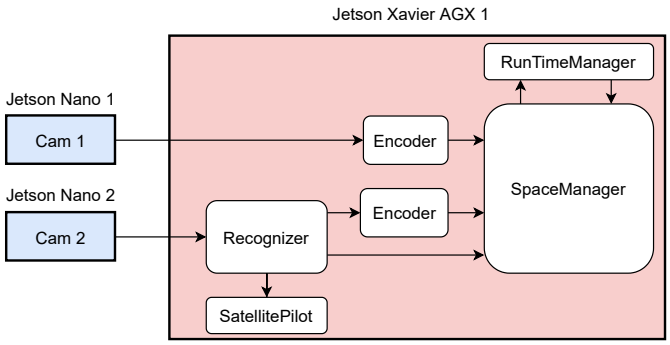

Fig. 6. Configuration "s2"

B. System configurations

The system has five different configurations depending on the distribution of the components on the different platforms shown in Figure 5. Since all components in the *Ground Segment* will always be implemented in a PC, the configurations refer to components in the *Space Segment*.

Configuration "s0", or default configuration, consists of all components being mapped to the main board. In this configuration, both *Camera* components use the "Turned off" implementation, so no image is captured.

Configuration "s1" also consists of all components being mapped to the main board. But, in this configuration, *Cam1* uses its "RGB" implementation while *Cam2* uses the "3D" implementation.

In configuration "s2" both *Camera* components are implemented remotely, each in one Jetson Nano [10] as shown in Figure 6. *Cam1* uses the "RGB remote" implementation and *Cam2* uses "3D remote".

Configuration "s3" uses the remote implementation of both *Camera* components as in configuration "s2", and the remote implementation of the *Recogniser* and *Satellite Pilot*, as shown in Figure 7.

Configuration "s4" uses the remote implementation of the *Camera*, *Recogniser* and *Satelite Pilot* components. Both *Encoder* use their FPGA implementation.

V. Reconfiguration Algorithm

The component implementation methodology proposed in [3] has been used to develop the software components and to allow the system to reconfigure itself at runtime. To control system operation and reconfiguration, all components software must implement the following functions derived from the methodology's library, RIE (Runtime reconfiguration Implementation of Embedded systems) [3].

- `run()`: Starts the component's threads for the first time. If the component does not use threads, nothing is done.

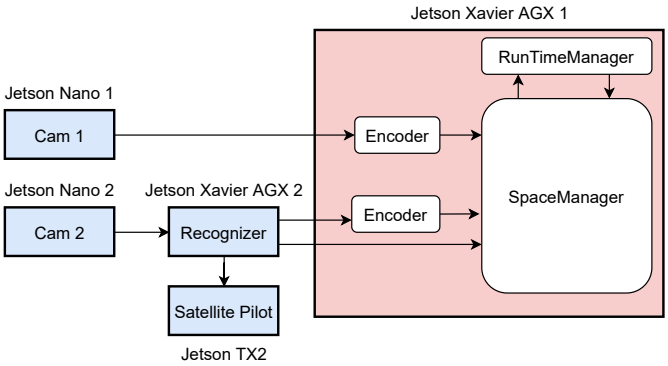

Fig. 7. Configuration "s3"

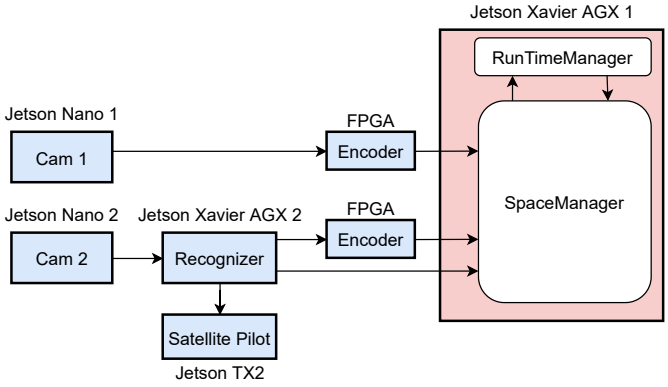

Fig. 8. Configuration "s4"

- `isStopped()`: Checks if the component is stopped to be able to reconfigure it.
- `stop()`: Stops the active threads that the component has but it does not wait for them to have stopped. This is why the above function is necessary.
- `resume()`: Restarts the threads of the component once it has been reconfigured.
- `assignSetPoint(string s)`: Changes the component's implementation.

The algorithm used to reconfigure the system follows these steps:

1) **Put the system in a safe state.** By calling the `RIEstop()` function, RIE accesses the `stop()` function of all system components to stop them. This process might take some time because some components may need time to complete their current task. For this reason, the `RIEisStopped()` function is used, which calls the `isStopped()` function of each component to check the status of each one until all components are in a safe state.
2) **Reconfigure the system.** To reconfigure the entire system, the `reconfig(string sp)` function of `RIEComponent` shall be called. This function takes the name of the new configuration to which you want to reconfigure the system must be passed to this function as a parameter.
3) **Resume execution.** Once the system has been reconfigured, the `RIEresume()` function must be called to resume the tasks of the components. The `RIEresume()` function calls the `resume()` function of each component.

Since the proposed system is capable of reconfiguring itself at runtime, a reconfiguration strategy must be defined. The defined strategy is based on the system's latency, power consumption, and battery life.

The system starts on configuration "s0" and after waiting 1 second switches to configuration 's1". Then it checks if any of the following scenarios is occurring.

If the system is in configuration "s1" and power consumption of the main board goes above a threshold, the configuration changes to "s2".

If the system is in configuration "s2" and latency goes above a threshold, the configuration changes to "s3".

If the system is in configuration "s3" and latency goes above a threshold, the configuration changes to "s4".

If the system is in configuration "s4" and system's battery life goes below a threshold, the configuration changes to "s3".

If the system is in configuration "s3" and system's battery life goes below a threshold, the configuration changes to "s2".

If the system is in configuration "s2" and system's battery life goes below a threshold, the configuration changes to "s1".

VI. Evaluation

The latency of the proposed system has been measured for all the possible configurations. The obtained results are shown in Figure 9, which shows the average latency for each configuration except for "s0", whose latency information is irrelevant to the performance of the system.

The component that most affects the system's latency is the *Recogniser*, since it is the one who needs more computational

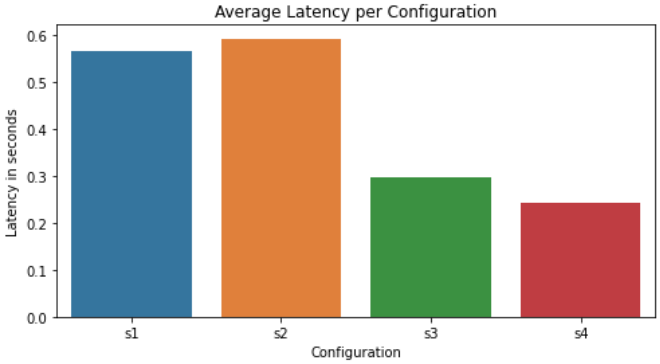

Fig. 9. Average Latency per Configuration

resources. As it can be seen in Figure 9, the average latency for configuration "s1" is around 0.55 seconds which equals to processing 2 frames per second. In configuration "s2" the latency increases a little due to the connection delay caused by the remote implementation of the cameras.

For configuration "s3" the average latency is around 0.3 seconds, which means that implementing the *Recogniser* on a dedicated platform decreases the system's latency. This platform is the Jetson Xavier AGX, which uses Tensorflow Lite, a special version of Tensorflow [6] purposely made for embedded systems. Configuration "s4" improves this value by using the FPGA implementation of the *Encoder* components.

Reconfiguration time has also been evaluated, obtaining an average time of 0.5742 seconds when switching configurations.

VII. Conclusions

This work presents the development of an application for space exploration using a methodology [3] that allows the dynamic reconfiguration of the system. The purpose of the application is the identification of decommissioned satellite models. The system is divided into two segments, *Space Segment* and *Ground Segment*. The *Space Segment* is made up of all the components that will be integrated into the satellite that performs the mission, while the *Ground Segment* integrates the satellite's remote control system. Both subsystems would be connected by a radio link.

In order to develop the proposed application, several architectures have been studied to select those most suitable for the system. For the development of the *Recogniser* component, the available object recognition models have been evaluated, selecting the one which had the best trade off between accuracy and speed. To implement the *Encoder* component, a high level synthesis of specific `C` code has been carried out.

The application achieves latency times between 0.6 and 0.2 seconds, which allows the balance between latency and power consumption. System's power consumption can be reduced due to the reduction of active platforms using the proposed reconfiguration strategy. At the same time, latency can be reduced by expanding the number of available platforms and thus increasing the amount of computational resources for each component.

The reconfiguration process takes an average of 0.5742 seconds. This time is small enough to not lose a significant amount of image frames and as a consequence, not to damage the performance of the system.

References

[1] "Space environments." https://www.nasa.gov/centers/johnson/pdf/584742main_Wings-ch5g-pgs444-458.pdf.

[2] "About space debris." https://www.esa.int/Safety_Security/Space_Debris/About_space_debris.

[3] F. van den Berg, V. Camra, M. Hendriks, M. C. Geilen, P. Hnetynka, F. Manteca, P. Sanchez, T. Bures, and A. T. Basten, "QRML: A component language and toolset for quality and resource management," in *2020 Forum on Specification and Design Languages*, 2020.

[4] "Image Data Compression." Issue 2. Recommendation for Space Data System Standards (Blue Book), CCSDS 122.0-B-2., Washington D.C.: CCSDS, September 2017.

[5] "Tensorflow 1 detection model zoo." https://github.com/tensorflow/models/blob/master/research/object_detection/g3doc/tf1_detection_zoo.md.

[6] M. Abadi, A. Agarwal, P. Barham, E. Brevdo, Z. Chen, C. Citro, G. S. Corrado, A. Davis, J. Dean, M. Devin, S. Ghemawat, I. Goodfellow, A. Harp, G. Irving, M. Isard, Y. Jia, R. Jozefowicz, L. Kaiser, M. Kudlur, J. Levenberg, D. Mané, R. Monga, S. Moore, D. Murray, C. Olah, M. Schuster, J. Shlens, B. Steiner, I. Sutskever, K. Talwar, P. Tucker, V. Vanhoucke, V. Vasudevan, F. Viégas, O. Vinyals, P. Warden, M. Wattenberg, M. Wicke, Y. Yu, and X. Zheng, "TensorFlow: Large-scale machine learning on heterogeneous systems." https://www.tensorflow.org/, 2015. Software available from tensorflow.org.

[7] W. Liu, D. Anguelov, D. Erhan, C. Szegedy, S. Reed, C.-Y. Fu, and A. C. Berg, "SSD: Single shot multibox detector," in *European conference on computer vision*, pp. 21–37, Springer, 2016.

[8] "Intel RealSense SDK 2.0 – Intel RealSense Depth and Tracking cameras." https://www.intelrealsense.com/sdk-2/.

[9] "Jetson AGX Xavier Developer Kit." https://developer.nvidia.com/embedded/jetson-agx-xavier-developer-kit.

[10] "Jetson Nano Developer Kit." https://developer.nvidia.com/embedded/jetson-nano-developer-kit.

[11] "Jetson TX2 Developer Kit." https://developer.nvidia.com/embedded/jetson-tx2-developer-kit.