Information and Communication Technologies (ICT) Programme

Project Nº: H2020-ICT-2016-1-732105

D6.2: Planetary Exploration Demonstrator (Final version)

Lead Beneficiary:	TASE
Workpackage:	WP6
Date:	31-12-2019
Distribution - Confidentiality:	Public

Abstract:
This document contains the description of the M36 Planetary Exploration demonstrator. The description is based on the skeleton defined in D6.1 and includes the scope and purpose of the demonstrator, its detailed description and the accomplished results.

Disclaimer

The CERBERO Consortium is the following:

Num.	**Beneficiary name**	**Acronym**	**Country**
1 (Coord.)	IBM Israel – Science and Technology LTD	IBM	IL
2	Università degli Studi di Sassari	UniSS	IT
3	Thales Alenia Space Espana, SA	TASE	ES
4	Università degli Studi di Cagliari	UniCA	IT
5	Institut National des Sciences Appliquees de Rennes	INSA	FR
6	Universidad Politécnica de Madrid	UPM	ES
7	Università della Svizzera Italiana	USI	CH
8	Abinsula SRL	AI	IT
9	Ambiesense LTD	AS	UK
10	Nederlandse Organisatie Voor Toegepast Natuurwetenschappelijk Ondeerzoek TNO	TNO	NL
11	Science and Technology	S&T	NL
12	Centro Ricerche FIAT	CRF	IT

For the CERBERO Consortium, please see the http://cerbero-h2020.eu web-site.

Document Authors

The following list of authors reflects the major contribution to the writing of the document.

Name(s)	Organization Acronym
Pablo Sánchez de Rojas	TASE
Francesca Palumbo	UNISS
Tiziana Fanni	UNISS
Carlo Sau	UNICA
Francesco Regazzoni	USI
Eduardo de la Torre	UPM
Claudio Rubattu	UNISS / INSA
Eduardo Juarez	UPM
Leonardo Suriano	UPM
Daniel Madroñal	UPM

Document Revision History

Date	Ver.	Contributor (Beneficiary)	Summary of main changes
21-10-2019	0.1	TASE	First draft
15-12-2019	0.2	TASE	TASE inputs added
23-12-2019	0.3	UNISS, UPM	UNISS and UPM inputs added
23-12-2019	0.4	TASE	Final integration
13-01-2020	0.5	TASE	Revision and correction

Table of contents

1. Executive Summary

This document provides a detailed description of the final Planetary Exploration demonstrator, being one of the three CERBERO use-cases.

1.1. Structure of Document

This document is organized as follows:

- Section 2 presents the scope and purpose of the demonstrator;
- Section 3 includes the description of the developed demonstrator;
- Section 4 outlines the tests that were performed in order to validate the demonstrator, their results and the feedback obtained;
- Section 5 summarizes the conclusion from the use case leader;
- Section 6 provides the references for the document.

1.2. Related Documents

This document is related to the following CERBERO deliverables:

- D2.1 – Description of Scenarios (Final version)
    - The Planetary Exploration demonstrator will be based on the use case scenario as defined in D2.1
- D2.2 – Technical Requirements (Final version)
    - The development of the demonstrator will contribute to satisfy and validate the requirements listed in D2.2
- D3.1 – Modelling of KPI (Final version)
    - The addressed KPIs are based on the generic list of KPIs as defined in D3.1
- D4.2 – Self Adaptation Manager (Final version)
    - The methodology for CERBERO self-adaptation management as applied in the demonstrator is described in D4.2.
- D5.2 and D5.1 – Framework Components (Final version) and Holistic Methodology and Integration Interfaces (Final version)
    - The framework used for setting up and interfacing CERBERO tools in the (M36) demonstrator are based on the framework components described in D5.2 and their point-to-point integrations described in D5.1.
- D6.1 – Demonstration Skeleton (Final version)
    - The generic skeleton used to build the smart travelling demonstrator is described in D6.1
- D6.8 – Planetary Exploration Demonstration (version I)

- The first version of this document, describing the status of the demonstrator in M18

2. Scope and purpose

This document is the final version of D6.8, where the status of the Self-Healing System for Planetary Exploration demonstrator was originally described. In these lines, the M36 demonstrator of the Planetary Exploration use-case is reported in order to validate the CERBERO tool-chain for deployment in a computational CPS.

The M36 demonstrator provides the second iteration of the demonstrator, where the main functionalities of the M18 demonstrator have been extended and CERBERO tools and technologies have been applied both at the design-time and at run-time.

The drivers of demonstration activities have been defined in D2.2. In the following table an excerpt of Table 4 of D2.2 is provided, where the "planned month" column has been removed for the sake of simplicity. Moreover, an in-depth analysis of goals accomplished in M18 and M36 demonstrators is presented in Section 2 of this document.

Table 1. Requirement validation table

User Requirement	Technical Requirement(s)	Validation Test

PE1. Enable Hardware / Software (HW/SW) co-design for Rad-Tolerant control of robotic arm for planetary exploration using adaptable COTS FPGAs.	**5** (CERBERO framework/technology SHOULD provide multi-viewpoint multi-objective correct-by-construction high-level architecture.) **6** (CERBERO framework/technology SHOULD ensure energy efficient and dependable HW/SW co-design using cross-layer run-time adaptation of reconfigurable HW.) **7** (CERBERO framework/technology SHALL define methodology and SHOULD provide library of reusable functional and non-functional KPIs.) **8** (CERBERO framework/technology SHALL define methodology and SHOULD provide library of reusable energy related components.) **6** (CERBERO framework/technology SHOULD ensure energy efficient and dependable HW/SW co-design using cross-layer run-time adaptation of reconfigurable HW.)	Integrate CERBERO tools and technologies in the Robot Control Unit in order to provide HW/SW co-design.
PE2. Develop integrated “open” toolchain environment for development of robotic arms for space missions with focus on multi-viewpoint system-in-the-loop virtual environment.	**1** (CERBERO framework/technology SHOULD increase the level of abstraction at least by one for HW/SW co-design and for System Level Design. **2** (CERBERO framework/technology SHOULD provide interoperability between cross-layer tools and semantics at the same level of abstraction.)	Use CERBERO tools/technologies for development of robotic arm applications both at design-time and at run-time. CERBERO components can be in principle used stand-alone or combined and this composability is particularly relevant in the space domain. Moreover, all the platform deployment tools as ARTICo3 and MDC are

	4 (CERBERO framework/technology SHOULD provide software and system in-the-loop simulation capabilities for HW/SW co-design.)	compliant with Xilinx development environment, one of the de-facto standards in the Field Programmable Gate Array domain.
PE3. Development of a (self-)adaptation methodology with supporting tools.	**6** (CERBERO framework/technology SHOULD ensure energy efficient and dependable HW/SW co-design using cross-layer run-time adaptation of reconfigurable HW.) **20** (CERBERO framework/technology SHALL provide methodology and tools for development of adaptive application.)	Extensive use of the CERBERO self-adaptation loop and its supporting tools to adapt the computing infrastructures for different purposes (e.g. energy efficiency, throughput, fault robustness, etc).

3. Description of the Planetary Exploration demonstrator

For the use case Planetary Exploration, the goal was to develop a computing platform capable of controlling a robotic arm in the uncertain environment of a space mission. A set of CERBERO tools and technologies are used during both design-time and run-time in order to address these challenges.

The high level CERBERO requirements related to the Planetary Exploration use case are (see Table 2 in D2.2) are:

1. (PE1) Enable HW/SW co-design for rad-tolerant control of robotic arm for planetary exploration using adaptable COTS FPGAs.
2. (PE2) Development of integrated open-source or commercially available toolchain for development of robotic arms for space missions, with focus on multi-viewpoint system-in-the-loop virtual environment.
3. (PE3) Development of a self-adaptation methodology with supporting tools.

Three main goals have been derived from these requirements, defining what was accomplished during the development of M18 and M36 demonstrators (see section 3.2 in D2.1):

1. Fault tolerance to single event effects that may be produced by the impact of subatomic radiation particles in the electronic components.
2. Environment adaptation to the harsh physical environment.
3. Capability of run-time metrics (i.e. power and throughput) evaluation to be able to optimize the the cyber part of the computing platform, by means of a complete infrastructures including monitoring, decision making, and adaptation elements.

The robotic arm part of the final demonstrator for this use case is composed of two main processing elements: the Robot Control Unit (RCU) in charge of performing high level computing and implemented in a Zynq UltraScale+ FPGA; and the Servo Control Unit (SCU) included in each one of the actuator joints that performs functions of torque limit, PID regulation, etc. As stated in D2.1, the final demonstrator will focus on the RCU to provide Adaptive motion Planning and Self-Healing capabilities to the robotic arm.

Up to month 18, HW accelerators in ARTICo^3 were implemented in order to accelerate the execution of the path planning, achieving a fault-tolerant behavior of the algorithm. Besides, different adaptation strategies were highlighted thanks to ARTICo^3 and MDC integration. During M18-M36, a full exploration of the adaptation potential took place.

In month 36, the design tools were adopted to facilitate adaptive fabric deployment. After the exploration of Key Performance Indicators (KPI) run-time trade-offs concluded, the self-adaption loop including SPIDER, PAPIFY, MDC and ARTICo^3 was assessed, providing autonomous complete HW/SW reconfiguration.

As anticipated in D6.1 the PE use case supports different adaptivity types and triggers, dynamically supporting self-awareness through internal execution monitors (see Section 3.2.2), user-commanded adaptation as the operator may chose the tasks and trajectories and environmental awareness, which is currently mimicked through the Graphical User Interface (see Section 3.1.7). Initially, there was the intention to embed also proximity sensors to capture awareness, but in the end the effectiveness of the demonstrator in supporting diversity of adaptivity types was favored with respect to the work on another possible triggers for adaptation. With respect to this latter all types of adaptivity are supported.

- Functional oriented adaptivity is implemented by allowing the switch among two different Inverse Kinematic algorithms (described in Section 3.1.3) according to the execution needs (as reported in Section 3.1.4).
- Non-functional oriented adaptivity allows for performance boost by playing with parallelism of execution (see Section 3.1.5) and for dynamic trade-off executions (see Section 3.2.4).
- Finally, in the PE use case, the computing fabric may be damaged by radiation effects. The selected execution target naturally supports on-demand HW redundancy and exploits HW monitors to check error status and ensure the correct operation of the system (see Section 3.1.6). Nevertheless, in particular conditions (i.e. a solar storm or a sand storms are arriving) also complete configurable logic switch off is also possible, bringing back the computation on the main CPU.

At design-time, basically all the more mature computing level tools are adopted: PREESM, PAPIFY, MDC and ARTICo^3. While at run-time the target executing the robotic arm controller is an heterogeneous Multi Processor System on Chip (MPSoC), fully developed and customized using the above-mentioned CERBERO computing-layer framework components: HW/SW self-adaptation not there by nature, but it is guaranteed thanks to the adoption of the CERBERO tools, which enrich the deployed system with monitoring capabilities, decision making capabilities and efficient support of different types of HW reconfiguration.

The management of this complex computing infrastructure is implemented as part of the SW application and leverages on SPIDER and ARTICo^3 run-time management SW. The complete description of adopted design-time and run-time support is detained in Section 3.2, while a high-level schematic block diagram of the demonstrator is provided in Figure 1.

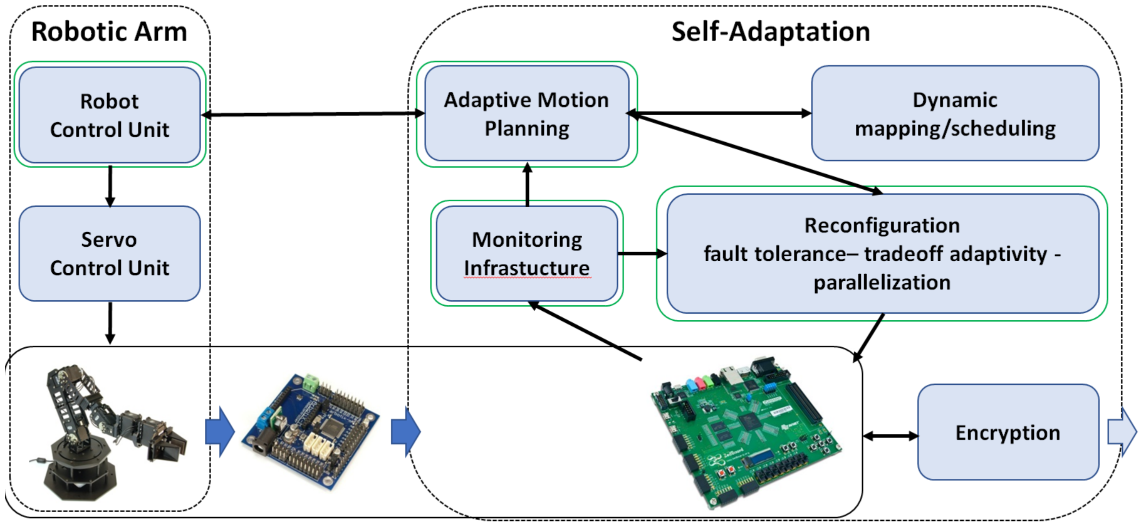


Figure 1. PE demonstrator block diagram

Due to their very high computational complexity, the embeddability of the studied reinforcement learning algorithms was difficult. Standalone tests on the arm were not conclusive enough to justify their integration in the demonstrator.

All the CERBERO use cases are developed according to the CERBERO demonstration skeleton defined in D6.1. The demonstrator components are identified with the skeleton parts, helping also for future incremental developments. A graphical mapping between the use case architecture and the CERBERO skeleton is presented in Figure 2 below:

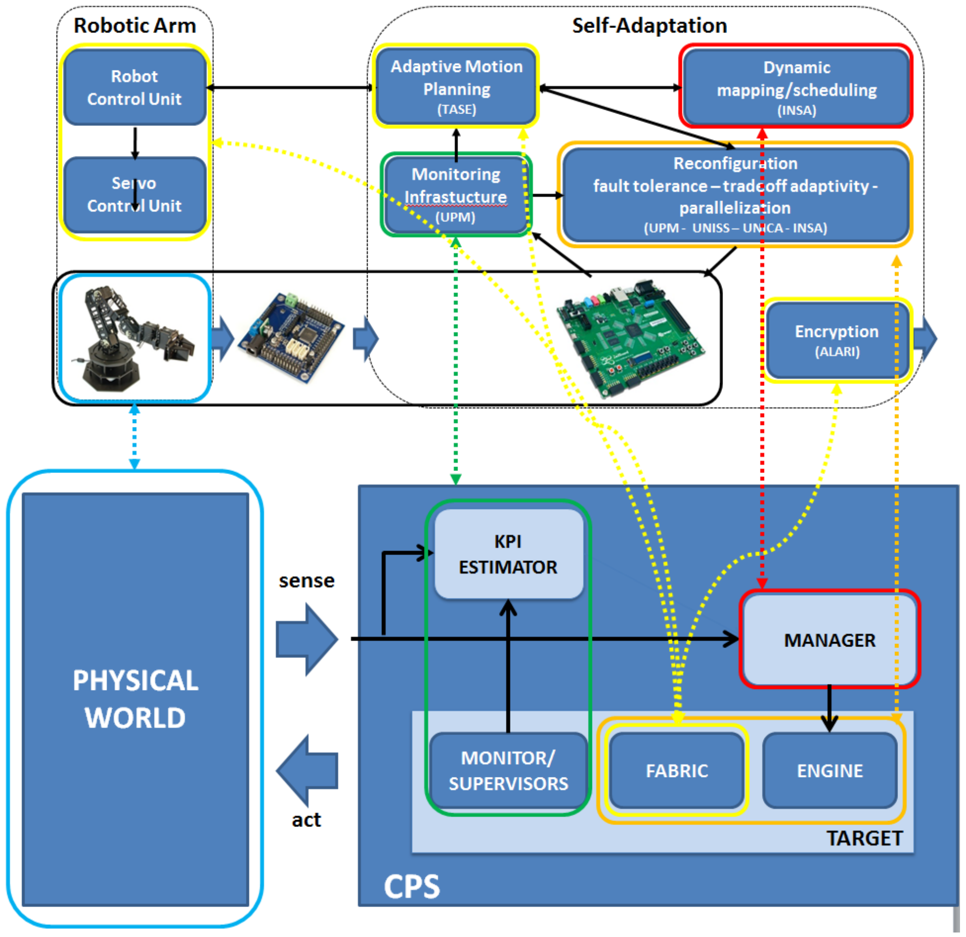

Robotic Arm
Self-Adaptation
Robot Control Unit
Servo Control Unit
Adaptive Motion Planning (TASE)
Dynamic mapping/scheduling (INSA)
Monitoring Infrastucture (UPM)
Reconfiguration fault tolerance – tradeoff adaptivity - parallelization (UPM - UNISS – UNICA - INSA)
Encryption (ALARI)
PHYSICAL WORLD
sense
act
KPI ESTIMATOR
MANAGER
MONITOR/ SUPERVISORS
FABRIC
ENGINE
TARGET
CPS

Figure 2. Planetary Exploration use case - CERBERO skeleton mapping

More details about the implemented functionalities of the demonstrator may be found in the following sections.

3.1. Functionalities

The implementation of the M36 Planetary Exploration demonstrator splits functionalities into the following parts, according to set goals and requirements in D2.2:

1. Physical and energy modeling
2. Trajectory computation and generation
3. Diversity, scalability, parallelization and redundancy
4. HW/SW monitoring

5. User command and simulation
6. Adaptation manager
7. Encryption

In the context of the tasks of the project, the demonstrator involved integration of results from WP3, WP4 and WP5; and was developed within WP6. In the following paragraphs the functionalities of different parts of the demonstrator are described.

3.1.1. *Physical modeling*

The WidowX robot arm that has been used for the validation of the computing platform; it has six degrees of freedom, corresponding to each one of its actuator joints as shown in Figure 2.

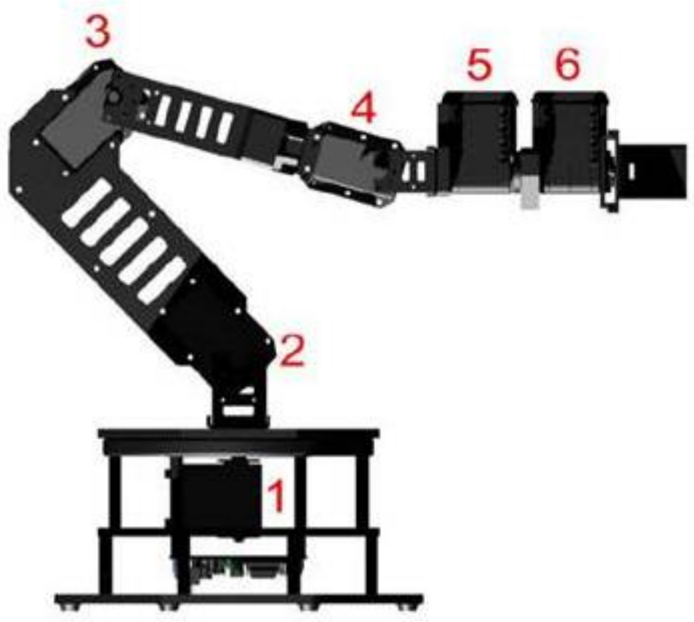


Figure 2. Degrees of freedom of robotic arm

The physical model is built based on the Denavit-Hartenberg parametrization [DH1][DH2], which provides a standard way of attaching reference frames from the physical links of a robot manipulator. The joints are selected depending on the degrees of freedom of the arm and constructional constraints. For the elaboration of the model, five joints are identified:

- Joint 1: Servo 1
- Joint 2: Servo 2
- Joint 3: Elbow (fixed 90º due to arm structure)
- Joint 4: Servo 3
- Joint 5: Servo 4

Since servos 5 and 6 do not affect to the final position of the grip (only to its orientation and to the gripping movement), neither they intervene in the physical model equations, nor they are considered when sending the commands to the arm.

Knowing the z-axis for each frame matches the joint axis, these parameters are defined as:

- J: number of joint, i.e. index;
- r: value of the radius about previous z-axis;
- d: offset of the origin of the reference frame along previous z-axis;
- α: angle between the z-axis of previous and current joint.

A diagram of simple kinematic chain representing these parameters is presented in Figure 3 (note that designation ‘a’ is used instead of ‘d’ in the figure).

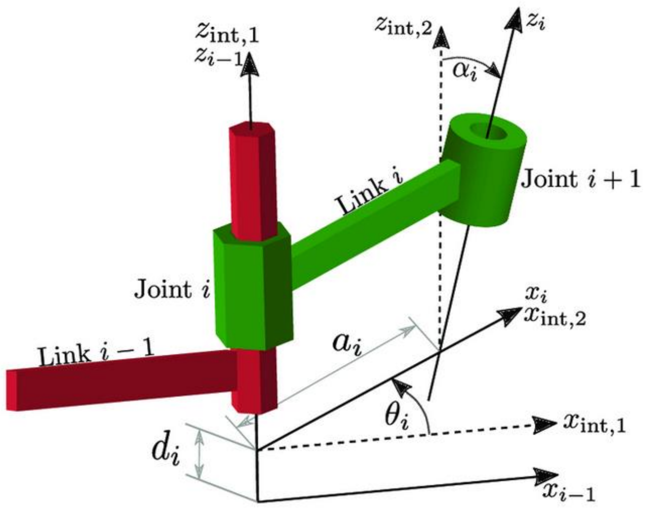


Figure 3. Denavit-Hartenberg parameters for schematic kinematic chain [VALVERDE]

Table 2 shows the Denavit-Hartenberg convention parameters for the model of the robot.

Table 2. Denavit-Hartenberg parameters of the physical model

J_i	r_i	d_i	α_i	Θ_i
1	0	0	π/2	Θ_1
2	15	0	-π	Θ_2
3	5	0	0	Θ_3
4	15	0	0	Θ_4
5	15	0	0	Θ_5

Some of these parameters could have been chosen differently, but after considering the direction of the rotation of each servo, the resulting transformation equations should be equivalent. The correctness of the physical model was ensured by obtaining a 3-D representation of the arm based on calculated parameters. A view of this three-dimensional representation generated by the Peter Corke toolbox [CORKE] is depicted in Figure 4.

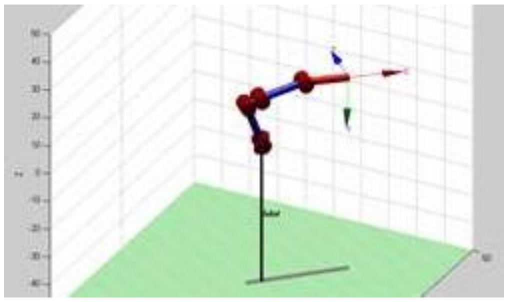


Figure 4. Output of Peter-Corke robotics toolbox

Mathematical equations for this model are too large to be included in this document, but they may be found in the open data files for this use case [PEUCOD].

3.1.2. Energy modeling

In order to implement power consumption optimization, it was necessary to build an energy model of the application. This model is twofold: on one hand it refers to the power consumed during the computation of the cyber part, and on the other hand it must consider the energy consumed by the actuators when moving the physical part to the endpoint.

The energy consumption corresponding to the mechanical movement of the arm is orders of magnitude above the computing power of the platform, but since the last was found to be interesting in order to highlight specific steps in the adaptation loop, it has been also considered.

Moreover, please note that in realistic environment the robotic and the computational parts could be also powered by different set of batteries. Therefore, splitting the models make specific sense to mimic the real flying model deployment.

3.1.2.1 Physical part model

There are several methods for the power consumption optimization on the movement of a robotic arm. These strategies are based on the calculation of the trajectories with the minimum effort, the minimum torque rate, or the minimum mechanical energy among other optimization functions [CARABIN].

For this purpose, the mechanical energy approach has been adopted, being widely explore in literature and found compatible with different optimization strategies [BAILON][SENGUPTA][MOHAMMED]. The mathematical expression of the consumption function for each actuator is:

$$P_i = \int_0^T \dot{\theta}(t)\tau(t)dt$$

The following considerations may be applied:

- The robot is going to work under zero (or very low) gravity;
- The robot is going to work in vacuum or very thin atmosphere;
- Rotation speed of servomotors is constant over time (we will command them accordingly).

Under the previous assumptions, being torque constant over time at a given rotation speed, the consumption equation can be expressed as:

$$P_i = k_i \cdot \Delta\theta_i$$

Therefore, the energy consumed by every servo after a given movement is directly proportional to its rotation angle. By applying the principle of superposition, the total energy consumption for the physical part will be the summation of power consumed by every actuator, i.e. directly proportional to the summation of all rotation angles.

$$P = \sum_{i=1}^{6} P_i = \sum_{i=1}^{6} k_i \cdot \Delta\theta_i = K \cdot \sum_{i=1}^{6} \Delta\theta_i$$

The previous equation represents final mathematical expression for the simplified energy model. The constant of proportionality is found through calibration of the power source setup as depend on the desired units of measurement.

Extensive power monitoring tests has been performed in the lab in order to achieve a first validation of the correctness of the model. Since it targets a low gravity, thin atmosphere work environment, the model prediction could not be directly compared to the power measurements: it was necessary to scale these measurements of different trajectories (for and against gravity). Further experiments replicating these conditions correspond to more advance stages of space missions and would be very costly to design and produce.

3.1.2.2 Cyber part model

The cyber part, running all diverse trajectory planning algorithms, should account on the SW running on the R5 cores operating in lock-step mode, for reliability reasons, and so, running as in an equivalent single-core mode. For the HW version, directly mapped HW accelerators (such as the Nelder Mead one) or based on MDC (such as the Dumped Least Squares one) rely on the ARTICo^3 infrastructure.

The behavior of the ARTICo^3 infrastructure in terms of additional power consumption is, in general, characterized by three factors:

- There is a quite linear increment of power consumption with an increasing number of accelerators.
- However, if good performance scalability is obtained, the computation of energy (power along time) gives better energy utilization when the highest number of accelerators are used.

- Under the previous conditions, scalability is driven by the computation. However, if the bus gets saturated, increasing the number of used accelerators just increases power and energy since there are no more possible time reduction.
- Power consumption in double redundancy and in triple redundancy increase linearly. Two accelerators in DMR consume slightly less than two accelerators operating with different data.

Figure 5 shows a typical design space of solutions provided by ARTICo^3. Continuous lines are actual measurements while dotted lines are projections with the model.

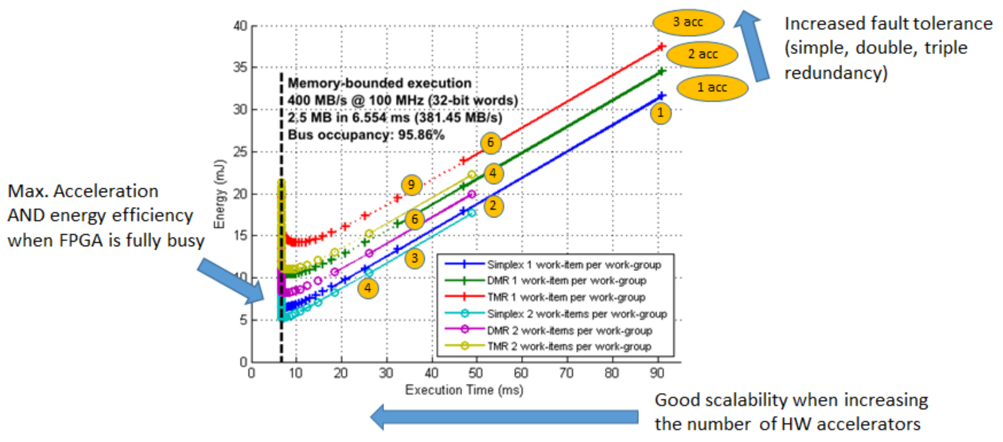


Figure 5. Design Space and Energy model for ARTICo^3 infrastructure

As it can be seen, all the previously mentioned effects can be observed, and the model may be built by the measurement of the power used for one slot, and taking into account memory transfers and execution times, verify memory-bounded or computing-bounded execution.

The results section shows some measurements that show the behavior with the Inverse Kinematic problem. The adaptation manager could work with look-up tables of consumption based on such results, although actual power measurements will be used also.

3.1.3. Trajectory Generation

Any robotic manipulator is composed, as depicted in Figure 6, of different parts, namely: 1) a base; 2) rigid links; 3) joints (each of them connecting two adjacent links); and 4) an end effector.

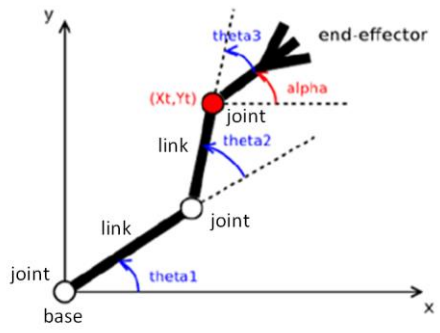


Figure 6 – Generalized robotic manipulator.

As you can see in Figure 6, physically an angle can be associated to each rotational joint, or a displacement for a prismatic joint, and the end effector position can be determined by:

- computing the spatial coordinates of the end effector from all the joint angles, resolving a *Forward Kinematics* (FK) problem.
- computing the joint angles from a desired end-effector spatial coordinate, resolving an *Inverse Kinematics* (IK) problem.

Indeed, implementing an arm controller we had to work on IK problems to derive the angles of the joints starting from a given desired position (or set of positions). Solutions of IK problems are not straightforward, since:

- more than one solution can be found for a desired end-effector position, for example elbow-up or elbow-down poses could both lead to reach the same spatial coordinate;
- there could be joint angles limitations; and finally,
- out-of-reach end-effector spatial coordinates could be experienced.

In the CERBERO project, despite different algorithms are available at the state of the art, TASE adopted as a baseline the Nelder-Mead one (see Section 3.1.3.2). They also provided the definition of an operational workspace (see Section 3.1.3.1).

In general, many numerical and non-numerical solutions are available at the state of the art [Aristidou 2017]. Numerical methods, the category Nelder-Mead belong to, require various iterations to converge over a solution, but are better capable of dealing with Degrees of Freedom, and multiple end effectors (e.g. fingers of a hand or arms of a body) with respect to analytic solutions and data-driven methods. Among the numerical solutions we can list: Heuristic methods and Cyclic Coordinate Descendant ones [Wright 2012]; Newton-based methods (exploiting second-order Taylor expansion), and Jacobian-based methods (exploiting inverted, pseudo-inverted and transposed Jacobian matrices) [Buss 2009].

In the CERBERO project, to prove the possibility of effective deployment of functional adaptive infrastructure, beside the Nelder-Mead also a second numerical algorithm known as Damped Least Square (DLS) has been implemented (see Section 3.1.3.3).

3.1.3.1 Workspace and Trajectory Generator

In robotics, many times it is interesting to know what is the set of locations that can be reached. This is what is called the “workspace” of the robot and depends on constructional parameters of the robot itself, but also on the end-effector mounted. The workspace for a six-axis robotic arm is a six-dimensional space that consists of all possible combination of values for the six degrees of freedom of the arm, i.e. the complete set of positions and orientations of the robot.

Also, singularities are a very important phenomenon that limit the usable parts of the workspace. A singularity is a position where IK fail, and the robot end-effector becomes blocked in certain directions. Moreover, passing close to a singularity results in high joint velocities which may be unexpected and may be dangerous for the robot itself and its surroundings. Main singularities have been identified (wrist, elbow and shoulder singularity) for wrist-partitioned, vertically articulated robotic arms, but these occurrences can only be avoided through the appropriate design of the whole robot cell.

Typically, the workspace is a combination of several singularity-free workspace subsets. For this use case this analysis has not been performed since it is outside the scope of the project. A “safe zone” has been specified instead in cylindrical coordinates, where the robot can be moved without the risk of encountering a singularity if specific guidelines are followed. The 3D representation of this safe zone is depicted in Figure 7:

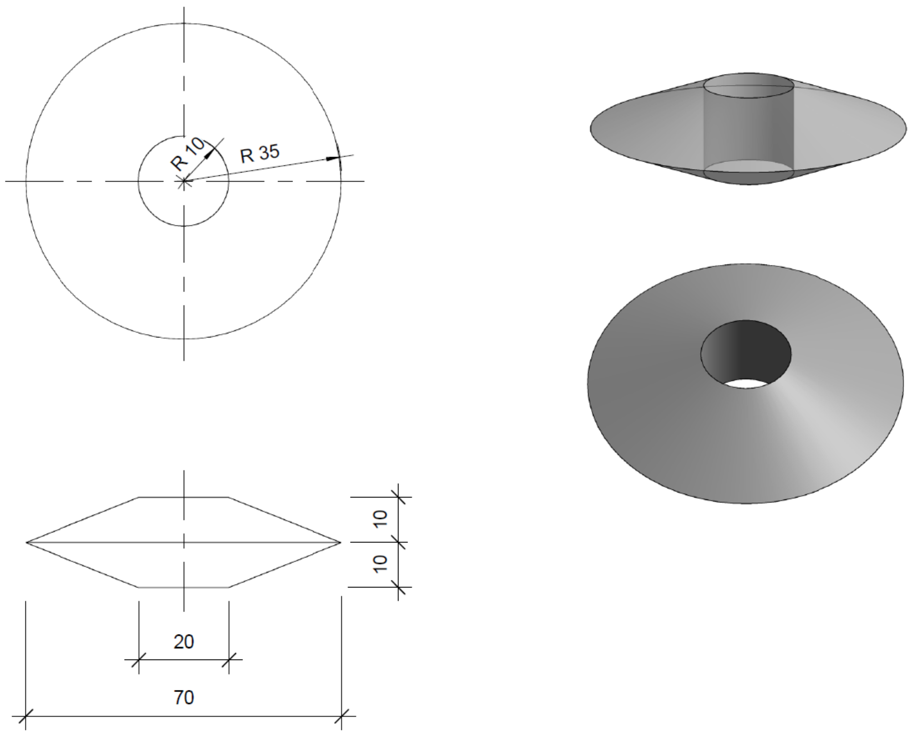


Figure 7. Three-dimensional view of the safe zone of the robot (dimensions in cm.)

The description of this safe zone, mathematical equations and guidelines for the movements are included as open data [PEUCOD].

According to the workspace definition provided by TASE, UNISS has derived a trajectory generator available as open data [TRJ]. The reference trajectory set comprises 100 different trajectories, sorted by length, and available for comparisons in the trajectories.csv file [TRJ].

Within the given workspace different possible tasks, with different possible requirements and characteristics, have been considered.

- Task 1: Retrieval of rock sample from planetary surface of Mars
  - Approximate distance of trajectories: 50 cm
  - Required positioning accuracy: 10 mm
- Task 2: Manipulation of sample containment
  - Approximate distance of trajectories: 10 cm
  - Required positioning accuracy: 5 mm

3.1.3.2 Nelder-Mead optimization

The first motion planning algorithm that was implemented is based on the Nelder-Mead optimization method [NM]. It is often used in non-linear searches of the minimum or maximum of a cost function since it is a direct search method that does not need the calculation of the function derivatives.

The main drawback of this method is its convergence is not mathematically guaranteed, and the optimization process could lead to non-stationary points. However, after extensive engineering testing the adopted NM based solution has not given convergence problems, being capable of always finding a solution in a low number of iterations and proven itself to be a valuable tool for the generation of trajectories.

The optimization process looks for a local optimum of a multidimensional problem, finding the approximate solution when the cost function varies smoothly. The accuracy of the searching process can be customized impacting on the number of iterations, but given a required accuracy the length of the iterative process cannot be known in advance.

This mechanism starts with a 6-dimensional simplex: in this scenario, each dimension represents a degree of freedom of the arm. The centroid of the starting simplex is determined by the joint angles of the starting position (which are known).

Then, a series of transformations are applied to this simplex as seen in Figure 8:

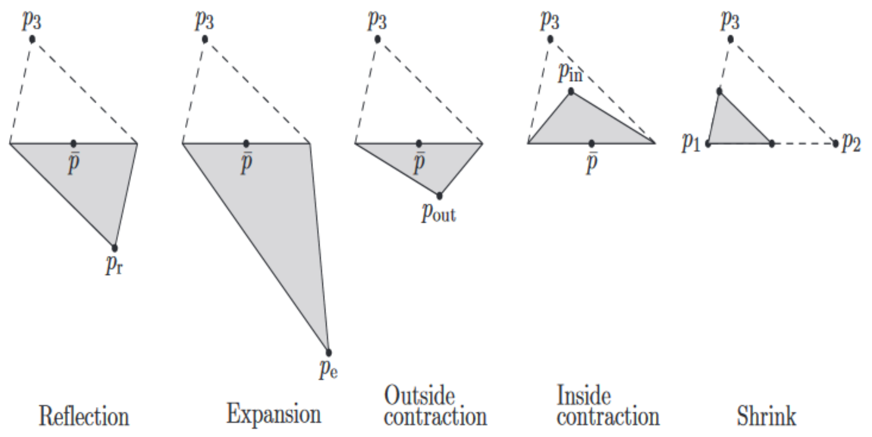


Figure 8. Simplex transformations during the Nelder-Mead optimization process [Wright 2012]

The cost function is evaluated in all vertices and then, a transformation in the simplex is produced. Depending on the values of the simplex, it will experience expansions, contractions or reflections in order to narrow the search more and more after each iteration.

A contingency mechanism is implemented, in such a way that if the solution is not found in a number of iterations, the initial simplex is recalculated with random values generated around the starting angles, and the process starts over.

In this scenario, the cost function is calculated as follows: first, the forward kinematics equations are applied to the vertex in order to obtain its corresponding Cartesian coordinates. Then, the least squares fitting between this point and the target point is calculated. In this step, the constraint of canceling the z coordinate of two last joints is added in order to keep the last section of the arm straight and avoid weird positions during the movement. When the cost function value is smaller than the required accuracy, the search process concludes and the angles of the joints for the final position are provided.

This process will need to be repeated a number of times until the final position is reached since, in order to avoid jerky movements and increase its smoothness, the algorithm works with a list of intermediate points that can be generated through linear interpolation or any other method.

3.1.3.3 Damped Least Squares Method

To guarantee algorithmic diversity, UNISS has studied and implemented, together with UNICA, another trajectory IK solver, the Damped Least Squares (DLS). The DLS approach belong to the algorithms in the Jacobian-based methods category. All of them, as the name says, are based on the Jacobian matrix [Lewis 2003] that is defined as follow:

$$f\colon \Omega \subseteq \mathbb{R}^n \to \mathbb{R}^m \quad J_f(x) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(x) & \cdots & \frac{\partial f_1}{\partial x_n}(x) \\ \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(x) & \cdots & \frac{\partial f_m}{\partial x_n}(x) \end{pmatrix} \in \mathbb{R}^{m,n}$$

The Jacobian matrix represents a transformation between two time derivative-related spaces: the Cartesian Space and the Velocity Space. This latter depends on the Velocity Joints Space according to the equation:

$$\Delta\vec{\theta} = J^{+}\Delta\vec{e}$$

where $\Delta\vec{\theta}$ is the joint space vector, $\Delta\vec{e}$ is the error or displacement vector and J^{+} is a matrix computed from the Jacobian matrix *J*. In the paper [Fanni L. 2019] more details are provided on how to calculate all these terms.

In the following we briefly focus just on J^{+} that is computed, according to the DLS algorithm proposed in [Buss 2015], as:

$$J^{+} = \left(J^T J + \lambda^2 I\right)^{-1} J^T =$$

and, by substituting this expression into the $\Delta\vec{\theta}$ one, we obtain:

$$\Delta\vec{\theta} = (J^T J + \lambda^2 I)^{-1} J^T \Delta\vec{e}$$

The dumping factor, lambda, in the proposed implementation is a static parameter (fixed to 0.5) that is used, more in general, to handle singularities in the workspace.

To compute the entire trajectory according to this methodology, the steps are summarized in Figure 6. This process is clearly iterative: the new angles are obtained according to the current ones $\theta_{new} = \theta_{old} + \Delta\vec{\theta}$.

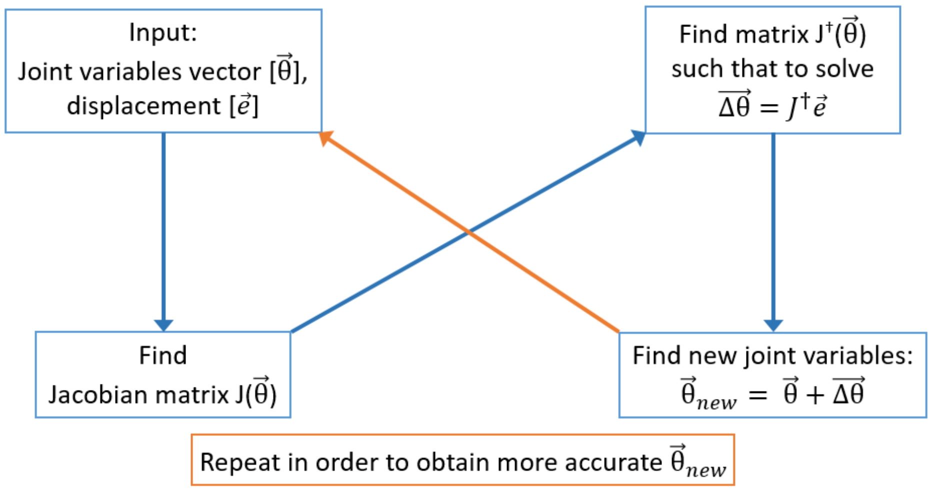

Input:
Joint variables vector [$\vec{\theta}$], displacement [$\vec{e}$]
Find matrix J†($\vec{\theta}$) such that to solve $\overline{\Delta\theta} = J^{\dagger}\vec{e}$
Find Jacobian matrix J($\vec{\theta}$)
Find new joint variables: $\vec{\theta}_{new} = \vec{\theta} + \overline{\Delta\theta}$
Repeat in order to obtain more accurate $\vec{\theta}_{new}$

Figure 6. Simplex transformations during the Nelder-Mead optimization process [WRIGHT]

Please note that the process shown in Figure 6 is normally iterated several times. The number of iterations, *N*, is a parameter of the DLS algorithm. This parameter is directly proportional to the accuracy: the larger is N, the smaller will be the error among the desired (the pedix *Th* in the formula below, standing for theoretical desired point) and computed/reached (the pedix Cl in the formula below, standing for classical DLS) end-effector spatial coordinate. The final error point is computed as:

$$\Delta_{err-Cl-Th} = \sqrt{[(x_{Cl} - x_{Th})^2 + (y_{Cl} - y_{Th})^2 + (z_{Cl} - z_{Th})^2]}$$

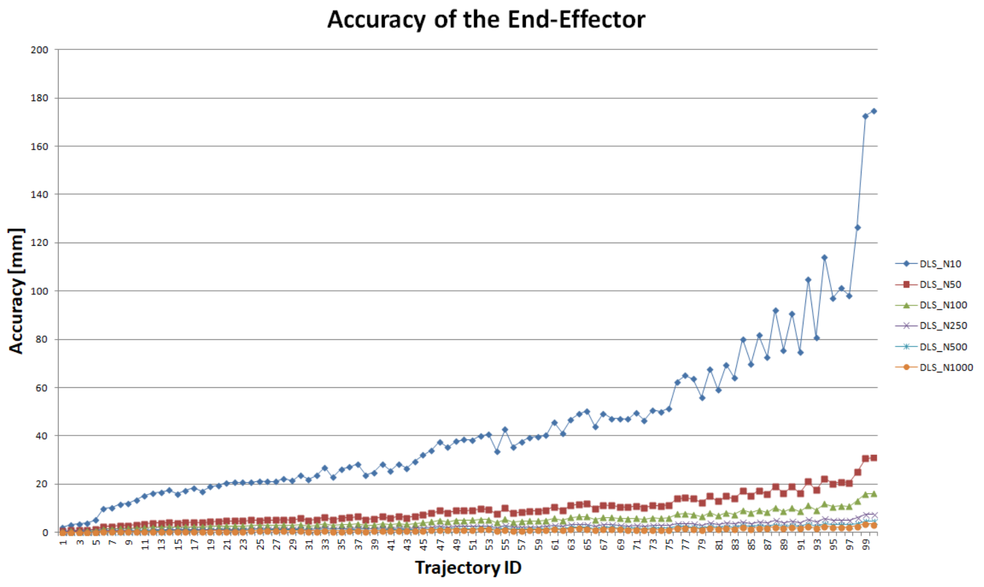
An important design decision in the DSL algorithm regards the value of N to be chosen. To empirically determine such value, we have analyzed (prior to the implementation) the behavior of the DLS for variable values of N. This analysis is reported in Figure 9.

Regardless of the *N* value, the error trend is growing in general with the trajectory length. Therefore, longer trajectories may turn out to be critical with respect to the accuracy when *N* is fixed. At the same time, the error decreases by increasing the number of iterations *N* to perform the job over the same trajectory. Let’s assume a tolerance fixed to 0.5 mm:

- for *N=10*, only the first short trajectories can satisfy the tolerance;
- by increasing *N* from 50 to 100, more trajectories can maintain the error below the tolerance.
- setting *N* from 500 to 1000, all the trajectories can meet the tolerance.

From our empirical analysis in the given scenario, we found out that there is a relationship between the density of points per trajectory, defined as the total length of the calculated trajectory divided by the number of iterations to execute it, and the tolerance that can be guaranteed. Our empirical analysis on the reference set of 100 trajectories, derived as described in Section 3.1.3, proved that given a tolerance of 5 mm, 8 points per centimeter should be computed to satisfy the given constraint; while relaxing the tolerance constraint to 10 mm, just 4 points per centimeter are sufficient.

All the data that permit to verify these assumptions and to reproduce our analysis are available as open data [DLS].



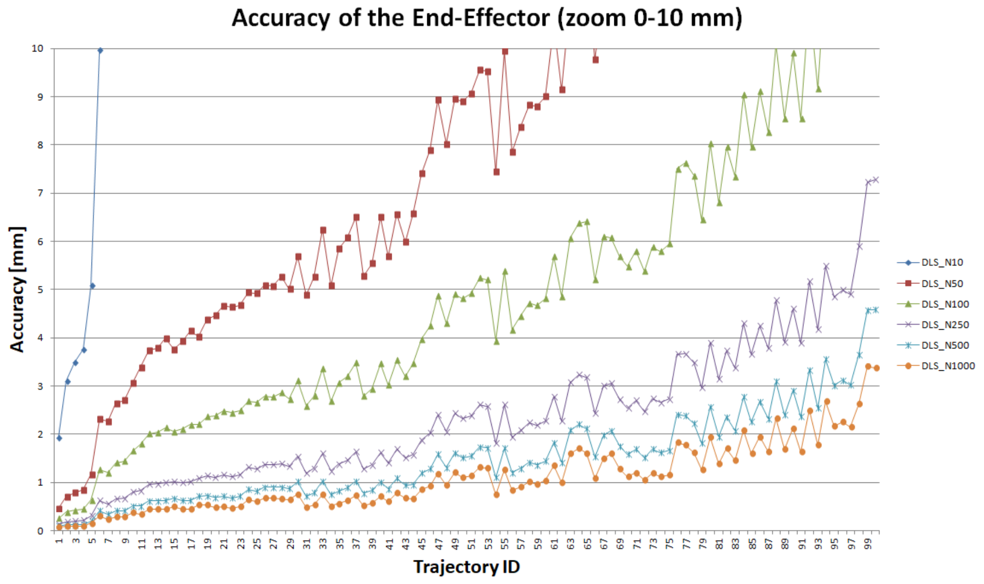


Figure 9. Accuracy of the end-effector position: (top) variation with the number of iterations N in the DLS algorithm and (bottom) zoom in of the behavior in the target accuracy range.

3.1.4. Diversity and scalability

The work done in WP4 proved that on a self-adaptive infrastructure different types of adaptivity can be supported by a self-adaptive system. On the demonstrator we were able to prove that the outcome and integrations carried out in WP4 and WP5 facilitate the possibility of supporting various types of execution, based on SW and on HW, as well as providing adaptive diversity, scalability and fault tolerance leveraging on the combined benefits of mixed-grained reconfiguration support offered by ARTICo^3 and MDC [FANNI 2018, RODRIGUEZ 2018].

Diversity in terms of HW and SW execution is a good fault tolerance mechanism, because it offers an ultimate solution in the case of common mode failures that affect completely the execution on a given computing fabric. For instance, if a fault on the FPGA part, or in a power supply that is used only by this one, HW execution would be stopped and the execution would be moved to SW-only execution.

Algorithmic diversity has also been used, by implementing the two different trajectory IK solvers described in the previous section: the Nelder-Mead (NM) and the Damped Least Squares (DLS).

- The drawback of NM algorithm is that the computation time is not predictable, given the fact that the number of iterations of the simplex algorithms is dependent on the specified accuracy. i.e., the distance between the specified point (the input to the algorithm) and the proposed one (the cartesian point obtained by the proposed set of absolute angles) should be kept under a specified limit.
- In DLS accuracy is only given by the number of segments the trajectory is split in, which in turn depend on the length of the same trajectory. Thus, for a given trajectory and for a fixed accuracy, the number of steps required to the DLS is known. Being a numerical solver, the computation time of a single run of the algorithm is constant, meaning that, once defined desired accuracy and trajectory length, the whole DLS computation time is predictable.

CERBERO self-adaptation loop is capable of mastering different types of adaptation, including non-functional ones. As it will be discussed below in Section 3.2, besides the algorithmic implication of offering predictable computation time, useful in critical tasks, the DLS has also been modeled in order to exploit a further degree of adaptivity offered by the CERBERO framework. In fact, the HW version of DLS has been designed with MDC, which enables intra-slot low overhead adaptivity. Two different variants of the HW version of DLS have been modeled: the baseline (BL) and the high performance (HP) one. The two variants provide different trade-offs in terms of computing time and power consumption:

- BL is slower but less power consuming,
- HP is faster but more power hungry.

In such a way, according to the current state of the system in terms of power budget and/or level of criticality, it is possible to switch between BL and HP variants in an extremely lightweight manner, avoiding ARTICo^3 complete reconfiguration of the whole slot.

To summarize, the algorithmic diversity is useful for different quality and performance solutions. While guaranteed and exact precision movement[1] is to be solved by NM,

[1] NM algorithm, when converging, is capable to meet exactly the given accuracy (provided as input), while in the DLS case accuracy depends on the chosen number of iterations (that are the intermediate point of the trajectory), empirically determined on the basis of the length of the trajectory. In the second case, we DLS prove to be able to stay within a range, but not to provide a certain accuracy number.

guaranteed timing[2] execution is achieved by DLS. Diverse fabric execution and diverse algorithmic solving offer four different execution possibilities:

- NM execution of SW (single core)
- DLS execution on SW (single core)
- NM execution using ARTICo^3 accelerators
- DLS execution using ARTICo^3 accelerators, with additional MDC adaptivity.

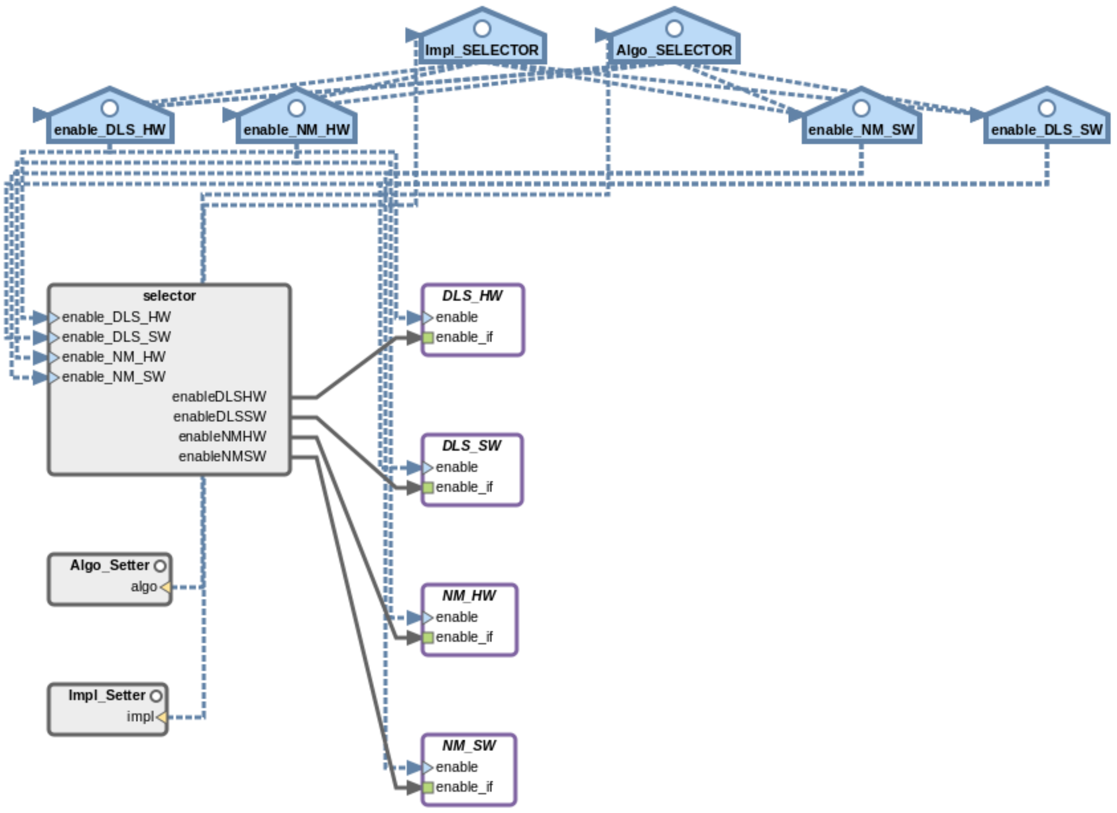


Figure 10. Diversity on the execution of the reverse kinematics, observable in the PiSDF graph.

Note: The four versions of the algorithm produce a result that is set to a file. This file is then sent to one of the two following possibilities:

- To the real robotic arm in order to make the appropriate servo operation
- To the Spyder graphic simulator of the robotic arm to check the behavior when the node is available.

[2] The number of iterations of the DLS is provided as input, therefore its execution time is known. On the contrary, NM one is not since simplex optimizations are carried out until convergence is reached.

3.1.5. Parallelization and adaptable redundancy

In the demonstrator, different levels of parallelization turned out to be useful:

- Acceleration through instantiation of multiple cost-function cores (related to previous section)
- Direct parallelization of the algorithm (computing new points without the calculation of immediate previous ones)

The scalability possibilities offered by ARTICo^3 allow different modes of execution, all for both NM and DLS:

- Execution using one accelerator
- Execution in scalability mode using two/three/four accelerators
- Execution in DMR mode (dual redundancy) with two accelerators in redundant mode
- Execution in DMR mode with two sets of redundant accelerators with an equivalent performance of two accelerators (using four)
- Execution in TMR mode, using three accelerators with voting for fault masking.

The following listing shows an example on how to invoke different potential parallelization levels, as an example of the functions to be used by the adaptation manager to easily allow for adaptation with no deep knowledge of the underlying functions.

```
void changeNumberSlots(int nbSlots){
      //Release previous instances of a3 kernels
      artico3_kernel_release("costfunction");
      //create new SW struct for a3 kernel
      artico3_kernel_create("costfunction", 512, 8, 1);
      for(int i = 0; I < nbSlots; i++){
         // load nBSlots artico bitstreams
         artico3_load("costfunction", i, 0, 0, 0);
      }
      // set all registers to zero and set internal state to reset
      artico3_kernel_reset("costfunction"); //activate reset signal
      // define variable a3data_t with values to be written in the
      // configuration registers. in this case, simplexPerAccelInit is
      // 10 (simplex to be computed by every accelerator
      a3data_t wcfg[8] = {simplexPerAccelInit, simplexPerAccelInit,
                          simplexPerAccelInit, simplexPerAccelInit,
                          simplexPerAccelInit, simplexPerAccelInit,
                          simplexPerAccelInit, simplexPerAccelInit};
      // write configuration data into configuration registers
      artico3_kernel_wcfg("costfunction", 0,wcfg);
}
```

The code that runs in parallel against several accelerators is described below. It is not intended to give a detailed operation of the execution, but to show the different elements that make up the code:

- Data allocation for memory exchanges (input parameters and output parameters)

- A loop that, for every kernel invocation (in charge of execution several points in parallel each), performs:
  - Preparation of input data
  - Invocations (in a loop) of the subsequent sub-trajectories that are sent for the complete trajectory computation
  - Memory disposal (free)
- Memory free

```
void cost_fun_HW_Parallel(int nbParameters, int simplexPerAcc, int
ParallelPoints, simplexPlus_t *inputs) {
//printf("cost_fun_HW_Parallel\n");
     artico3_kernel_reset("costfunction");
     a3data_t wcfg[8] =
{simplexPerAcc,simplexPerAcc,simplexPerAcc,simplexPerAcc,simplexPerAcc,
simplexPerAcc,simplexPerAcc,simplexPerAcc};
     artico3_kernel_wcfg("costfunction", 0, wcfg);

     int GlobalValues = simplexPerAcc * ParallelPoints;

      x_0 = artico3_alloc(GlobalValues * sizeof *x_0, "costfunction",
"port0", A3_P_I);
      x_1 = artico3_alloc(GlobalValues * sizeof *x_1, "costfunction",
"port1", A3_P_I);
      x_3 = artico3_alloc(GlobalValues * sizeof *x_3, "costfunction",
"port2", A3_P_I);
      x_4 = artico3_alloc(GlobalValues * sizeof *x_4, "costfunction",
"port3", A3_P_I);

      pos_final_1 = artico3_alloc(GlobalValues * sizeof *pos_final_1,
"costfunction", "port4", A3_P_I);
      pos_final_2 = artico3_alloc(GlobalValues * sizeof *pos_final_2,
"costfunction", "port5", A3_P_I);
      pos_final_3 = artico3_alloc(GlobalValues * sizeof *pos_final_3,
"costfunction", "port6", A3_P_I);

      cost = artico3_alloc(GlobalValues * sizeof *cost, "costfunction",
"port7", A3_P_O);

     //filling buffers
     unsigned int simplexPlusID = 0;
     for (simplexPlusID = 0; simplexPlusID < ParallelPoints ;
simplexPlusID++){
      copyAccInputSimplex(nbParameters, simplexPlusID, simplexPerAcc,
inputs);
     }

     artico3_kernel_execute("costfunction",       simplexPerAcc       *
ParallelPoints /*gsize*/, simplexPerAcc/*lsize*/);
     artico3_kernel_wait("costfunction");

     for(simplexPlusID = 0; simplexPlusID < ParallelPoints ;
simplexPlusID++){
      //copyAccOutputSimplex(int nbParameters, int simplexPlusID, int
simplexPerAcc, simplexPlus_t *inputs)
```

```
      copyAccOutputSimplex(nbParameters, simplexPlusID, simplexPerAcc, 
inputs);
     }
     artico3_free("costfunction","port0");
     artico3_free("costfunction","port1");
     artico3_free("costfunction","port2");
     artico3_free("costfunction","port3");
     artico3_free("costfunction","port4");
     artico3_free("costfunction","port5");
     artico3_free("costfunction","port6");
     artico3_free("costfunction","port7");

     return;
}
```

As it may be observed, the procedure is quite similar to the invocation of GPU kernels. The ARTICo^3 integrated with PREESM includes a function to include these calls automatically on architectures that account on ARTICo^3 execution.

Same as for different parallelization modes shown before, redundant modes of operation are set by specifying the mode of operation of a set of kernels/slots. These modes are selectable by the adaptation manager by making use of simple functions available in the ARTICo^3 run-time library.

3.1.6. HW/SW monitoring

The CERBERO toolchain offers also monitoring capabilities support, which has been exploited in the project in different manners. The positive aspects is that PAPIFY, which is based on the well-known and established PAPI library, masks the access to the monitors, allowing to treat them in the same way despite what is monitored (KPI under observation) and where (HW or SW).

Instrumentation of the PiSDF for performance monitoring is done by using the PAPIFY tool embedded in PREESM. The PAPIFY-VIEWER tool is employed to represent the collected performance monitoring information. The work achieved along the project allows to monitor both SW and HW components.

- Specifically, at SW side, actor execution time per iteration is monitored as a performance monitoring event to be provided to the Adaptation Manager.
- At HW side, execution time, error events and power measurements per actor iteration are monitored and fed to the Adaptation Manager.

3.1.6.1 Radiation failures monitoring

When operating in redundant modes, the voter/comparator in the fixed infrastructure of ARTICo^3 accounts with error counters that are incremented in every word transaction when resulting data are read from the accelerators, before sending it to the main memory.

Injection of faults use a simple model where a specified accelerator is set to operate under faulty conditions. Whenever a given accelerator is set to be faulty, transactions with the results are all set to ‘all ones’. If the mode of operation is in either DMR or TMR, the associated fault registers start to increase. These registers, accessible via the corresponding ARTICo^3 PAPI component, may be read.

Whenever a given accelerator is identified to be faulty, the corresponding bitstream may be rewritten. This improves repair times with respect to conventional scrubbers, since the reconfiguration time for a given slot is approximately 7 times smaller than the complete bitstream (determined by the ratio of approx. 26 MB for the complete bitstream to 3-4 MB of a partial bitstream).

3.1.7. User command and simulation

The Planetary Exploration demonstrator intends to recreate a number of operating conditions distinctive from the scenario that are hardly replicable on the Earth surface. In order to overcome this issue, a Graphic User Interface has been developed. It provides an appealing, intuitive and easy-to-use environment to command the arm.

Commands are sent by serial protocol to the computing platform, where are decoded.

The main functionalities implemented in this GUI include:

- Set end-point in Cartesian coordinates (x, y, z)
- Set end and intermediate points in Cartesian coordinates
- Launch execution of pre-set trajectories
- Trigger environmental events (start and end solar storm or sand storm)
- Alternate between two different scenarios: sample retrieving and container manipulation, with different requirements of end-effector positioning accuracy (see section 3.1.3.1).

The graphic appearance of the GUI in shown in Figure 11:

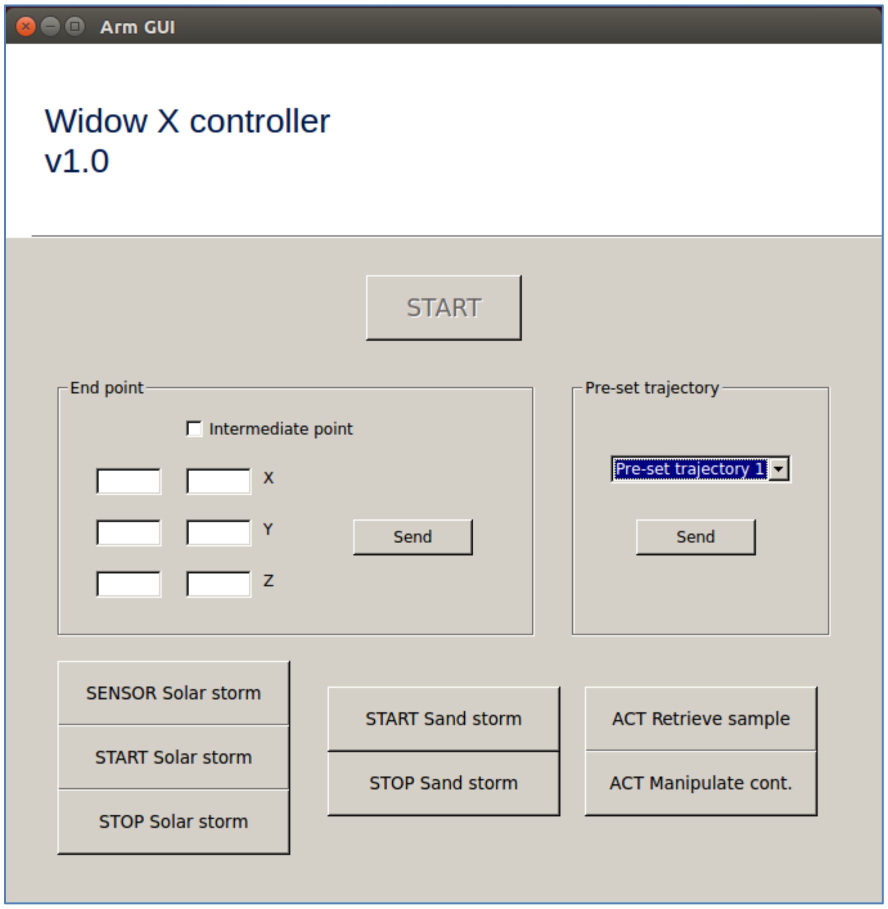


Figure 11. Planetary Exploration demonstrator GUI interface

While setting a target end-point directly commands the computer platform to initiate the motion planning algorithm, configuring a pre-set trajectory will switch between the different internal CSV files specifying the coordinates of the trajectory. The same occurs when alternating between scenarios: it will change the accuracy requirement, stored in an internal file read during the execution of the algorithms. Last, the commands that indicate environmental events are received by the adaption manager that will trigger adaption accordingly.

The GUI has been developed in Python 2.7 and has been tested under Python Spyder. This framework has been used to develop a Motion Simulator, as well in order to validate the calculated trajectories without having to connect the arm every time.

This simulator application receives the angles calculated by the computation platform and draws a schematic representation of the movement of the arm. It considers two different scenarios, one in free space and the other with an obstacle placed in a known position. The simulation of the arm avoiding an obstacle is displayed in Figure 12.

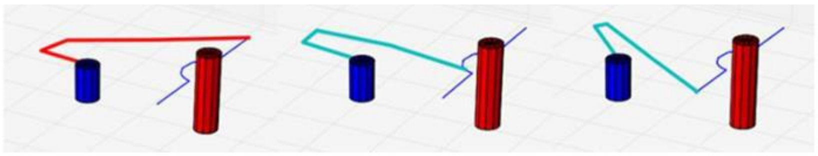


Figure 12. Sequential captures on Motion Simulator

The files for the graphic user interface, a document explaining the implemented communication protocol and the movement simulator are provided as open data [PEUCOD].

3.1.8. Encryption accelerator

Despite not being the main focus of this use case, encryption functionalities could be useful in certain aerospace applications and in future phases of the development of the planetary exploration robots. To be ready for these future needs, we designed and independently tested a dedicated robust accelerator implementing authenticated encryption.

Robustness here is needed to reliable operating in a harsh environment, thus we do not need to address challenges related with the presence of a smart adversary voluntary injecting fault to mount an attack for extracting secret information.

The accelerator implements the AES algorithm used in Galois/Counter Mode, with a MAC size of 128 bit, being this the configuration is currently widely used in aerospace [CCSDS].

The accelerator includes error detection and correction capabilities. The choice to include these capabilities directly in the module instead of using the one provided by ARTICo^3 comes from the fact that robustness of the AES algorithm and the development of optimal error detection and correction codes for it is a problem that has been widely studied in literature and the dedicated design previously proposed have been shown to achieve sufficient robustness at minimal cost. These designs are thus a natural solution for the particular problem of ensuring robustness of the AES algorithm. The selection of the amount of redundancy (and, as direct consequence, the provided robustness) can be configured by the user by means of a dedicated input port. Being designed to be a library component for future use, the accelerator has an interface that allows an easy integration with different types of components.

3.2. Integrated and evolved tools

The PE Demonstrator, and all its functionalities described in the previous section, has been designed and managed using the tools at the computational level of CERBERO environment. In particular Fig. Figure 13 illustrates tools adopted at design-time (on the left) and at run-time (on the right).

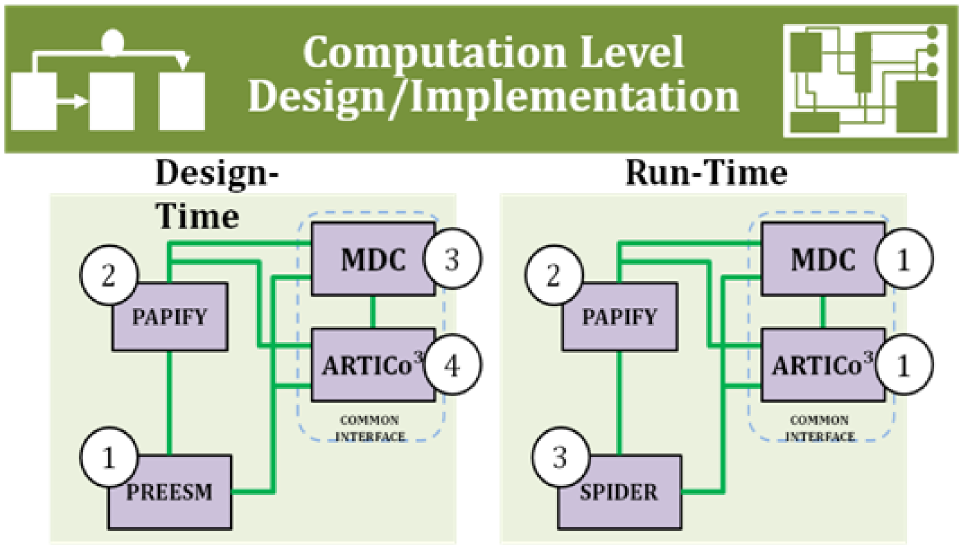


Figure 13. CERBERO tools used by PE demonstrator

Design-Time Support

At the design-time, the application is modeled adopting PREESM that allows to early stage analysis and facilitate the definition of the partitioning of the application among the cores and the HW fabric (1). The application, at this stage, can be automatically instrumented with PAPIFY calls necessary to read monitors present in both SW and HW (2). The HW computing fabric leverages on the multi-grain reconfiguration given by the combination of MDC (3) and ARTICo^3 (3). Hereinafter it is reported a short recap on the design-time adopted tools:

- PREESM enables parallel-application development with design-time prediction, and provides also code generation capabilities, generating an optimized code to execute the application on the given parallel and heterogeneous architecture [PREESM].
- PAPIFY generalizes PAPI to provide monitoring capabilities for heterogeneous HW/SW architectures. PAPIFY is capable of instrumenting any PREESM dataflow applications according to user-defined monitoring configurations [PAPIFY-DEMO] [PAPIFY] [PAPIFY-VIEWER] [Madroñal 2019].
- MDC and ARTICo^3 offer support for the design and implementation of a mixed grain architecture able to offer different degrees of reconfiguration [FANNI 2018, RODRIGUEZ 2018, MULTIGRAIN_TUT]. This architecture can be automatically instrumented with PAPI compliant monitors [Fanni 2019].

Run-Time Support

The mixed-grain architecture given by the combination of MDC and ARTICo^3 offers high flexibility, by enabling different execution trades-off in terms of performance, surfing among execution set-points and enabling scalable parallelism as well as achieving fault-tolerant designs through redundancy (1). Instrumentation of the architecture with ad-hoc PAPI-compliant HW monitor, allows PAPIFY and PAPIFY VIEWER (2) to pass relevant information to SPIDER, where a dedicated actor manager triggers the adaptation of the design (3). Hereinafter it is reported a short recap on the design-time adopted tools:

- SPIDER is the dynamic counter part of PREESM and performs dynamic mapping and scheduling of dataflow applications on a parallel heterogeneous architecture under a given constrained scenario [SPIDER].
- PAPIFY provides a set of run-time execution information to SPIDER. Monitors data are sent to SPIDER global run-time to enable informed decision taking [PAPIFY-DEMO] [PAPIFY] [PAPIFY-VIEWER] [Madroñal 2019].
- MDC and ARTICo^3 deploy and configure proper engines over the physical substrate at design-time. These engines are used at run-time to execute all the actions needed to support run-time reconfiguration of the HW [FANNI 2018, RODRIGUEZ 2018, MULTIGRAIN_TUT, Fanni 2019].

The following sections provide more technical details on these tools and how they have been adopted into the presented PE Demonstrator.

3.2.1. *PREESM and SPIDER*

In Figure 14, the PiSDF describes the DLS application in PREESM.

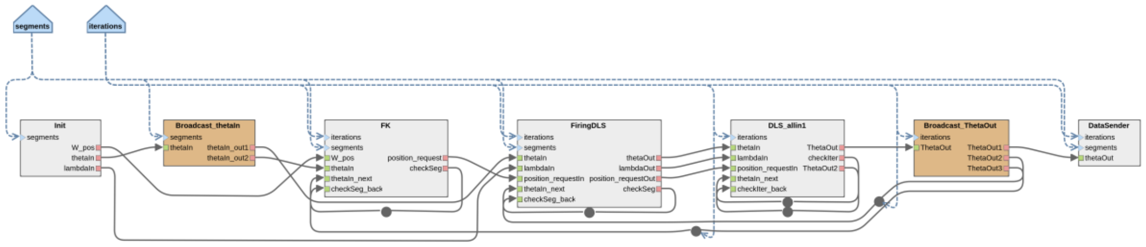


Figure 14. PiSDF description of the DLS application for design-time implementation using PREESM

In this implementation, the parameters (the blue pentagons) are set at design-time cannot be assigned at run-time, such as the characteristics of the communication links (FIFO sizes and delays) that depend on them. The actor *Init* sends the starting angles of the joints and the required points of the trajectory. From the information related to the starting configuration, the *FK* actor evaluates the end-effector position by using the Forward Kinematics. The actor *Firing_DLS* takes this as input and forwards it with the other desired points, managing the firing rate of the actor *DLS_allin1* that performs the algorithm always *iterations* times for each required point. At the end of the process, *DataSender* can send all the angles of the considered trajectory to the robotic arm. Thus, in order to handle the application at run-time with a variable number of iterations per segment, a SPIDER implementation has been needed.

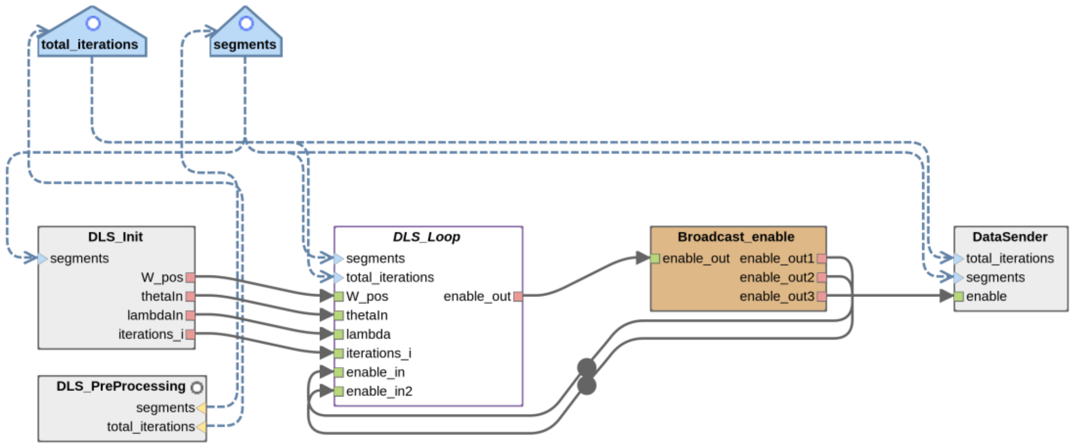


Figure 15. PiSDF description of the DLS application for run-time configuration using SPIDER

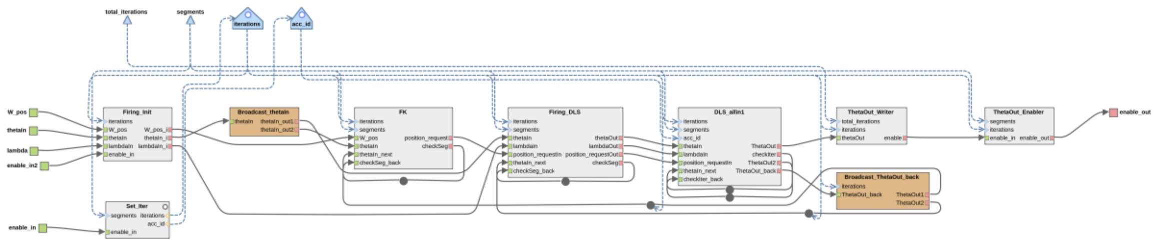


Figure 16. PiSDF description of the subgraph DLS_Loop

Figure 15 and Figure 16 show the dataflow description of the DLS application and of its sub-graph respectively, that have been designed by using PREESM and SPIDER. The configuration actor (*DLS_PreProcessing*) reads the inputs given by the user related to the accuracy and the required points of the trajectory. From the distance between subsequent input points, it calculates the number of the iterations required to the DLS to achieve every specific point, given the input accuracy. Then, it writes a file ("segmentsInfo.txt") in which all the points are reported with the related number of iterations. Moreover, it evaluates the total number of iterations needed to perform the entire movement (sum of the iterations of each point).

However, its main effect is to set the dynamic parameters: number of input segments/points (*segments*) and number of total iterations (*total_iterations*). The actor *DLS_init* fires one time per graph firing, only to read and send all the point information contained in the "segmentsInfo.txt", and enable the N firings of the next actor *Firing_Init* (where N corresponds to the parameter *segments*). For each point of the trajectory, *Set_Iter* updates the number of iterations (*iterations*), on which the FIFO sizes and delays (grey circles on the edges) depend. Moreover, depending on the battery level, it enables high or low performance of the accelerator by setting its ID (parameter *acc_id*) in the HW case (subgraph *DLS_HW* in the Figure 10).

As in the static version, *FK* provides the proper angles of the joints associated to the starting point of the trajectory (that is statically defined). The actor *Firing_DLS* handles input data tokens to send to the *DLS_allin1*. For the whole trajectory, *DLS_allin1* fires *iterations* times for each input point, performing the DLS algorithm and sending the output to the *ThetaOut_Writer* that stores it into the file “thetas.txt” for each DLS iteration. *ThetaOut_Enabler* fires at the end of every segment present in the trajectory only to enable the *DataSender*. This actor properly sets the angles of the joints, send the data of the whole trajectory to the arm, and enables a new reading of the input in the configurator (*DLS_PreProcessing*).

3.2.2. PAPIFY/PAPIFY VIEWER

PAPIFY is employed to monitor the HW/SW NM and DLS dataflow implementations. Specifically, the PAPIFY toolbox is used:

- To specify and collect execution time per actor iteration as a performance monitoring event for actors mapped to HW.
- To specify and collect execution time, error event counter values and power measurements per actor iteration during HW actor execution in a centralized manner at the SW side of the heterogeneous system using the concept of component
- To provide the collected counter values (execution times, error events and power measurements) to the adaptation manager for proper actor parameter adaptation in SPIDER.

The PAPIFY-VIEWER tool is employed to graphically represent the monitoring data per either actor or computing resource.

3.2.3. ARTICo^3

As it was mentioned, both NM and DLS were designed so that they were compatible with ARTICo^3 slots to allow for scalable and dynamically redundant operation modes. In order to achieve these goals, ARTICo^3 was used to:

- Define an architecture that splits the FPGA into the static part and four reconfigurable slots.
- Use the toolflow to generate the HW versions of both NM and DLS. They have been generated by either transferring the cost function from C to VHDL in the case of NM, or use HLS → MDC → ARTICo^3 wrapper addition in the case of DLS
- Use the provided ARTICo^3 run-time support libraries in the function calls available in the adaptation manager.

3.2.4. MDC

As described previously, in order to introduce an additional degree of non-functional adaptivity in the system, the HW version of the DLS has been modeled with MDC enabling two different execution profiles. The dataflow corresponding to such DLS HW model, in the BaseLine (BL) variant, is depicted in Figure 14. It corresponds to the actor DLS_allin1 within the SPIDER PiSDF description of the DLS, depicted in Figure 15. The DLS HW model involves, in turn, five HW actors, each executing a different step of the processing. HW actors have been designed by means of Vivado HLS starting from the C code of the SPIDER DLS_allin1 monolithic actor. The DLS HW model, besides having 5 inputs, as shown in Figure 17, also takes two dynamic parameters, compliantly with the corresponding SPIDER PiSDF model, and produces 4 outputs.

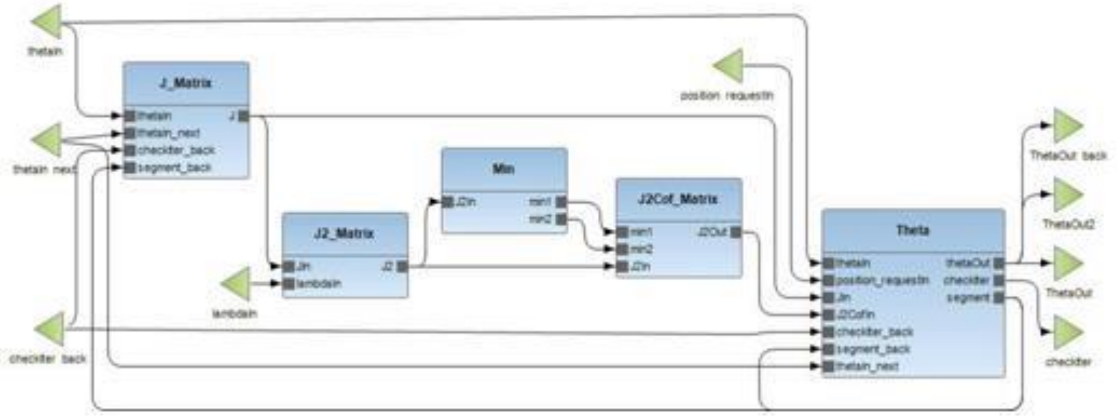


Figure 17. MDC model of the HW version of the DLS algorithm (BL variant).

Among all the HW actors, J_Matrix is the one responsible for about half of the computing time in the BL variant of the HW implementations. This actor, starting from the initial angles of the joints, is in charge of calculating the Jacobian matrix J, that is then used in the following computation by the other HW actors.

To provide an alternative to the BL configuration of the DLS HW implementation, the High Performance (HP) version has been derived by substituting the BL J_Matrix HW actor with a corresponding HP one. The HP J_Matrix HW actor has been generated with Vivado HLS as well, but UNROLL directives have been adopted to provide a faster, but more resource hungry J_Matrix actor. The HP variant of the DLS HW model is than equal to the BL one, but the J_Matrix actor has been replaced with the corresponding HP version, resulting in a speed up of the whole computation at a price of increased resource occupancy and, in turn, power consumption.

By running MDC on BL and HP DLS HW models, a reconfigurable DLS accelerator able to switch between these two execution profiles, BL and HP, has been generated. Being the reconfiguration provided by MDC a virtual reconfiguration, meaning that the resources of the two profiles are always present within the accelerator, the achievable trade-off is in terms of computing time vs. power consumption.

***3.2.5.** Mixed-Grain Adaptivity*

The integration of MDC and ARTICo^3 offers the possibility of automatically implement, deploy and manage mixed-grain reconfigurable architectures, able to provide different degrees of flexibility. Applications to be accelerated are defined as dataflow specifications and merged by MDC in an ARTICo^3-compliant CGR kernel. Then ARTICo^3 toolchain processes this kernel to automatically embed it in the HW architecture [FANNI 2018, RODRIGUEZ 2018, MULTIGRAIN_TUT].

In this particular demonstrator this allowed us to support:

1. functional changes of the execution from NM to DLS, exploiting potential of partial reconfiguration;
2. robust execution of both NM and DLS, by leveraging on ARTICo^3 slots redundancy;
3. different execution trade-off of the DLS algorithm, by implementing HP and LP configurations concurrently on the same slot with no partial reconfiguration penalty;
4. exploit parallelization potentials of the simplex optimization computation in the NM implementation by distributing calculations upon multiple ARTICo^3 slots.

3.3. Development and deployment environment

The robotic arm used is a Widow X by Trossen Robotics [TROSSEN]. It is a wrist-partitioned, vertically articulated robotic arm with six degrees of freedom. This robot mounts dynamixel actuators in each joint. This kind of servomotors is not representative of space technology, where typically brushless DC (BLDC) motors are used [NASA]. The main reasons are related to their limited life due to brush wear, and the generation of sparks. A BLDC motor control scenario was envisioned in early stages of the project, but the challenges of the robotic arm scenario were found to be more interesting in order to highlight the benefits of the complete CERBERO toolset.

The board that has been selected to implement the RCU is a ZCU-102 (see Figure 18) that contains a XCZU9EG-2FFVB1156E MPSoC SRAM based device. This device contains a variety of computing fabrics, which is valid for exploiting the heterogeneous execution support devised in the adaptation WP (WP4). Among the fabrics selected for execution, there are:

- An A53 quad-core 64 bits core from ARM, where Linux OS and SW supported tools are held.
- A programmable logic component (i.e. FPGA fabric) which holds the ARTICo^3 infrastructure, enriched with MDC-compliant slots and PAPIFY monitors, for HW accelerated execution.

Figure 18. Xilinx ZCU102 development board

The device also contains other cores, such as R5 cores for real-time execution, and an embedded GPU. This last one would be a desirable fabric, but the support provided by the manufacturer restricts its use for processing graphics with OpenGL libraries, which is out of the scope of this project.

The ARTICo^3 architecture was implemented within the project using a 4 slot architecture. An 8-slot based architecture was also used for verifying performance and scalability. The ARTICo^3 toolflow includes a tutorial on how to set up the virtual architecture partition (i.e., the number of slots and the architecture of the static part).

4. Tests, results and feedback

4.1. Tests

For testing and validating the M36 Planetary Exploration demonstrator the scenario described in D2.1 has been implemented by using PREESM, PAPIFY, SPIDER, ARTICo^3 and MDC tools.

As stated in section 3.1.3, initially the Nelder-Mead optimization was adopted as the baseline approach for the path planning algorithm implementation. Initial testing of a standalone application was performed in order to assess its usability in the scenario. Figure 19 shows the lab set-up in TASE where the validation tests took place.

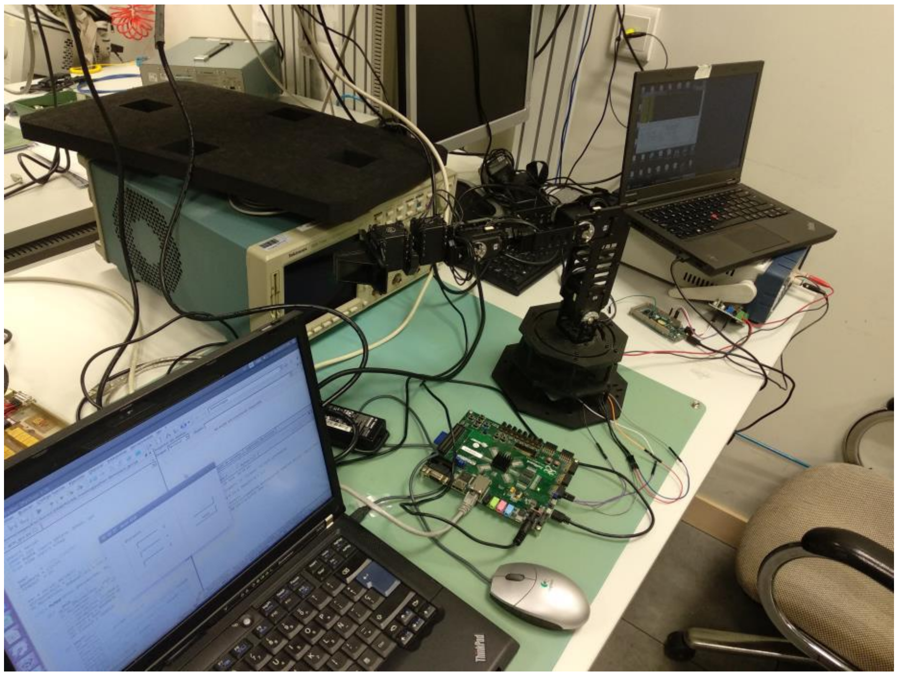


Figure 19. Set-up in TASE for testing of N-M implementation

These validation tests were favorable and they confirmed that the Nelder-Mead algorithm would not throw convergence problems during the optimization process, and that the defined “safe zone” was indeed a valid workspace free of singularities.

After M18, in order to be capable of highlight more features of the toolset, it was decided to add the algorithm diversity feature to the demonstrator. In that moment, a development started in parallel for another version of the motion planning algorithm based on the Damped Least-Squares that overcomes some of the limitations of the Nelder-Mead algorithm. Thanks to the combined effort of all partners involved in the project, this algorithm was promptly validated with the real arm, and its implementation under CERBERO toolset could start.

Figure 20 shows the set-up for the DLS implementation of the motion planning algorithm as displayed in Alghero for the Summer School 2019 demonstration.

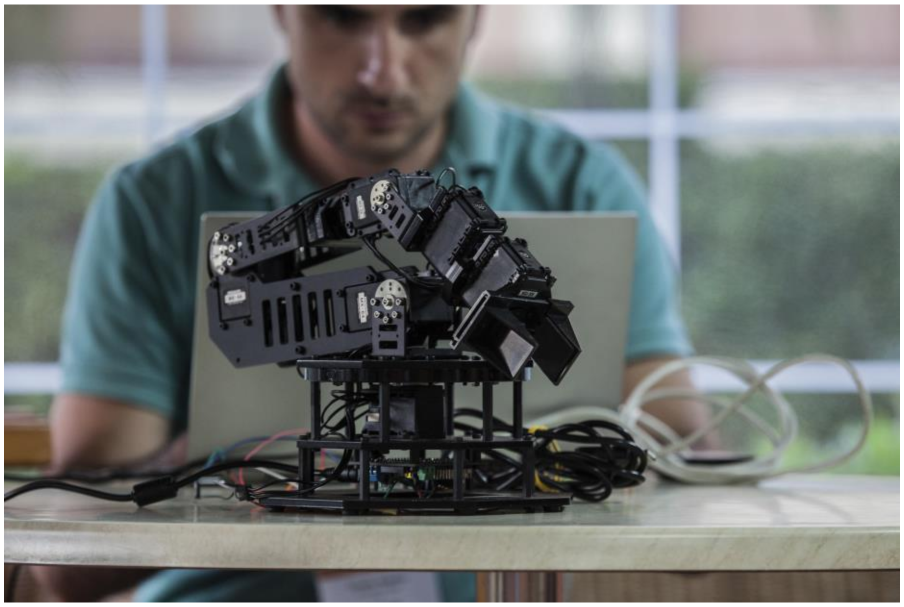


Figure 20. Set-up in Alghero for the demonstration of DLS implementation

These two development environments with the physical robotic arm were used to develop, integrate and validate the different parts of the demonstrator.

Final tests were focused on the ability to execute the scenarios as defined in D2.1 and also prove that expected behavior of the computing system is produced after every mechanism of adaptation is triggered; e.g. change in robot functionality with different accuracy constraints automatically impacts performance, redundancy mechanisms are implemented when the solar storm is commanded, or algorithm diversity is properly exploited.

4.2. Test result

4.2.1. Quantitative results

4.2.1.1 Nelder-Mead on ARTICo^3

The following performance and power measurements, that correspond with actual values of the execution in the target board, are to exemplify the performance of ARTICo^3 accelerator system behavior. For this, values on execution time, power and resulting energy values (computed by multiplying the previous two values) are shown in the Table 3:

Table 3. Performance metrics for Nelder Mead on ARTICo^3

# accelerators	Elapsed Time (s)	Meas. Power (mW)	Energy (J)
1	11.5	375	4.31
2	6.5	400	2.60
3	5.5	450	2.47
4	4.5	450	2.02
5	3.4	500	1.70
6	3.4	525	1.78
7	3.4	550	1.87
8	3.4	575	1.95

The time measurement is for a complete 100 points trajectory, solving 8 points in parallel assuming, as a starting point, the same set of initial arm angles for all point computations within the same accelerator invocation. Every point iteration needs a total of 10 simplex evaluations. Evaluating 8 points in parallel implies, therefore,80 simplex evaluation per iteration. The number of iterations is not fixed and depends on the precision and the distance angles between the original starting point of the computation and the resulting final one. Although the architecture used in the PE use case contains only 4 slots, the previous table shows data for up to 8 accelerators, to show the scalability property. It may be seen that, using more than 5 accelerators, no further performance savings are obtained, on the contrary and increased power and energy are used.

Time per iteration per point for a variable number of accelerators is shown in Figure 21. Curves apply only for the feasible parallelization achievable, i.e., since a point cannot be split into two accelerators, it is useless to use a number of accelerators higher than the number of points in parallel.

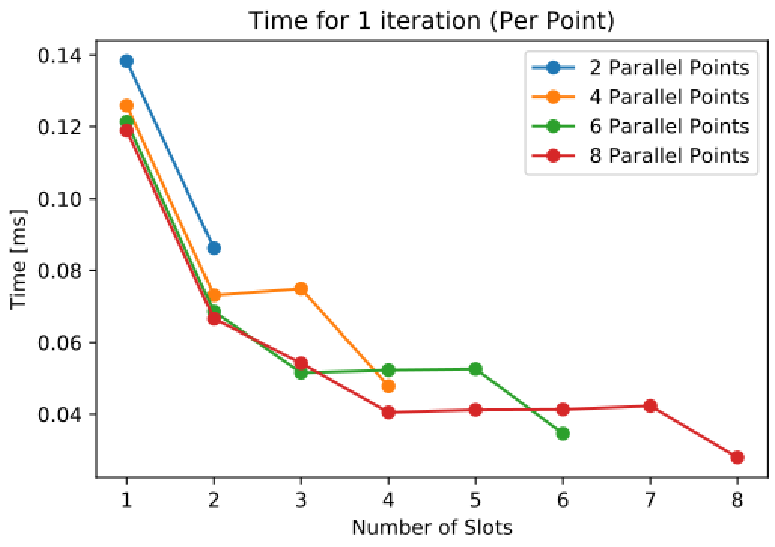


Figure 21. Time per single iteration for different number of parallel points and slots

As mentioned before, the number of iterations to solve each point is not fixed. Figure 22 shows a statistical boxplot for the number of iterations required when splitting the number of points in every accelerator invocation.

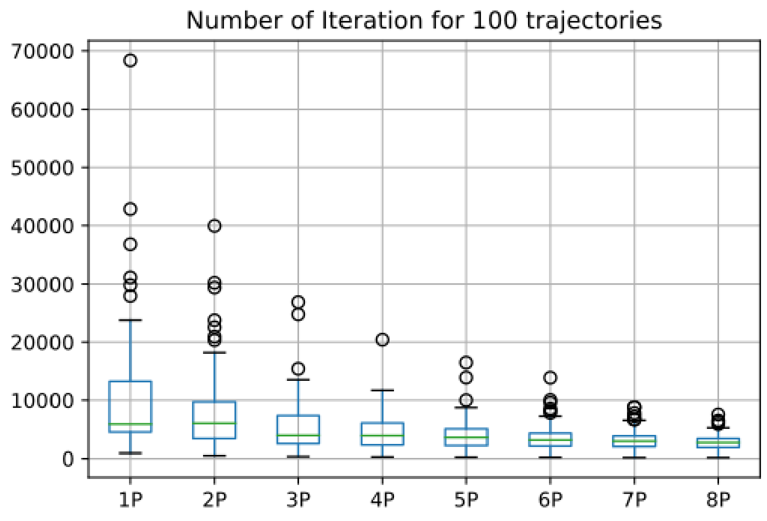


Figure 22. Number of iterations (-3 sigma, avf, 3 sigma) per parallel points invocation

As it may be seen, the total number gets reduced with higher parallel points. This is due to the fact that the points that are invoked in parallel are close to each other and start from the same angle, and so, the precision gets closer for all parallel computations, and so, release the accelerator for the next point set computation.

4.2.1.2 Damped Least Square with MDC

Table 4 reports resource occupancy numbers coming from the High Level Synthesis of the actors involved in the DLS reconfigurable HW accelerator, as well as from the synthesis of the isolated reconfigurable datapath provided by MDC. As discussed previously, even if such datapath is capable of executing the BL and HP variants of the DLS HW implementation, the corresponding resources are always instantiated in the device, meaning that the resource occupancy for such implementation is the same for both BL and HP cases.

Table 4. Synthesis results for Zynq UltraScale+ target device.

entity	LUT	FF	DSP	BRAM
J_Matrix	4729	3577	18	15
J_Matrix_HP	13880	12110	88	5
J2_Matrix	1074	833	5	0
Min	738	676	5	0

J2Cof_Matrix	1872	1402	8	2
Theta	1656	1295	5	0
reconfigurabl e datapath	18925	15448	129	14

Dealing with latency (see Table 5) and power consumption (see Table 6) the situation is different: the HP variant is over-performing the BL one, employing barely 25% less time for executing one iteration, that is one step, of the DLS algorithm. Being the number of iterations fixed for a given length of the trajectory and desired accuracy, this percentage of saving is kept constant also for the considered test trajectories.

Table 5. Post-synthesis latency analysis at 100 MHz

latency [ms]					
configuration	1 iteration	trajectory 1	trajectory 2	trajectory 3	trajectory 4
BL	0.0452	11.3	9.04	4.52	9.04
HP	0.0329	8.22	6.58	3.29	6.58

If the resource trade-off between the two variants, BL and HP, is flattened by the MDC virtual reconfiguration, it is still present a trade-off in terms power consumption. In fact, for calculating one iteration, that is one step, of the DLS algorithm, the BL variant consumes about 4% less power than the HP one. This difference could be very important in the considered context where the system is supplied by batteries and the power of these latter can be limited.

Table 6. Post-synthesis power consumption analysis at 100 MHz

power [mW]								
configuration	total	static	dynamic	clocks	signals	logic	BRAM	DSP
BL	751	621	130	50	5	13	4	18
HP	780	621	159	50	14	24	4	27

The DLS reconfigurable HW accelerator generated by MDC is thus successfully providing a system capable to quickly adapt depending either on the desired computation time (latency) or the amount of power consumption. The adaptation is reached by changing configuration from the BL profile, that is low power but slow, to the HP profile, that is fast but power hungry, and vice versa.

4.2.2. End-user results

In the table below the results from the demonstrator development activity were added (in *italic*) to the expected results as specified in Table 3-1 of D2.1:

ID	Goal	Results M36 demonstrator
PE1	Enable Hardware / Software (HW/SW) co-design for Rad-Tolerant control of robotic arm for planetary exploration using adaptable COTS FPGAs.	Minimization of energy consumption and costs, while keeping/improving resiliency. o *PiSDF applications developed through PREESM get all benefits from the design time, from architecture modeling to code generation, and ARTICo^3 architecture enables heterogeneous systems with resiliency to radiation failures.* o *The implementation of a HW/SW self-adaptation loop on heterogeneous FPGA technologies has been facilitated by the adoption of the available tools (PREESM, SPIDER, ARTICo^3, MDC and PAPIFY) and offered the possibility of switching among different working points and configurations to guarantee variable performance and resiliency.* o *The Robot Control Unit has been implemented in a Zynq UltraScale+ FPGA, which has proven to hold the computing power and reconfigurable resources necessary to demonstrate this requirements.*
PE2	Develop integrated "open" toolchain environment for development of robotic arms for space missions with focus on multi-viewpoint system-in-the-loop virtual environment.	Provide multi-objective design space exploration and multi-view analysis. Reduce development time of complex heterogeneous systems by increasing the level of abstraction. Increase quality and verification level to ensure proper operation of the system. o *The complete toolset has been used to deploy and run the self-adaptation computing infrastructure capable of controlling the robotic arm.* o *A number of library elements that can be used for future developments have been produced.*
PE3	Development of a (self-)adaptation methodology with supporting tools.	Efficient support of architectural adaptivity, according to radiation effects and harsh environmental conditions. o *The usage of the CERBERO tools shows the efficient support of functional, non-functional and repair-oriented adaptation, according to internal triggers from processing platform, user commands and emulation of environmental conditions.*

4.3. Feedback

Based on the high-level functionalities and the demonstrator skeleton description defined in early versions of the respective deliverables, an initial test scenario was defined. This scenario was tackled in the M18 demonstrator, which highlighted the following achievements:

- Verification of the proper operation of the inverse kinematics and optimization algorithms to provide collision-free motion planning.
- Improvement of execution time by using parallel methods of computation.
- Optimal parallelization to find a trade-off between execution time and soft motion planning.
- Support of dual and triple mode redundancy, enabling component-level fault tolerance.

After the development of M18 demonstrator, additional functionalities were added.

First, the initial PiSDF graph implementing the algorithm in PREESM evolved into a more complex description with a higher number of actors, increasing the monitoring granularity of internal KPIs achieved by the PAPIFY tool. This graph included a HW description for the most critical, time consuming actor, enabling the HW scalability.

In parallel, algorithm diversity was explored and DLS solution for the inverse kinematic problem was developed. The concurrent adaptive implementation of different algorithms contributes to achieve the requested QoS versus computational efficiency and energy consumption trade-off.

Finally, ARTICo^3 was integrated with SPIDER, enhancing the reconfigurable behavior of the model by allowing to set new values for parameters for the ARTICo^3 accelerators at run-time.

The final solution provided all envisioned functionalities, leading to the results described in section 4.2. A final evaluation on the fulfillment of the initial requirements and the tools utilized during the development of M36 demonstrator for the Planetary Exploration use case is provided in Section 4 of D6.5.

5. Conclusion

The final demonstrator of the Self-Healing System for Planetary Exploration use case uses CERBERO methodology, tools and technologies to overcome the main challenges of the scenario.

Fault tolerant capabilities are provided to the computing platform, automatically adjusting the redundancy of the HW accelerators and being capable of moving the execution flow of the path planning algorithm from HW to SW and vice versa in an autonomous way.

Adaptation to the harsh physical environment is achieved due to the run-time trade-offs evaluation by the adaption manager between the internal KPIs of the system, user commands and environmental conditions emulation.

The implemented performance monitoring counters in the cyber part of the computing platform provide simultaneous HW/SW power measurement and optimization capabilities.

This demonstrator highlights an extensive amount of work from the CERBERO consortium. The collaboration between scientific and industrial community during this time has produced important synergies: advances in relevant, state-of-the art challenges of the field of space robotics were addressed, and also the evolving needs defined by this activity helped refine the tools to better fit real use case applications.

6. References

[Aristidou 2018]	Aristidou, A., Lasenby, J., Chrysanthou, Y. and Shamir, A. (2018), Inverse Kinematics Techniques in Computer Graphics: A Survey. Computer Graphics Forum, 37: 35-58. doi:10.1111/cgf.13310
[BAILON	Bailón,W.; Cardiel, E.; Campos, I.; Paz, A. Mechanical energy optimization in trajectory planning for six DOF robot manipulators based on eighth-degree polynomial functions and a genetic algorithm. In Proceedings of the 7th International Conference on Electrical Engineering, Computing Science and Automatic Control, Tuxtla Gutierrez, Mexico, 8–10 September 2010; pp. 446–451.
[Buss 2009]	Samuel R. Buss. Introduction to inverse kinematics with Jacobian transpose, pseudoinverse and damped least squares methods , 2009.
[Buss 2015]	Buss, Samuel & Kim, Jin-Su. (2005). Selectively Damped Least Squares for Inverse Kinematics. J. Graphics Tools. 10. 37-49. 10.1080/2151237X.2005.10129202.
[CARABIN]	Carabin, G.; Wehrle, E.; Vidoni, R. A Review on Energy-Saving Optimization Methods for Robotic and Automatic Systems. *Robotics* 2017, *6*, 39.
[CCSDS]	https://public.ccsds.org/Pubs/352x0b2.pdf
[CERBERO 2018]	http://www.cerbero-h2020.eu
[CORKE]	http://petercorke.com/wordpress/toolboxes/robotics-toolbox
[D1.3]	CERBERO D1.3 Open Data Management Plan (Final version)
[D2.1]	CERBERO D2.1 Scenarios description (Final version)
[D2.2]	CERBERO D2.2 Technical requirements (Final version)
[D3.1]	Modelling of KPI (Final version)
[D4.2]	D4.2 – Self Adaptation Manager (Final version)
[D5.1]	CERBERO D5.1 Holistic methodology and integration interfaces (Final version)
[D6.1]	CERBERO D6.1 Demonstration Skeleton (Final version)
[D6.8]	CERBERO D6.8 Planetary Exploration Demonstrator (Version I)
[DH1]	Denavit, Jacques; Hartenberg, Richard Scheunemann (1955). *"A kinematic* notation for lower-pair mechanisms based on matrices". Trans ASME J. Appl. Mech. 23: 215–221
[DH2]	Hartenberg, Richard Scheunemann; Denavit, Jacques (1965). Kinematic synthesis of linkages. McGraw-Hill series in mechanical engineering. New York: McGraw-Hill. P. 435.
[DLS]	https://www.cerbero-h2020.eu/wp-content/uploads/2019/12/DLS-implementation.zip
[FANNI 2018]	Fanni, T., Rodríguez, A., Sau, C., Suriano, L., Palumbo, F., Raffo, L., & de la Torre, E. (2018, December). Multi-Grain Reconfiguration for Advanced Adaptivity in Cyber-Physical Systems. In 2018 International Conference on ReConFigurable Computing and FPGAs (ReConFig) (pp. 1-8). IEEE.
[Fanni 2019]	T. Fanni et al., "Run-time Performance Monitoring of Heterogenous Hw/Sw Platforms Using PAPI," FSP Workshop 2019; Sixth International Workshop on FPGAs for Software Programmers, Barcelona, Spain, 2019, pp. 1-10.
[Fanni L. 2019]	Luca Fanni, Leonardo Suriano, Claudio Rubattu, Pablo Sánchez, Eduardo de la Torre, Francesca Palumbo. A Dataflow Implementation of Inverse Kinematics on Reconfigurable Heterogeneous MPSoC. Proceedings of the Cyber-Physical Systems PhD Workshop 2019, an event held within the {CPS} Summer School "Designing Cyber-Physical Systems - From concepts to implementation", Alghero, Italy, September 23, 2019. http://ceur-ws.org/Vol-2457/11.pdf
[Lewis 2003]	Lewis, F.L. and Dawson, D.M. and Abdallah, C.T., Robot Manipulator Control: Theory and Practice}. 2003, Edition 2. CRC Press.

[Madroñal 2019]	D. Madroñal, F. Arrestier, J. Sancho, A. Morvan, R. Lazcano, K. Desnos, R. Salvador, D. Menard, E. Juarez and C. Sanz. PAPIFY: Automatic Instrumentation and Monitoring of Dynamic Dataflow Applications Based on PAPI. IEEE Access, Vol 7, pp. 111801-111812, Sept. 2019.
[MOHAMMED]	Mohammed, A.; Schmidt, B.;Wang, L.; Gao, L. Minimizing energy consumption for robot arm movement. Procedia CIRP 2014, 25, 400–405.
[MULTIGRAIN_TUT]	http://www.cpsschool.eu/previous-editions/cps-summer-school-2018/schedule/multi-grain-reconfiguration-advanced-adaptivity-cyber-physical-systems-2/
[NASA]	NASA-CR-2506, NASA, United States, 1975.
[NM]	Nelder, John A.; R. Mead (1965). "A simplex method for function minimization". Computer Journal. 7 (4): 308–313
[PAPIFY]	https://gitlab.citsem.upm.es/papify/papify/wikis/Papify-Website
[PAPIFY-DEMO]	https://www.youtube.com/watch?v=9QbqtEjKI2U
[PAPIFY-VIEWER]	https://gitlab.citsem.upm.es/papify/papify/wikis/Papify-Viewer-website
[PEUCOD]	https://www.cerbero-h2020.eu/wp-content/uploads/2019/12/Open-Data-PE.zip
[PREESM]	https://preesm.github.io/
[RODRIGUEZ 2018]	Rodriguez, A., & Fanni, T. (2018, December). Multi-Grain Adaptivity in Cyber-Physical Systems. In 2018 30th International Conference on Microelectronics (ICM) (pp. 44-47). IEEE.
[SENGUPTA]	Sengupta, A.; Chakraborti, T.; Konar, A.; Nagar, A. Energy efficient trajectory planning by a robot arm using invasive weed optimization technique. In Proceedings of the 3rdWorld Congress on Nature and Biologically Inspired Computing, Salamanca, Spain, 19–21 October 2011; pp. 311–316.
[SPIDER]	https://preesm.github.io/tutos/spider/
[TRJ]	https://www.cerbero-h2020.eu/wp-content/uploads/2019/12/Trajectory-Generator.zip
[TROSSEN]	https://www.trossenrobotics.com/widowxrobotarm
[VALVERDE]	Valverde, Alfredo & Tsiotras, Panagiotis. (2018). Spacecraft Robot Kinematics Using Dual Quaternions. Robotics. 7. 64. 10.3390/robotics7040064.
[Wright 2012]	Wright, M. (2012). Nelder, Mead, and the other simplex method. Documenta Mathematica, Extra volume: Optimization Stories, 271-276