UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE INGENIERÍA ELÉCTRICA

NANOSATELLITE CONSTELLATIONS CONTROL FRAMEWORK USING EVOLUTIONARY CONTACT PLAN DESIGN AND COMMAND ARCHITECTURE FLIGHT SOFTWARE

TESIS PARA OPTAR AL GRADO DE DOCTOR EN INGENIERÍA ELÉCTRICA

CARLOS EDUARDO GONZÁLEZ CORTÉS

PROFESOR GUÍA:
MARCOS DÍAZ QUEZADA

PROFESOR CO-GUÍA:
ALEXANDRE BERGEL

MIEMBROS DE LA COMISIÓN:
SANDRA CÉSPEDES UMAÑA
JUAN FRAIRE
SHINICHI NAKASUKA

Este trabajo ha sido parcialmente financiado por:
CONICYT-PCHA/Doctorado Nacional/2016-21161016.

Powered@NLHPC: Esta investigación/tesis fue parcialmente apoyada por la infraestructura de supercómputo del NLHPC (ECM-02)

SANTIAGO DE CHILE
2022

RESUMEN DE LA TESIS PARA OPTAR AL GRADO
DE DOCTOR EN INGENIERÍA ELÉCTRICA
POR: CARLOS EDUARDO GONZÁLEZ CORTÉS
FECHA: 2022
PROF. GUÍA: MARCOS DÍAZ QUEZADA
PROF. CO-GUÍA: ALEXANDRE BERGEL

NANOSATELLITE CONSTELLATIONS CONTROL FRAMEWORK USING EVOLUTIONARY CONTACT PLAN DESIGN AND COMMAND ARCHITECTURE FLIGHT SOFTWARE

Agencias espaciales, instituciones educacionales, y empresas usan Cubesats para la investigación científica, educación, demostraciones tecnológicas e industria espacial en la era del *New Space*. El siguiente paso en la cambiante industria espacial es la construcción y operación de mega constelaciones de cientos a miles de pequeños o nanosatélites. Este contexto agrega nuevos requerimientos y desafíos para las líneas de producción y operación de proyectos espaciales. Este trabajo se enfoca en la operación ágil de una mega constelación satelital con comunicaciones inter-satélite. Este trabajo propone utilizar la topología de la constelación para diseñar planes de contacto usando algoritmos evolutivos y la información del plan de contacto para controlar la operación de la constelación. El plan de contactos se usa para crear una tabla con un plan de vuelo global que resumirá las operaciones necesarias para ejecutar una tarea. Así, los satélites y estaciones terrenas solo necesitan un software de vuelo capaz de encolar, ejecutar y transferir los comandos del plan de vuelo. Este trabajo presenta el diseño e implementación del sistema completo así como casos de estudio para validar el funcionamiento con constelaciones de hasta 1000 nodos. La propuesta de usar algoritmos evolutivos para diseñar el plan de contactos muestra resultados prometedores abriendo la posibilidad de controlar mega constelaciones de cientos a miles de nanosatélites.

ABSTRACT OF THE THESIS FOR THE DEGREE
OF DOCTOR IN ELECTRICAL ENGINEERING
AUTHOR: CARLOS EDUARDO GONZÁLEZ CORTÉS
DATE: 2022
ADVISOR: MARCOS DÍAZ QUEZADA
CO-ADVISOR: ALEXANDRE BERGEL

NANOSATELLITE CONSTELLATIONS CONTROL FRAMEWORK USING EVOLUTIONARY CONTACT PLAN DESIGN AND COMMAND ARCHITECTURE FLIGHT SOFTWARE

Space agencies, educational institutions, and private companies have adopted CubeSat nanosatellites to do scientific research, training, technology demonstration, and space-based industries in the New Space era. The next step in this changing space sector corresponds to the assembly and operation of large satellite constellations consisting of hundreds or thousands of small- or nanosatellites. This context adds new requirements and challenges to the production and operation lines of these space projects. This work focuses on the agile operation of a large nanosatellite constellation with inter-satellite communications. This work proposes utilizing the constellation contact topology to design contact plans using evolutionary algorithms and contact plan information to control the constellation operations. The contact plan is then used to create a Global Flight Plan table that summarizes all the operations required to execute a proposed task. Thus, satellites and ground station nodes only need flight software capable of queuing, executing, and transferring Flight Plan commands. This work presents the design and implementation of the complete system and case studies to validate framework functioning with constellations up to 1000 nodes. The evolutionary contact plan design approach shows promising scalability results opening the possibility of controlling satellite mega constellation of hundreds or thousands of nanosatellites.

A mis padres y hermanas por que les debo todo lo que soy,
y porque gracias a su gran esfuerzo y apoyo incondicional
he podido llegar a instancias que nunca imaginé.

Los amo, siempre estaré con y para ustedes

Agradecimientos

A mi familia, padres y hermanas, por su incondicional apoyo y comprensión durante este largo proceso. En especial a mis padres por su gran esfuerzo, por su confianza, por haber sembrado en mí, desde muy pequeño, la motivación para ser una gran profesional. Con su ejemplo, me han enseñado la lección más importante: el trabajo dedicado, el esfuerzo y la perseverancia son las claves para lograr las metas en la vida, la felicidad personal y la de quienes te rodean. Por siempre les estaré agradecido. Espero que mis metas sean también parte su felicidad.

A mi novia, mi compañera, mi amiga y mi amor: Tamara. Muchas gracias por tu incondicional apoyo y paciencia en momentos difíciles. Compartir contigo en tantos aspectos de la vida ha sido una experiencia inigualable. Gracias por motivar mis esfuerzos y abrir las puertas a nuevas perspectivas de futuro. Este trabajo está lleno de tu gracia.

A Pacu, quien ama, apoya y enseña sin decir nada.

A los miembros de mi comisión, por la confianza depositada en mí, su apoyo y aporte, que ha sido vital en mi formación como profesional. Al profesor Marcos Díaz por la oportunidad invaluable de poder participar en un proyecto tan fuera de serie, por compartir su visión sobre el rol profesional y social que implica este grado académico. Al profesor Alexandré Bergel quien amablemente ha compartido su conocimiento y entusiasmo, entregando un apoyo de otro nivel para cumplir esta meta. Sus valores humanos, éticos y profesionales son una base inigualable para crecer como persona.

A todos los miembros del proyecto SUCHAI, porque este trabajo integra, de manera directa o indirecta, el esfuerzo de un gran equipo de personas. En el intento de compartir mi experiencia, he aprendido de ustedes conocimientos y virtudes que no puedes encontrar en ningún libro. En mi persona hay una amalgama de cada conversación, risas, logros, fatigas y sueños que hemos compartido en las arduas horas de trabajo. A mis amigos y colegas Alex Becerra, Camilo Rojas, Elias Obreque, Francisco Anguita. Ustedes le dan sentido a las palabra equipo, poder compartir la alegría y el rigor con ustedes es un privilegio para mí. A todas mis amistades, gracias por estar más allá de lo que el tiempo significa.

A mis alumnos y colegas de la Universidad de Santiago de Chile. Probablemente no sepan que han sido un soporte muy importante para desarrollar este trabajo. Enseñar es un privilegio y una responsabilidad y sobre todo una gran motivación. He aprendido demasiado de ustedes. Finalmente, recordar que este trabajo ha sido apoyado por becas y fondos científicos estatales. De forma muy directa la sociedad depositó su confianza y me entregó la responsabilidad de retribuir desde mi disciplina y humanidad a una mejor sociedad para todos.

Table of contents

List of tables

List of illustrations

Chapter 1

Introduction

1.1. Motivation

CubeSats are standardized satellites shaped like a 10cm edge cube. According to the CubeSat Design Specification, this basic unit (1U) can be extended to 2U, 3U, 6U, and more [1]. CubeSat nanosatellites have demonstrated that cost- and time-effective access to space is possible. Space agencies, educational institutions, and private companies have adopted this technology to do scientific research, training, technology demonstration, and space-based industries in the New Space era [2]. In general, satellites can operate independently or in formation flying, i.e., several vehicles are used to accomplish a mission cooperatively. The most common satellite formations flying are trailing, cluster, and constellation [3] using Walker-delta or Walker-star geometries [4]. However, it is unlikely that a large or mega CubeSat constellation can be deployed with a specific geometry. Instead, Ad-hoc configurations resulting from several secondary payload launching opportunities can be expected [5, 6]. If the satellites are equipped with an inter-satellite communication system, the capabilities of the constellation are enhanced, enabling missions such as: servicing or proximity operations, autonomous operations, fractionated spacecraft, or distributed processing [3].

Today, the trend in this changing space sector is the assembly and operation of large satellite constellations consisting of hundreds or thousands of small or nanosatellites [7, 8, 9]. Such a number of satellites have no precedents and propose new challenges. Satellite operations largely depend on human operators, statically assigned ground capabilities, or homogeneous satellite networks. Constellations with inter-satellite capabilities are not widely deployed yet except for a few companies. Small and nanosatellites add numerous challenges to this problem: heavy restrictions in space, power, and communications capabilities, plus different configurations, short life cycle, and rapid technological evolution. These challenges may stress satellites' production lines [10]. Meanwhile, the deployment, maintenance, data acquisition, routing, and optimization of constellation operations are complex scheduling problems [11], so novel techniques are required to achieve repeatability, autonomy, and scalability in nanosatellite constellation missions [10].

Radhakrishnan *et al.* (2016) [3] review the challenges of constellations missions with small satellites from Physical to Network layers of the OSI model. However, the assembly and operation of these space systems also present many challenges in the Application Layer. A high level of automation is required to optimize the usage of small satellite constellation

capabilities. Assigning earth and space resources, distributing goals, or propagating changes in the constellation system are complex problems that require intelligent algorithms to be solved [12, 13]. Even the simplest versions of these scheduling problems are Mixed Integer Linear Programming (MILP) and hence NP-hard class [14]. Therefore exact solution algorithms can be impractical for large constellations of thousands of nodes. On the other hand, heuristics approaches such as Evolutionary Algorithms have been used with promising results.

From the Network and Transport layers perspective, traditional TCP/IP protocols are not well suited for a satellite constellation due to long-delay and low reliable communication links. CCSDS protocols are more widely used in space applications but with limited use cases in large constellations. The family of DTN routing protocols is better suited in LEO constellations with ISL. In contrast with TCP/IP stack protocol, Delay or Disruption Tolerant Network (DTN) protocols assume that contacts among nodes are sporadic, so nodes require a buffer to store and carry messages until a link is available. Despite the time-evolving nature of the connections in a Low Earth Orbit (LEO) satellite network, the contact opportunities can be predictable due to orbital mechanics [15]. The contact information can be used to design contact plans with different goals[15, 16, 17].

All the operations mentioned above must be supported in the satellite flight software as well as the ground control nodes [18, 11]. Previous works have remarked that modular, extensible, and reliable flight software architectures are required to deliver quality software in less time and with less effort [19, 20].

Despite the incipient deployment of small- and nanosatellite mega-constellations, the studies in task scheduling, and advances in flight software development, there is a gap in solutions that integrate those concepts to scale the production and operation of nanosatellites constellations from tens to hundreds or thousands of nodes. Thus, in this work, we present a nanosatellite constellation control framework (See section 3) that uses the contact topology information to design a global flight plan which must be executed cooperatively by the satellites to solve a particular task. This global flight plan is created using a contact plan, just as in DTN routing protocols. Thus, an evolutionary algorithm is used to design this contact plan, considering the scalability of a large number of nodes. Based on typical CubeSats' hardware and software capabilities, delegating the scheduling problem to the ground nodes and delivering a global flight plan table to the satellites seems to be a more suitable solution. Validation of these ideas is shown in Section 4 through a case study that includes constellations with tens, hundreds, and thousands of satellites.

1.2. Problem definition

Constellations with a large number (hundreds to thousands) of small or nanosatellite with Inter-satellite link (ISL) are challenging. These challenges, decomposed using the OSI framework, ranging from the Physical to the Application layers, are summarized in Table. 1.1. In lower layers, inter-satellite communication systems for nanosatellites with limited energy resources and pointing capabilities and proper medium access control (MAC) protocols are required. Emerging technologies based on optical communication links and phased array an-

tennas have shown promising results [3]. From the Network and Transport layers perspective, traditional TCP/IP protocols are not well suited for a satellite constellation due to long-delay and low reliable communication links. CCSDS protocols are more widely used in space applications but with limited use cases in large constellations. A new family of routing protocol for DTN is required [15]. Finally, in the Application layer, a more flexible and autonomous software solution to control a small nanosatellite constellation operation is an active research area [21]. Additionally, the flight software should include tools to simplify the constellation operation, task scheduling, and options to optimize the system resource usage [18, 11]. Despite the incipient deployment of small- and nanosatellite mega-constellations, the studies in task scheduling, and advances in flight software development, there is a gap in solutions that integrate those concepts to scale the production and operation of nanosatellites constellations from tens to hundreds or thousands of nodes.

Table 1.1: Small- and nanosatellite constellation challenge summary

OSI model	**Challenges**
Application Presentation Session	Flight software architectures capable of scaling to assembly hundreds or thousands of small or nanosatellites in an agile fashion [21]. Support an autonomous operation of the constellation. Optimize the usage of the constellation resources (computational, energy, lifetime, etc.) [18, 11]
Transport Network	Network protocols that support long delays and/or interrupted communication links [15]. Protocols aware of computational resources, energy, and link capacity limitations [17].
Data-link Physical	Inter-satellite communication links for small or nanosatellites with energy, pointing, and space limitations [3]. MAC protocols for large wireless network [22].

1.3. Research questions

Considering the problem stated Section 1.2, this Ph. D. thesis addresses the following research question:

"How to control the operation of a large nanosatellite constellation with ISL optimizing the system resources usages "

This general research question generates further relevant secondary questions:

1. *Should the operation control strategy be implemented in the ground or space segment?*
2. *What optimization technique should be implemented to solve the scheduling problem?*
3. *What are the software requirements, in terms of architectural design, to control a nanosatellite constellation operation?*

These questions intend to provide a general guideline to find a feasible framework design to operate large nanosatellite constellations with ISL.

1.4. Hypotheses

The Hypotheses presented in this Ph. D. thesis are the following:

H1 A nanosatellite constellation can be controlled from a central entity by setting a global flight plan that nodes must execute cooperatively.

H2 The flight plan that solves the scheduling problem can be derived from the contact topology designing a valid contact plan as seen in DTN routing problems.

H3 Evolutionary algorithms can find a valid contact plan, and thus a global flight plan, in bounded time even for large constellations of hundreds or thousands satellites.

1.5. Research objectives

1.5.1. General objective

The general objective of this Ph.D. thesis is to provide a framework to automate a nanosatellite constellation's operations with ISL and evaluate the scalability of the solution to a constellation of hundreds to thousands of nodes.

1.5.2. Specific objectives

The specific objectives required to fulfill the work are:

- Design and implement a mission control framework to schedule and execute tasks in a satellite network automatically.
- Design and implement a nanosatellite flight software that satisfies the framework operation model's requirements.
- Define metrics and tool-chains to test, measure, and validate the framework results. For example, measure the solution's scalability to scenarios of thousands of satellites and the constellation resources usage when tasks are executed.

1.6. Contributions of this work

The contributions of this work are summarized below:

- **Nanosatellite constellations mission control framework:** This work presents the design and the implementation of a framework to schedule and execute tasks in a nanosatellite constellation. The framework was implemented as a set of libraries and scripts in the Python programming language. It includes modules to define scenarios and tasks, calculate the constellation contact list from the satellite Two-line Elements Set (TLE)s and target locations, calculate valid contact plans using a genetic algorithm, and obtain the flight plan that solves the defined task. In addition, the framework can be extended to set new constraints and optimization goals.

- **Flight software for nanosatellite constellations:** As a way to close the gap between the problem definition of this work and the practical deployment of future nanosatellite constellation, this work presents a nanosatellite flight software designed to work with the proposed control framework and to meet the principal requirements of CubeSat constellation missions. This work also proposes a verification technique, based on the visual software architecture tracking, to ensure quality requirements are fulfilled even in nanosatellite massive production. The software was implemented in the C programming language, ported to the FreeRTOS and GNU/Linux operating systems, and licensed as a Free/Libre and Open Source Software (FLOSS) project to be used in a wide range of CubeSat projects.
- **Nanosatellite constellation software in the loop simulation suite:** The software in the loop simulation suite is used to validate the results of the control framework. This suite uses the proposed flight software and the results of the control framework as input. Implemented in the Python programming language, the suite interacts with several flight software instances simulating satellites or ground control nodes. The simulation results make it possible to evaluate the proposed solution's performance. This suite is intended to be used in future production environments to accurately predict the constellation system's functioning before executing a task.

Only Free and Open Source Software (FOSS) projects were used in this work. Hoping to contribute to these efforts, all the software results of this work are available in a GitLab repository and licensed as FLOSS. The intention is to expand the CubeSat community's tools into developing better and complex missions in the future.

1.7. Publications

1.7.1. Scientific journals (ISI)

- **C. E. Gonzalez**, C. J. Rojas, A. Bergel and M. A. Diaz, *An Architecture-Tracking Approach to Evaluate a Modular and Extensible Flight Software for CubeSat Nanosatellites*, in IEEE Access, vol. 7, pp. 126409-126429, 2019, doi: 10.1109/ACCESS.2019.2927931.

    - In this publication, the author contributed to the complete design and implementation of novel flight software for CubeSats. The author also studied state of the art, defined the software requirements, performed tests and analyzed the results. The visualization tool as wheel as the software implementation was developed in collaboration with the co-authors. This work is relevant for this thesis and the CubeSat program because it facilitates the development of the mission, payloads, and derived works. It also opens the ability to operate constellations, as explored in this Ph.D. thesis.

- T. Gutierrez, A. Bergel, **C. E. Gonzalez**, C. J. Rojas, and M. A. Diaz, *Systematic Fuzz Testing Techniques on a Nanosatellite Flight Software for Agile Mission Development*, in IEEE Access, vol. 9, pp. 114008-114021, 2021, doi: 10.1109/ACCESS.2021.3104283.

    - In this publication, the thesis's author contributed to designing the flight software and guidelines to interact with external tools such as the fuzz testing system. In collaboration with the co-authors, analyzed and solved the errors found by the novel testing techniques and helped characterize the errors. Thanks to previous collaborations with other CubeSats

teams, the author also provided guidelines to implement this testing technique in third-party flight software. This work is relevant to the Ph.D. thesis because it also studies agile and automated techniques to develop satellites which are key to deploying large constellations.

1.7.2. International conferences and workshops

- **C. E. Gonzalez**, A. Bergel and M. Diaz, *Nanosatellite constellation control framework using evolutionary contact plan designs* Conference Paper. Presented in the 8th IEEE International Conference on Space Mission Challenges for Information Technology (SMC-IT), 2021.
- M. Diaz, J. Rojas, **C. E. Gonzalez**, C. Rojas, E. Obreque, I. Portillo, *Preliminary analysis of the new space communication scenario: the ground segment perspective*. In proceedings. 2020 IEEE Biennial Congress of Argentina (ARGENCON). 2020.
- T. Gutierrez, A. Bergel, **C. E. Gonzalez**, C. J. Rojas, and M. A. Diaz, *Toward Applying Fuzz Testing Techniques on the SUCHAI Nanosatellites Flight Software*. In proceedings. 2020 IEEE Biennial Congress of Argentina (ARGENCON). 2020.
- Diaz, M. A. ; **Gonzalez, C**. ; Moya, P. S. ; Martinez Ledesma, M., *Preliminary results of the first year of operation of the SUCHAI-1 Cubesat: Langmuir probe and particle counter measurements*. Poster. American Geophysical Union, Fall Meeting. 2018.
- **C Gonzalez**, C Rojas, A Becerra, J Rojas, T Opazo, M Diaz, *Lessons Learned from Building the First Chilean Nanosatellite: The SUCHAI Project*. In proceedings. 32nd Annual AIAA/USU Conference on Small Satellites. 2018.

1.8. Outline of the thesis

This Ph. D. thesis continues in Section 2 presenting a background of concepts and state of the art regarding nanosatellite constellations, task scheduling problems, delay or disruptions tolerant networks, and nanosatellite flight software. Then, the proposed constellation control framework is described in detail in Section 3. Section 4 present the results of applying the framework to case studies. Finally, conclusions and future work are presented in Section 5.

Chapter 2

Background

This chapter presents the main topics related to this doctoral thesis. From general to specific, this chapter describes definitions and state of the art about large satellite constellations, task scheduling in satellite constellations, Delay and Disruption Tolerant Networks, and nanosatellite flight software.

2.1. Large satellite constellation

Distributed Space Missions (DSMs), such as formation flight and constellations [23], are being recognized as key solutions to enhance satellite services by increasing the spatial and temporal measurement frequency or providing global communication networks. According to the satellites' spatial relationship [23], the most popular constellations distributions are the Walker Delta and the Walker Star geometries [4]. Both geometries are shown in Fig. 2.1. However, it is unlikely that a large or mega CubeSat constellation can be deployed with a specific geometry. Instead, Ad-hoc configurations resulting from several secondary payload launching opportunities have been used more frequently. Marinan *et al.* (2013) [6] analyze this case and show that it is possible to achieve a global coverage earth-observing constellation from successive secondary launch opportunities with performance comparable to uniform Walker constellations.

As shown in Table 2.1, mega-constellations consisting of hundreds and thousands of small-

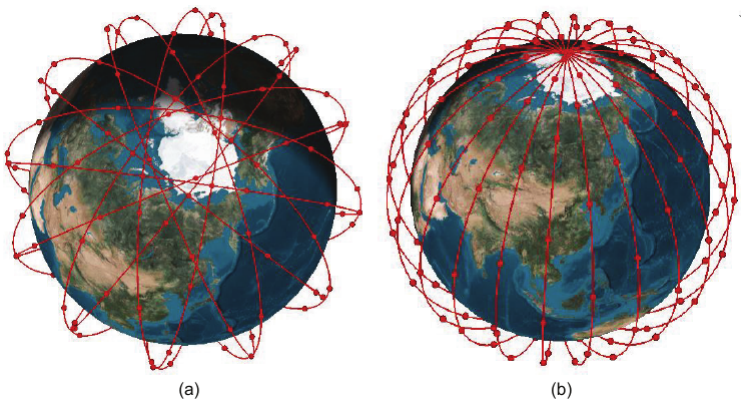


Figure 2.1: Constellation geometries: a) Walker Delta constellation; b) Walker Star Constellation [4]

Table 2.1: Review of small and nano-satellite constellations

Project	N° Sat.	Orbit alt.	Mass	Frequency	ISL	Usage
Telesat	117	1000-1248 km.	168 kg*	Ka-band	No	Broadband communications
OneWeb	720	1200 km.	145 kg.	Ku-, and Ka-bands	No	Broadband communications
StarLink	4425	1110-1325 km.	260 kg.	Ku-, and Ka-bands	Yes	Broadband communications
Astrocast	64	450-600 km.	4 kg. (3U)	L-band	Yes	IoT
Fleet	100	NA	NA (1.5U, 3U, 12U)	NA	Yes	IoT
Kepler	140	500-650 km.	5 kg. (3U)	Ku-, and Ka-bands	Yes	Satellite backhaul
Planet	160*	390-500 km.	5 kg (3U)	S-, and X-bands	No	Earth-observation

*: Estimation from available information
NA: Not available in public sources

satellites have started to be deployed by private companies such as Telesat, OneWeb, and SpaceX with 117, 720, and 4425 planed nodes respectively [9, 24]. Nanosatellites are also part of this trend. Akyildiz *et al.* (2019) [8] summarizes the Astrocast, Fleet, and Kepler commercial constellations with 64, 100, and 104 proposed nanosatellites, respectively. Meanwhile, Planet company has launched more than 160 earth-observing 3U CubeSats [25]. The future of nanosatellite constellations will be heavily influenced by current research and technology demonstration missions. For example, Bandyopadhyay *et al.* (2016) [26] presents a review of 39 research and technology demonstration of nanosatellite constellations projects. Su *et al.* (2017) [27] study a global communication constellation with CubeSats, and Kak *et al.* (2019) [28] proposes a framework to design and evaluate large-scale CubeSat constellation for IoT applications.

However, due to the commercial nature of the mentioned constellation projects, there is little technical information about hardware and software capabilities, constellation control strategies, communication protocols, among others. As detailed in table 2.1 it is possible to find information about orbits and frequency allocations thanks to the communication regulations.

2.2. Task scheduling in satellite constellations

In satellite constellations, mission operations scheduling or task scheduling means assigning system resources to distribute the operation's workload. System resources are satellites, instruments, ground stations, or inter-satellite links, while tasks can refer to earth observation activities, provide communications services, or operate scientific instruments. This problem may imply mapping discrete temporal resources, or time slots, optimally considering technical and economic constraints.

In earth-observing satellite constellations where satellites work as a gathering, storing, and downloading data system, the problem is defining the time, location, and satellites to take pictures to cover one o more geographical areas. Constrains are related to satellite maneuvering capabilities, optical sensor capabilities, energy, memory consumption, meteorology, and ground station availability. Lemaitre *et al.* (2002) [14] describes the formulation of the Earth-Observing satellite scheduling problem and explores exact algorithms to solve it, showing that the solution is computationally hard. Cho *et al.* (2018) [29] delve into a two-step linear programming approach to solve both imaging and data downloading schedule, including energy constraints; they simulated results for a constellation of up to 12 satellites. The onboard scheduling problem in a bi-satellite cluster is studied by Chu *et al.* (2017) [30] where a low-resolution satellite sets targets to the high-resolution satellite, which generates online planning; an Anytime Branch and Bound Algorithm was applied. The work of Kennedy *et al.* (2017) [18] evaluate two algorithms, Resource-Aware SmallSat Planner (RASP) and Limited Communication Constellation Coordinator (LCCC), to coordinate a constellation of earth-observation CubeSat satellites with ISL. Both RASP and LCCC algorithms perform online and onboard planning of imaging tasks, sharing the planning with the constellation using the ISL. They run simulations for Plain Walker, Stitched Walker, and Ad-hoc constellations geometries up to 18 satellites. Also, in the field of CubeSats, Nag *et al.* (2018) [11] present a framework for scheduling the attitude control operations for a constellation of small Earth-observing satellites using Dynamic Programming and Mixed Integer Linear Programming (MILP) algorithms; up to 4 satellites were simulated. Another approach is the usage of meta-heuristic such as evolutionary algorithms to solve scheduling problems in satellite constellations. Its simplicity and efficiency can be an advantage in finding high-quality solutions to these kinds of optimization problems. For example, Li *et al.* (2007) [31] uses a combination of Genetic Algorithm (GA) and Simulating Annealing to solve selecting and scheduling tasks of agile earth-observing satellites. Yuan *et al.* (2014) [32] used a similar approach but provided high-quality starting solutions to the GA improving the overall performance.

Communication satellite constellations, especially those equipped with inter-satellite links, may need to find the optimal path to transfer data from one point to another, considering restrictions such as link capacity, antenna pointing, energy consumption, among others. The work of Spangelo and Cutler (2015) [33] solves the single-satellite, multi-ground station communication scheduling problem to maximize the total amount of data downloaded from space focused on small satellites. Jia *et al.* (2017) [34] propose a collaborative scheme that allows satellites to offload data among themselves using inter-satellite links before they come to contact the ground station. They used an iterative algorithm to solve the optimization problem, and the schema was tested with simulations in the Globalstar and Iridium constellations. The work of Deng *et al.* (2018) [35] analyses the scenario of user satellites and data relay satellites to propose a two-phase dynamic task scheduling method. In this method, the first step uses a GA to solve the scheduling problem and a second step manages the possible disturbances in the initial task scheduling by preemptive task-switching. The simulation scenario includes 3 data relay satellites and 8 user satellites. Bisgaard *et al.* (2019) [36] presents a CubeSat-related work where a battery-aware scheduling problem for the GomX-3 satellite is studied. In this work, a two-step procedure to perform task scheduling is formally presented using a timed automata model. Battery models and scheduling results are contrasted with the satellite in-orbit operation. Later, Fraire *et al.* (2020) [37] studied the scalability of the batter-aware scheduling problem solving the MILP with a commercial solver. They were able

to solve scenarios up to 50 satellites.

In summary, even the simplest versions of these scheduling problems are MILP, and hence NP-hard class [14]. Exact algorithms such as Greedy Algorithms (GA) [18, 29] or Dynamic Programming (DP) [11]. However, due to these algorithms' complexity, the solution can be impractical for large constellations of thousands of nodes. On the other hand, heuristics approaches such as Evolutionary Algorithms, Genetic Algorithms [31, 38] have been used with promising results.

2.3. DTN and contact plan design

In contrast with ground networks based on the TCP/IP protocol stack where continuous end-to-end connectivity is assumed, in a constellation of LEO satellites with ISL connectivity among nodes could be sporadic. The latter is an example of Delay or Disruption Tolerant Network (DTN). These networks use a store-carry-forward message scheme, so each node requires non-volatile memory to store messages until the next hop is available. Despite the time-evolving nature of the connections in LEO satellite networks, the contact opportunities can be predictable due to orbital mechanics [15].

The work of Fraire *et al.* (2015) [15] defines a contact as the opportunity to establish a temporal communication link among two DTN nodes when physical requirements are met. They also define the *contact topology* as the set of all feasible contacts within an interval and explain two ways to model the topology: Contact List (CL) and Finite State Machine (FSM) representations. As shown in Fig. 2.2 in the CL representation each contact is in the form of a source, destination, start time, and stop time table. For example, its possible to observe a contact between node N_1 and node N_2 starting at 608 s. and finishing at 2228 s. In the FSM representation, the contact list is discretized in states k_1 to k_7. Each state describes the contacts available in a time window. In the example $c_{2,3,4}$ represent a contact between node N_3 and N_4 at state k_2. Is is always possible to translate from CL to FSM and *vice versa*.

The process of selecting the contact set that meets a specific problem's communication requirements is called Contact Plan Design (CPD). The resulting contact set is defined as Contact Plan (CP) [15]. Depending on the specific problem to be solved, different strategies Contact Plan Design (CPD) have been proposed in the literature; however, as the number of satellites in the constellation increases, meta-heuristics approaches as evolutionary algorithms have shown promising results [38].

2.4. Nanosatellites flight software

Flight Software (FS) refers to the software system that controls the spacecraft operations and is executed by the On-Board Computer (OBC), usually a real-time embedded system. The FS implements most of the mission operational requirements and is a key element to facilitate the development and operation of satellite constellations [39, 40]. Significant efforts have been developed to provide better FS for nanosatellites, as detailed in Table 2.2. Also, Miranda *et al.* (2019) [21] present a survey of available flight software frameworks for the New Space era [2]. They show that nowadays, it is possible to find ready to use FS solutions for CubeSats being the NASA Core Flight System (cFS) [41] and the Kubos initiative [42],

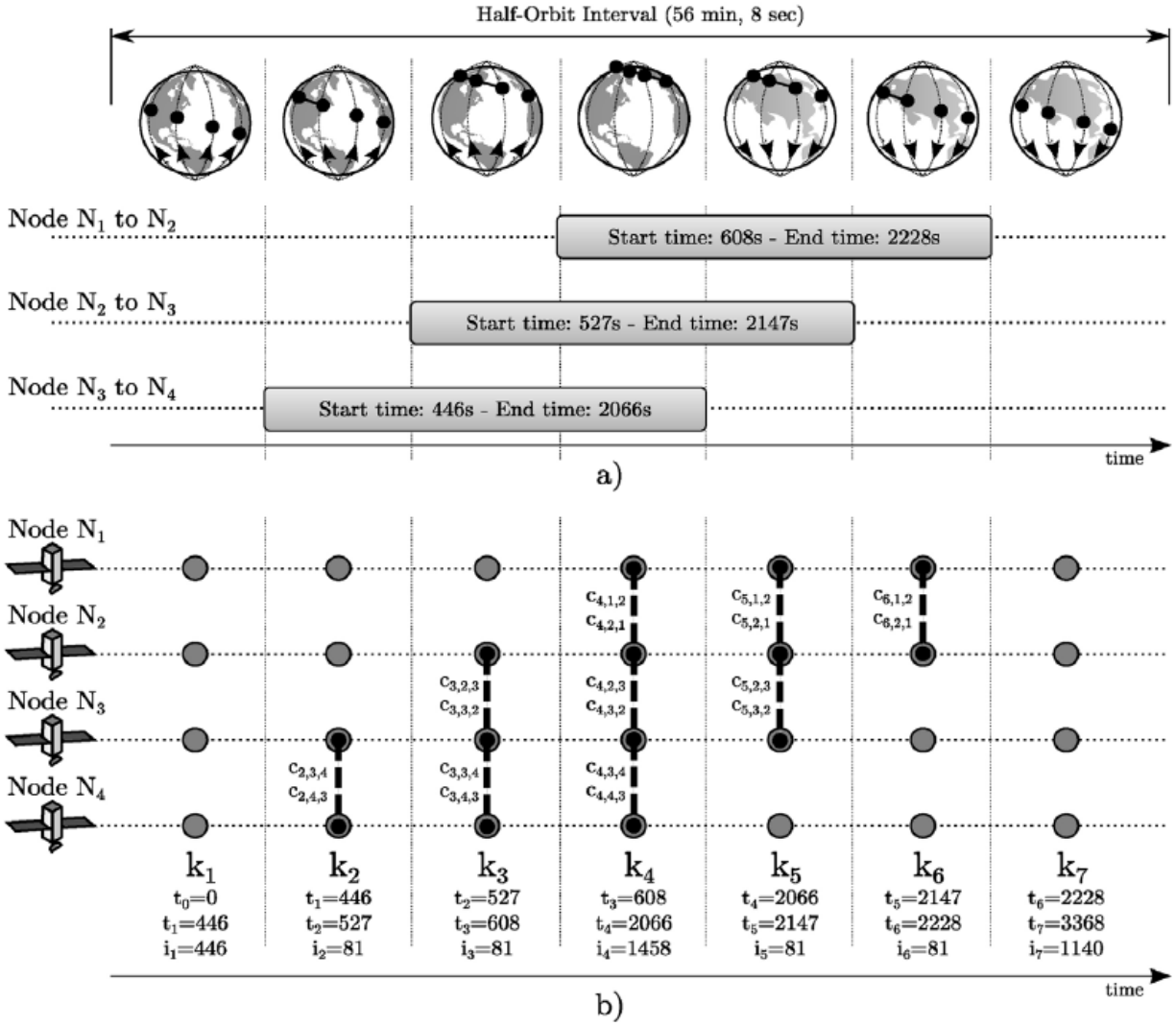


Figure 2.2: Example of a DTN contact topology modeled as: a) CL; b) FSM [38].

some of the well documented and FOSS alternatives.

CubeSats are using a variety of On-Board Computers (OBCs) and Operating System (OS) from 16-bits microcontrollers running FreeRTOS to modern ARM-Cortex microprocessors capable of running GNU/Linux. Undoubtedly, the processor's selection, the OS, and the programming language are closely related, and this decision directly affects other mission variables. On the one hand, if the FS requires a high-level programming language or entirely depends on features available only in GNU/Linux, then a more powerful processor is needed, which impacts the power consumption. On the other hand, using an OS for embedded systems can save power at the cost of the processing capabilities and limit the availability of developers since they are required either to know or to learn a low-level programming language.

Most of the articles and missions listed in Table 2.2 and Miranda *et al.* (2019) [21] present some level of software architecture definition. State machines, component-based, and

service-oriented architectures, messaging systems-based architectures, and command-centric architectures are proposed. They also present some non-functional requirements such as modularity, extensibility, flexibility, robustness, and fault-tolerance [43] as relevant features of the FS for nanosatellites. Non-functional requirements refer to the quality attributes of a system [43]. Beyond the functional aspects of a satellite mission, it is necessary to define some design guidelines that affect architectural decisions, especially in the context of agile, flexible, and fast-growing CubeSat projects. This context makes non-functional requirements relevant in the design and development phases. Except for NASA cFS and Kubos, all the reviewed works are not open source or do not report the actual implementation details. Without this information, it is challenging to evaluate software quality criteria beyond the design phase; moreover, as far as we know, there is no standard and low-cost methodology to verify software quality criteria for space systems neither in an agile fashion nor in real-time. Araguz *et al.* [19] present three structured criteria to develop nanosatellite FS: robustness through hierarchy, payload-oriented modularity, and onboard planning capabilities. The cFS developers have used, among others, unit testing and graphical tools to verify the architecture and quality of the software [44, 45] but these tools are not continuously integrated into the development process in a way that might allow real-time monitoring of the architecture after the contribution of different developers.

The numbers of satellites proposed for the coming constellations are unprecedented [60, 61]. It is critical for current and future FS solutions to facilitate satellites' mass production and the operation of a large number of spacecraft. Therefore, it is imperative that the architectural guidelines and declared quality features of the FS, such as modularity, flexibility, and extensibility, could be evaluated in an agile manner. That is, tracking software quality during the development and integration phases instead of delegating these analyses to the final phases.

2.5. Chapter highlights

In this section, nanosatellite constellation operation challenges and state of the art have been discussed, and the following conclusions are extracted.

- A nanosatellite constellation will likely be constructed by successive secondary launches, creating an Ad-hoc configuration instead of a Walker configuration.
- Task scheduling in satellite constellations is an NP-hard optimization problem. Although different exact algorithms have been explored in the literature, heuristics approaches such as evolutionary algorithms may scale better in large constellation scenarios.
- A LEO constellation of nanosatellites with ISL can be modeled as a delay and disruption tolerant network. Using the constellation contact topology, designing a contact plan that meets certain design criteria is possible. Evolutionary algorithms have been used to design contact plans.
- Nanosatellite flight software is not standardized. There is a variety of solutions, architectures, and design goals reported in the literature. There is a lack of flight software design for large nanosatellite constellations, including agile methods to test and verify software quality.

Table 2.2: Review of flight software architectures used in CubeSat projects

Project	Architecture details	OS supported	Language	Hardware supported	Source code	License
PilsenCUBE[46]	State machine	N/A	C	NXP LPC2148	No	N/A
Delfi-n3Xt[47]	Layered. State machine. State machine in the application layer.	N/A	C	TI MSP430F1611	No	NI
RACE, ARMADILLO[48]	Layered. Component based modules. State machine to execute modules functionalities in the application layer.	GNU/Linux	C, C++	NXP LPC3250	No	N/A
UWE-2[49]	Centralized. Modules controlled by a central module.	uClinux	C	Hitachi H8	No	N/A
Kysat[50]	Layered. Component based modules. Centralized tasks organization.	Salvo RTOS	C	TI MSP430F1611	No	N/A
Kysat-2[51]	Layered. SPA distributed messaging system.	SPA middleware	NI	SL 8051F930	No	N/A
PolySat[52]	Modules separated in processes, inter process communications with UDP sockets	GNU/Linux	C	Atmel AT91SAM9G20	No	N/A
ESTCUBE-1[53]	Layered. Modules as independent tasks.	FreeRTOS	C	STM32F1	No	N/A
WinCube[54]	Layered. Modules as independent tasks.	Salvo RTOS	C	TI MSP430F169	No	N/A
Asundi *et al.*[55]	Distributed. Functionalities distributed across two microprocessors.	NI	NI	MSP430, TI C6000	N/A	N/A
3Cat-1[56]	Layered. Two high level layers: System Core and Process Manager. Modules are threads with messaging system and task scheduler.	Linux	Prolog	AT91SAM9G20	No	N/A
NUTS[57]	Layered. Service oriented. Inter task and inter processor communications with CSP.	FreeRTOS	C	Atmel AVR32UC3, SAMV71	Yes	NI
CubETH[58]	Component-based model, verified and validated with BIP framework.	N/A	C, C++	SL EFM32GG880	No	N/A
EQUULEUS, PROCYON[59]	Layered. Command Centric Architecture (C2A).	N/I	C, C++	N/I	No	N/A
NASA Core Flight System[41]	Layered. Service-oriented. Publisher-Subscriber inter task messaging system.	GNU/Linux, VxWork, RTEMS	C	x86, RAD750, MCP750, Coldfire, and others	Yes	NASAs Open Source Agreement
Kubos[42]	Layered, N-thier architecture. Service oriented.	GNU/Linux	Rust, Python, C	BeagleBone Black (Cortex A8), ISIS OBC (ARM9)	Yes	Apache 2.0
Brightascension GenerationOne	Layered. Component-based framework. Component generator using XML definitions.	GNU/Linux, FreeRTOS, RTEMS	XML, C	Nanomind A712(ARM7TDMI), Clayspace OBC (FPGA based ARM Cortex M3), BeagleBone Black (Cortex A8), TI MSP430, Vorago VA10820, Xiphos Q7	No	Commercial

N/A: Not applicable. NI: No information available

- There is a lack of solutions integrating all the ingredients necessary to operate a nanosatellite constellation with a certain level of automation.

Chapter 3

Nanosatellite constellation control framework

This chapter describes the design of a nanosatellite constellation operation framework to schedule, distribute and execute tasks automatically. First, the framework's global design is presented, including the definitions of scenarios, tasks, and contacts. Second, an evolutionary algorithm to generate contact plans is described. Third, this section presents the restrictions of the problem studied and the genetic algorithm details. Finally, the SUCHAI Flight Software design is discussed in detail. This software was designed to execute the global flight plan derived from the evolutionary contact plan design. A constellation software in loop simulation suite to validate the proposed scheduling is also presented.

3.1. Constellation control framework

Figure 3.1 shows a scenario with three nanosatellites and two ground stations. Satellites are equipped with some earth observing payload and inter-satellite communication. Let us consider that the satellites and ground station software support the execution of commands scheduled in the flight plan table. Also, let us define a task as taking an instrument's data over a certain location and downloading that data in the designated ground station in the shorter possible period. Two key variables must be considered to solve this problem: commands and contact times. Commands are satellites and ground stations' actions, including sending data to nodes, taking data from instruments, pointing, and attitude maneuvers. Contact opportunities may refer to two ideas: the instants where nodes can establish radio links or targets visible to nodes. These two variables can be expressed in a global flight plan that includes the time, node, and command to be executed to accomplish a task. This flight plan must be distributed to all nodes using the inter-satellite links.

Thus, the constellation control framework aims to generate a global flight plan that solves a specific task for a certain scenario. Generating a valid flight plan is a scheduling problem. Variables such as delivery time, starting time, or resource usages are optimized under restrictions such as contacts feasibility and node capabilities. This work uses an evolutionary contact plan design approach to solve the scheduling problem and generate a valid flight plan. Fig. 3.2 shows the general framework functioning scheme. It consists of three main modules: a contact list generator, the contact plan design, and the flight plan generator. The contact list generator module uses the scenario definition to determine future satellite to

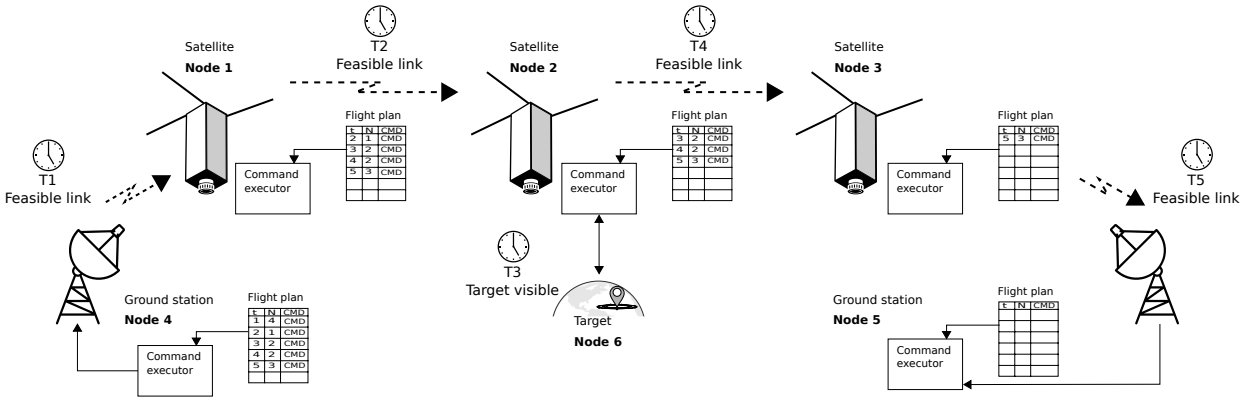


Figure 3.1: Constellation working scheme

satellite, satellite to ground stations, and satellite to targets contacts; satellite capabilities such as communication system and payload instruments parameters are used to define a feasible contact. The contact plan design module searches for a valid solution, according to the restrictions defined in the task, in the space of all contact opportunities; thus, solving the scheduling problem. Finally, the selected contact plan is translated into a flight plan containing the commands that nodes must execute to complete the proposed task. The command sequence's validity and feasibility are analyzed through the software on the loop simulation of the constellation; unfeasible solutions are rejected, and a new solution is obtained. Satellites operation is simulated using the flight software (FS), so its architecture and functioning will be explained in detail.

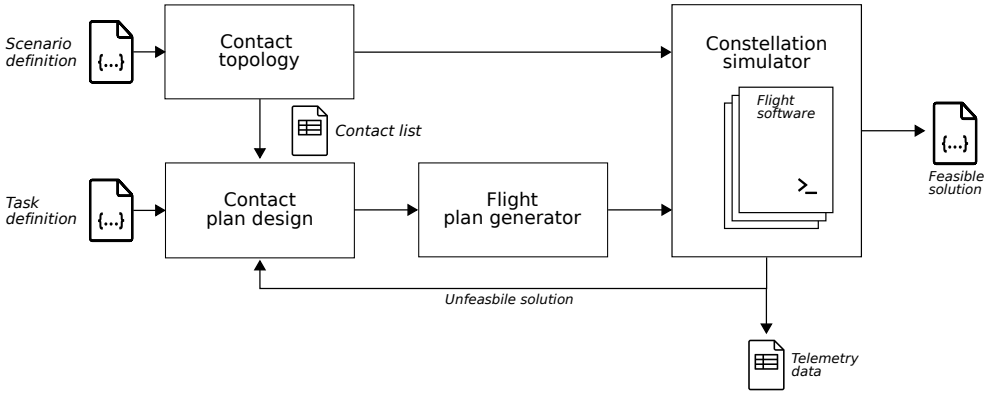


Figure 3.2: Constellation control framework block diagram

3.1.1. Scenario and task definitions

Scenarios are objects that describe the time constraints, the satellites in the constellation, the ground station, and the targets of the simulation case. As described in Table 3.1, satellites are defined by their TLE, node number, and id. Targets and ground stations are defined by their spatial coordinates and id. This object can also store a cache with calculated tracks and contacts files.

Table 3.1: Scenario definition

Scenario object definition		
field	**type**	**description**
id	string	Scenario identifier
start	number	Scenario start time in seconds
duration	number	Scenario duration in seconds
step	number	Time resolution in seconds
tracks	string	Path to file with satellite tracks, or null
contacts	string	Path to file with the contact list, or null
satellites	list	List of satellite objects
stations	list	List of ground station objects
targets	list	List of targets objects

Ground station object definition		
field	**type**	**description**
id	string	Ground station id or name
node	number	Node number in the constellation
lat	number	Latitude coordinate
lon	number	Longitude coordinate
alt	number	Altitude coordinate

Satellite object definition		
field	**type**	**description**
id	string	Satellite id, name or catalog number
node	number	Node number in the constellation
tle1	string	TLE line 1
tle2	string	TLE line 2

Target object definition		
field	**type**	**description**
id	string	Target id or name
node	number	Node number in the constellation
lat	number	Latitude coordinate
lon	number	Longitude coordinate
alt	number	Altitude coordinate

Task objects define what actions must be performed by the constellation. As shown in Table 3.2, a task definition includes the starting node (the ground station that starts the task execution), the list of targets to visit, and the commands to execute on it (usually points where collect data from an instrument), and the end node where result data is retrieved (the ground station where data will be downloaded). This object also stores the solution (flight plan).

Table 3.2: Task definition

Task object definition		
field	**type**	**description**
id	string	Task id
start	string	Ground station object id
end	string	Ground station object id
targets	list	List of task target objects
solution	string	Flight plan file path, or null

Task target object definition		
field	**type**	**description**
id	string	Target id
command	string	Command to execute
result	string	Result data id
prio	int	Target priority

An example scenario and task definition as JSON files are shown in Listing 3.1 and 3.2. These files describe a scenario with three satellites, two ground stations, and one target. The task consist on execute the `take_data` command over the `SAA` node, starting from the Santiago (`stgo`) ground station and ending in the Tokyo (`tokyo`) ground station.

Listing 3.1: Example scenario definition JSON

```
1 {
2     "id": 1,
3     "start": 1588054885,
```

```
    "duration": 43200,
    "step": 30,
    "satellites": [
        {"id": "1244", "node": 1, "tle1": "1 45196U 20012U    20094.58334491 -.00029761  00000-0
        ↪ -20303-2 0  9999", "tle2": "2 45196  52.9956  50.6558 0001754  72.9283 341.9461 15.05609076
        ↪ 1716"},
        {"id": "1102", "node": 2, "tle1": "1 44920U 20001G    20094.58334491  .00050489  00000-0
        ↪ 34604-2 0  9999", "tle2": "2 44920  53.0009 350.6416 0001892  65.4641  85.7463 15.05571452
        ↪ 13382"},
        {"id": "1032", "node": 3, "tle1": "1 44737U 19074AA   20094.58334491 -.00005148  00000-0
        ↪ -33442-3 0  9997", "tle2": "2 44737  53.0002 210.6425 0001072  82.8131 134.9394 15.05576243
        ↪ 22184"}
    ],
    "stations": [
        {"id": "stgo", "node": 4, "lat": -33.3833, "lon": -70.7833, "alt": 476},
        {"id": "tokyo", "node": 5, "lat": 35.6830, "lon": 139.7670, "alt": 5}
    ],
    "targets": [
        {"id": "saa", "node": 6, "lat": -15.0, "lon": -15, "alt": 0}
    ],
    "tracks": "logs/track__1__1588054885.csv",
    "contacts": "logs/contacts__1__1588054885.csv"
}
```

Listing 3.2: Example task definition JSON

```
{
    "id": 1,
    "start": "stgo",
    "end": "tokyo",
    "targets": [
        {"id": "saa", "command": "sim_take_data data1", "result": "data1", "prio":  1}
    ],
    "solution": null
}
```

3.1.2. Contact list generation

Contacts are generally described as the communication opportunities between two nodes in the satellite constellation. The CL is the summary of all communication opportunities for a given scenario. This list is obtained by propagating satellite orbits and calculating their proximity based on their communication system capabilities or assumptions. The CL can be expressed as table with *(from, to, start, end)* entries or in the FSM representation [15]. Figure 3.3 shows an example topology diagram and the contact list in its FSM representation. In this work, not only the satellites are included in the contact topology but also the ground stations and targets. The main caveat is that contacts with targets do not produce data transfers; these contacts are only used as opportunities to get data from instruments and not as real data communication opportunities. In detail, three types of contacts and conditions are defined:

1. **Satellite to satellite**: Omnidirectional antennas were considered in the inter-satellite

links. Thus, an inter-satellite contact occurs when the euclidean distance between satellites is in an N km range.

2. **Satellite to ground stations**: Ground stations are defined by their latitude, longitude, and altitude coordinates. A feasible contact is defined by the satellite footprint and the minimum elevation angle α of the ground station.

3. **Satellite to targets:** As ground stations, targets are also defined by their latitude, longitude, and altitude coordinates. In this case, a contact is defined by the footprint of the satellite instrument over the target. For example, in earth-observing satellites, the contact is defined by the camera footprint. It is also possible to define contacts with high attitude targets, for example, magnetometers, plasma instruments, or particle counters, interested in taking samples over particular areas such as the poles or the South Atlantic Anomaly (SAA) [62]. This type of contact differs from the other two because no data is exchanged (from the telecommunications point of view), and this restriction must be taken into consideration to generate the CP

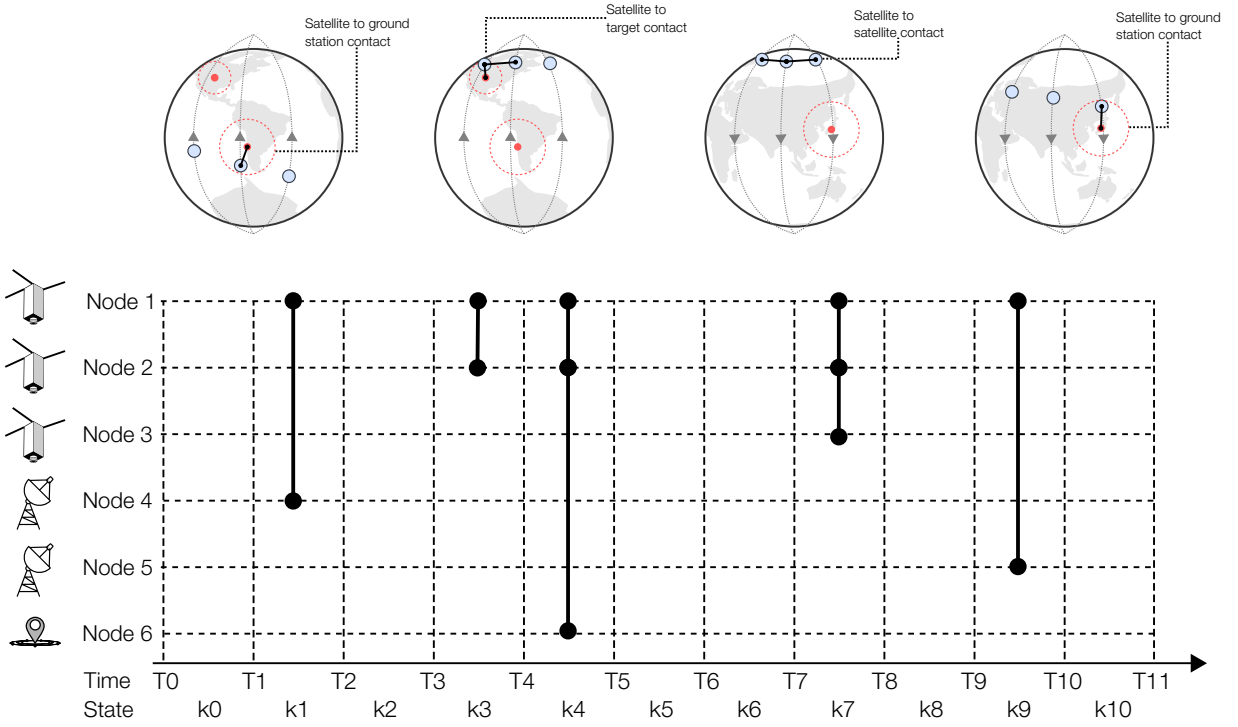


Figure 3.3: Contact topology and using it FSM representation. It includes ground stations and targets as nodes

3.1.3. Contact plan design

A nanosatellite constellation with ISL can be considered a DTN, so it is possible to use the contact information to schedule which nodes are used in a task. In a DTN, for a given CL there are many possible paths to visit the targets defined in a task. A valid set of contacts is defined as a CP and the process to find a CP under certain restrictions is called CPD.

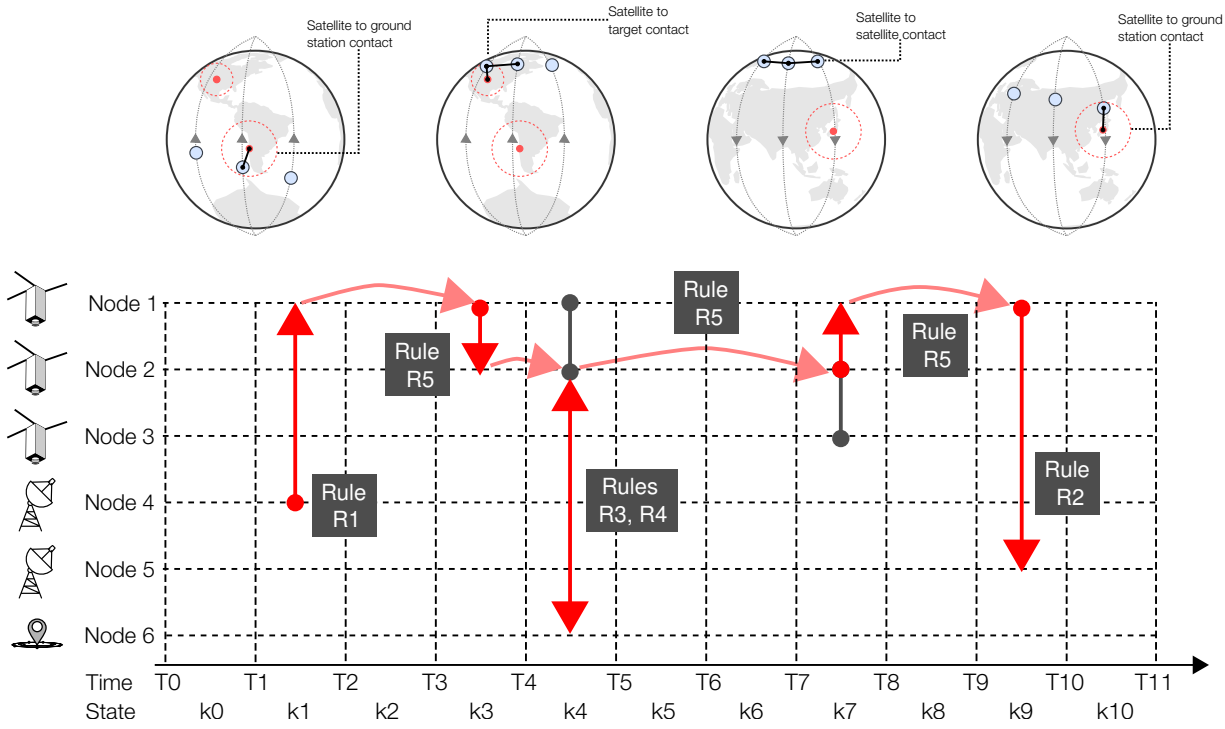


Figure 3.4: Contact Plan Design rules and example Contact Plan

Using the Contact List FSM representation (see Fig 3.3), the CPD consists on selecting a sequence of nodes $S = [S_1, ..., S_L]$ at states $K = [k_1, ..., k_L]$. For example, for the task defined in Listing 3.2 the Fig. 3.4 show a valid CP that selects the contacts of the sequence $S_1 = [4, 1, 2, 6, 2, 1, 5]$ at states $K_{S1} = [k_1, k_3, k_4, k_7, k_9]$. Note that this is not the only valid sequence, and that in real scenarios the solution space grows significantly.

Using this example CP, it is possible to define the **delivery time** of the task as $D_{S1} = T_10 - T_1$, the sequence length $L = 7$ and the **validity** of the solution $V = 0$. These variables are described below:

1. **Validity (V):** Because this work includes the targets and ground stations in the CL, not all sequences of contacts are valid and some rules are added to the CPD to determine if a given CP is valid or not:

   R1. The CP must start in the ground station defined by the task.

   R2. The CP must end in the ground station defined by the task.

   R3. The CP must visit all targets defined in the task.

   R4. Targets are not data relays. That is, if a contact from satellite A to target T occurs, the next contact must start from the same satellite A

   R5. Satellites are data relays. That is, if a contact between satellite A to satellite B occurs, the next contact must start from the satellite B.

   If one or more of these rules are not satisfied in a CP, it is defined as invalid. Thus, validity is defined as the sum of contact rules not satisfied per state k of the CP: $V_k = 0$

if R1. to R5. are meet, else $V_k = 1$. So, the sum of contacts validity $V = \sum_{k=1}^{n} V_k$ must be zero to define a sequence as valid.

2. **Delivery time (D):** Delivery time is defined as the total time the task takes to execute. That is the time between the state k start time (T_k^i) and the state l end time (T_l^f). Then, $D = T_l^f - T_k^i$.

3. **Number of contacts (L):** The length of the solution or the number of contacts used in the contact plan $L = length(S)$.

Thus, the CPD is an optimization problem, primarily over the *delivery time* variable, subject to the *validity* of the solution.:

$$minimize : D = T_l^f - T_k^i \tag{3.1}$$

$$subject\ to : V = \sum_{k=1}^{L} V_k = 0 \tag{3.2}$$

$$0 < D \leq T_n^f - T_0^i \tag{3.3}$$

$$0 < L \leq n \tag{3.4}$$

$$\tag{3.5}$$

Different approaches can be used to solve the CPD problem; however, as the number of nodes increases, classical optimization techniques are not practical and evolutionary algorithms have been proposed in the literature [38]. In this work, a genetic algorithm is used to find a CP.

3.1.3.1. Genetic algorithm

The proposed GA for the Constellation Control Framework CPD is designed to generate multiple CP candidates and evaluate its *validity* and optimize the *delivery time* function. The algorithm is designed to find valid contact paths in an evolutionary fashion, a problem similar to finding the escape route in a labyrinth. Then, the algorithm focuses on minimizing the cost of this path. This approach is described in Algorithm 1 and throughout this section.

The algorithm requires the scenario and task definitions as inputs because this information will be used to generate the initial population according to the encoding and validity rules (See section 3.1.3.2 and 3.1.3.4. The CL is used to evaluate the individuals in the fitness function to contrast the CP sequence with actual contact opportunities. Thus, infeasible or low-quality solutions will be discarded in the evolution (See section 3.1.3.3). Other parameter are: L, the sequence length; P_size, the population size; $Iter$, the maximum number of iterations; p_{mut}, the mutation probability; and p_cr, the crossover probability. Individuals are generated as random sequences, but they are fixed using the task information, so individuals always contain the start, target, and end node described in Section 3.1.3.4. Then, a population is evaluated obtained the $Validity$ and $Fitness$ values. As described in Section 3.1.3.3, $Validity$ represent how feasible is the solution and $Fitness$ evaluates the *delivery time*. During the tournament, individuals are sorted first by $Validity$ and then by $Fitness$; thus, the algorithm first finds feasible solutions and then optimizes the *delivery time*. The old population is replaced in the next iteration with the best individuals, new individuals product of

the crossover operation, or a mutation according to the probabilities and operators described in Section 3.1.3.6 and 3.1.3.5. Finally, after $Iter$ iterations or the stop condition described in Section 3.1.3.4 the algorithm returns the best individual.

Algorithm 1: Genetic algorithm

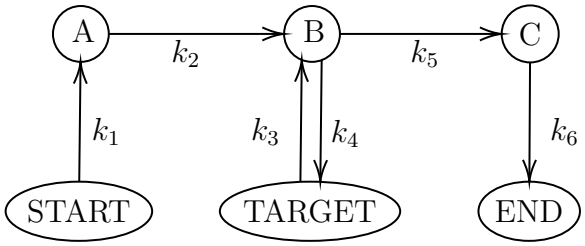
Input: $Scenario$, $Task$, CL, L, P_{size}, $Iter$, p_{mut}, p_{cr}
Output: CP
// Create initial population
for $i \in P_{size}$ **do**
    $indv \leftarrow random\_sequence(0, Scenario.nodes, L)$
    $P[indv] \leftarrow fix\_individual(indv, Task)$
// Run until max. iterations reached
for $Iter$ **do**
    // Evaluate population
    for $\forall indv \in P$ **do**
        $Valid[indv] \leftarrow evaluate\_valid(indv, CL, Task)$
        $Fitness[indv] \leftarrow evaluate\_fitness(indv, CL)$
    // Population replacement
    for $\forall indv \in P$ **do**
        $parent_1, parent_2 \leftarrow tournament(P, Valid, Fitness)$
        $indv\_new \leftarrow parent_1$
        if $random(0,1) \leq p_{cr}$ **then**
            $indv\_new \leftarrow crossover(parent_1, parent_2)$
        if $random(0,1) \leq p_{mut}$ **then**
            $indv\_new \leftarrow mutation(indv\_new)$
        $P[indv] \leftarrow indv\_new$
    if $termination\_condition(Valid, Fitness)$ **then**
        break
$CP \leftarrow indv | Valid[indv] -> 0 \wedge Fitness[indv] -> min$

3.1.3.2. Encoding

In this work, individuals are encoded using a list of integers. Thus, the sequence $S = [s_1, ..., s_L]$ represent a list of nodes s_i to visit in the execution of the CP or FP associated with a task definition. This work's approach is to use the information of the task and the validity rules to encode valid individuals (but not necessarily feasible) from the beginning. Consider the situation described in Fig. 3.4, where the task is take a data sample from *node 6 (Target)*, starting from *node 4 (Start)*, and finishing in *node 5 (End)*. The following diagram describes the situation.

Of course, at this point, we do not know how to travel to these nodes. Let say we can move trough $Start$, $Target$ and End nodes, using the nodes (satellites) $\{A, B, C\}$ encoding to the sequence $S = [Start, A, B, Target, B, C, End]$ ($L = 7$). The genotype S expresses as

a phenotype $K = [k_1, ..., k_6]$ which encodes the sequence of states k_i where the contacts are feasible according to the CL. This situation is described in the following diagram.



In the sequence, $S = [Start, A, B, Target, B, C, End]$, the $Start$, $Target$, and End nodes are well known (Fixed in the task definition). Thus, the algorithm has to find the values $A, B,$ and C. These values are generated randomly, and the genotype K is obtained during the fitness function evaluation by searching sequentially in the CL for states that makes the sequence S feasible. If there are less than $L - 1$ states for the sequence S, it is an unfeasible or invalid solution. Since L is a parameter, it can be set arbitrarily large because redundant contacts are allowed. Figure 3.5 shows the encoding of a possible solution to the example scenario and task definition.

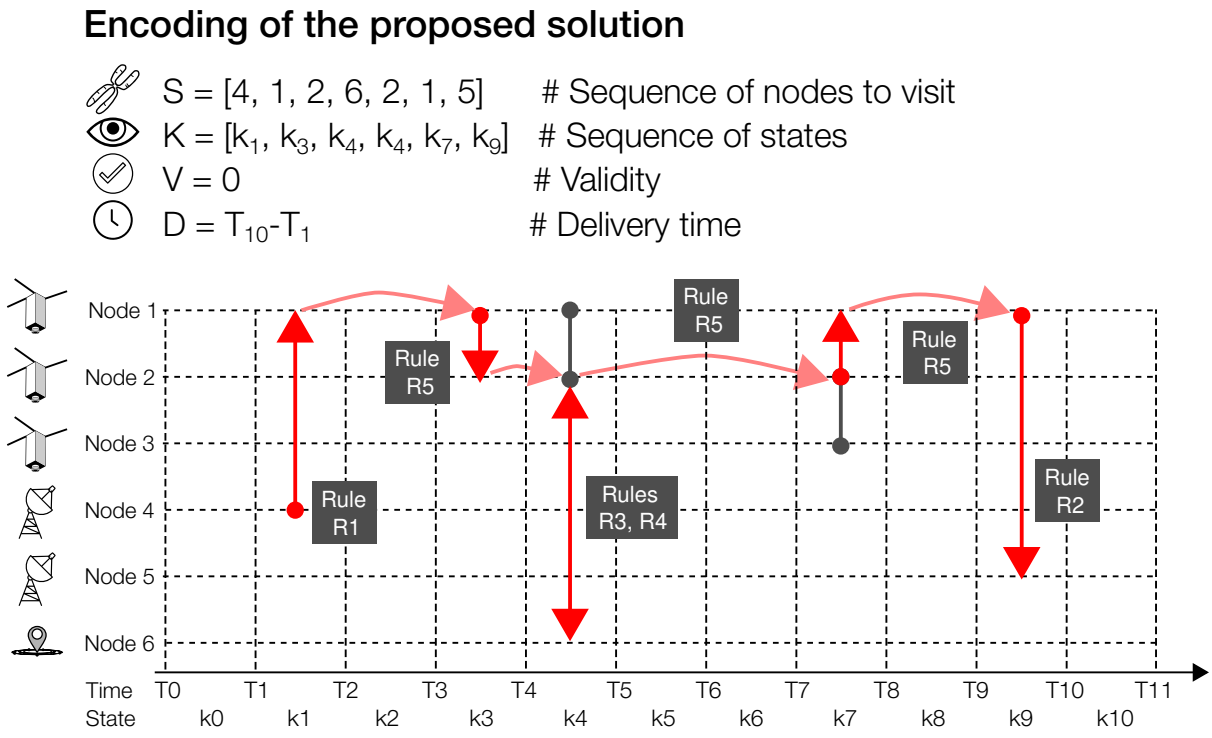


Figure 3.5: Genotype, phenotype, validity and delivery time of a possible solution for the example scenario and task definition

3.1.3.3. Constraints and fitness function

The fitness function is the objective function in Eq. 3.2, *i.e.,* the delivery time (also mentioned as sequence duration, or simply duration). However, in this problem, the validity restrictions (see rules R1. to R5.) are absolutely relevant to evaluate a solution. Therefore,

validity is also considered in the fitness function, creating a multi-objective optimization problem. Thus, the following fitness function is used:

$$
\begin{aligned}
fitness: \ & F_i = (V_i, D_i) && (3.6)\\
& 0 \leq V_i \leq L && (3.7)\\
& D_i > 0 && (3.8)\\
& && (3.9)
\end{aligned}
$$

Where V_i is the validity of the sequence S_i and is calculated as the sum of invalid contacts. Let define V_k as 1 if the contact at state k breaks any rule R1. to R5. or 0 if not. If a sequence of $S_i = \{s_1, ..., s_L\}$ of length L contains only valid contacts then $V_i = 0$:

$$V_i = \sum_{k=1}^{L} V_k \tag{3.10}$$

Delivery time D_i is calculated as the time difference between first and last contact in sequence S_i:

$$D_i = (T^{end}_{k_{L-1}} - T^{start}_{k_0}) \tag{3.11}$$

3.1.3.4. Initialization and stopping criteria

Individuals are created from three parameters: the task, the maximum number of nodes allowed (L), and the target's position in the sequence (I). The *start*, *target* and *end* node numbers are obtained from the task definition. The maximum number of nodes is a user-defined parameter to limit the sequence length. This number can be arbitrarily large because redundant contacts are allowed, and the final sequence can be simplified to a short version without repeated contacts. The target nodes' position is a randomly chosen index, so $I \in [2, L-2]$ and an individual tracks this value to maintain its validity during the genetic operations. A couple of examples of individuals are shown in Fig. 3.6. This idea can be extended to arbitrary large sequences and an arbitrary number of targets.

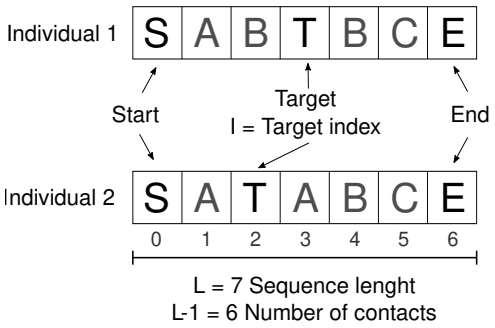


Figure 3.6: Individuals initialization. S, T, and E are the Start, Target, and End nodes' numbers. These values are known from the task definition. The index I of node T is particular to an individual. A, B, and C are unknown and so generated randomly for each individual

Thus, a population of N individuals of length L is created. The target index I and the

nodes (other than the *start*, *target* and *end*) are chosen randomly. After the evaluation, each individual keeps track of the sequence of valid contacts K reached by its sequence.

If an individual of length L reaches $L-1$ valid contacts, *i.e.*,, $V=0$, then its sequence is a valid contact plan. The delivery time or sequence duration D is also evaluated using the information of the CL.

If individuals reached the validity condition ($V=0$) and if no observable improvement of the delivery time D of the best individual during five consecutive generations, GA is assumed to have reached convergence, and the stop condition is satisfied.

3.1.3.5. Mutation operation

The mutation operation starts selecting an index i of the sequence to mutate. Depending on the value of i there are several cases:

- $i \in \{0, I, L-1\}$: If the mutation index i points to the start, end, or target position, do nothing. These values cannot be changed.
- $i \in \{I-1, I+1\}$: In any case, replace both $I-1$ and $I+1$ with a new random value. This operation respects the validity rule R4..
- Otherwise: replace the node at index i with a new random value.

Figure 3.7 graphically describes the mutation operation. Note that this operation always generates a valid sequence.

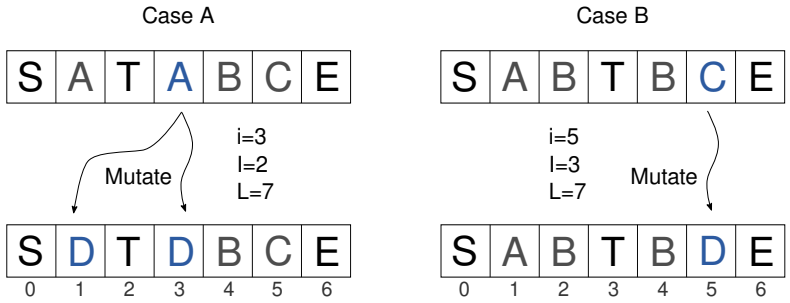


Figure 3.7: Mutation operation. Left, case A: $i \in \{I-1, I+1\}$, so both indexes 1 and 3 are mutated. Right, case B: $i \notin \{0, I-1, I, , I+1, L-1\}$ so mutate node at index 5.

3.1.3.6. Cross-over operation

To do the crossover between two individuals, a cut point j is selected. The cut point is always the index next to the target index to maintain the sequences valid, i.e. $j = I+1$. However, the sequence to cut is chosen randomly. Thus, as described in Fig. 3.8 the crossover operation consists of mixing section $S_A[0:j]$ of the first individual with section $S_B[j:L-1]$ of the second individual, or vice versa.

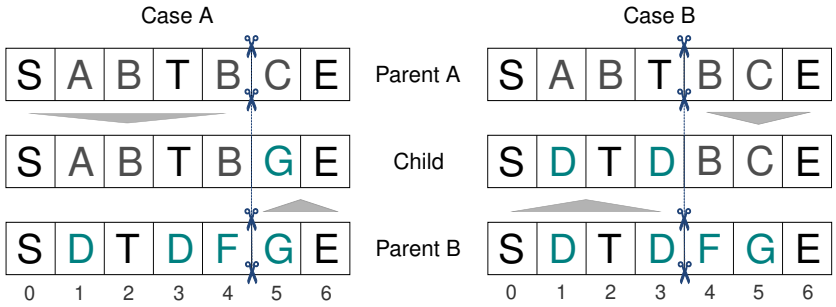


Figure 3.8: Cross-over operation. Left, case A: the cut point is chosen from parent A so $j = I_A + 1 = 4$ and the first section comes from parent A while the second section comes from parent B. Right, case B: in this case the cut point is chosen from parent B so so $j = I_B + 1 = 3$ and the first section also comes from parent B

3.1.4. Flight plan design

This module has to find the FP that is a valid solution to execute a task using the constellation's capabilities. Figure 3.9 describes how the CP is related to the FP to solve the problem. The CP defines the instants and operations required to visit the nodes involved in the solution; then, the last step is to define the commands in the FP. Commands depend on the contact type and the task definition. Four different directives should be implemented in the nodes' software:

- `set_fp <time> <command>`: This directive will be used to set a FP entry. It can be used to queue commands that will be executed in future contact. The `<time>` is defined in the CP. The `<command>` can be any of the following directives.
- `send_fp <node>`: This directive can be used to transfer FP entries to another node during a contact. The `<node>` parameter is obtained from the CP. In combination with the previous directive its possible to automate FP transfers.
- *`get_data <data_id>`*: Satellites must implement specific commands to operate its payloads and get data from targets. This directive is used in a satellite-to-target contact and is obtained from the task definition.
- `send_data <node> <data_id>`: This directive can be used to transfer payload data back to the ground stations. In combination with the `set_fp` directive its possible to automate data transfers.

Contact plan

From	To	Start	End
4	1	T1	T2
1	2	T2	T3
2	6	T3	T4
2	3	T4	T5
3	5	T5	T6

Ground station to satellite or satellite to satellite contact: transfer flight plan

Satellite to target contact: get data from target

Ground station to satellite or satellite to satellite contact: return data and transfer flight plan

Flight plan

Time	Node	Command
T1	4	`fp_send 1`
T2	1	`fp_send 2`
T3	2	`get_data 55a1`
T4	2	`send_data 3 55a1`
T4	2	`fp_send 3`
T5	3	`send_data 5 55a1`

Figure 3.9: Relation between Contact Plan (CP) and Flight Plan (FP)

The FP contains all the information required to execute the task. This information is sent to the first node, a ground station, to start the task's automated execution. The execution of this global FP can be simulated to validate the feasibility of the solution or detect any problem. The remaining question is how to build a constellation capable of executing this kind of FP, *i.e.,*, a homogeneous FS solution (capable of running commands in the satellites and ground station nodes) is required.

3.1.5. Implementation details

The complete suite has been implemented in the Python programming language, and it is available under the LGPL v3 license in the following GitLab repository: https://gitlab.com/carlgonz/constellation-framework

Interested readers can find installation and execution instructions in the repository `README.md` file.

3.2. Flight software

To control a nanosatellite constellation, a Flight Software (FS) solution capable of interacting with the proposed framework is required. Also, to implement a more realistic software in the loop simulator, it will use the same software solution as the spacecraft once in orbit. For these reasons and with the aim of closing the gap between theoretical and practical developments, in this work, a nanosatellite constellation FS solution is presented and described in detail. The requirements and architectural decisions are inspired by the Satellite of the University of Chile for Aerospace Investigation (SUCHAI) nanosatellite program being developed at the University of Chile. In fact, base ideas were developed for the SUCHAI I CubeSat, launched into space in 2017, and operated for about 1.5 years. This university space program continues to the date, building a small constellation of three 3U CubeSat. For this reason, the solution developed in this work is named *SUCHAI Flight Software*.

3.2.1. Requirements analysis

A comprehensive review of available FS solutions, and common requirements for FS development was presented in section 2.4. Below in this section is a summary of the desired FS characteristics, both non-functional and functional requirements, in the context of New Space satellite missions, agile CubeSat development, high flexibility of mission tasks, and a constellation of large numbers of nanosatellites.

3.2.1.1. Non-functional requirements

Non-functional requirements refer to the quality attributes of a system [43]. Beyond the functional aspects of a satellite mission, it is necessary to define some design guidelines that will affect the architectural decisions, especially in agile, flexible, and fast-growing CubeSat projects. This context makes non-functional requirements more relevant not only in the design phase but also in the development process. Based on the literature review, the list of the quality attributes considered in this work includes:

Q1 The FS must be **extensible**, in the sense that any change or improvement should be localized, avoiding affecting the system structure.

Q2 The FS must be **modular**, so that non-critical modules, such as payloads, might be added or removed without affecting the entire system (e.g., needing to modify or recompile the entire system).

Q3 The FS design should reduce failure points and help in the implementation of fault tolerance techniques to improve the mission **reliability**.

Q4 The FS must be **portable** to multiple hardware and software platforms, such as embedded systems supported by FreeRTOS or computers capable of running GNU/Linux. Hardware portability is also required to being **reusable** along current and future missions.

Q5 The FS must be **scalable**, in the sense that the solution can support the manufacturing and operation of an increasing number of nanosatellite in large constellations.

3.2.1.2. Functional requirements

To determine the functional requirements of the FS, the operation model (or use case) described in Fig. 3.10 is considered. When the satellite is within the ground station range, the operators can send telecommands. These commands include the setup of the flight plan table. The satellite can execute these commands immediately (for example, download telemetry, download payload data, or modify settings) or queue flight plan commands (also known as time-tagged commands) for later execution (for example, sample sensors, take payload data, or transfer the flight plan to another node). During the remaining orbit, the satellite has to perform some activities autonomously. These activities include periodic tasks (such as housekeeping, sending a beacon, resetting watchdog timers, and checking for incoming telecommands), executing commands queued in the flight plan at a specific time, and reacting non-deterministic events such as malfunctions, unexpected resets, batteries discharge, among others. As a reference, a low orbit satellite can establish contact with the ground station three or four times in 24 hours, and each contact may last between 5 to 10 minutes. The remaining time, the satellite executes autonomous or scheduled operations.

This operation model is heavily based on the satellite's ability to execute commands, both remote or self-generated [63, 59]. The satellite operators should break down the satellite mission in a series of remote commands. Autonomous operations are translated to self-generated commands with a well-defined execution logic (for example, periodical or event-based).

The concept of remote and self-generated command execution can also be extended to constellation operations. Let us suppose that each satellite in the constellation can execute a set of commands. Mission control can decompose (manually or automatically) the mission goals into a series of commands, time, and nodes that each satellite in the constellation has to execute. Let us call this list of commands a *Global Flight Plan (FP)* table. If satellites have inter-satellite communication capabilities, then any satellite can transfer FP entries to surrounding satellites, thus facilitating the operation by distributing the mission goals over the constellation. The information exchange can also include data generated by payloads and

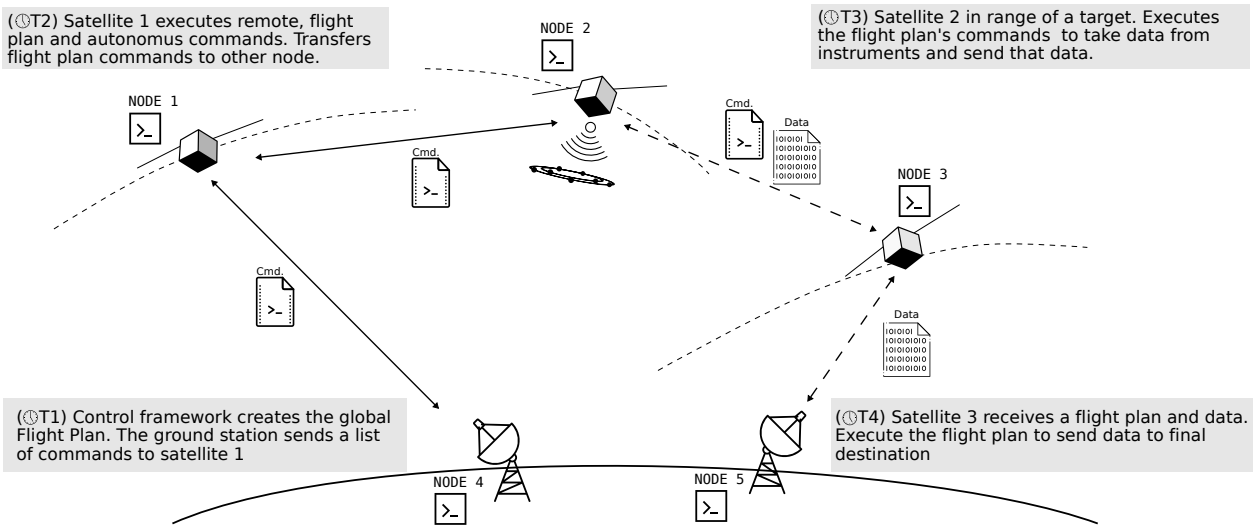


Figure 3.10: Example of satellite constellation operations. Ground station nodes send commands and flight plans to satellites. Satellites execute flight plan commands and remote commands (on-demand operations) and autonomous commands (housekeeping, attitude control, etc.). Using the ISL, nodes can send commands and data to other nodes. All nodes in the constellation execute the same flight software.

instruments. Therefore, the FS functional requirements can be expressed as follows:

F1 The FS must queue and execute commands at specific times in a Flight Plan table.

F2 The FS must execute remote (on-demand) commands generated from ground satellite operators

F3 The FS must execute self-generated (autonomous) commands. The execution logic of these commands can be single event, periodical, or event-based.

F4 The FS must store and download telemetry data.

Note that any specific mission functional requirements should be translated to commands and command execution logic. Thus, the FS is flexible enough to implement a variety of functional requirements. For example: “to send a beacon every minute”, “to collect particles-counter samples over the South Atlantic Anomaly”, or “to transfer a FP entry to node N”, are all possible mission functional requirements that can be implemented as commands and executed with a defined logic. Particularly important is supporting and implementing the commands defined by the constellation control framework in Sec. 3.1.4.

3.2.2. General design

Following the experiences of similar projects [44, 19] the proposed software design follows the layer architectural pattern dividing the system into hardware drivers, operating system, and application layers. This design provides a portable solution [64, 65, 66] that satisfies the

requirement **Q4** because the operating system and the device drivers layer can be exchanged by design. This approach allows us to integrate existing solutions in the drivers and operating systems layers and focus on the design and implementation of an application layer that satisfies the operation model discussed in Fig. 3.10.

Each layer functionality is divided into modules and dependencies between modules. Starting from the lower abstraction level, the minimum set of functionalities required are real-time clock, data storage system, access to input/output devices, and drivers for external devices or peripherals. The operating system layer will use these functionalities to provide high-level features such as threads, thread/task scheduler, timing functionalities, queues or message systems, and synchronization structures. Finally, functionalities to implement tasks, data repositories, and the concept of commands and flight plan are required in the application layer. The dependency tree of these modules is described in Fig. 3.11 Maintaining this dependency tree helps to maintain portability because, by design, the operating system and drivers are totally independent of the application code. The following sections describe each layer's design and implementation details, with a special focus on the application layer implementation.

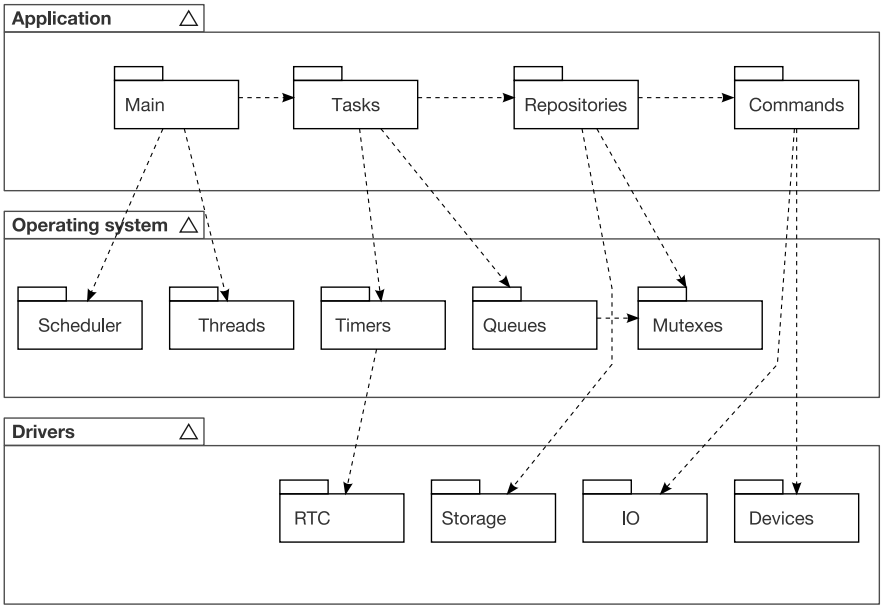


Figure 3.11: SUCHAI Flight software architecture: UML model diagram. Each layer consists of a number of coarse-grain modules, a module resulting from compiling several C files and headers. A direct dependency between modules is indicated with an arrow. The architecture follows a top-down interaction: higher-level layers can interact with layers below, but a lower level layer should never depend on layers above.

3.2.2.1. Drivers layer

This layer is populated by hardware and vendor-dependent software, created to interact with peripherals and devices at a low level. Any supported device should provide a set of drivers, libraries, or frameworks that help to interact with the device's features. From experience working with embedded systems, the diversity in this layer is so extended that the

author recommends following each vendor standard and solving the upper layer's differences through interfaces and wrappers. This recommendation includes managing different build systems at this level.

3.2.2.2. Operating system layer

The Operating System (OS) adds an abstraction level between the hardware and application layers so more advanced solutions can be implemented in the application layer using utilities such as multi-tasking, message queues, timers, and files. From requirement **Q4** and **Q5** at least two operating systems should be supported: GNU/Linux and FreeRTOS. Supporting GNU/Linux is useful for simulating the satellite functions on personal computers (a developer's laptop or testing servers) and supporting powerful embedded computers such as the Raspberry Pi or the ARM™ Cortex A9 found in the Zynq 7000 family. Meanwhile, FreeRTOS is more suitable for low-power embedded systems, which are usually 16 or 32-bit microcontrollers such as the Microchip™ PIC24 and PIC32, the Atmel™ AVR32, the Espressif™ ESP32, to name a few. This portability layer is required to map specific operating system functionalities to a common custom interface. For example, a custom function `osTaskCreate()` to create Tasks is implemented, which is a wrapper to `pthread_create()` in GNU/Linux and to `xTaskCreate()` in FreeRTOS.

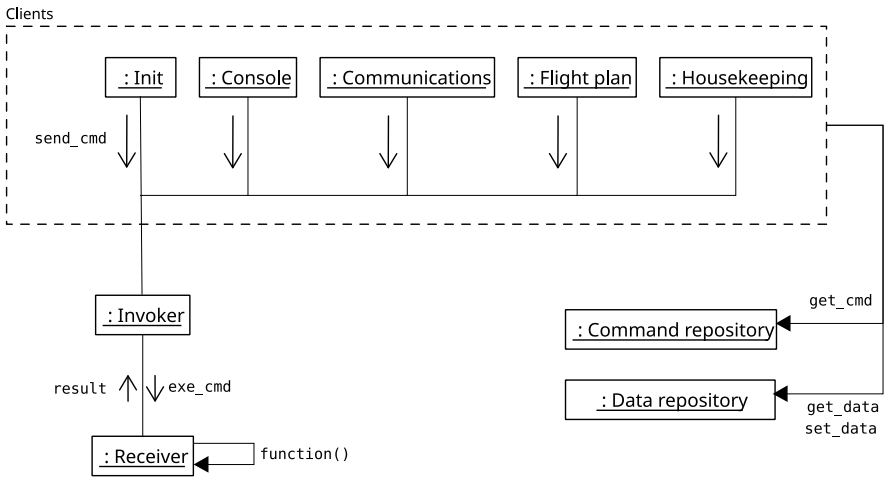


Figure 3.12: SUCHAI Flight software architecture: UML communication diagram. In this architecture, clients only generate requests to execute commands, depending on the control strategy that each client implements. These requests are sent as messages to the invoker that may implement some control strategies over the command execution, such as filtering, priorities, and logging. If the invoker decides that the command can be executed, it sends the request to the receiver. The receiver actually executes the command by calling the corresponding function. The command and data repositories provide an interface to handle commands creation and data storage, respectively.

3.2.2.3. Application layer

The application layer architecture is based on the command processor design pattern. This pattern explains how to build an application that separates the service request from its execution, encapsulating each requirement in different commands [65]. Nevertheless, this pattern was used at an architectural level and adapted for implementation in the C programming language [66]. The software architecture is described by the UML communication diagram shown in Fig. 3.12 and the UML sequence diagram shown in Fig. 3.13. The execution logic is the following: when a client is required to do a particular action, it creates a specific command and requests its execution to the invoker by sending the "`send_cmd`" message. The invoker checks if the command is executable and sends the requirement to the receiver as the "`exe_cmd`" message. The receiver actually executes the command by calling the corresponding function and sends the return value back to the invoker in a "`result`" message. Furthermore, the satellite needs to store at least the available commands, a list of settings or status variables, and data generated by payloads. Therefore, a set of repositories are included in the architecture, which are modules designed to encapsulate the data handling.

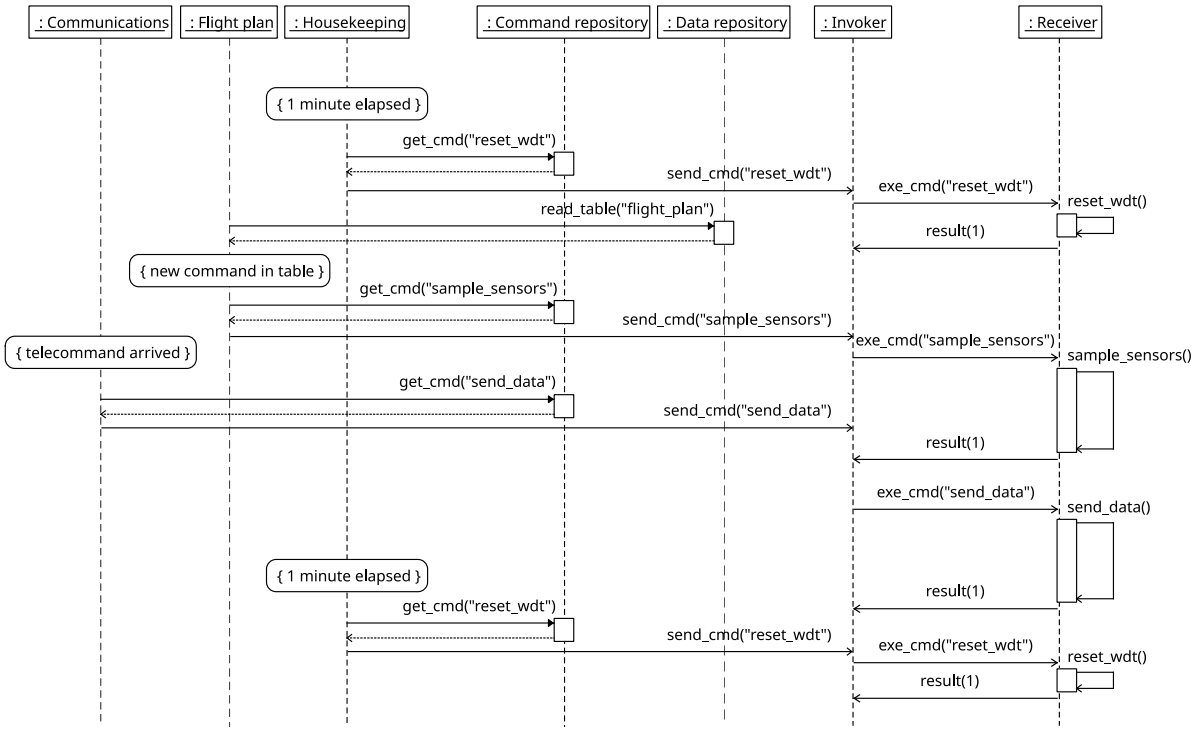


Figure 3.13: SUCHAI Flight software architecture: UML sequence diagram. Each client implements a control strategy and can request commands execution under certain circumstances. To execute a command, the client has to create it using the command repository and then send an asynchronous message to the invoker. The invoker receives all client messages and organizes the execution by sending the request to the receiver. The receiver actually executes the command by calling the corresponding function. Once the command is executed, the receiver sends a message back to the invoker with the execution result.

The architecture's necessary modules are the clients, the invoker, and the receiver because

they implement the command execution logic. These modules are implemented as concurrent tasks (FreeRTOS) or threads (GNU/Linux) that use a messaging system to communicate clients' requests. These requests or commands are data structures (C structs) that contain all the relevant information to execute the target code, such as a function pointer and parameters. Any of the many existing clients can generate commands. The messaging mechanism might be a shared queue where clients can push the commands as C structs; thus, the invoker can pop commands to be processed one at a time.

The SUCHAI FS presents a clear execution logic because clients just generate the requirements and only the receiver actually calls functions for its execution; if additional control is required (such as command priorities, safe mode, execution logging, etc.), the invoker can implement these functionalities. This execution logic (described in Fig. 3.12 and 3.13) requires the definition of the set of client modules, commands and repositories.

3.2.2.4. Flight plan

In the SUCHAI FS the Flight Plan is one of the Client modules, so it is implemented as a concurrent task. This module's main goal is to schedule commands to be executed at a certain date and time. The schedule can be modified by specific commands to dynamically change the mission plan once the satellite is in orbit. The command list is stored in non-volatile memory to maintain the state between resets. The logic of this task is very simple. As described in Algorithm 2, the idea is to periodically check the system timestamp with 1-second resolution and search in the list for commands queued at a specific time. Additionally, this implementation also supports the schedule of periodic commands, *i.e.,* the command can be rescheduled *period* seconds in the future up to *repetitions* times. Table 3.3 show the list of available commands to control de FP operation.

Algorithm 2: Flight plan algorithm

```
   // Run forever
1  while True do
2      sleep(1second)
3      current_time ← get_unix_time()
4      command, node, repetitions, period ← get_command(current_time)
       // Check if a command is scheduled for this node
5      if command = null ∨ node ≠ this_node then
6          continue
       // Support command repetition
7      if period > 0 ∧ repetitions > 0 then
8          set_command(current_time+period, command, node, repetitions−1, period)
       // Execute the command
9      if repetitions ≥ 0 then
10         execute_cmd(command)
```

Table 3.3: Flight plan related commands

Command	Arguments	Description
fp_set_cmd	<time><node><rep><period><command>[args]	Set a new flight plan table entry
fp_del_cmd	<time>	Delete a flight plan table entry
fp_show		Print current flight plan table
fp_reset		Clear flight plan table
fp_send	<node>	Transfer current flight plan table to <node>
fp_recv	<fp_table>	Parse and load a flight plan table

3.2.3. Implementation details

The SUCHAI Flight Software has been implemented in the C programming language and supports the GNU/Linux and FreeRTOS operating systems. It has been tested and used in X86 computers, the Raspberry Pi minicomputer, the Espressif ESP32 microcontroller, and the Nanomind A3200 (ATMEL AVR32) onboard computer. The software is released under the GPL v3 license in the following GitLab repository: https://gitlab.com/spel-uchile/suchai-flight-software

Interested readers can find installation and execution instructions in the repository `README.md` file and a complete usage guide in the following link: https://docs.google.com/presentation/d/1CzChzQjQzATod_S-Zj9ivqEMR4P_FI5_3RXnxfiKWzM/edit?usp=sharing

3.3. Simulator

A software in the loop simulator is a flexible tool-chain to evaluate the functioning of complex systems such as nanosatellites. The idea of implementing an operational simulator for nanosatellites has been explored as an option to reduce mission risk and provide agility to space missions [67, 68]. A similar idea is required in this work but extended to simulate a set of satellites working as a constellation with inter-satellite capabilities. The goal is to validate solutions obtained by the Flight Plan Generator module using the software on the loop simulation methodology.

Figure 3.14 details the simulator modules integrated into the SUCHAI FS. The simulator will use the same IO system to get/put data commands because the communication libraries (CSP or TCP/IP) support local loopback interfaces. Simulator commands and simulated devices are required for hardware operations not supported in the host system (GNU/Linux), such as payloads, instruments, and actuators. The OS layer is intervened with a Simulator Scheduler to control the execution time. Thus, all software timers such as system tick, system calendar, and threads delays run in an accelerated simulated time. The Simulator main module controls the starting, ending, and tick update. It also controls time synchronization with other nodes in the constellation simulation using POSIX shared memory and shared mutex. One node runs the *Primary Timer* task, which updates the simulator ticks counter to the shared memory space. The rest of the nodes run the *Secondary Timer* task, which reads the current simulator tick from the shared memory and signals all waiting Tasks.

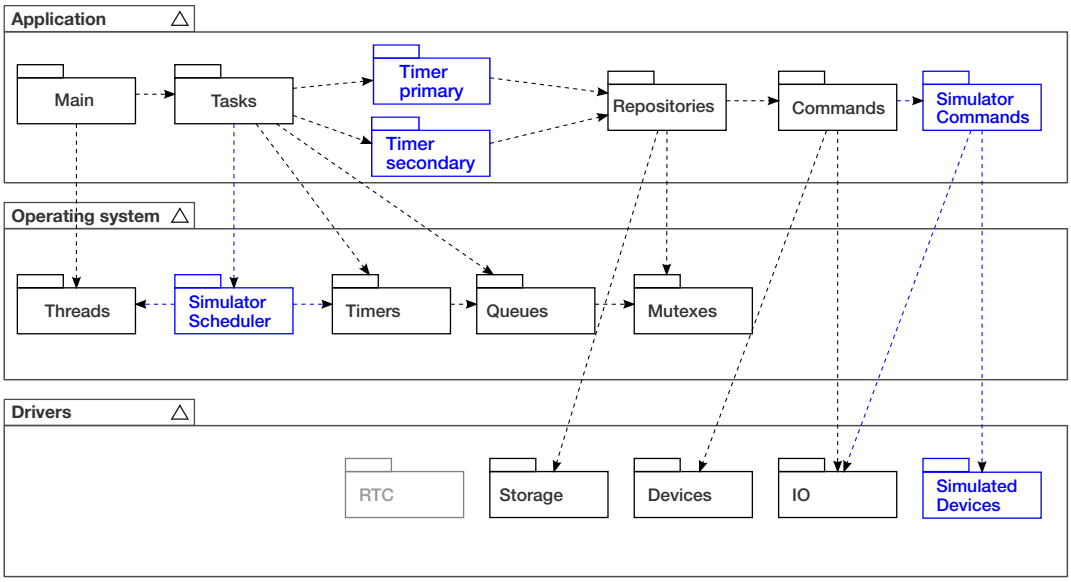


Figure 3.14: Simulator modules integrated in the SUCHAI FS. In blue: new simulator related modules. In gray: modules no longer used

3.3.1. Implementation details

Figure 3.15 describes the architecture designed for the simulator. The simulator uses the SUCHAI Flight Software as the base framework and creates a new application. The main idea is to launch and set up multiple instances of the simulator application, each representing a node in the constellation (ground station or satellite). FS instances shared the simulated time and also a communication bus. Using the communication bus and the CL information, it is possible to simulate the inter-satellite and ground station links. Thus, simulated instances can share commands and data interacting as in reality from the software point of view. A simulator controller was written in Python to take the scenario and task definitions and launch the simulated satellite instances. The FS instances collect periodic telemetry and the

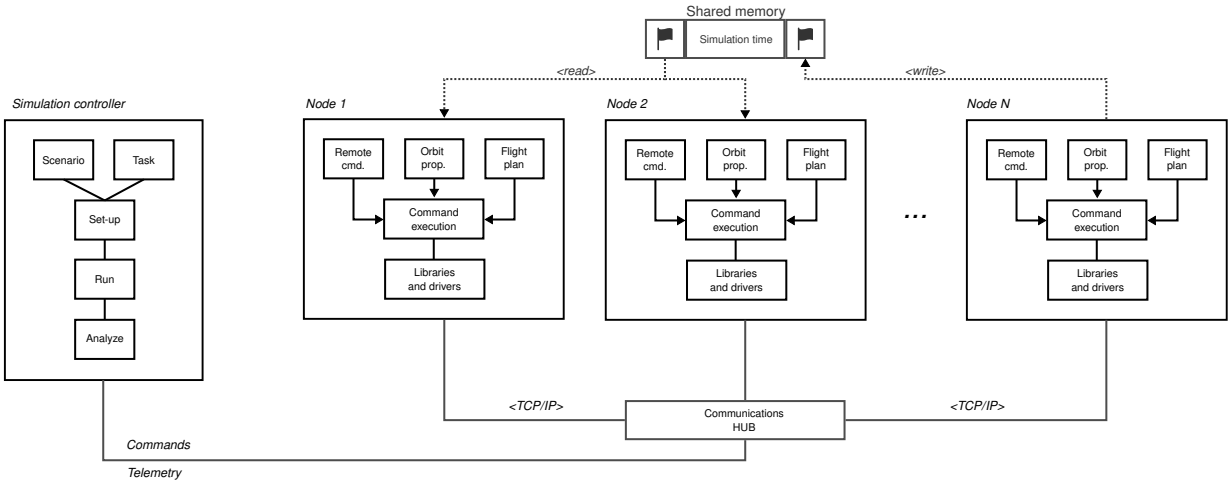


Figure 3.15: Software in the loop simulator for nanosatellites constellation using the SUCHAI FS

list of commands executed. This information is used to validate the execution of the task and analyze the functioning of the constellation.

The simulator is a separated SUCHAI FS based application so it can be found in the following GitLab repository: https://gitlab.com/carlgonz/suchai-constellation-simulator/-/tree/framework

3.4. Chapter highlights

A nanosatellite constellation operation framework was presented in this section, including the task scheduler, the flight software, and a simulation suite. The following conclusions were obtained.

- In the scenario of a nanosatellite constellation with ISL, considering CubeSats' typical size, power, and computational resources limitations, the operations can be summarized in a global flight plan table. Ground nodes can calculate this global flight plan table to reduce the space segment computational workload. The framework does not distinguish between ground nodes, satellite nodes, or targets. It requires that all nodes support the same software characteristics, including commands and flight plan execution.
- The framework is composed of three main modules: a contact list generator, a contact plan generator, a flight plan table generator, and software in loop simulation suite.
- The contact plan generation presents specific restrictions to satisfy de data flow of the studied problem. Five validity rules and two optimization parameters are described. The goal is to design a contact plan that satisfies the validity rules while optimizing the flow of data to accomplish the tasks.
- A genetic algorithm was used to solve the contact plan design. A particular encoding, a two-step fitness function, and particular genetic operators (mutation and crossover) were created to solve the problem under study.
- It is possible to derive a global flight plan using the contact plan information. The constellation must cooperatively execute this flight plan to accomplish the proposed task.
- It was possible to create a flight software based on the command design pattern to meet all functional a non-functional requirements derived from a CubeSat constellation operation. The designed software can execute the global flight plan and communicate data with surrounding nodes.
- It was found that the quality attributes of the flight software are key to supporting the agile assembly and operation of a mega constellation. Therefore, it will be necessary to create benchmarks and techniques to verify software quality criteria during the constellation assembly and operation.

Chapter 4

Results

This section presents the results of applying the proposed constellation control framework to various scenarios. In particular, the task scheduler performance is evaluated in constellations up to 1000 nodes in walker and Ad hoc configurations. It also shows the scalability tests to the contact list calculation tool and the evolutive contact plan design algorithm. This section also presents the flight software's quality verification results using agile visualization techniques. Finally, it presents the constellation resource usage results obtained from the constellation operational simulator tool.

4.1. Contact list generation scalability

The proposed framework requires generating the contact list of constellations up to 1000 nodes. As explained in Figure 4.1, calculating the contact list requires several stages. First, each satellite track is propagated to future time using the TLE orbital parameters and a SGP4 propagator. Second, for each node and time instant, the software evaluates if contact is feasible with any other node. According to the radio link model programmed (in this case, omnidirectional antennas), this step generates an intermediate result with all possible contacts among all nodes in the scenario. Finally, the tool generates the contact list data file according to the format commonly used in DTN studies (contact start time, start node, end time, destination node, and contact span). For simplicity, contacts are considered bidirectional in all calculations.

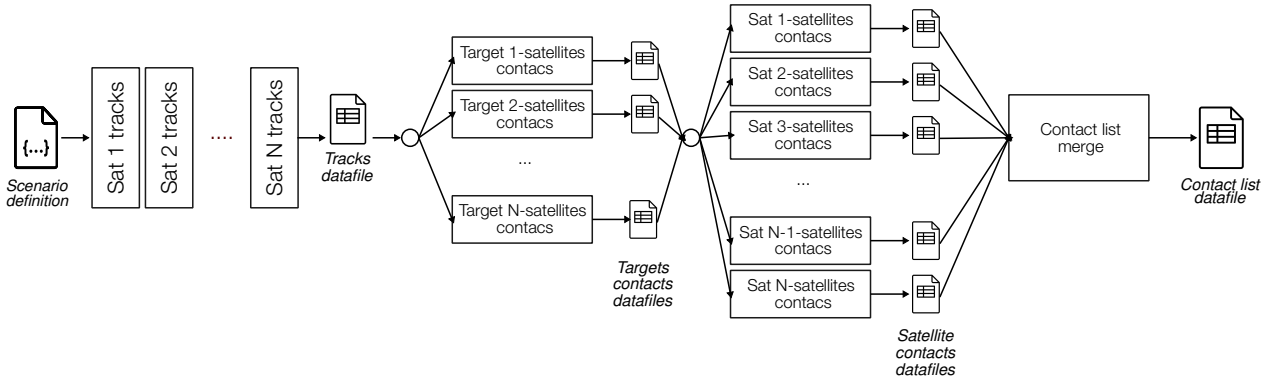


Figure 4.1: Contact list generation flow diagram.

Calculating N to N nodes contacts opportunities is a computationally intense task, es-

pecially for large constellations. Thus, several optimizations were implemented to calculate contact lists up to 1000 nodes. Parallel processing was required to accelerate results. RAM usage optimization and intermediate files were necessary to limit memory usage. Scalability tests where performed in the NLHPC[1] cluster using the *slims* nodes (up to 20 cores)[2] for the 10 satellites scenario and the *general* nodes (up to 44 cores)[3] for the 100 satellites scenario. Results are shown in Figure 4.2. As Figure 4.1 explains, not all the process is parallelized. Using Amdahl's law

$$speedup = 1/(s + p/N) \tag{4.1}$$

It is possible to determine the portion of execution time spent in the serial part (s). For the 10 satellites scenario $s = 0.19$ on average, at least 19 % of the workload was not parallel. For the 100 satellites scenario $s = 0.04$ on average or a 4 % of serial workload. The later results are explained because calculating satellite to satellite contacts in parallel is more computationally expensive than calculating satellite tracks and merging the contact lists data files serially. Therefore parallelization is exploited better in large scenarios.

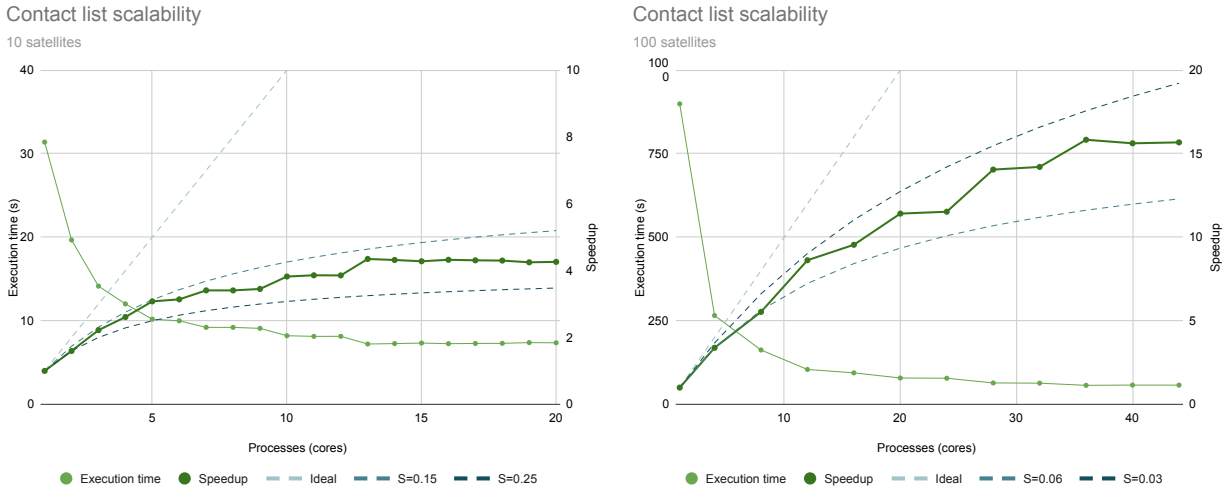


Figure 4.2: Contact list generator scalability results for a 10 satellites constellation (left) and 100 satellites constellation (right). Propagation time 16200 seconds (3 orbits) with 30 seconds resolution, and 60 seconds contacts resolution.

Finally, total execution times to calculate the contact list for 10, 100, 1000 scenarios are shown in Figure 4.3. The `general` nodes with 44 cores were used to propagate orbits during 16200 seconds (3 orbits) with 30 seconds resolutions. Contact lists were calculated with 60 seconds resolution. Results show that less than 60 seconds are required to calculate contacts of 100 satellites during three orbits. Meanwhile, about 1 hour is required to calculate contacts of 3 orbits with 1000 satellites which is a ratio of 4.5 times between the scenario period and calculation time.

[1] https://www.nlhpc.cl/
[2] 128 nodes with 2 x Intel Xeon E5-2660v2 @ 2,20GHz, 10 cores each, 48 GB of RAM
[3] 48 nodes with 2 x Intel Xeon Gold 6152 CPU @ 2.10GHz, 22 cores each, 192 GB of RAM

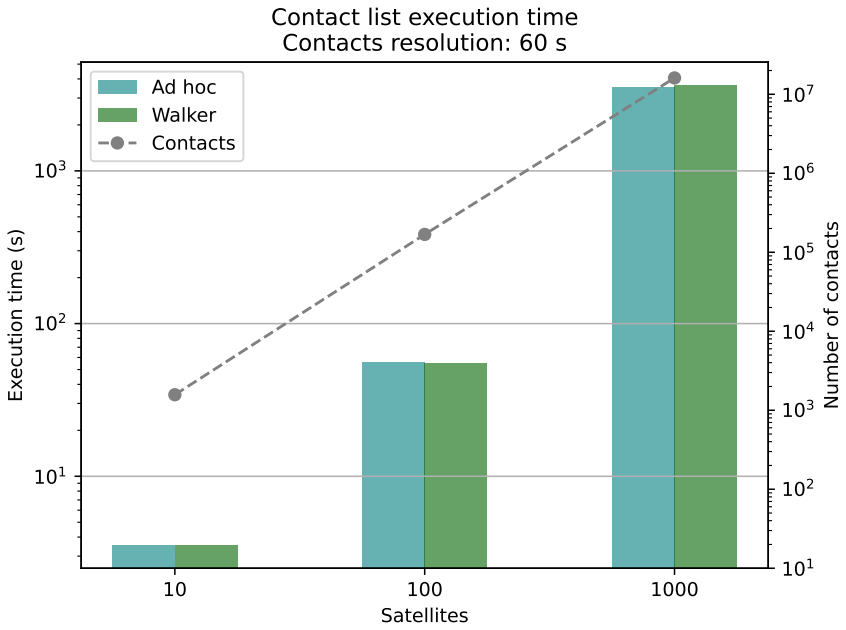


Figure 4.3: Contact list calculation execution times for 10, 100 and 1000 satellites scenarios. Propagation time 16200 seconds (3 orbits) with 30 seconds resolution, and 60 seconds contacts resolution.

4.2. Genetic algorithm hyper-parameters tuning

Case studies were used to evaluate the performance of the genetic algorithm generating contact plans. These cases were also used to tune the algorithm's hyper-parameters, including mutation rate and population size. A small constellation with 10 satellites will be considered at the beginning to keep the results, visualizations, and analysis treatable. Then, scenarios with 100 and 1000 satellites are studied.

4.2.1. Scenario A: 10 satellites Walker constellation

Scenario A consists of a constellation with ten satellites in Walker configuration (5 orbital planes, 2 satellites per plane, 500 km altitude), two ground stations, and one target. Details are described in Table 4.1. An SGP4 orbit propagator[4] implementation was used to calculate satellite ground tracks, Inter-satellite link (ISL) and ground station contacts. Inter-satellite links consider omnidirectional antennas with a 1500 km range. A maximum elevation of 5° was considered for the ground to satellite links. It is also assumed that radio links are fast enough to transmit all dada during the contact and that commands are executed fast enough during the contact. A simulation of 16200 seconds with a resolution of 60 seconds to calculate the contact opportunities resulted in 1929 contacts.

[4] https://rhodesmill.org/skyfield/api-satellites.html

Table 4.1: Scenario A description

Simulation time				
Start time: 2020-09-30T00:00:00 UTC (1601424000 Unix time)				
Simulation time: 16200 seconds (∼3 orbits)				
Simulation resolution: 30 seconds				
Contact list resolution: 60 seconds				
Number of contacts: 1929				
Satellites				
Node	Period (min)	Incl	Mean anom.	R. ascension
0	94.47	97°	0.0°	0.0°
1	94.47	97°	90.0°	0.0°
2	94.47	97°	0.0°	36.0°
3	94.47	97°	90.0°	36.0°
4	94.47	97°	0.0°	72.0°
5	94.47	97°	90.0°	72.0°
6	94.47	97°	0.0°	108.0°
7	94.47	97°	90.0°	108.0°
8	94.47	97°	0.0°	144.0°
9	94.47	97°	90.0°	144.0°
Ground stations and targets				
Node	Lat.	Lon.	Alt.	Reference
10	-33.3833°	-70.7833°	476 m	Santiago, Chile
11	35.6830°	139.7670°	5 m	Tokyo, Japan
12	-15.0°	-15.0°	500 km	S. Atlantic Anomaly

Figure 4.4 shows a static representation of the satellites' ground tracks after 45 minutes (approx. half orbit). Satellites will eventually get closer to each other at some points (especially at the poles), and ISL are possible. The contact list FSM representation shown in Fig. 4.5 presents a better view of this scenario. In this figure, contact opportunities are shown as arcs connecting two nodes. Contact opportunities are discretized and calculated with 60 seconds resolution but displayed with 300 seconds resolution. Thus, the goal of the GA is to search in this graph for the best possible combination of contact opportunities to solve the proposed task.

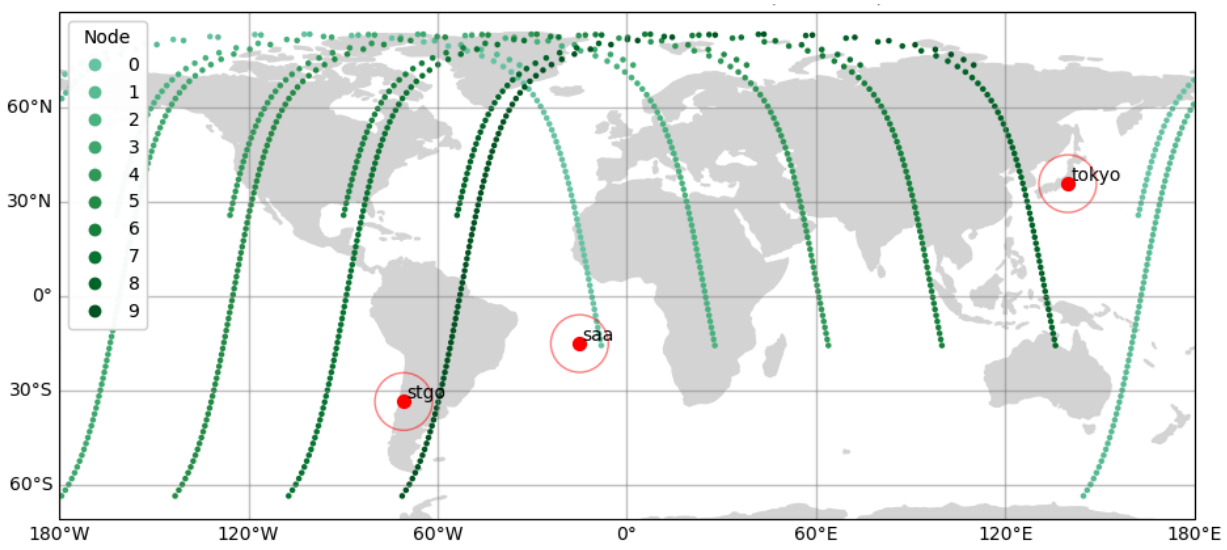


Figure 4.4: Scenario A satellite tracks after 45 minutes (Approx. half orbit), ground stations and targets locations (red).

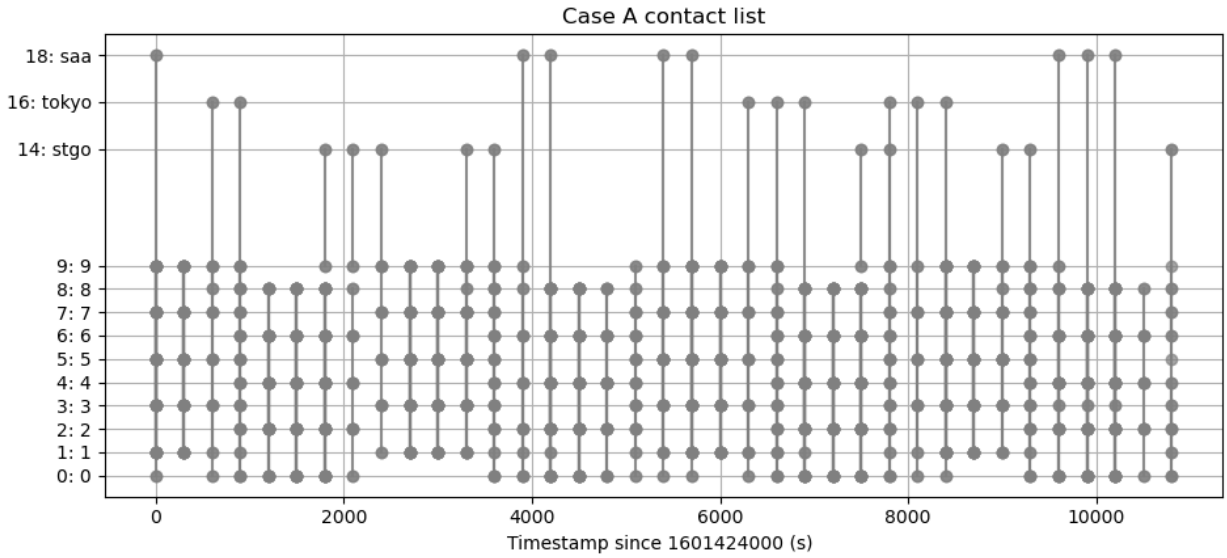


Figure 4.5: Scenario A contact list in FSM representation. Contact opportunities are grey lines connecting two nodes. For clarity, a simplified version with 300 seconds contacts resolution is shown. Note that lines connecting nodes in a particular state may be overleaped.

4.2.1.1. Task 1

The first task to test was a storage and forward mission. The goal is to execute the command `sim_get_data 1` over the *South Atlantic Anomaly (SAA)*, starting in *Santiago* ground station to download the result id `1` in *Tokyo* ground station. The definition file for this task, according to Section 3.1.1, is described in Listing 4.1.

Listing 4.1: Task 1 definition file

```
{
    "id": 1,
    "start": "stgo",
    "end": "tokyo",
    "targets": [
        {"id": "saa", "command": "sim_get_data 1", "result": "1", "prio":  1}
    ],
    "solution": null
}
```

The scenario and task definition were loaded into the constellation controller framework to obtain the case's contact and flight plans. The hyper-parameters evaluation included 4 population sizes (50, 100, 150, and 200), and 4 mutation rates (0.2, 0.4, 0.6, and 0.8). Because of the stochastic nature of GA, 100 independent runs of the algorithm for each combination were executed, each run with a different random seed. A summary of the results is presented in Fig. 4.6.

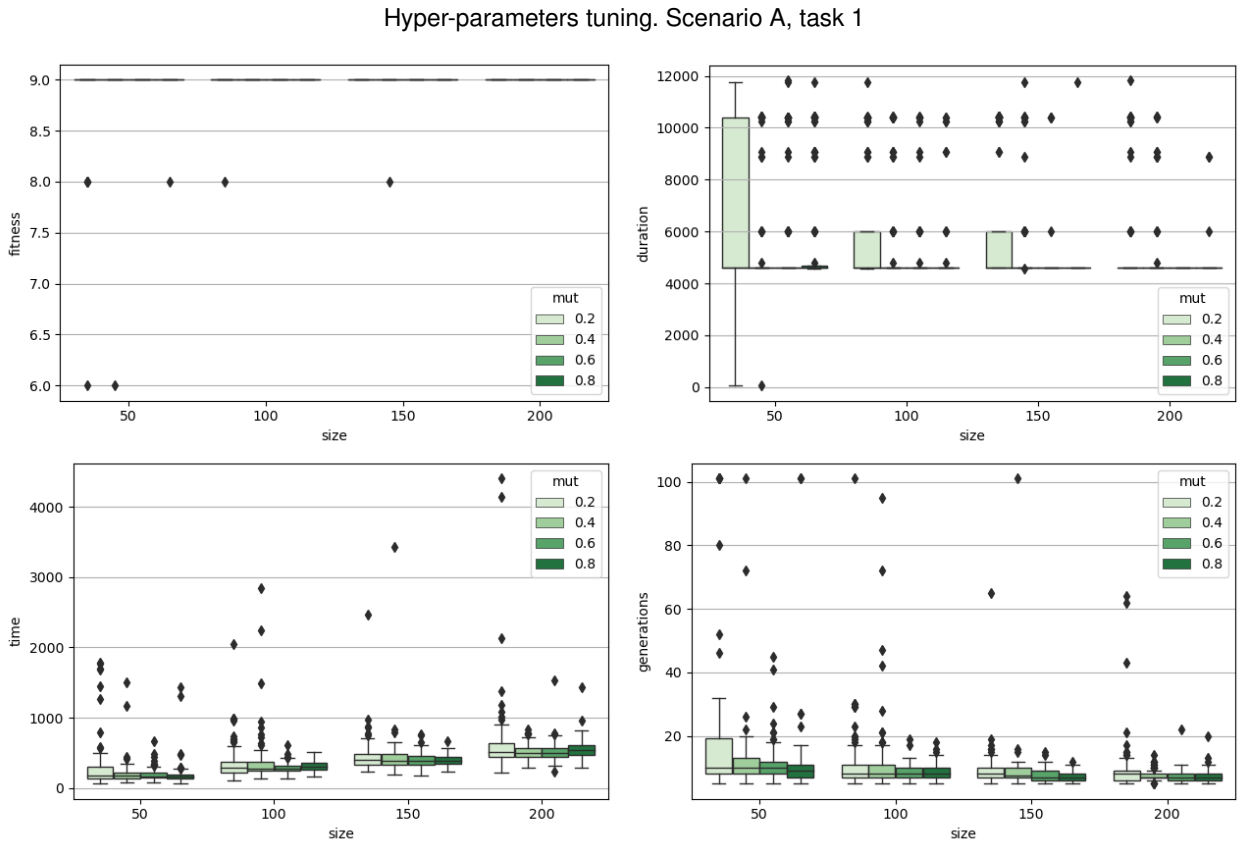


Figure 4.6: Scenario A, task 1 hyper-parameters tuning with maximum 9 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

These results show that the algorithm can find valid solutions to the proposed task and converges to the optimal value of 4620 seconds[5] (duration or delivery time) for mutation rates larger than 0.4. In this case, greater mutation rate or population size do not significantly impact the algorithm's performance; however, large population size and high mutation rate favor convergence. The maximum length of the sequence (or the number of hops) was not relevant because the algorithm allows redundant contacts in a sequence. On average, the algorithm obtains solutions in 276.8 seconds average (64.5 seconds minimum) using the laboratory workstation[6] and 10 generations average (5 generations minimum). Two solutions were randomly chosen to be displayed in Fig.4.7. These solutions are not identical, but they are valid and have the same fitness value. The algorithm can choose the intermediate contacts among several options while the task definition and validity rules are met (See section 3.1.3). The solution shown in Fig. 4.7-right is the sequence `S=[10, 8, 8, 0, 12, 0, 0, 12, 0, 11]` that can be simplified to `S=[10, 8, 0, 12, 0, 11]` and contacts `K=[417, 519, 643, 643, 938]`. This sequence corresponds to the contact plan and flight plan described in Table 4.2.

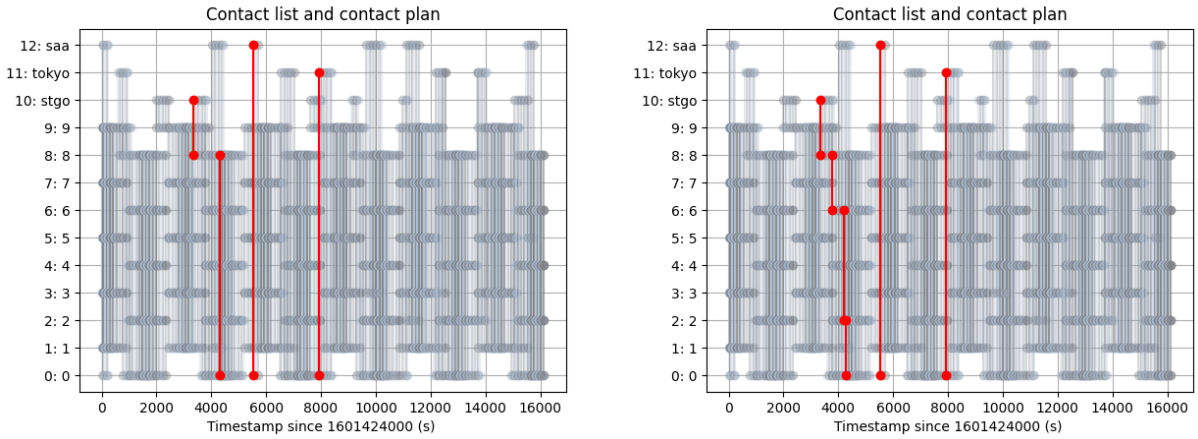


Figure 4.7: Scenario A, task 1 example results with maximum 9 hops. Contacts resolution 60 seconds. Left: Mutation=0.6, population=150, fitness=9, duration=4620 s, sequence=`[10, 8, 8, 0, 12, 0, 0, 12, 0, 11]`, contacts=`[417, 417, 519, 643, 643, 643, 643, 643, 938]`. Right: Mutation=0.4, population=50, fitness=9, duration=4620 s, sequence=`[10, 8, 6, 2, 0, 12, 0, 11, 11, 11]`, contacts=`[417, 462, 499, 512, 643, 643, 938, 938, 938]`

Table 4.2: Scenario A, task 1 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
417	12	8	1601428140	1601428200	60
519	8	0	1601428320	1601428380	60
643	0	12	1601429520	1601429580	60
643	12	0	1601429520	1601429580	60
938	0	11	1601431920	1601431980	60

Flight plan		
Time	**Node**	**Command**
1601428140	12	sim_send_fp 8
1601428320	8	sim_send_fp 0
1601429520	0	sim_get_data 1
1601429521	0	sim_send_data 11 1
1601431920	0	sim_send_fp 11

[5] Preliminary results with 300 seconds contact list resolution converged to 4800 seconds
[6] Intel(R) Core(TM) i7-990X @3.47GHz, 6 cores/12 threads, 24 GB of RAM

4.2.1.2. Task 2

A second task was analyzed, consisting of a store and forward operation over two targets. The idea is to execute the command `sim_get_data 1` over *Santiago*, and the command `sim_get_data 2` over *SAA* starting in *Tokyo* ground station to download both `data 1` and `data 2` also in *Tokyo* ground station. The definition file for this task is described in Listing 4.2.

Listing 4.2: Task 2 definition file

```
{
   "id": 2,
   "start": "tokyo",
   "end": "tokyo",
   "targets": [
      {"id": "stgo", "command": "sim_get_data 1", "result": "1"},
      {"id": "saa", "command": "sim_get_data 2", "result": "2"} ],
   "solution": null }
```

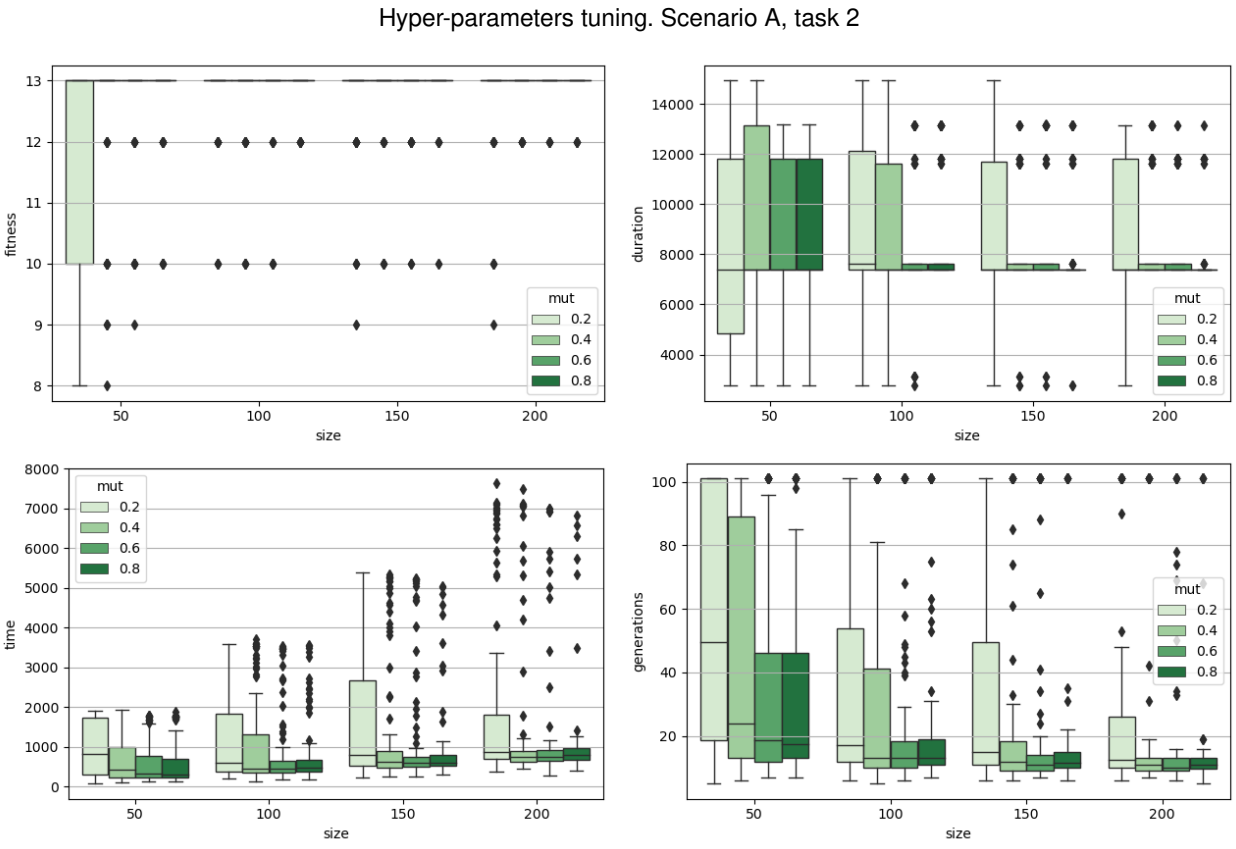


Figure 4.8: Scenario A, task 2 hyper-parameters tuning with maximum 13 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

This task used the constellation controller framework to obtain the contact and flight plans similar to the previous case. The hyper-parameters evaluation included 4 population sizes (50, 100, 150, and 200), 4 mutation rates (0.2, 0.4, 0.6, and 0.8), and 100 realizations for each combination. A summary of the results is presented in Fig. 4.8.

These results show that the algorithm converges to the fitness value of 13 contacts and a duration of 7380 seconds[7]; however higher mutation rate and larger population size are relevant to ensure convergence. In average, the algorithm obtain solutions in 1126.1 seconds and 30 generations with a minimum of 75.7 seconds and 5 generations. Two solutions are presented in Fig 4.9. The solution shown in Fig. 4.9-right is the sequence `S=[11, 8, 10, 8, 0, 12, 0, 11]` that can be simplified to `S=[11, 8, 10, 8, 0, 12, 0, 11]` and contacts `K=[97, 417, 417, 519, 643, 643, 938]`. This sequence can be translated to the contact plan and flight plan described in Table 4.3.

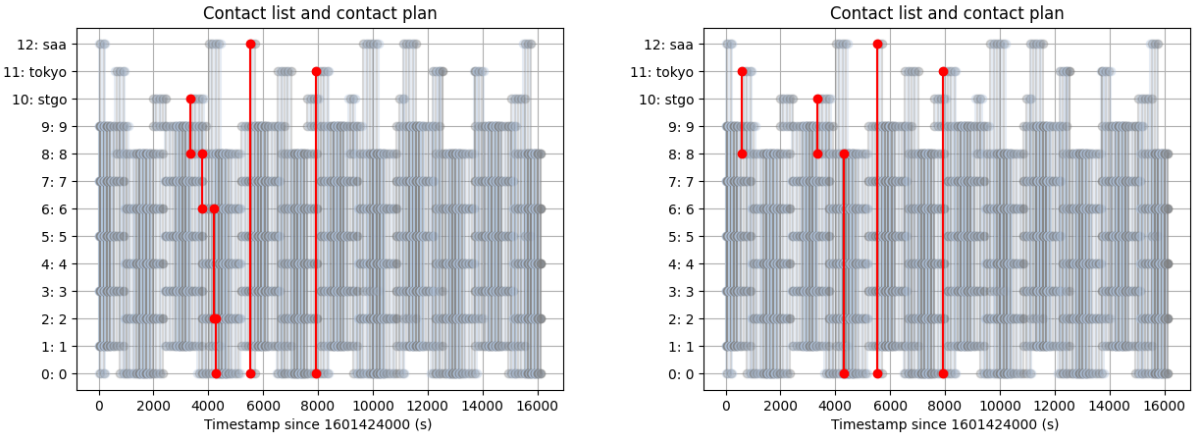


Figure 4.9: Scenario A, task 2 example results with maximum 13 hops. Contacts resolution 60 seconds. Left: Mutation=0.6, population=150, fitness=13, duration=7380 s, sequence=`[11, 8, 6, 8, 10, 8, 0, 0, 6, 0, 0, 12, 0, 11]`, contacts=`[97, 134, 134, 417, 417, 519, 519, 543, 543, 543, 643, 643, 938]`. Right: Mutation=0.4, population=200, fitness=13, duration=7380 s, sequence=`[11, 8, 10, 8, 0, 8, 8, 0, 8, 0, 12, 0, 11, 11]`, contacts=`[97, 417, 417, 519, 519, 519, 519, 519, 519, 643, 643, 938, 938]`

Table 4.3: Scenario A, task 2 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
97	11	8	1601424600	1601424660	60
417	8	10	1601427360	1601427420	60
417	10	8	1601427360	1601427420	60
519	8	0	1601428320	1601428380	60
643	0	12	1601429520	1601429580	60
643	12	0	1601429520	1601429580	60
938	0	11	1601431920	1601431980	60

Flight plan		
Time	**Node**	**Command**
1601424600	11	sim_send_fp 8
1601427360	8	sim_get_data 1
1601427361	8	sim_send_data 0 1
1601428320	8	sim_send_fp 0
1601429520	0	sim_get_data 2
1601429521	0	sim_send_data 11 1
1601431920	0	sim_send_data 11 2
1601431802	0	sim_send_fp 11

[7] Preliminary results with 300 seconds contact list resolution converged to 7500 seconds

4.2.2. Scenario B: 10 satellites Ad hoc constellation

Scenario B consists of a constellation with ten satellites in Ad hoc configuration, two ground stations, and one target. In this scenario, the orbital parameters are chosen randomly with an altitude between 500 km and 600 km. A simulation of 16200 seconds with 60 seconds resolution to calculate the contact opportunities resulted in 1573 contacts. The complete details of this configuration are shown in Table 4.4.

Figure 4.10 shows a static representation of the satellites' ground tracks after 45 minutes (Approx. half orbit). A better view of this scenario is obtained from the contact list FSM representation shown in Fig. 4.11 (using a contact resolution of 300 seconds to improve clarity). The reader can note that in contrast with Fig. 4.5 in the ad-hoc configuration, the contact opportunities are less regular and frequent.

Table 4.4: Scenario B description

Simulation time				
Start time: 2020-09-30T00:00:00 UTC (1601424000 Unix time)				
Simulation time: 16200 seconds (∼3 orbits)				
Simulation resolution: 30 seconds				
Contact list resolution: 60 seconds				
Number of contacts: 1573				
Satellites				
Node	Period (min)	Incl	Mean anom.	R. ascension
0	95.58	99.0°	24.0°	136.0°
1	95.36	89.0°	146.0°	78.0°
2	95.53	96.0°	66.0°	161.0°
3	95.45	97.0°	80.0°	178.0°
4	96.21	98.0°	126.0°	177.0°
5	94.97	83.0°	52.0°	59.0°
6	94.71	91.0°	150.0°	73.0°
7	95.37	92.0°	171.0°	84.0°
8	96.18	95.0°	61.0°	20.0°
9	95.81	82.0°	121.0°	14.0°
Ground stations and targets				
Node	Lat.	Lon.	Alt.	Reference
14	-33.3833°	-70.7833°	476 m	Santiago, Chile
16	35.6830°	139.7670°	5 m	Tokyo, Japan
18	-15.0°	-15.0°	500 km	S. Atlantic Anomaly

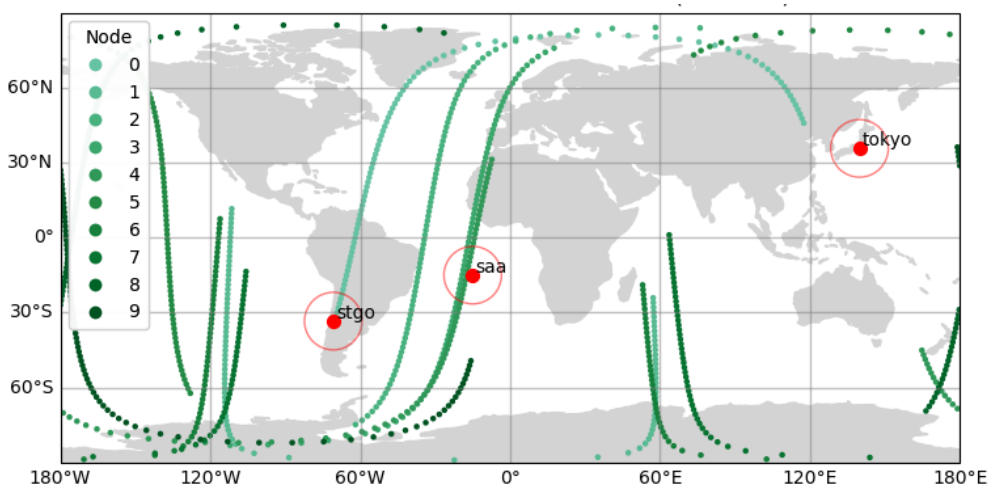


Figure 4.10: Scenario B satellite tracks after 45 minutes (Approx. half orbit), ground stations and targets locations (red).

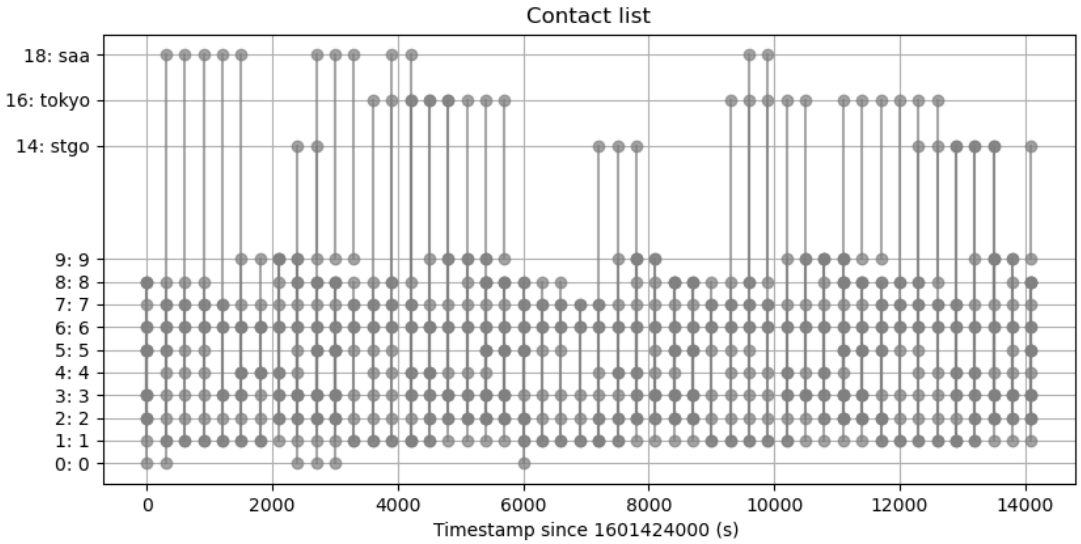


Figure 4.11: Scenario B contact list in FSM representation. Contact opportunities are grey lines connecting two nodes. Note that lines connecting nodes in a particular state may be overleaped. Contacts resolution is 300 seconds to improve clarity

4.2.2.1. Task 1

The framework was used to solve Task 1 as described in the previous scenario (See 4.2.1.1). The results of the hyper-parameters tuning are shown in Fig. 4.12. Results show convergence of the algorithm when the population size is larger than 100 individuals and converges to the fitness value (duration or delivery time) of 4860 seconds[8]. Compared with the 4620 seconds of the Walker configuration, there is a 240 seconds difference or 4 states. The latter means that in terms of performance, the Walker configuration presents slightly better results.

[8] Preliminary results with 300 seconds contact list resolution converged to 5100 seconds

Figure 4.13 show two example of valid solutions. In particular the solution in Fig. 4.13-left correspond to the sequence `S=[10, 2, 2, 2, 5, 5, 8, 12, 8, 11]` and contacts `K=[732, 732, 732, 825, 825, 835, 952, 952, 1196]` obtained with a mutation rate of 0.6 and a population size of 150 individuals. This solutions evaluates to 4860 seconds (delivery time) and took 1935.3 seconds in 100 generations to get the result. The complete contact list and contact plant for this solution is detailed in Table 4.5

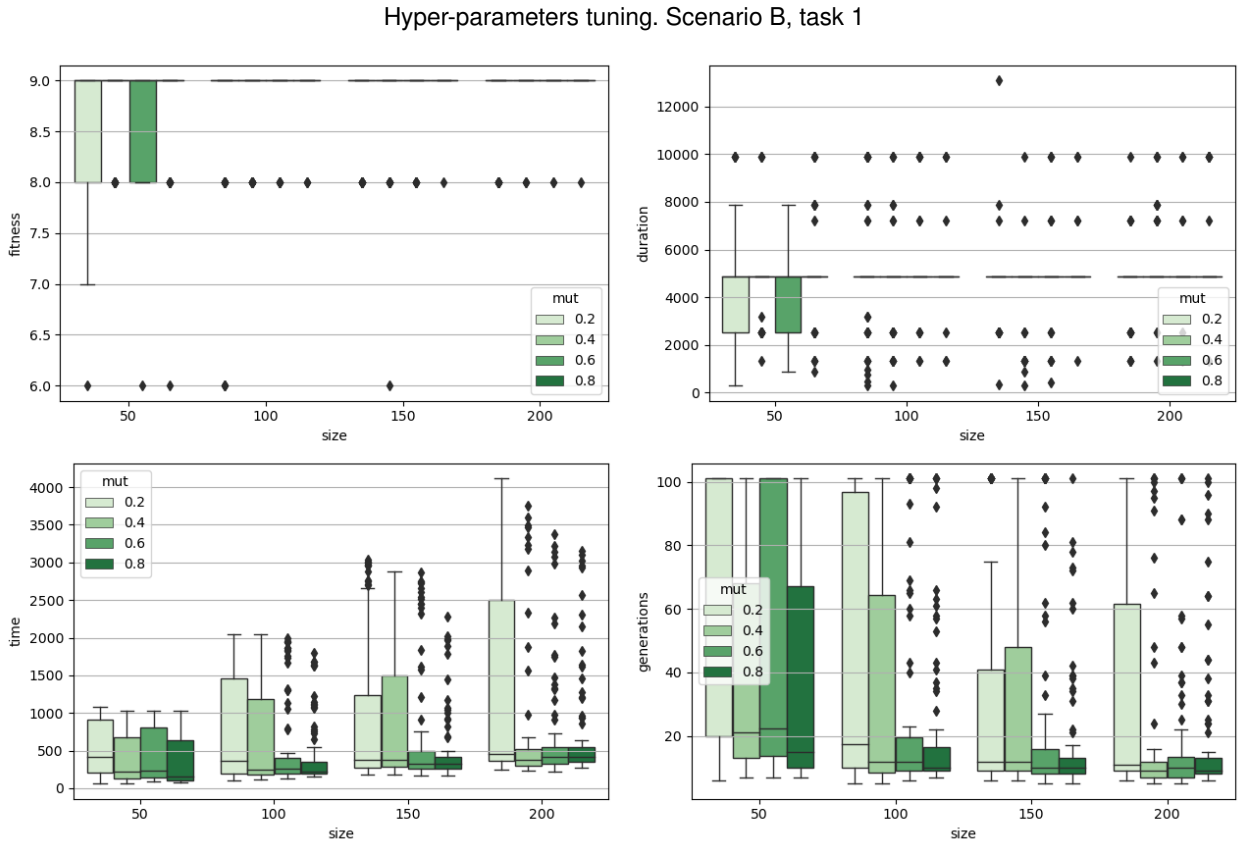


Figure 4.12: Scenario B, task 1 hyper-parameters tuning with maximum 8 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

Table 4.5: Scenario B, task 1 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
732	10	2	1601431440	1601431500	60
825	2	5	1601432400	1601432460	60
952	8	12	1601433720	1601433780	60
1196	8	11	1601436240	1601436300	60

Flight plan		
Time	**Node**	**Command**
1601431440	10	sim_send_fp 2
1601432400	2	sim_send_fp 5
1601433720	8	sim_get_data 1
1601436240	8	sim_send_data 11 1
1601436241	8	sim_send_fp 11

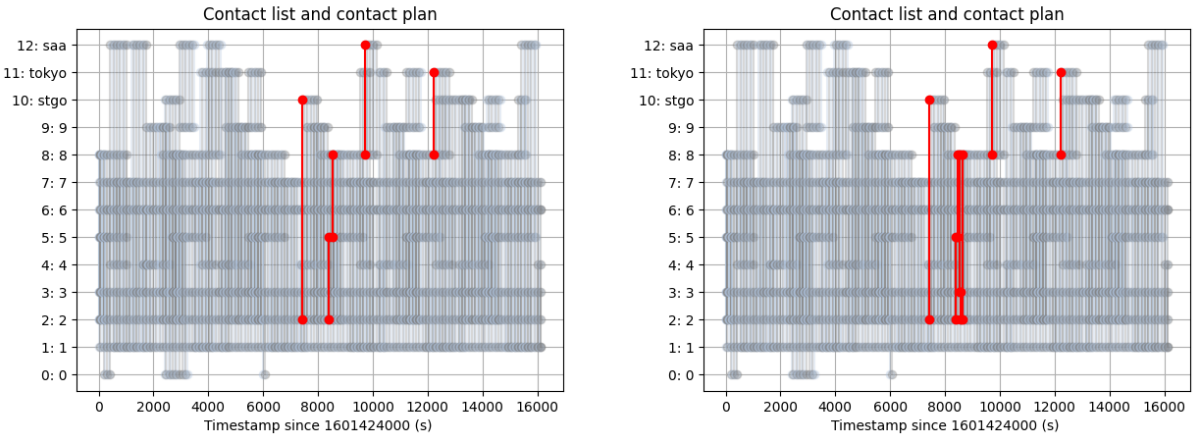


Figure 4.13: Scenario B, task 1 example results with maximum 9 hops. Left: Mutation=0.6, population=150, fitness=9, duration=4860 s, sequence=[10, 2, 2, 2, 5, 5, 8, 12, 8, 11], contacts=[732, 732, 732, 825, 825, 835, 952, 952, 1196]. Right: Mutation=0.8, population=200, fitness=9, duration=4860 s, sequence=[10, 2, 5, 8, 3, 2, 8, 12, 8, 11], contacts=[732, 825, 831, 833, 842, 856, 952, 952, 1196]

4.2.2.2. Task 2

Similar to the previous section, task 2 was solved, and the performance of the GA was evaluated by hyper-parameters analysis. Fig. 4.14 shows the results for this task, and scenario combination. The results exhibit convergence for a population size of 100 individuals or greater. The algorithm converges to the value of 7680 seconds[9] (delivery time), solving the problem in 623 seconds. The difference with scenario A in Sec. 4.2.1 (Walker configuration) is 300 seconds (5 states). Thus, Walker configurations show slightly better performance.

Figure 4.15 show two example of valid solutions. In particular the solution in Fig. 4.15-left correspond to the sequence S=[16, 2, 2, 14, 2, 2, 5, 8, 18, 8, 16, 16] and contacts K=[111, 111, 176, 176, 176, 202, 209, 234, 234, 289, 289] obtained with a mutation rate of 0.6 and a population size of 150 individuals. This solutions evaluates to 7800 seconds (delivery time) and took 82.16 seconds in 7 generations to get the result. Note that, despite this task is slightly more complex than Task 1, the time and generations required to find a solution are not affected significantly. Details of the contact plan and flight plan corresponding to this solution are found in Table 4.6

[9] Preliminary results with 300 seconds contact list resolution converged to 7800 seconds

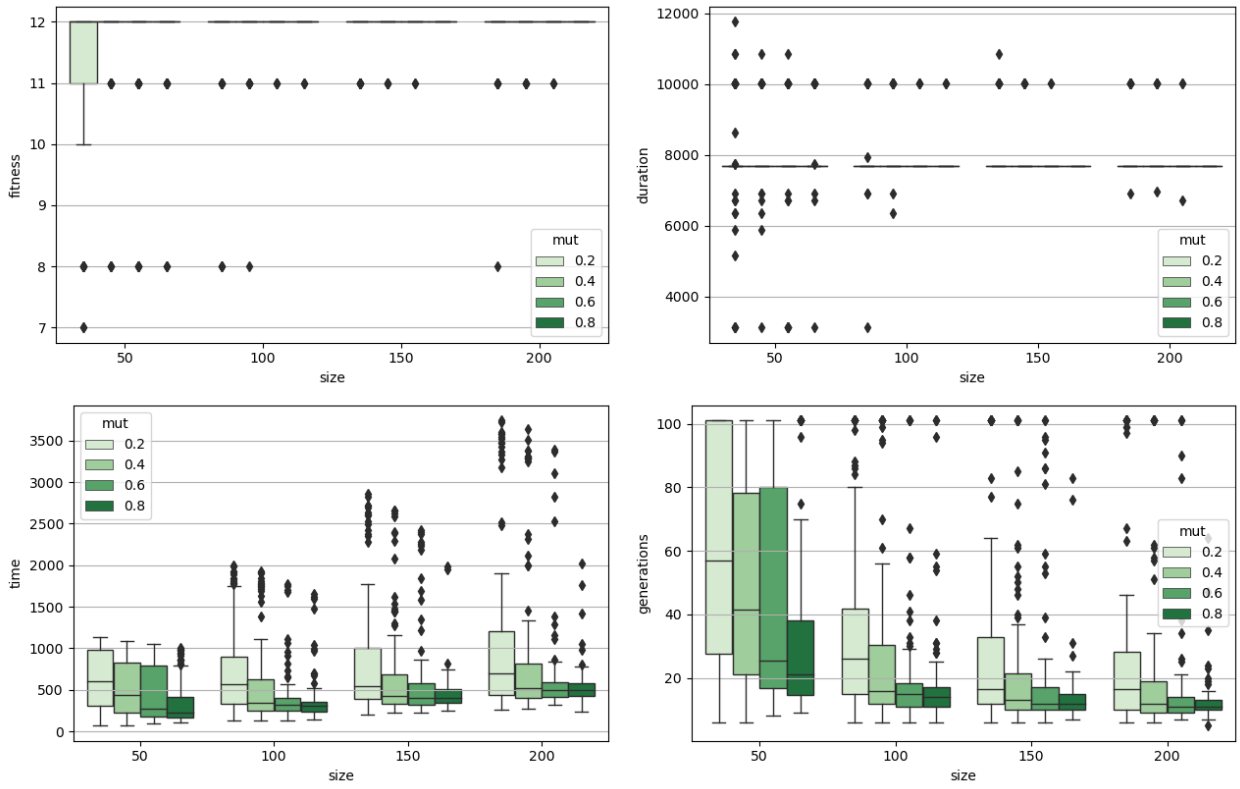


Figure 4.14: Scenario B, task 2 hyper-parameters tuning with maximum 8 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

Table 4.6: Scenario B, task 2 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
464	11	2	1601428620	1601428680	60
732	2	10	1601431440	1601431500	60
782	2	9	1601431980	1601432040	60
793	9	3	1601432100	1601432160	60
816	3	5	1601432340	1601432400	60
831	5	8	1601432460	1601432520	60
952	8	12	1601433720	1601433780	60
1196	8	11	1601436240	1601436300	60

Flight plan		
Time	**Node**	**Command**
1601428620	11	sim_send_fp 2
1601431440	2	sim_get_data 1
1601431980	2	sim_send_data 9 1
1601431981	2	sim_send_fp 9
1601432100	9	sim_send_data 3 1
1601432101	9	sim_send_fp 3
1601432340	3	sim_send_data 5 1
1601432341	3	sim_send_fp 5
1601432460	5	sim_send_data 8 1
1601432461	5	sim_send_fp 8
1601433720	8	sim_get_data 2
1601436240	8	sim_send_data 11 1
1601436241	8	sim_send_data 11 2
1601436242	8	sim_send_fp 11

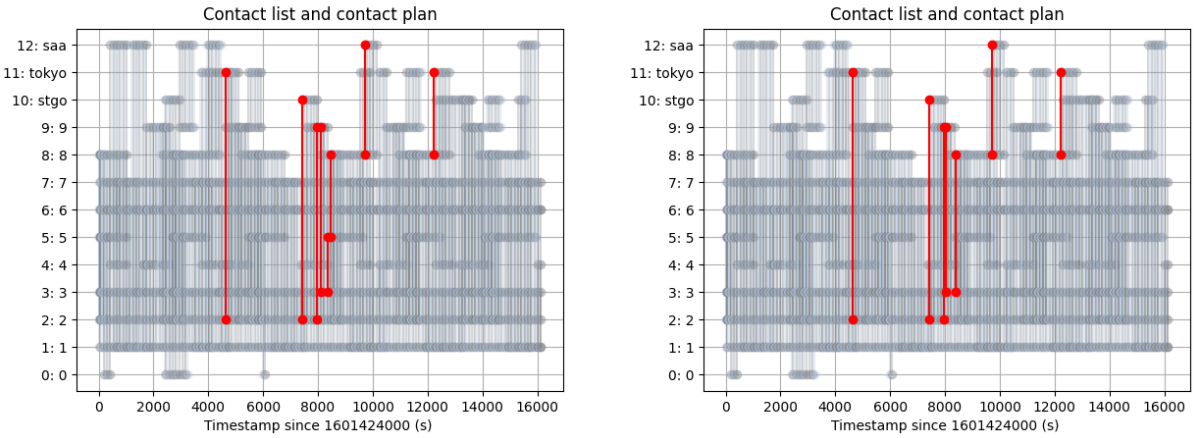


Figure 4.15: Scenario B, task 2 example results with maximum 12 hops. Left: Mutation=0.6, population=150, fitness=12, duration=7680 s, sequence=[11, 2, 10, 2, 9, 9, 3, 5, 8, 12, 8, 11, 11], contacts=[464, 732, 732, 782, 782, 793, 816, 831, 952, 952, 1196, 1196]. Right: Mutation=0.8, population=50, fitness=12, duration=7680 s, sequence=[11, 2, 10, 2, 9, 3, 8, 2, 8, 12, 8, 11, 11], contacts=[464, 732, 732, 782, 786, 822, 838, 838, 952, 952, 1196, 1196]

4.2.3. Scenario C: 100 satellites Walker constellation

This case study uses 100 satellites to evaluate the genetic algorithm's performance in a large constellation scenario. In particular, the effect of the number of the satellites and the GA hyper-parameters. Scenario C consists of a constellation with 100 satellites in Walker configuration (20 orbital planes, 5 satellites per plane, 500 km altitude), two ground stations, and one target. Details are described in Table 4.7. A simulation of 12000 seconds with a resolution to calculate the ISL of 600 seconds resulted in 16057 contacts. Restrictions and assumptions remain equal.

Table 4.7: Scenario C description

Simulation time
Start time: 2020-09-30T00:00:00 UTC (1601424000 Unix time)
Simulation time: 12000 seconds (2.1 orbits)
Simulation resolution: 30 seconds
Contact list resolution: 600 seconds
Number of contacts: 16057
Satellites
Node: 0-99
Period: 94.47 min
Inclination: 97°
Mean anomaly: 0, 36°, 72°, 108°, and 144°
Right ascension step: 9° (0°, 9°, 18°, ..., 171°)

Figure 4.16 shows an static representation of the satellites ground tracks after 45 minutes (Approx. half orbit) and Fig. 4.17 shows the contact list FSM representation. The tracks and contacts' visualization is more challenging here due to the data density.

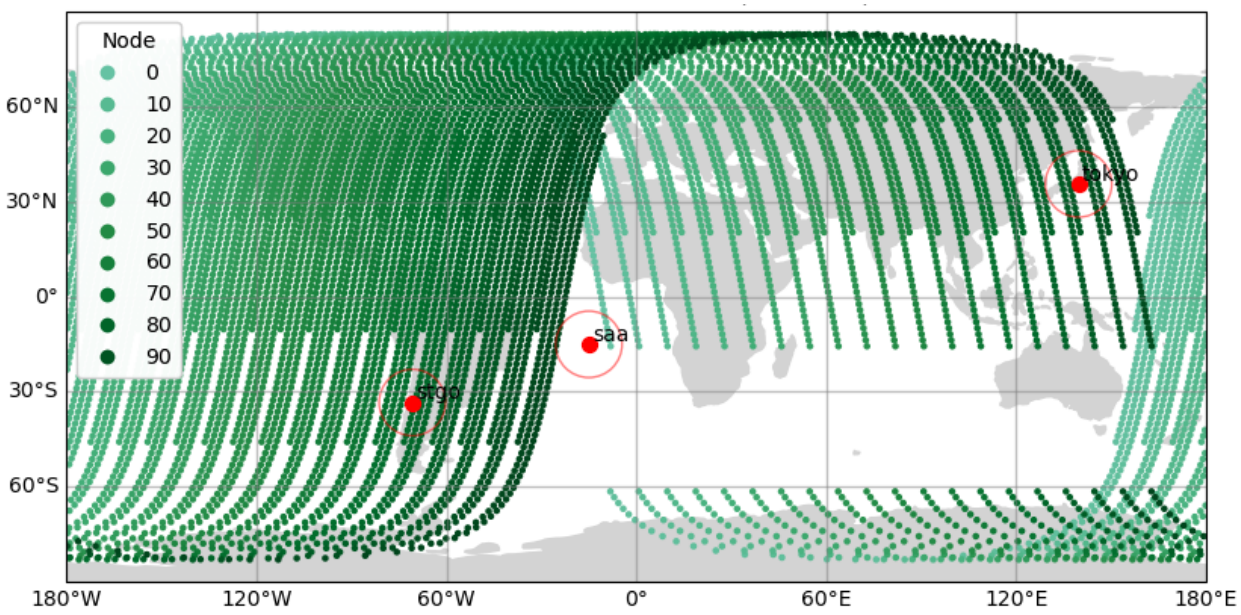


Figure 4.16: Scenario C satellite tracks after 45 minutes (Approx. half orbit), ground stations and targets locations (red).

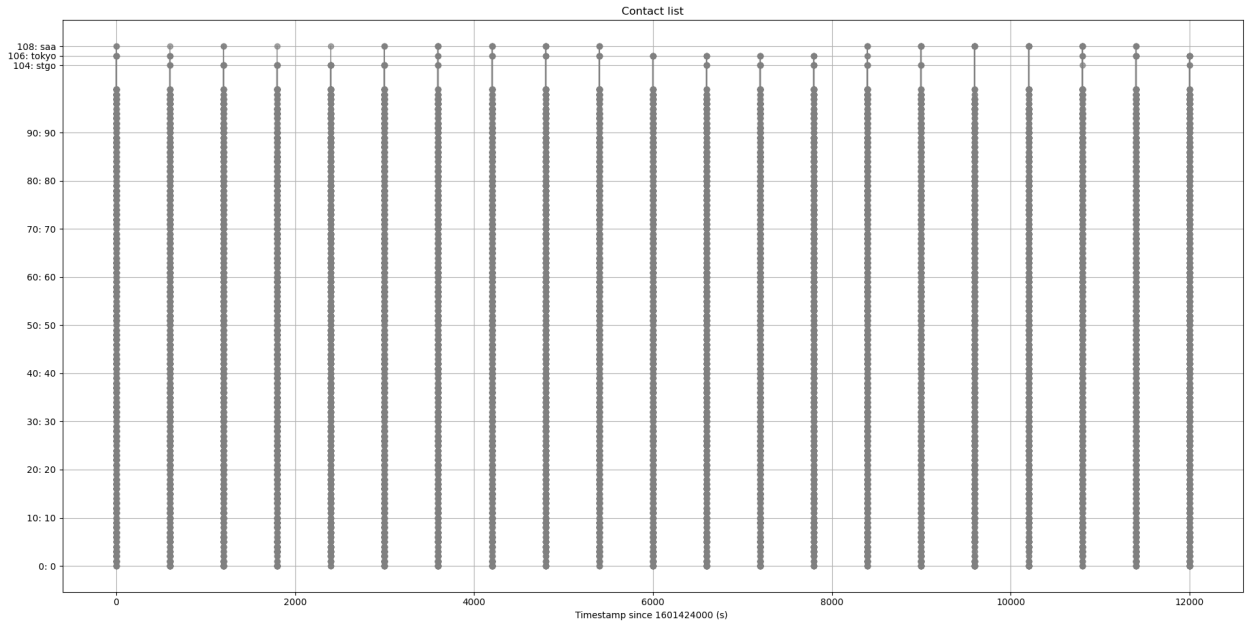


Figure 4.17: Scenario C contact list in FSM representation. Contact opportunities are grey lines connecting two nodes. Note that lines connecting nodes in a particular state may be overleaped, please refer to the annexes to see the full table

4.2.3.1. Task 1

The framework is used to solve task 1 described in the previous scenario. The result of the hyper-parameters tuning is shown in Fig. 4.18. Increasing the number of satellites produces a greater number of contacts. As the results show, increasing the population variability is crucial to ensure the convergence of the GA. In this case, a population larger than 200 individuals is required. On the other hand, it is possible to lower the contact resolution to reduce the contact list length and speed-up calculations. For this population size and contact resolution, solutions are obtained in 187 seconds on average.

Graphically display the contact plan is not practical in this case due to the density of the data. For this reason, only an example solution is detailed here, in this case the sequence `S=[104, 85, 6, 6, 108, 6, 6, 46, 0, 106]` with contacts `K=[4092, 4985, 4985, 6203, 6203, 6203, 8813, 9786, 10493]` with mutation rate of 0.4 and population size of 200. This solution was obtained after 15 generations in 153.6 seconds with a fitness value of 5400 seconds (delivery time). Table 4.8 details the corresponding contact plan and flight plan.

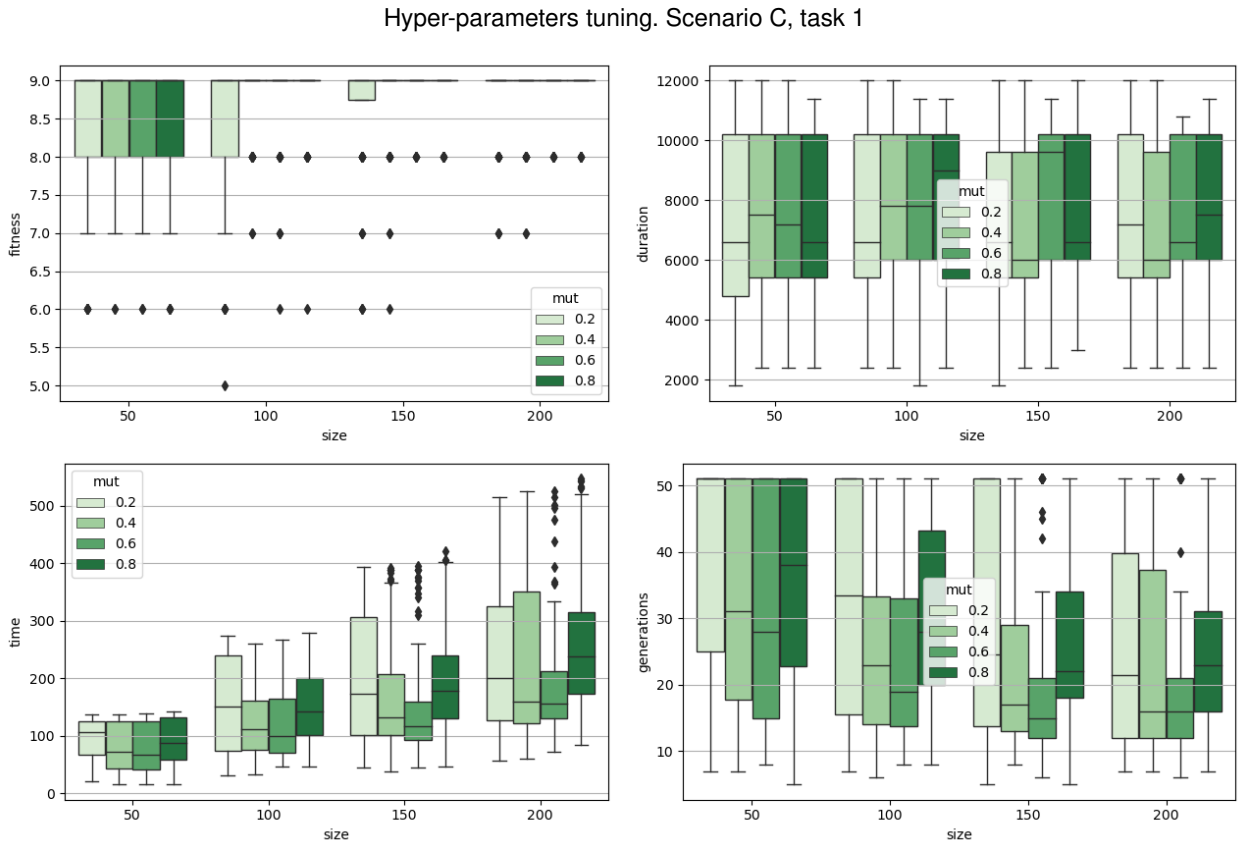


Figure 4.18: Scenario C, task 1 hyper-parameters tuning with maximum 9 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

Table 4.8: Scenario C, task 1 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
4092	104	85	1601427000	1601427600	600
4985	85	6	1601427600	1601428200	600
6203	6	108	1601428800	1601429400	600
6203	108	6	1601428800	1601429400	600
8813	6	46	1601430600	1601431200	600
9786	46	0	1601431200	1601431800	600
10493	0	106	1601431800	1601432400	600

Flight plan		
Time	**Node**	**Command**
1601427000	104	sim_send_fp 85
1601427600	85	sim_send_fp 6
1601428800	6	sim_get_data data1
1601430600	6	sim_send_fp 46
1601430601	6	sim_send_data 46 data1
1601431200	46	sim_send_fp 0
1601431201	46	send_data 0 data1
1601431800	0	sim_send_fp 106
1601431801	0	sim_send_data 106 data1

4.2.3.2. Task 2

Task 2 was also evaluated in the 100 satellites scenario. As Figure 4.19 shows, the hyper-parameters space used is not enough to ensure the convergence of the GA. This situation restates the idea of requiring great population variability as the size of the problem increases. Here the test limited the maximum number of generations to 50, sufficient to get valid results.

Despite of the results of the hyper-parameters analysis in this case (the values was maintained equal among the different tests) valid results as obtained using a mutation rate of 0.6 and a population size of 1000 individuals. With this values, the solution with sequence `S=[104, 85, 6, 6, 108, 6, 6, 46, 0, 106]` and contacts `K=[4092, 4985, 4985, 6203, 6203, 6203, 8813, 9786, 10493]` was obtained after 16 generations in 401.0 seconds with a fitness value of 8400 seconds (delivery time). Table 4.9 details the corresponding contact plan and flight plan.

Table 4.9: Scenario C, task 2 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
249	106	91	1601424000	1601424600	600
1006	91	71	1601424600	1601425200	600
3377	71	104	1601426400	1601427000	600
4915	71	86	1601427600	1601428200	600
5908	86	0	1601428200	1601428800	600
7364	0	108	1601429400	1601430000	600
9051	0	5	1601430600	1601431200	600
10042	5	106	1601431800	1601432400	600

Flight plan		
Time	**Node**	**Command**
1601424000	106	sim_send_fp 91
1601424600	91	sim_send_fp 6
1601426400	71	sim_get_data data1
1601427600	71	sim_send_fp 86
1601427601	71	sim_send_data 86 data1
1601428200	86	sim_send_fp 0
1601428201	86	sim_send_data 0 data1
1601429400	0	sim_get_data data2
1601430600	0	sim_send_fp 5
1601430601	0	sim_send_data 5 data1
1601430602	0	sim_send_data 5 data2
1601431800	5	sim_send_fp 106
1601431801	5	sim_send_data 106 data1
1601431802	5	sim_send_data 106 data2

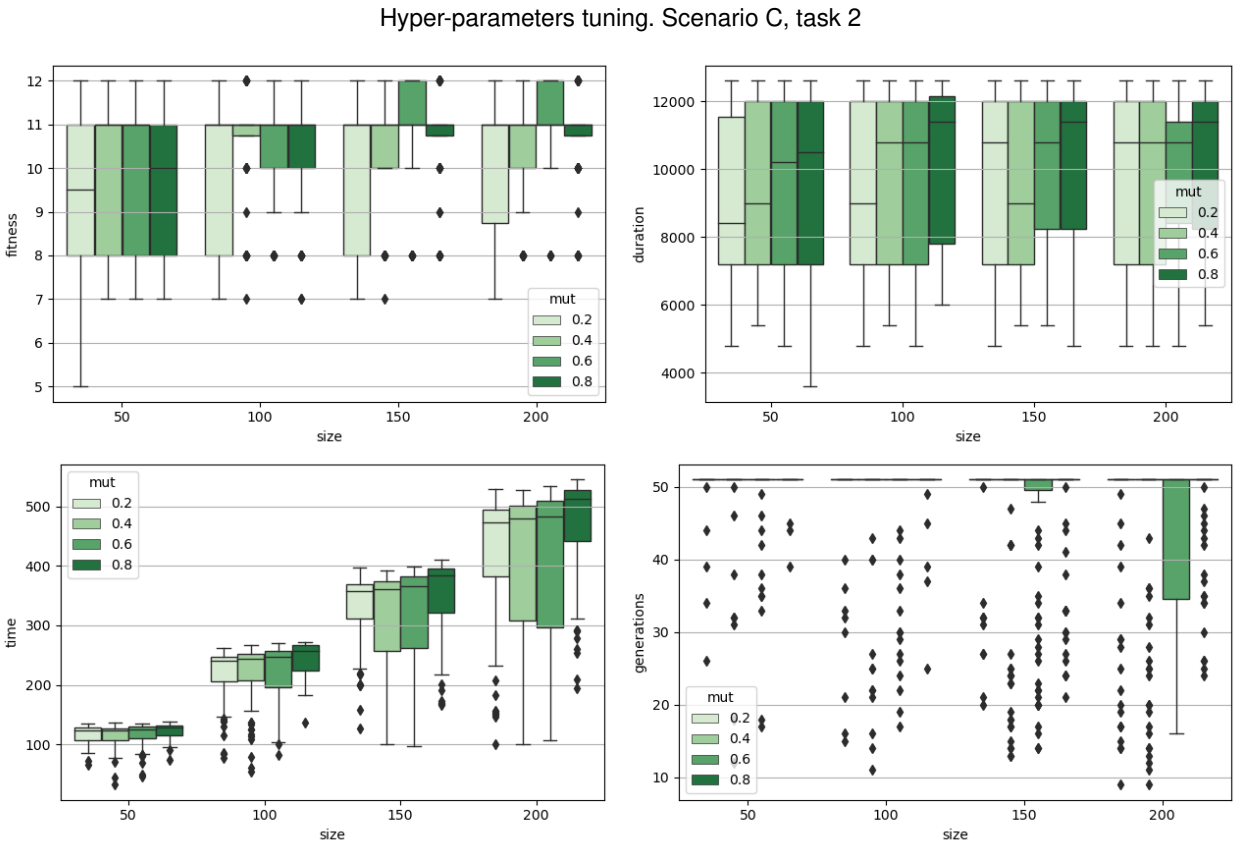


Figure 4.19: Scenario C, task 2 hyper-parameters tuning with maximum 9 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

4.2.4. Scenario D: 100 satellites Ad hoc constellation

Another large constellation scenario was tested, this case with 100 satellites in Ad hoc configuration. In this scenario, the orbital parameters are chosen randomly with an altitude between 500 km. and 600 km. Details are described in Table 4.13. A simulation of 14400 seconds with a resolution to calculate the ISL of 600 seconds resulted in 22371 contacts. Restrictions and assumptions remain equal.

Table 4.10: Scenario C description

Simulation time
Start time: 2020-09-30T00:00:00 UTC (1601424000 Unix time)
Simulation time: 14400 seconds (2.67 orbits)
Simulation resolution: 30 seconds
Contact list resolution: 600 seconds
Number of contacts: 22371
Satellites
Node: $[0, 99]$
Period: $unif$(94.47, 96.54) min
Inclination: $unif$(80°, 100°)
Mean anomaly: $unif$(0°, 180°)
Right ascension angle: $unif$(0°, 180°)

Figure 4.20 shows an static representation of the satellites ground tracks after 45 minutes (Approx. half orbit) and the contact list FSM representation is shown in Fig. 4.21. In this case, the tracks and contacts' visualization is more challenging due to the data's density.

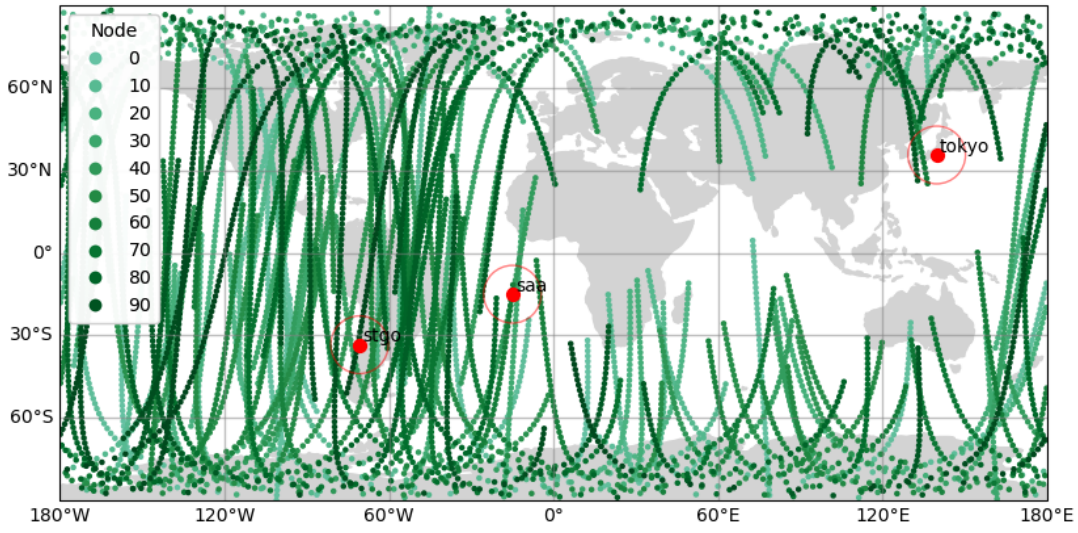


Figure 4.20: Scenario D satellite tracks after 45 minutes (Approx. half orbit), ground stations and targets locations (red).

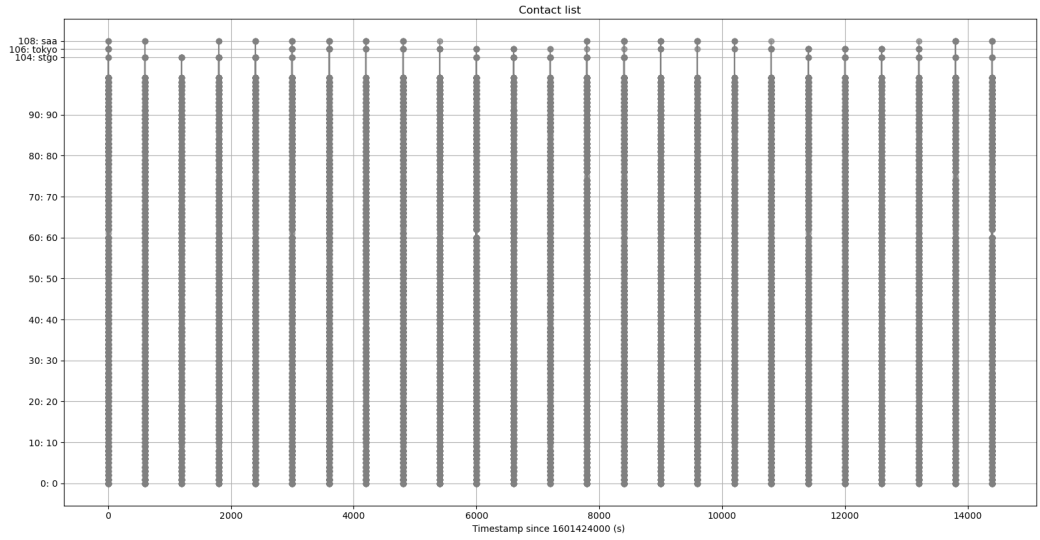


Figure 4.21: Scenario D contact list in FSM representation. Contact opportunities are grey lines connecting two nodes. Note that lines connecting nodes in a particular state may be overleaped, please refer to the annexes to see the full table

4.2.4.1. Task 1

Results of the hyper-parameters tuning for task 1 are shown in Fig. 4.22. As a solution visualization is not practical, an example solution is described. The following solution was obtained with a mutation rate of 0.2 and a population size of 100 individual and correspond to the sequence `S=[104, 49, 37, 38, 87, 97, 108, 97, 77, 106]` and contacts `K=[14, 977, 2498, 3509, 4003, 4833, 4833, 6647, 7545]`. This solution value is 5400 seconds and is obtained in 61.7 seconds or 10 generations. These results are exactly equal to the Walker configuration; thus, it is impossible to determine which configuration performs better with this contacts resolution. The contact plan and flight plan are detailed in Table 4.11.

Table 4.11: Scenario D, task 1 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
14	104	49	1601424000	1601424600	600
977	49	37	1601424600	1601425200	600
2498	37	38	1601425200	1601425800	600
3509	38	87	1601425800	1601426400	600
4003	87	97	1601426400	1601427000	600
4833	97	108	1601427000	1601427600	600
6647	97	77	1601428200	1601428800	600
7545	77	106	1601428800	1601429400	600

Flight plan		
Time	**Node**	**Command**
1601424000	104	sim_send_fp 49
1601424600	49	sim_send_fp 37
1601425200	37	sim_send_fp 38
1601425800	38	sim_send_fp 87
1601426400	87	sim_send_fp 97
1601427000	97	sim_get_data data1
1601428200	97	sim_send_fp 77
1601428201	97	sim_send_data 77 data1
1601428800	77	sim_send_data 106 data1

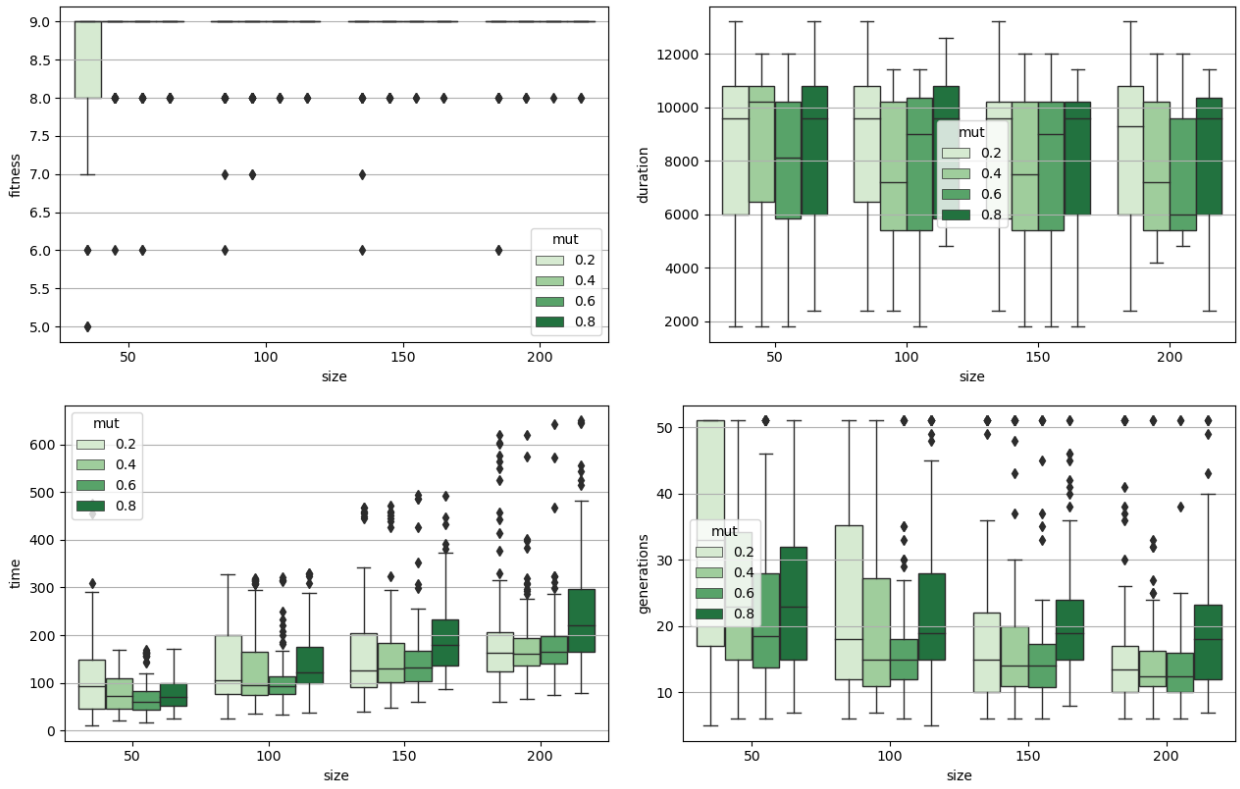


Figure 4.22: Scenario D, task 1 hyper-parameters tuning with maximum 9 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

4.2.4.2. Task 2

Similarly the hyper-parameters tuning for task 2 is shown in Fig. 4.23. The following solution was obtained with a mutation rate of 0.2 and a population size of 200 individuals, corresponding to the sequence S=[106, 73, 8, 104, 8, 70, 78, 78, 108, 78, 76, 78, 106] and contacts K=[422, 1091, 3027, 3027, 4863, 5812, 5812, 7551, 7551, 9453, 9453, 11209]. This solution evaluates to 7800 seconds and was obtained in 158.42 seconds or 12 generations. The variance of the results does not enable a fair comparison of each configuration performance. The contact plan and flight plan are detailed in Table 4.12.

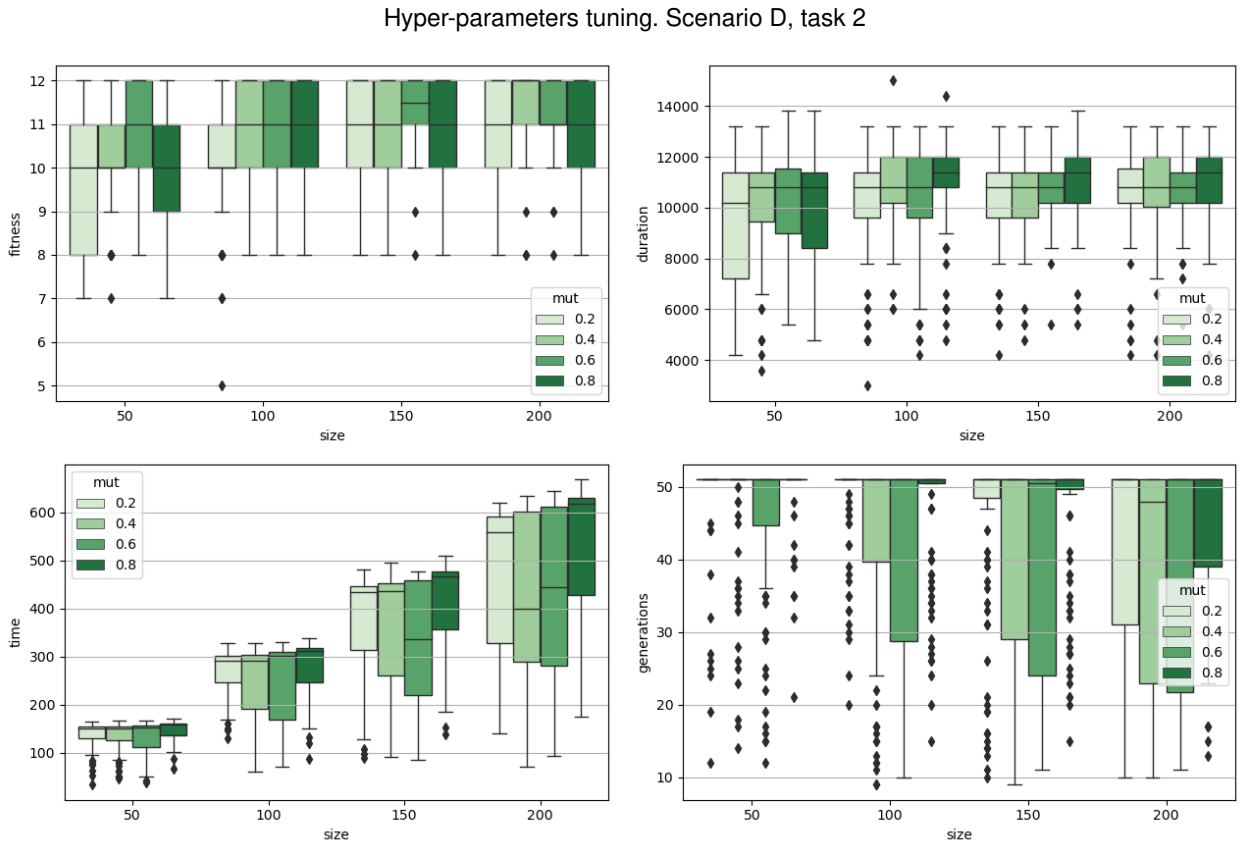


Figure 4.23: Scenario D, task 2 hyper-parameters tuning with maximum 12 hops. The box plots show the fitness value as the number of valid contacts in the solution sequence (upper left), the duration -or delivery time- of the solution in seconds (upper right), the time required to find the solution (lower left), and the number of generations required to find the solution (lower right).

Table 4.12: Scenario D, task 2 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
422	106	73	1601424000	1601424600	600
1091	73	8	1601424600	1601425200	600
3027	8	104	1601425800	1601426400	600
4863	8	70	1601427000	1601427600	600
5812	70	78	1601427600	1601428200	600
7551	78	108	1601428800	1601429400	600
9453	78	76	1601430000	1601430600	600
11209	78	106	1601431200	1601431800	600

Flight plan		
Time	**Node**	**Command**
1601424000	106	sim_send_fp 73
1601424600	73	sim_send_fp 8
1601425800	8	sim_get_data data1
1601427000	8	sim_send_fp 70
1601427001	8	sim_send_data 70 data 1
1601427600	70	sim_send_fp 78
1601427601	70	sim_send_data 78 data 1
1601428800	78	sim_get_data data2
1601430000	78	sim_send_fp 76
1601430001	78	sim_send_data 76 data 1
1601430002	78	sim_send_data 76 data 2
1601431200	78	sim_send_fp 106
1601431201	78	sim_send_data 106 data 1
1601431202	78	sim_send_data 106 data 2

4.2.5. Scenario E: 1000 satellites Ad hoc constellation

A mega constellation scenario with 1000 satellites is tested in this section. As shown in Sections 4.1 and 4.2.6 these kind of scenarios are computationally expensive so only the Ad hoc configuration was tested. Visualizing and analyzing data in these large scenarios is especially challenging, so displaying tracks, contacts, and contact plans in static plots is worthless. For these reasons, the results showed are limited to the contact plan and flight plan tables.

This scenario uses an Ad hoc configuration, so the orbital parameters are chosen randomly with an altitude between 500 km. and 600 km. Details are described in Table 4.13. A simulation of 16200 seconds with a resolution to calculate the ISL of 60 seconds resulted in 16106905 contacts. Restrictions and assumptions remain equal.

Table 4.13: Scenario E description

Simulation time
Start time: 2020-09-30T00:00:00 UTC (1601424000 Unix time)
Simulation time: 16200 seconds (3.0 orbits)
Simulation resolution: 30 seconds
Contact list resolution: 60 seconds
Number of contacts: 16106905
Satellites
Node: $[0, 999]$
Period: $unif$(94.47, 96.54) min
Inclination: $unif$(80°, 100°)
Mean anomaly: $unif$(0°, 180°)
Right ascension angle: $unif$(0°, 180°)

4.2.5.1. Task 1

The hyper-parameters analysis shown in previous scenarios is impractical here due to the required computational time. Also, the visualization of the solution is not useful due to the information density. Instead, a limited set of trials were executed with a fixed set of hyper-parameters: mutation rate of 0.6 and population size of 100 individuals and 20 generations maximum. An example solution is corresponding to the sequence `S=[2649754, 3090991, 3644957, 3736247, 4857548, 4857548, 4857548, 8410834, 10909304]` and contacts `K=[14, 977, 2498, 3509, 4003, 4833, 4833, 6647, 7545]` is detailed in Table 4.14. This solution value is 8340 seconds and was obtained in 7855.38 seconds or 13 generations.

4.2.5.2. Task 2

Similar to the previous case, task 2 tests were executed the following set of hyper-parameters: mutation rate of 0.6 and, population size of 100 individuals and 20 generations maximum. An example solution is corresponding to the sequence `S=[1001, 380, 212, 1000, 212, 518, 432, 746, 298, 1002, 298, 521, 1001]` and contacts `K=[5689021, 5739086,`

8617418, 8617418, 9301191, 9333460, 9392656, 9495685, 10047152, 10047152, 10163231, 12714813] is detailed in Table 4.15. This solution evaluates to 7200 seconds and was obtained in 10027.53 seconds or 18 generations.

Table 4.14: Scenario E, task 1 example contact plan and flight plan solution.

Contact plan					
contact	**from**	**to**	**start**	**end**	**duration**
2649754	1000	932	1601426640	1601426700	60
3090991	932	848	1601427060	1601427120	60
3644957	848	826	1601427600	1601427660	60
3736247	826	346	1601427660	1601427720	60
4857548	346	1002	1601428860	1601428920	60
8410834	346	758	1601432400	1601432460	60
10909304	758	1001	1601434920	1601434980	60

Flight plan		
Time	**Node**	**Command**
1601426640	1000	sim_send_fp 932
1601427060	932	sim_send_fp 848
1601427600	848	sim_send_fp 826
1601427600	826	sim_send_fp 346
1601428860	346	sim_get_data 1
1601432400	346	sim_send_data 346 1
1601432401	346	sim_send_fp 758
1601434920	758	sim_send_data 1001 1
1601434921	758	sim_send_fp 1001

Table 4.15: Scenario E, task 2 example contact plan and flight plan solution.

Contact plan					
contact	from	to	start	end	duration
5689021	1001	380	1601429640	1601429700	60
5739086	380	212	1601429700	1601429760	60
8617418	212	1000	1601432580	1601432640	60
9301191	212	518	1601433240	1601433300	60
9333460	518	432	1601433300	1601433360	60
9392656	432	746	1601433360	1601433420	60
9495685	746	298	1601433420	1601433480	60
10047152	298	1002	1601434080	1601434140	60
10163231	298	521	1601434200	1601434260	60
12714813	521	1001	1601436780	1601436840	60

Flight plan		
Time	Node	Command
1601429640	1001	sim_send_fp 380
1601429700	380	sim_send_fp 212
1601432580	212	sim_get_data 1
1601433240	212	sim_send_data 518 1
1601433241	212	sim_send_fp 518
1601433300	518	sim_send_data 432 1
1601433301	518	sim_send_fp 432
1601433360	432	sim_send_data 746 1
1601433361	432	sim_send_fp 746
1601433420	746	sim_send_data 298 1
1601433421	746	sim_send_fp 298
1601434080	298	sim_get_data 2
1601434200	298	sim_send_data 521 1
1601434201	298	sim_send_data 521 2
1601434202	298	sim_send_fp 512
1601436780	521	sim_send_data 1001 1
1601436780	521	sim_send_data 1001 2
1601436780	521	sim_send_fp 1001

4.2.6. Genetic algorithm scalability

Finally, the scalability of the evolutive contact plan design algorithm versus the number of satellites in the constellation was studied. The Ad hoc scenario was re-executed under similar conditions with 10, 100, and 1000 satellites constellation sizes. Thus, all test were executed in the *slims* nodes[10] of the NLHPC cluster, using one core per realization, and 10 realizations per test in parallel. All test used the same mutation rate $m = 0.6$, population size $s = 200$, and maximum number of generations $i = 50$. The contact lists calculated in Section 4.1 with a resolution of 60 seconds were used, so these test measures only the time required by the genetic algorithm to create a valid contact plan. Note that this is a sequential process, and scalability is measured against the number of satellites.

[10] 2 x Intel Xeon E5-2660v2 @ 2,20GHz, 10 cores each, 48 GB of RAM

Figure 4.24 summarizes the results. It shows that the execution time does not vary significantly with the task complexity but with the length of the contact list. The contact list size scales exponentially so does the contact plan design complexity. Results confirm that managing large constellations of hundreds of nodes is feasible. It is possible to plan several hours in a portion of time. However, planning for a mega constellation of thousands of nodes might be challenging in terms of computational resources and execution time. Scenarios of hundreds of satellites can be managed in regular workstation computers, but larger scenarios require high-performance computing capabilities and implementing some level of parallelization inside the genetic algorithm. For example, it is worth studying parallelizing the population fitness function evaluation.

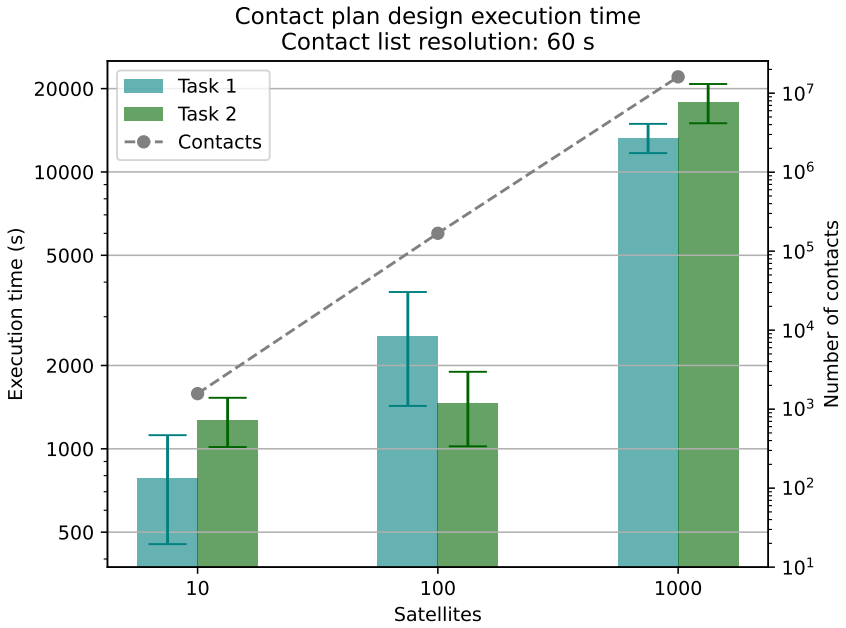


Figure 4.24: Evolutive contact plan design scalability results

4.3. Flight software verification and validation

As discussed in Section 2.4, the CubeSat community is concerned about the design and quality of the Flight Software (FS). However, defining and ensuring rigorous software quality criteria is not simple. CubeSat projects apply different techniques such as extensive testing [69, 57], hardware in the loop simulation [70], static analysis, or the use of of of certified language standards [50]. This section provides a methodology that utilizes software engineering tools, in the context of embedded systems development, to track the quality attributes of the *SUCHAI FS* using a visual architecture evaluation tool. When this visual tool is integrated with the development process, developers can monitor the non-functional requirements' evolution and detect architecture disruptions that may deteriorate the software quality.

Visual displays allow the human brain to study multiple aspects of a complex problem simultaneously. It is well known that software visualization allows for a higher level of abstraction and closer mapping to the problem domain [71]. For this reason, several visualizations were produced to measure and assess the modularity of the components involved in the FS and to extract the architecture from the source code. Source code visualizations are generated using a script written in the Pharo programming language [72] and based on an agile visualization library called Roassal [73].

4.3.0.1. Evaluation of modularity using software visualization

Visualizing software dependencies is a common technique employed to communicate interaction between components [74]. In the global architecture, these interactions express the modularity of the FS solution. The visualization tool consists of a script that parses the source code files, classifying them as an application (including main, clients, invoker, receiver, repositories, and commands files), operating system, or drivers, modules to associate them a color. Then, it constructs a directed graph based on its dependencies. Dependencies are extracted from the `#include` directives contained in the source code. Edges between modules indicate a dependency as described in the UML model diagram shown in Fig. 3.11. Two snapshots of the SUCHAI FS source code are represented in Fig. 4.25. The first snapshot was produced in December 2017 (commit `765c128` on GitHub) while the second in May 2018 (commit `0ca21db` on GitHub).

The visualization shows modules (files having the extension `.c` with the corresponding `.h`) and their dependencies. Each file is represented as a colored box. The box's height indicates the number of code lines in the module, while the box's width represents the number of dependencies included in the represented module. For example, the central green box in Fig. 4.25 represents the module named `repoCommand.c`, which represents the command repository whose purpose is to store and give access to available commands. This module is one of the largest in the SUCHAI FS since it is the highest box. Similarly, the blue box on the top represents the `main.c` file. The `main` is the widest module. It includes a large number of dependencies, which makes sense because it is the software entry point.

In the commit `765c128` from December 2017 the FS contains only the fundamentals modules to support command executions. The diagram in Fig. 4.25 left shows that client tasks

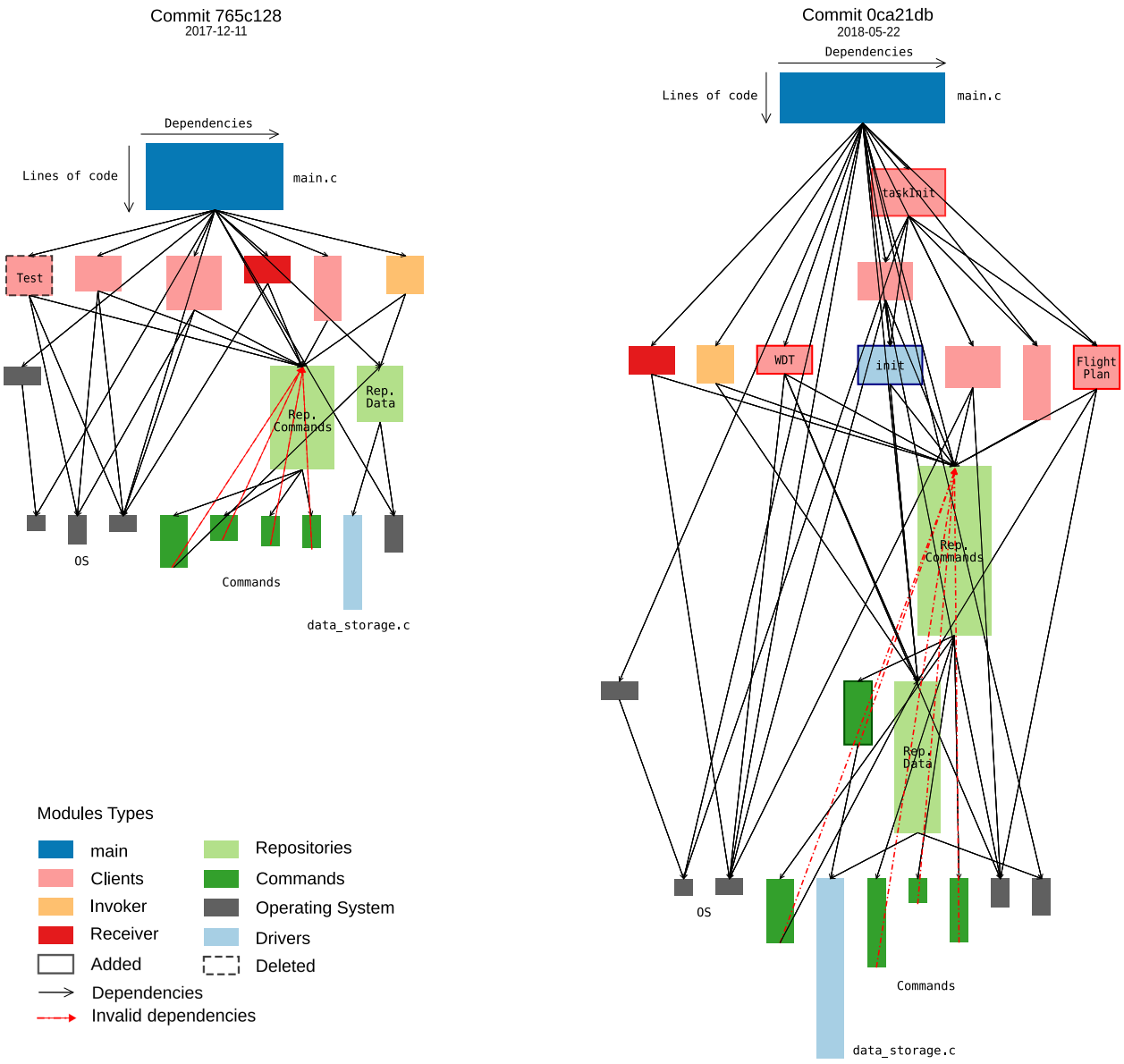


Figure 4.25: Modules dependencies comparison between commits 765c128 and 0ca21db.

depend on command and data repositories, command repository includes all command modules, and data repository includes drivers for data storage handling. This dependencies graph matches the proposed architecture described in Fig. 3.11, except for a circular dependency between the command repository and the command modules. This circular dependency is not described nor desired in the architecture shown in Fig. 3.11. However, an inspection of the source code reveals that using a command repository function inside command modules to register new commands in the system increases the code's readability and maintainability.

The diagram of the commit 0ca21db from May 2018 in Fig. 4.25 right shows the evolution of the software after several commits. A similar analysis makes it possible to determine that the architecture is preserved and that no extra dependencies were added. However, the visualization reveals that new clients were added, one was deleted, new commands were added,

while some modules changed the amount of code. The main module has reduced code lines since some initialization routines were moved to the new `taskInit` client. On the one hand, new commands and clients were added, which means that the software was augmented with new functionalities. However, on the other hand, the `Invoker` and the `Receiver` remain intact, which means that the commands execution logic was not intervened. A significant number of lines of code have been added to the data and command repository; hence, developers should concentrate on testing these modules.

The same visualization tool is used to validate the architectural rules of the application layer. The application layer architecture is based on the command pattern as the UML diagram in Fig. 3.12 details. The application layer modules are related by a messaging system to send commands from Task modules to the Invoker and the Receiver. The messaging system was implemented as queues in FreeRTOS and Linux, so the idea is to visualize Task modules and queues' relation. Figure 4.26 shows the structure and evolution of the application layer after several months of development. Only Task modules of type Client use the queue to send commands for execution. Although some Client modules were added and removed from commit `765c128` to `0ca21db`, the architecture of the application layer remains intact. The evolution shown in Fig. 4.25 contrasts with the results in Fig. 4.26 because with time, new features, lines of code, and modules were added, but the execution logic in the upper layer has remained intact.

These visualizations can be immediately exploitable by a software engineer. They are meant to be an early indicator of (i) a violation of the architecture and (ii) an anomaly due to exceptional entities.

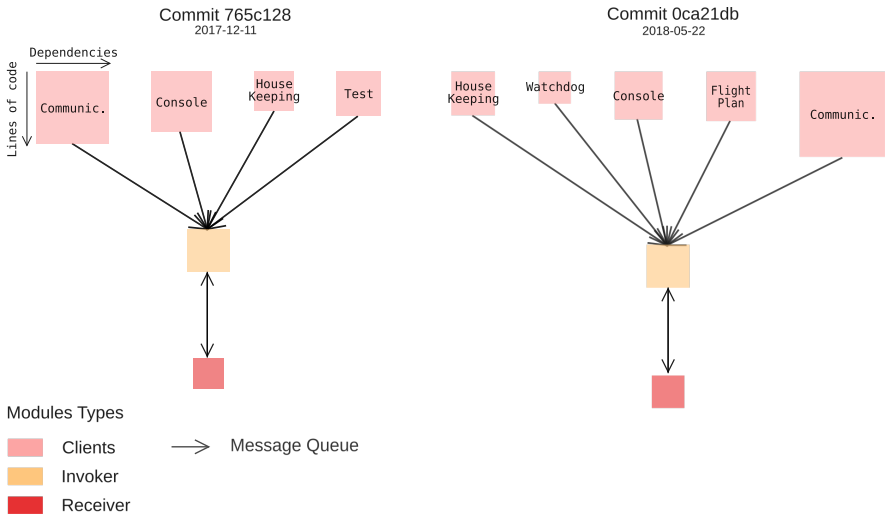


Figure 4.26: Application layer architecture visualization. Relation between Task modules, Invoker, Receiver and messages queues for commits `765c128` and `0ca21db`

Due to the architecture presented in Section 3.2 was inspired by a well-known design pattern, the implementation is simple and clear. However, to corroborate that the architecture and the quality requirements are effectively accomplished, visual support of the source code was developed to detect architecture disruptions in perfectly working code. This situation can

represent potential errors or deterioration in software quality. Thus, dependencies, message paths, and differences between commits were visualized to quickly determine which components are affected by a source code change and keep track of the software's architectural attributes. Moreover, as these tools can be automated and integrated into the development cycle, the presented flight software and the verification techniques exhibit good scalability to support large constellations' development.

4.4. Constellation simulator results

The evolutionary flight plan design tool and a flight software solution are validated; thus, the last step was to integrate these results in software in the loop simulation tool. The constellation simulator will take scenario definition information, the flight plan designed to solve a specific task, and will deploy N instances of the SUCHAI FS interconnected via a local loop and synchronized using shared memory space. The simulation will run up to 100 times faster than in real-time, depending on the level of parallel cores available. The goal is to validate the correctness of the generated flight plan and obtain execution time data to analyze the whole constellation resources usage prior to deploying the task.

The SUCHAI FS uses the Cubesat Space Protocol (CSP) library as a communication stack. This library presents many advantages, such as Linux and FreeRTOS support, is lightweight enough to be used in microcontrollers, and has been tested in several space missions. However, to date, version 1.0 of this communication protocol supports up to 32 network addresses[11]. This limitation impedes test scenarios with 100 and 1000 nodes. For this reason, only scenarios A and B with 10 satellites each are simulated.

4.4.1. Set up

Let us assume a complete new install of the constellation control framework to illustrate its usage. First, clone and initialize the repository with the following commands to download and build the python scripts, the simulator application, and the SUCHAI Flight Software.

```
1  git clone https://gitlab.com/carlgonz/constellation-framework.git
2  cd constellation-framework
3  sh init.sh
```

Then, place the scenario and task definitions in a separated folder. Following the default repository setup the following files are placed in the `cases/scenario_a_b` directory: `scenario_walker_10_5.json`, `scenario_adhoc_10.json`, `task1.json` and `task2.json`.

Scenarios can be created using the `scenarios.py` script to match the definitions of table 4.16. In this case, a contact list of 300 seconds was used because minimum differences were observed with lower resolutions in previous tests (see Section 4.2.1 and 4.2.2). Use the following commands to generate the scenario definition files:

```
1  python3 scenario.py walker 10 -p 5 -o scenario_walker_10_5.json -s 1601424000 -d 16200
2  python3 scenario.py adhoc 10 -o scenario_adhoc_10.json -s 1601424000 -d 16200
```

[11] To date CSP v2.0 is under development increasing the address field to 14 bits or 16384 addresses

Listing 4.3: Task 1 and 2 definition file

```
# task1.json
{
   "id": 1,
   "start": "stgo",
   "end": "tokyo",
   "targets": [
      {"id": "saa", "command": "sim_get_data 1", "result": "1", "prio":  1}
   ],
   "solution": null
}

# task2.json
{
   "id": 2,
   "start": "tokyo",
   "end": "tokyo",
   "targets": [
      {"id": "stgo", "command": "sim_get_data 1", "result": "1"},
      {"id": "saa", "command": "sim_get_data 2", "result": "2"}
   ],
   "solution": null
}
```

Table 4.16: Scenarios description

Scenario A and B parameters
Start time: 2020-09-30T00:00:00 UTC (1601424000 Unix time)
Simulation time: 16200 seconds (∼3 orbits)
Simulation resolution: 30 seconds
Contact list resolution: 3000 seconds
Number of contacts: 493 and 383

4.4.2. Execution

The framework entry point is the `controller.py` script. This software executes the following steps:

1. Load scenario and task file definitions
2. Generate the contact list data file, or load a pre-calculated file.
3. Generate the contact plan and flight plan using the evolutionary algorithm, or load a pre-calculated solution.
4. Run a simulation by launching N instances of the SUCHAI FS and simulation app.
5. Collect each node telemetry and produced summary files.

Use the following commands to run each scenario and task:

```
python3 controller.py cases/scenario_a_b scenario_walker_10_5.json task1.json -s 200 -m 0.6 -i 100 -n 2 --dt 300
python3 controller.py cases/scenario_a_b scenario_walker_10_5.json task2.json -s 200 -m 0.6 -i 100 -n 2 --dt 300
python3 controller.py cases/scenario_a_b scenario_adhoc_10.json task1.json -s 200 -m 0.6 -i 100 -n 2 --dt 300
python3 controller.py cases/scenario_a_b scenario_adhoc_10.json task2.json -s 200 -m 0.6 -i 100 -n 2 --dt 300
```

Figure 4.27 show the contact plans generated for each scenario and task executed.

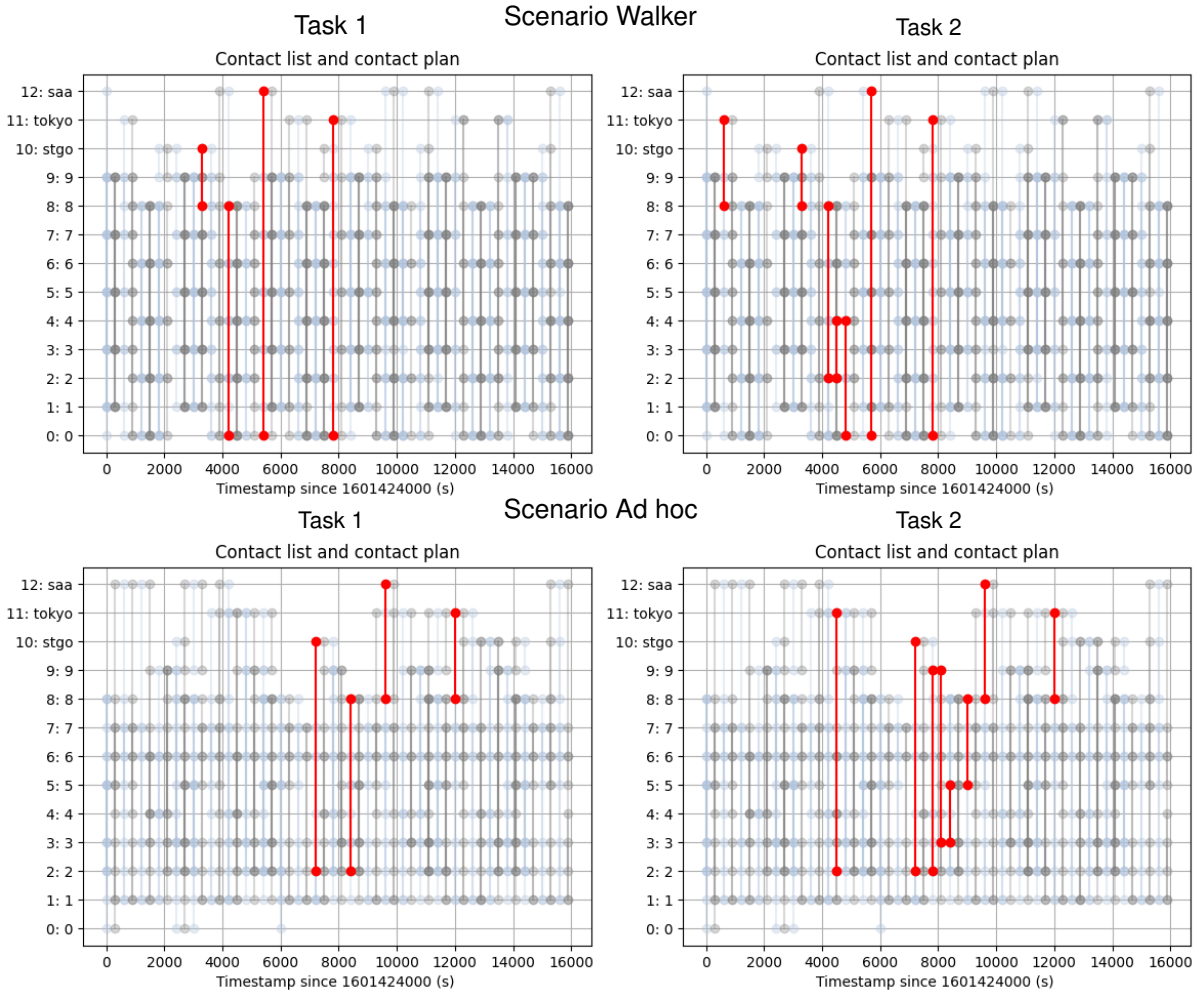


Figure 4.27: Contact plan for each scenario and task in the simulation

Finally, the controller will run the simulation by launching one instance of the SUCHAI FS simulator application for each node (satellites and ground stations). Using the parameters `–tty` it is possible to display the instances outputs in a set of previously opened terminals; else, the software log output is redirected to a file. Figure 4.28 is an example output of 8 SUCHAI FS instances running during a simulation.

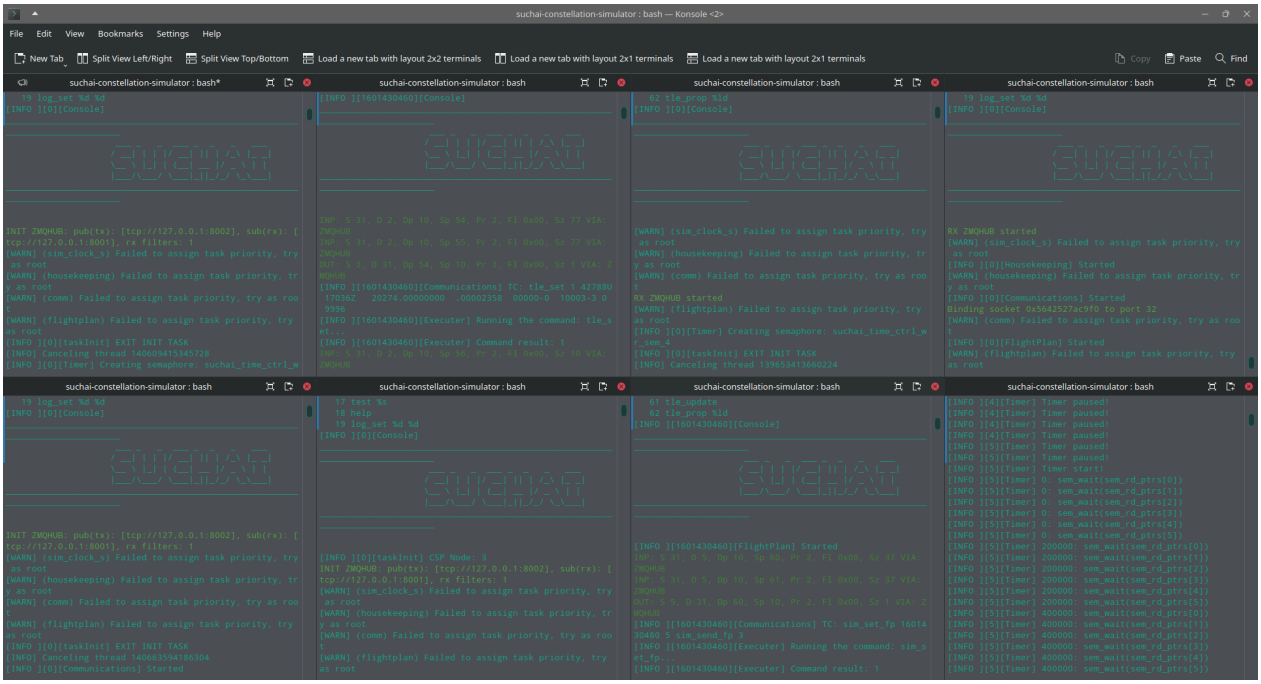


Figure 4.28: Contact plan for each scenario and task in the simulation

4.4.2.1. Results

Because the idea is to simulate the normal functioning of satellites in the constellation, each instance executes some commands by default. These operations include running the SGP4 propagator, housekeeping tasks such as updating software watchdogs and status variables, collecting status data telemetry, and collecting all time-tagged commands executed by the flight plan task. These two telemetries enable a posterior analysis of the constellation operation. With the commands telemetry, it is possible to validate that the proposed global flight plan was executed without errors. While with the status variables telemetry, it is possible to obtain resources usage metrics.

The command executing rate will be used as a constellation resources usage metric in this work. As discussed in Section 3.2 and 4.3 the SUCHAI Flight Software was designed to execute commands mainly, so measuring variations in the commands queue/execution rate is a relevant metric.

Figures 4.29 and 4.30 show the commands execution rate results for the Walker and Ad hoc scenarios respectively. They showed that node satellites have a base rate of 0.3 commands per second in both scenarios because the housekeeping task executes commands periodically (update date and time, update status, propagate orbit, among others). Ground station nodes do not propagate their orbits, so they execute fewer commands at a rate of 0.2 commands/-seconds. The plots show usage peaks coincident with the flight plan timing because those are new commands to execute. Despite the initial set-ups, the maximum usage is 0.5 and 0.6 for task 1, less than one command per second. These results confirm that task 1 is a lightweight and simple task, and the constellation has more resources available. For task 2, the maximum usages are 1.1 and 1.3 because each target visited also implies moving data between satellites resulting in more commands executed during a contact.

Despite the results revealing that both tasks are lightweight, they are relevant in showing the complete system working; thus, closing gaps between task scheduling, contact plan design, flight software design, and the constellation operation. Furthermore, it is possible to include more functioning details in the simulator, for example: simulate commands' actual duration, communication delays, battery usage, attitude control, among others. Models for these parameters can also be included in the contact plan design algorithm and contrasted with simulation results.

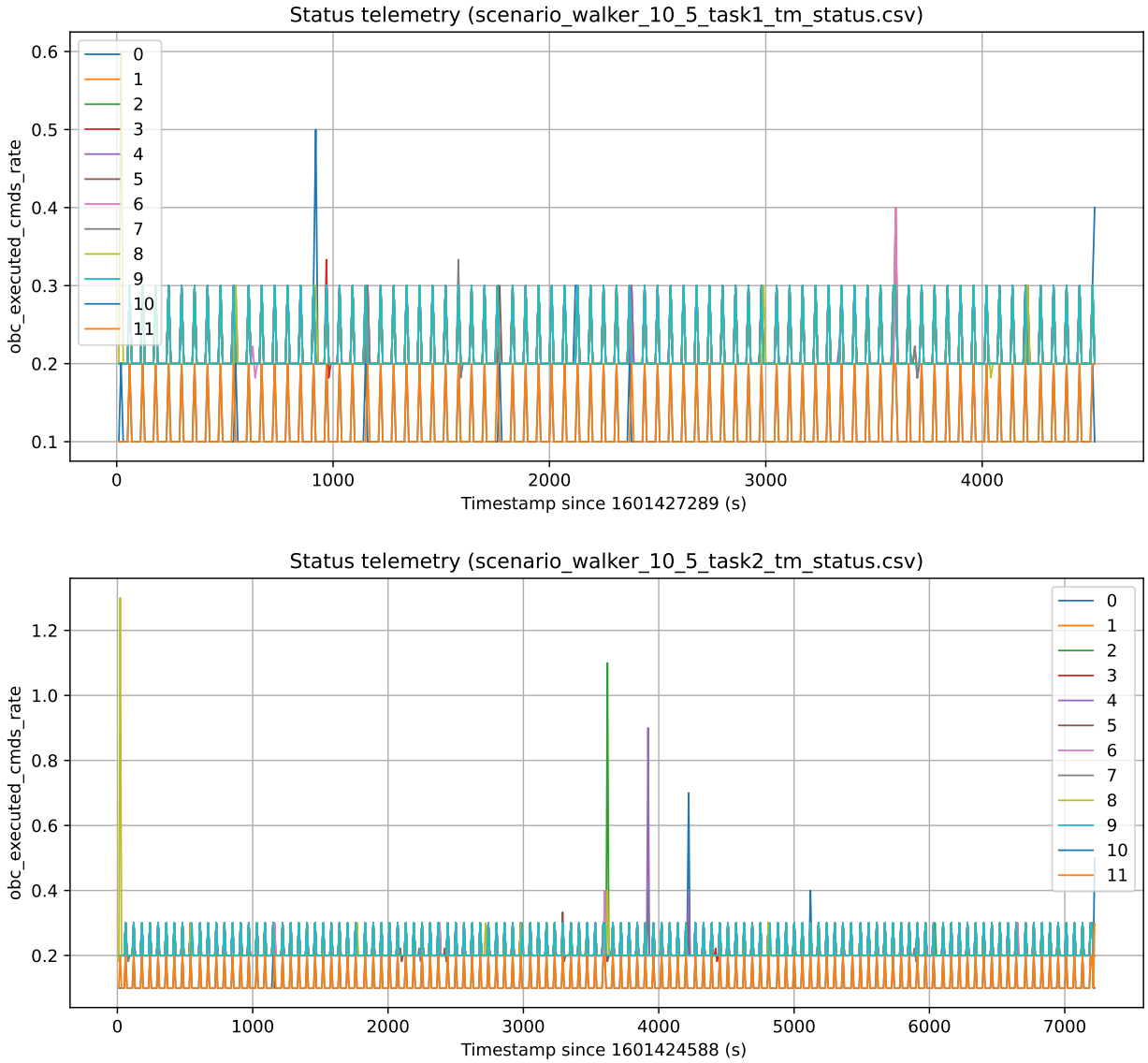


Figure 4.29: Scenario Walker command execution rate (commands/seconds).
Top: task 1 results. Bottom: task 2 results.

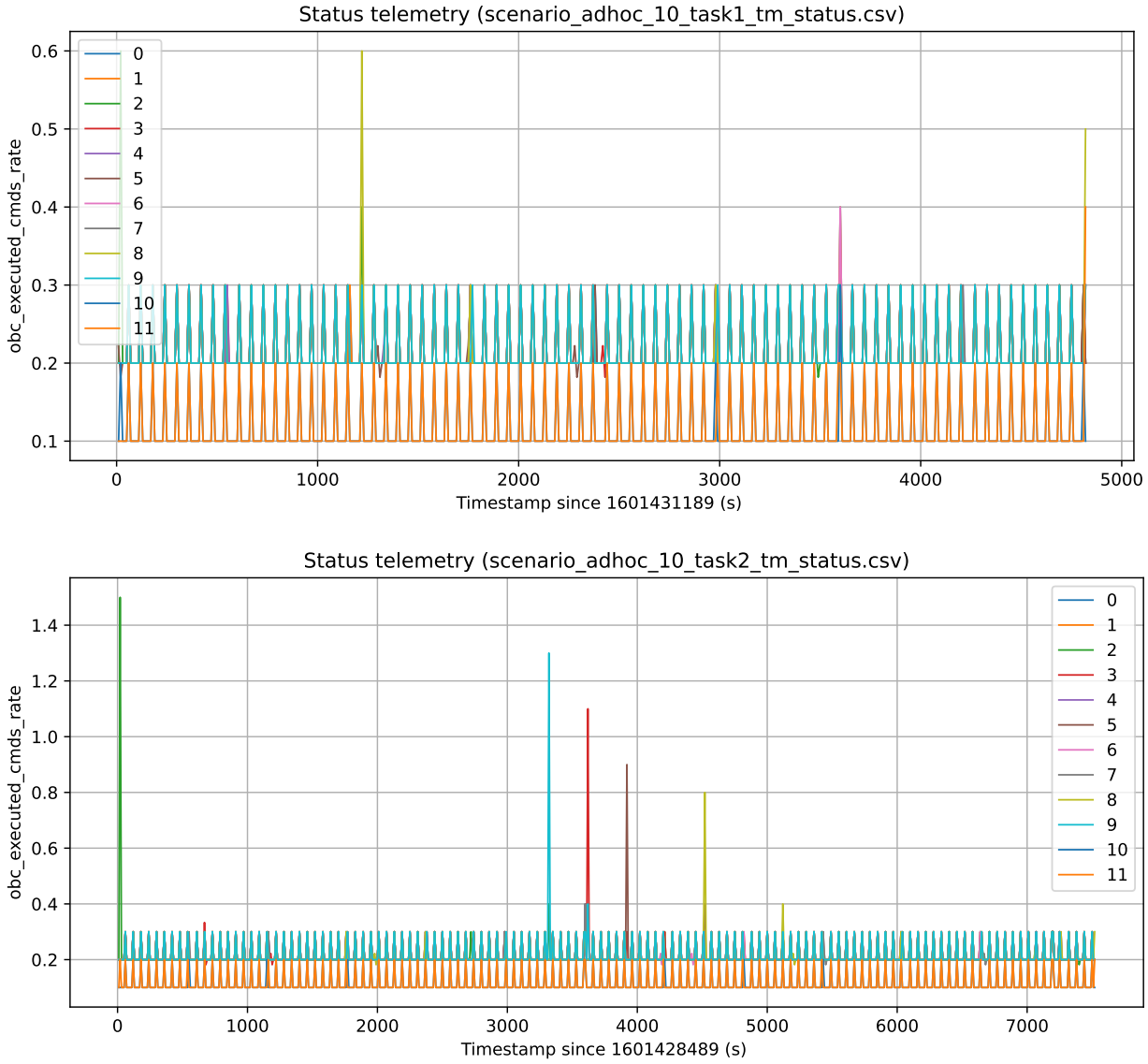


Figure 4.30: Scenario Ad hoc command execution rate (commands/seconds). Top: task 1 results. Bottom: task 2 results.

4.5. Chapter highlights

This chapter shows the results of the genetic algorithm for contact list design. The study focuses on the genetic algorithm scalability to a large number of nodes and the analysis of the flight software quality. The main conclusions are:

- The genetic algorithm converges to generate valid contact plans for several tasks and different scenarios. In particular, constellations with Walker and Ad hoc configuration and up to 1000 nodes were solved.
- The genetic algorithm performance versus the population size and mutation rate hyper-parameters was studied. It was possible to find a combination of parameters to obtain optimal or near to optimal solutions in all cases. Furthermore, optimal solutions were

verified by inspection in the smallest cases.

- A graphical tool was developed to extract the software architecture of the implemented code. This visualization of the architecture helps software engineers verify the code's quality attributes.
- Using visualizations, it is possible to detect architecture disruptions that may negatively impact the software quality, thus increasing mission risks.
- Scalability measurements show that the serial portion of the work limits the speedup of the contact list generation tool. However, optimizations and parallelization were critical to calculating contacts in scenarios up to 1000 nodes in reasonable times.
- Scalability measurements show that the time required to find a contact plan scales proportionally to the number of contacts in the scenario. The number of contacts scales exponentially with the number of satellites.

Chapter 5

Conclusions and future work

5.1. Conclusions

In this thesis, the problem of operating a large nanosatellite constellation was studied. To date, small- and nanosatellite constellations consisting of hundreds to thousands of nodes are being deployed. The New Space context adds new restrictions and requirements to this problem stressing the production and operation lines. Agility, flexibility, and autonomy are key concepts to succeed in the New Space era.

Literature shows that many efforts have been deployed to solve task scheduling problems in satellite constellations, deploy communication networks that support dynamic topologies, delays, and disruptions; and physically connect small- and nanosatellites using radio and optical links. CubeSat nanosatellites are part of this trend and present particular restrictions regarding space, power, computing resources, and constellation deployment logistics. In particular, it is not possible to assume that CubeSat constellations will be deployed in particular geometries, such as Walker Star or Walker Delta, but in Ad Hoc configuration due to several secondary payload launches. Also, the scheduling problem's complexity suggests that onboard planning and scheduling can be computationally demanding. This approach may not scale properly to a large number of nodes in the constellation. For these reasons, using meta-heuristic algorithms and ground-based planning is a valid approach to solve this problem.

The results of this work may also be applied to other contexts, including larger satellites (small o microsatellites). The original inspiration of this work is the increasing number of CubeSat satellites being deployed and the possibility to operate automatically as a coordinated constellation with this kind of spacecraft. The computational power and energy restrictions motivate offline planning and evolutionary algorithms. Commonly, larger satellites have more computational resources, larger solar panels, and batteries to afford onboard planning algorithms, which adds more autonomy to the system. On the other hand, this work may also be applied to satellites without ISL but with an extensive network of ground stations. The restrictions are similar, but adding connections between ground stations in the contact list will be necessary. The current implementation allows using ground stations as a bridge storing the flight plan until another satellite approaches if necessary.

In this work, satellites with Inter-satellite link (ISL) were considered. Thus, the LEO

constellation can be considered a Delay or Disruption Tolerant Network (DTN) and Contact Plan Design (CPD) technics were used to solve the scheduling problem. In particular, an evolutionary algorithm was designed and implemented to optimize the CPD under the study case's particular restrictions. The resultant Contact Plan is then used to generate a global Flight Plan table that describes the operations of the satellites to execute a task cooperatively. Therefore, the spacecraft complexity from the flight software perspective is bound to execute a set of time-tagged commands.

Case studies were used to validate the hypotheses. Simulated scenarios with 10, 100, and 1000 satellites in Walker and Ad Hoc configurations were used. Two tasks per scenario were used to analyze the performance of the GA. Results show that the algorithm con generates valid contact plan and flight plan tables in all cases. It was shown that it is possible to control a CubeSat constellation using the flight plan table derived from the contact plan design. It was also shown that the time required to solve the scheduled problem is bounded and remains feasible even for a large number of nodes. It was also shown that the command-based flight software architecture being used in the SUCHAI nanosatellites is suitable for this problem. This work also explains how to implement a Flight Software and software in the loop simulation tool to validate the scheduling solutions generated by the genetic contact plan design algorithm. Thus, this work expects to close an existing gap between the nanosatellite constellation software development and the operation of this complex system under heavy restrictions of computation resources.

Part of the results of this work have been published in the *IEEE Access* scientific journal under the title *"An architecture-tracking approach to evaluate a modular and extensible flight software for CubeSat nanosatellites"* [63] and in the 2021 Space-Terrestrial Internetworking Workshop (STINT) proceedings with the title *"Nanosatellite constellation control framework using evolutionary contact plan designs"*. Also, the derived work *"Systematic Fuzz Testing Techniques on a Nanosatellite Flight Software for Agile Mission Development"* [75] was recently published in the *IEEE Access* journal. The nanosatellite flight software and the nanosatellite constellation control framework are released as FLOSS projects in Git repositories[12][13][14]. The author aims to contribute to the CubeSat community with the tools developed during this work.

5.2. Challenges

This section aims to discuss some of the challenges addressed during the implementation of the thesis.

Scenarios and contact lists This work required generating the contact list of constellations up to 1000 nodes. Calculating the contact list requires defining each satellite orbital parameters and propagating orbits. Then, the contacts matrix is calculated: the instants where any pair of satellites are in the line of sight according to the communication system range.

[12] https://gitlab.com/spel-uchile/suchai-flight-software
[13] https://gitlab.com/carlgonz/constellation-framework
[14] https://gitlab.com/carlgonz/suchai-constellation-simulator

This task is usually done with third-party proprietary software such as STK. In this work, a free and open-source implementation of this problem was developed in Python. However, commercial solutions offer more advanced tools to calculate contacts with different radio link configurations, for example, directional antennas or several antennas. Still, this work only implements the omnidirectional antenna case. As the contact condition is evaluated in a Python function, it is possible to improve the radio link contacts model in the future. Several optimizations were implemented to calculate contact lists of 100 and 1000 nodes. Parallel processing was required to accelerate results. RAM usage optimization and intermediate files were necessary to limit memory usage.

Genetic algorithm encoding. The implementation of the genetic algorithm required several trials. The main challenge was to choose the correct encoding and the algorithm itself. The first approach was to replicate the algorithm presented in Fraire *et al.* (2018) [38]. However, binary encoding and permutation encoding did not work properly due to the validity restrictions. The contacts rules proposed in this work include the task targets as nodes. These target nodes are not data relays; instead, they are checkpoints, and satellites must cover these points to execute the task. These target nodes are also mandatory, so only contact plans that satisfy all validity rules can be considered. Thus, the validity of the sequences is also an optimization variable. The result was a trade-off between validity, delivery time, and the number of contacts variables. Even with multi-objective genetic algorithms such as NSGA-II, the encoding tends to create low variability in the population and thus converge to local optimal. The solution was to define a particular encoding to consider the validity restrictions during the initial population creation. Particular genetic operators were also created to maintain the validity of the individuals.

Contact list visualization Visualizing a contact list and contact plan is a powerful tool to analyze and communicate results quickly, but there were no available tools to visualize contact plans. Thus, Python scripts were developed to create these visualizations. The contact list FSM representation was useful to generate visualization, but it was still difficult to communicate the results on some occasions. Extra annotations were required, such as the direction of the data flow in the graph and connections between states. Also, visualizing large scenarios is challenging, and the proposed visualization was not practical for the 100 nodes case study. It is worth exploring better contact plan visualization tools.

Flight software This work aimed to create a solution feasible and usable in existing CubeSat projects. Thus, it is evident that a straightforward interface with the physical system was necessary together with the operating framework. This interface is the flight software of the nanosatellite. After studying the problem, it was decided to develop a flight software solution for the SPEL CubeSats. The proof of concept was the launch and operation of the SUCHAI 1 in June 2017 (during the second year of the Ph. D. program). The operation of the satellite was a success, in part due to the flight software design. Then, the construction of the SUCHAI 2, 3, and PlantSat nanosatellites started. From the experience of the first CubeSat, several improvements were necessary. First, the command architecture worked well, but more flexibility on the command's parameters was necessary. Second, extending the satellite as a computer network required standardizing the payloads and ground station interface. Thus, the OBC, payloads, and the ground station will use the SUCHAI Flight Software. Finally, this approach required rewriting the SUCHAI flight software to be portable

to different software architectures, including X86 computers, the Raspberry Pi embedded computer, and the NanoMind A3200 (AVR32UC3) microcontroller. To date, at least three SUCHAI Flight Software instances run in each of the three CubeSats, plus the ground station and other operators' terminals. Maintaining the development of more than 10 flight software instances with a team of no more than three to four part-time graduate and undergrad students is challenging. The team worked hard to automate testing using CI/CD tools, but new techniques were also required. Thus, the visual software architecture tracking and the fuzz testing tools were developed as part of this thesis and published in ISI journals.

5.3. Future work

The results of this thesis open several opportunities to explore. Promising results were obtained in the scalability of large constellations in relatively simple tasks. Therefore it would be interesting to explore more complex tasks and workflows, model those problems, and use the framework to validate the results. This work explores the execution of multiple independent tasks, which is useful for sampling multiple points on earth or monitoring a single point for a period. Future work should explore the execution of distributed programs, including loops, conditionals, and procedures. It will require creating a domain-specific language to define tasks so operators can create scripts in a high-level language that is aware of the actual constellation capabilities. This kind of distributed programs usually require sharing a global state, so it is necessary to explore more advanced network capabilities such as DTN routing protocols.

The GA, and evolutionary approaches in general, should be flexible enough to integrate more restrictions and variations to the original problem presented here. However, it will be necessary to model each problem, define the restrictions and measure the performance. Future works must keep the focus on scalability to large constellations and evaluate at least up to 1000 nodes. Also, it is important to consider the computational limitations of nanosatellites and ensure that the proposed models fit with the operational capabilities of the flight software. It could be necessary to extend the proposed flight software to support scripts, states, and routing protocols.

The analysis and visualization of the CPD results with a large number of nodes were challenging. The density of nodes and paths resulted in illegible contact list diagrams for scenarios with more than 100 nodes. Exact solutions can be challenging to obtain in large constellation scenarios, so expert analysis of the proposed contact plan will be valuable and faster. However, more interactive visualization tools with focus, filters, and adaptive capabilities are required to improve the analysis. Contributions in this direction might be valuable for researchers working in DTN.

A real-life demonstration is important to validate these ideas. The whole idea of this work was to close the gap between computational complex scheduling algorithms and the operation of a constellation of heavy limited CubeSat nanosatellites. The Space and Planetary Exploration Laboratory (SPEL) of the University of Chile is currently developing three CubeSats nanosatellites, the SUCHAI 2, 3, and PlantSat. Immediate work is to prepare the satellites to work with the proposed constellation operation framework. Part of this work is already developed. These satellites already execute the proposed flight software and use the

visualization tool to verify software quality daily. The ground station node also implements the software, and the whole system can operate as a computer network. However, additional work is required. The satellites must be physically connected using the UHF radio links. It is necessary to define operation scenarios based on the available payloads and instruments and run the case studies with the actual hardware. These tests will give more insights into the physical restrictions to include them in the models. With the satellites in orbit, it will be possible to conduct real-life experiments and collect operational data to complement these studies with realistic scenarios.

Finally, considering the ISL capabilities of the SPEL CubeSats, it is worth exploring the applicability of DTN routing protocols to the existing network stack based on the CubeSat Space Protocol (CSP). This exercise will open a full research line that could start validating the feasibility of existing DTN protocols found in the literature to finally propose improvements based on the restrictions of these spacecrafts. It will also allow to compare DTN routing protocols with the proposed constellation operation framework and define how to integrate both tools to solve more complex tasks.

List of acronyms

ALU	Arithmetic Logic Unit.
API	Application Programming Interface.
CL	Contact List.
CP	Contact Plan.
CPD	Contact Plan Design.
CPU	Central Processing Unit.
CSP	Cubesat Space Protocol.
DTN	Delay or Disruption Tolerant Network.
EEPROM	Electrically Erasable Programmable Read-Only Memory.
EPS	Energy Power System.
EULA	End-User License Agreement.
FLOSS	Free/Libre and Open Source Software.
FOSS	Free and Open Source Software.
FP	Flight Plan.
FS	Flight Software.
FSM	Finite State Machine.
GA	Genetic Algorithm.
GPL	General Public License.
HAL	Hardware Abstraction Layer.
I2C	Inter Integrated Circuit.
IARU	International Amateur Radio Union.
IDE	Integrated Development Environment.
ISL	Inter-satellite link.
MILP	Mixed Integer Linear Programming.
MVC	Model-View-Controller.
OBC	On-Board Computer.
RAM	Random Access Memory.
RISC	Reduced Instruction Set Computing.
RSSI	Received Signal Strength Indicator.
RTCC	Real Time Clock.
RTOS	Real Time Operating System.
SAA	South Atlantic Anomaly.
SPI	Serial Peripheral Interface.
SUCHAI	Satellite of the University of Chile for Aerospace Investigation.

TLE	Two-line Elements Set.
TNC	Terminal Node Controller.
UART	Universal Asynchronous Receiver-Transmitter.
UHF	Ultra High Frequency.
USB	Universal Serial Bus.
WDT	Watchdog timer.

Bibliography

[1] S. Lee, A. Hutputanasin, A. Toorian, W. Lan, R. Munakata, J. Carnahan, D. Pignatelli, and A. Mehrparvar, "Cubesat design specification rev. 13," Tech. Rep. 2, The CubeSat Program, Cal Poly San Luis Obispo, US, 2014.

[2] D. Paikowsky, "What is new space? the changing ecosystem of global space activity," *New Space*, vol. 5, no. 2, pp. 84–88, 2017.

[3] R. Radhakrishnan, W. W. Edmonson, F. Afghah, R. M. Rodriguez-Osorio, F. Pinto, and S. C. Burleigh, "Survey of inter-satellite communication for small satellite systems: Physical layer to network layer view," *IEEE Communications Surveys Tutorials*, vol. 18, pp. 2442–2473, Fourthquarter 2016.

[4] Y. Su, Y. Liu, Y. Zhou, J. Yuan, H. Cao, and J. Shi, "Broadband leo satellite communications: Architectures and key technologies," *IEEE Wireless Communications*, vol. 26, no. 2, pp. 55–61, 2019.

[5] S. Bandyopadhyay, R. Foust, G. P. Subramanian, S.-J. Chung, and F. Y. Hadaegh, "Review of formation flying and constellation missions using nanosatellites," *Journal of Spacecraft and Rockets*, no. 0, pp. 567–578, 2016.

[6] A. Marinan, A. Nicholas, and K. Cahoy, "Ad hoc cubesat constellations: Secondary launch coverage and distribution," in *2013 IEEE Aerospace Conference*, pp. 1–15, March 2013.

[7] C. Boshuizen, J. Mason, P. Klupar, and S. Spanhake, "Results from the Planet Labs Flock Constellation," in *AIAA/USU Conference on Small Satellites*, aug 2014.

[8] I. F. Akyildiz and A. Kak, "The internet of space things/cubesats," *IEEE Network*, vol. 33, no. 5, pp. 212–218, 2019.

[9] I. del Portillo, B. G. Cameron, and E. F. Crawley, "A technical comparison of three low earth orbit satellite constellation systems to provide global broadband," *Acta Astronautica*, vol. 159, pp. 123–135, 2019.

[10] J. Alvarez and B. Walls, "Constellations, clusters, and communication technology: Expanding small satellite access to space," in *2016 IEEE Aerospace Conference*, pp. 1–11, March 2016.

[11] S. Nag, A. S. Li, and J. H. Merrick, "Scheduling algorithms for rapid imaging using agile cubesat constellations," *Advances in Space Research*, vol. 61, no. 3, pp. 891 – 913, 2018.

[12] Z. Zheng, J. Guo, and E. Gill, ""swarm satellite mission scheduling & planning using hybrid dynamic mutation genetic algorithm"," *Acta Astronautica*, vol. 137, pp. 243 – 253, 2017.

[13] Z. Zheng, J. Guo, and E. Gill, "Onboard autonomous mission re-planning for multi-satellite system," *Acta Astronautica*, vol. 145, pp. 28 – 43, 2018.

[14] M. Lemaitre, G. Verfaillie, F. Jouhaud, J.-M. Lachiver, and N. Bataille, "Selecting and scheduling observations of agile satellites," *Aerospace Science and Technology*, vol. 6, no. 5, pp. 367 – 381, 2002.

[15] J. A. Fraire and J. M. Finochietto, "Design challenges in contact plans for disruption-tolerant satellite networks," *IEEE Communications Magazine*, vol. 53, pp. 163–169, May 2015.

[16] J. A. Fraire, P. G. Madoery, and J. M. Finochietto, "Traffic-aware contact plan design for disruption-tolerant space sensor networks," *Ad Hoc Networks*, vol. 47, pp. 41–52, 9 2016.

[17] J. A. Fraire, G. Nies, C. Gerstacker, H. Hermanns, K. Bay, and M. Bisgaard, "Battery-aware contact plan design for leo satellite constellations:the ulloriaq case study," *IEEE Transactions on Green Communications and Networking*, pp. 1–1, 2019.

[18] A. K. Kennedy and K. L. Cahoy, "Performance analysis of algorithms for coordination of earth observation by cubesat constellations," *Journal of Aerospace Information Systems*, vol. 14, no. 8, pp. 451–471, 2017.

[19] C. Araguz, M. Marí, E. Bou-Balust, E. Alarcon, and D. Selva, "Design Guidelines for General-Purpose Payload-Oriented Nanosatellite Software Architectures," *Journal of Aerospace Information Systems*, vol. 15, pp. 107–119, mar 2018.

[20] M. Tipaldi, C. Legendre, O. Koopmann, M. Ferraguto, R. Wenker, and G. D'Angelo, "Development strategies for the satellite flight software on-board Meteosat Third Generation," *Acta Astronautica*, vol. 145, pp. 482–491, apr 2018.

[21] D. J. A. F. Miranda, M. A.-c. Ferreira, F. Kucinskis, and D. McComas, "A Comparative Survey on Flight Software Frameworks for TNew Space Nanosatellite Missions," *Journal of Aerospace Technology and Management*, vol. 11, 00 2019.

[22] T. Ferrer, S. Céspedes, and A. Becerra, "Review and evaluation of mac protocols for satellite iot systems using nanosatellites," *Sensors*, vol. 19, no. 8, p. 1947, 2019.

[23] J. Le Moigne, J. C. Adams, and S. Nag, "A new taxonomy for distributed spacecraft missions," *IEEE Journal of Selected Topics in Applied Earth Observations and Remote Sensing*, vol. 13, pp. 872–883, 2020.

[24] J. N. Pelton and R. Laufer, *Commercial Small Satellites for Business Constellations Including Microsatellites and Minisatellites*, pp. 1–20. Cham: Springer International Publishing, 2019.

[25] D. Doan, R. Zimmerman, L. Leung, J. Mason, N. Parsons, and K. Shahid, "Commissioning the world's largest satellite constellation," 8 2017.

[26] S. Bandyopadhyay, R. Foust, G. P. Subramanian, S.-J. Chung, and F. Y. Hadaegh, "Review of formation flying and constellation missions using nanosatellites," *Journal of Spacecraft and Rockets*, vol. 53, no. 3, pp. 567–578, 2016.

[27] Weilian Su, Jianwen Lin, and T. Ha, "Global communication coverage using cubesats," in *2017 IEEE 7th Annual Computing and Communication Workshop and Conference*

(CCWC), pp. 1–7, Jan 2017.

[28] A. Kak and I. F. Akyildiz, "Large-scale constellation design for the internet of space things/cubesats," in *2019 IEEE Globecom Workshops (GC Wkshps)*, pp. 1–6, 2019.

[29] D.-H. Cho, J.-H. Kim, H.-L. Choi, and J. Ahn, "Optimization-based scheduling method for agile earth-observing satellite constellation," *Journal of Aerospace Information Systems*, vol. 15, no. 11, pp. 611–626, 2018.

[30] X. Chu, Y. Chen, and Y. Tan, "An anytime branch and bound algorithm for agile earth observation satellite onboard scheduling," *Advances in Space Research*, vol. 60, no. 9, pp. 2077 – 2090, 2017.

[31] Y. Li, M. Xu, and R. Wang, "Scheduling observations of agile satellites with combined genetic algorithm," in *Third International Conference on Natural Computation (ICNC 2007)*, vol. 3, pp. 29–33, 2007.

[32] Z. Yuan, Y. Chen, and R. He, "Agile earth observing satellites mission planning using genetic algorithm based on high quality initial solutions," in *2014 IEEE Congress on Evolutionary Computation (CEC)*, pp. 603–609, 2014.

[33] S. Spangelo, J. Cutler, K. Gilson, and A. Cohn, "Optimization-based scheduling for the single-satellite, multi-ground station communication problem," *Computers & Operations Research*, vol. 57, pp. 1 – 16, 2015.

[34] X. Jia, T. Lv, F. He, and H. Huang, "Collaborative data downloading by using inter-satellite links in leo satellite networks," *IEEE Transactions on Wireless Communications*, vol. 16, no. 3, pp. 1523–1532, 2017.

[35] B. Deng, C. Jiang, L. Kuang, S. Guo, J. Lu, and S. Zhao, "Two-phase task scheduling in data relay satellite systems," *IEEE Transactions on Vehicular Technology*, vol. 67, no. 2, pp. 1782–1793, 2018.

[36] M. Bisgaard, D. Gerhardt, H. Hermanns, J. Krčál, G. Nies, and M. Stenger, "Battery-aware scheduling in low orbit: the gomx-3 case," *Formal Aspects of Computing*, vol. 31, no. 2, pp. 261–285, 2019.

[37] J. A. Fraire, C. Gerstacker, H. Hermanns, G. Nies, M. Bisgaard, and K. Bay, "On the scalability of battery-aware contact plan design for leo satellite constellations," *International Journal of Satellite Communications and Networking*, vol. 39, no. 2, pp. 193–204, 2021.

[38] J. A. Fraire, P. G. Madoery, J. M. Finochietto, and G. Leguizamón, "An evolutionary approach towards contact plan design for disruption-tolerant satellite networks," *Applied Soft Computing Journal*, vol. 52, pp. 446–456, 3 2017.

[39] D. L. Dvorak, "NASA Study on Flight Software Complexity," in *AIAA Infotech@Aerospace Conference and AIAA Unmanned...Unlimited Conference*, (Reston, Virigina), p. 264pp, American Institute of Aeronautics and Astronautics, apr 2009.

[40] J. Alonso, M. Grottke, A. P. Nikora, and K. S. Trivedi, "An empirical investigation of fault repairs and mitigations in space mission system software," in *Proceedings of the International Conference on Dependable Systems and Networks*, pp. 1–8, IEEE, jun 2013.

[41] D. McComas, J. Wilmot, and A. Cudmore, "The Core Flight System (cFS) Community: Providing Low Cost Solutions for Small Spacecraft," in *AIAA/USU Conference on Small Satellites*, aug 2016.

[42] R. Plauche, "Building modern cross-platform flight software for small satellites," in *AIAA/USU Conference on Small Satellites*, Aug 2017.

[43] M. Glinz, "On non-functional requirements," in *15th IEEE International Requirements Engineering Conference (RE 2007)*, pp. 21–26, Oct 2007.

[44] D. Ganesan, M. Lindvall, C. Ackermann, D. McComas, and M. Bartholomew, "Verifying architectural design rules of the flight software product line," in *Software Product Lines, 13th International Conference, SPLC 2009, San Francisco, California, USA, August 24-28, 2009, Proceedings*, pp. 161–170, Carnegie Mellon University, 2009.

[45] D. Ganesan, M. Lindvall, D. McComas, M. Bartholomew, S. Slegel, and B. Medina, "Architecture-based unit testing of the flight software product line," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 6287 LNCS, pp. 256–270, Springer, Berlin, Heidelberg, 2010.

[46] P. Fiala and A. Vobornik, "Embedded microcontroller system for PilsenCUBE picosatellite," in *2013 IEEE 16th International Symposium on Design and Diagnostics of Electronic Circuits & Systems (DDECS)*, pp. 131–134, IEEE, apr 2013.

[47] A. van den Berg, "Fault-tolerant on-board computer software for the del fi-n3xt nanosatellite," Master's thesis, Delft University of Technology, Delft, Netherlands, August 2012.

[48] S. Johl, E. Glenn Lightsey, S. M. Horton, and G. R. Anandayuvaraj, "A reusable command and data handling system for university cubesat missions," in *IEEE Aerospace Conference Proceedings*, pp. 1–13, IEEE, mar 2014.

[49] M. Schmidt and K. Schilling, "An extensible on-board data handling software platform for pico satellites," *Acta Astronautica*, vol. 63, pp. 1299–1304, dec 2008.

[50] S. F. Hishmeh, T. J. Doering, and J. E. Lumpp, "Design of flight software for the KySat CubeSat bus," in *IEEE Aerospace Conference Proceedings*, pp. 1–15, IEEE, mar 2009.

[51] C. Mitchell, J. Rexroat, S. A. Rawashdeh, and J. Lumpp, "Development of a modular command and data handling architecture for the KySat-2 CubeSat," in *IEEE Aerospace Conference Proceedings*, pp. 1–11, IEEE, mar 2014.

[52] G. Manyak and J. M. Bellardo, "Polysat's next generation avionics design," in *2011 IEEE Fourth International Conference on Space Mission Challenges for Information Technology*, pp. 69–76, Aug 2011.

[53] I. Sünter, *Software for the Estcube-1 Command and Data Handling System*. PhD thesis, Institute of Computer Science, University of Tartu, 2014.

[54] D. Schor, J. Scowcroft, C. Nichols, and W. Kinsner, "A command and data handling unit for pico-satellite missions," in *Canadian Conference on Electrical and Computer Engineering*, pp. 874–879, 2009.

[55] S. A. Asundi and N. G. Fitz-Coy, "Design of command, data and telemetry handling sys-

tem for a distributed computing architecture CubeSat," in *IEEE Aerospace Conference Proceedings*, pp. 1–14, IEEE, mar 2013.

[56] C. Araguz López, *Towards a modular Nano-Satellite Software Platform: Prolog Constraint-based Scheduling and System Architecture*. PhD thesis, Universitat Politècnica de Catalunya, sep 2014.

[57] M. A. Normann and R. Birkeland, *Software Design of an Onboard Computer for a Nanosatellite*. PhD thesis, Norwegian University of Science and Technology, 2016.

[58] M. Pagnamenta, *Rigorous software design for nano and micro satellites using BIP framework*. PhD thesis, École polytechnique fédérale de Lausanne, 2014.

[59] S. Nakajima, J. Takisawa, S. Ikari, M. Tomooka, Y. Aoyanagi, R. Funase, and S. Nakasuka, "Command-centric architecture (c2a): Satellite software architecture with a flexible reconfiguration capability," *Acta Astronautica*, vol. 171, pp. 208–214, 2020.

[60] R. J. Barnett, "Oneweb non-geostationary satellite system: Technical information to supplement schedule s - attachment to fcc application sat-loi-20160428-00041," tech. rep., 2016.

[61] M. Albulet, "Spacex non-geostationary satellite system: Technical information to supplement schedule s - attachment to fcc application sat-loa-20161115-00118," tech. rep., 2016.

[62] M. A. Diaz, J. C. Zagal, C. Falcon, M. Stepanova, J. A. Valdivia, M. Martinez-Ledesma, J. Diaz-Peña, F. R. Jaramillo, N. Romanova, E. Pacheco, M. Milla, M. Orchard, J. Silva, and F. P. Mena, "New opportunities offered by Cubesats for space research in Latin America: The SUCHAI project case," *Advances in Space Research*, vol. 58, pp. 2134–2147, nov 2016.

[63] C. E. Gonzalez, C. J. Rojas, A. Bergel, and M. A. Diaz, "An architecture-tracking approach to evaluate a modular and extensible flight software for cubesat nanosatellites," *IEEE Access*, pp. 1–1, 2019.

[64] I. Sommerville, *Software Engineering*. Delhi, 2006.

[65] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns – Elements of Reusable Object-Oriented Software*. Pearson Education, 1994.

[66] F. Buschmann, R. Meunier, H. Rohnert, P. S. Stal, and A. Michael, *Pattern-Oriented Software Architecture: a system of patterns*, vol. 1. Wiley, 1996.

[67] M. Grubb, J. Morris, S. Zemerick, and J. Lucas, "Nasa operational simulator for small satellites (nos3): Tools for software-based validation and verification of small satellites," in *AIAA/USU Conference on Small Satellites*, 2016.

[68] J. Morris, S. Zemerick, M. Grubb, J. Lucas, M. Jaridi, J. N. Gross, J. A. Christian, D. Vassiliadis, A. Kadiyala, J. Dawson, *et al.*, "Simulation-to-flight 1 (stf-1): A mission to enable cubesat software-based verification and validation," in *54th AIAA Aerospace Sciences Meeting*, p. 1464, 2016.

[69] M. Pessans-Goyheneix, J. Bønding, M. Burchard, T. Kasper, and F. Jensen, "Software Framework for Reconfigurable Distributed System on Aausat3," tech. rep., Aalborg University, 2008.

[70] S. Corpino and F. Stesina, "Verification of a CubeSat via hardware-in-the-loop simulation," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 50, no. 4, pp. 2807–2818, 2014.

[71] M. Petre, "Why looking isn't always seeing: readership skills and graphical programming," *Communications of the ACM*, vol. 38, pp. 33–44, jun 1995.

[72] A. Bergel, D. Cassou, S. Ducasse, and J. Laval, *Deep Into Pharo*. Square Bracket Associates, 2013.

[73] A. Bergel, *Agile Visualization*. LULU Press, 2016.

[74] M. Lanza and S. Ducasse, "Polymetric views - A lightweight visual approach to reverse engineering," *IEEE Transactions on Software Engineering*, vol. 29, pp. 782–795, sep 2003.

[75] T. Gutierrez, A. Bergel, C. E. Gonzalez, C. J. Rojas, and M. A. Diaz, "Systematic fuzz testing techniques on a nanosatellite flight software for agile mission development," *IEEE Access*, pp. 1–1, 2021.