SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Autonomous Re-Configuration of Modular Spacecraft with Manipulator Arm

Author: Connall Shurey

Supervisors: Mark Post,
Cumanan Kanapathippillai

MEng in Electronic and Computer Engineering
4th Year Project Final Report

May, 2024

Abstract

The 2020 MOSAR (MOdular Spacecraft Assembly and Reconfiguration) project was initiated to develop innovative technologies aimed at standardising satellites and components by raising the degree of space system modularity. As part of the project, a ground demonstrator for on-orbit modular and reconfigurable satellites was created, consisting of reusable spacecraft modules and a repositionable symmetric walking robotic manipulator. This demonstrator can execute basic instructions to relocate the manipulator and move modules between positions. However, these instructions are currently generated and verified manually before transmission to the demonstrator.

In this project we develop a reconfiguration planning system to autonomously generate instructions for reconfiguring modular spacecraft. Despite the limited research in this field, we demonstrate the feasibility of developing a reliable reconfiguration planner that considers the physical constraints introduced by the mobile manipulator and local environment. This was achieved by dividing the discrete and continuous components of the reconfiguration planner into two separate systems that communicate through feedback strategies in a control-loop mechanism. Our results show the system's capability to solve complex reconfiguration problems and identifies areas for further development.

This project serves as a foundation for developing more advanced systems with enhanced capabilities, potentially benefiting the space industry in the assembly and reconfiguration of modular space systems.

Acknowledgements

I would like to give a big thank you to my project supervisor, Dr Mark Post, for providing guidance and feedback throughout the project; Along with giving me the freedom to explore and develop what interested me most. My exceptional peers at university that always pushed me to a higher standard throughout my time at university; And my family for making my time at university to further develop myself possible and providing endless support.

Statement of Ethics

After consideration of the University's code of practice and principles for good ethical governance no ethical issues were identified in this project.

Contents

1 Introduction

This section builds, and expands, on material previously included in the project Initial Report (see Appendix J - Initial Report)

1.1 Background and Context

In recent years, there has been rapid development in space systems driven by a global push for increased commercial accessibility. Current commercial systems are designed with a focus on minimizing mass and launch costs, resulting in highly customized configurations that often lack robust maintenance and repair capabilities. Consequently, the population of ageing satellites is expanding, and upon reaching the end of their operational life, they are either deliberately de-orbited using atmospheric deconstruction methods or left in orbit, contributing to the accumulation of space debris.

At present, there is little available technology to overcome these conditions. The HORIZON 2020 EU-funded MOdular Spacecraft Assembly and Reconfiguration (MOSAR) project [1] was therefore initiated to develop innovative technologies aimed at standardising satellites and components. Modularising and standardising space systems will benefit the European space industry by enabling mass production of standardised components, reducing assembly costs, shortening the time between customer orders and deployment in space, and facilitating direct in-orbit repair and component upgrades, thereby extending the lifetime of space systems.

MOSAR's primary objective is to create modular and reconfigurable satellites that can be assembled and adjusted in orbit. The project has developed a demonstrator for reconfiguring cubic modules using a mobile robotic manipulator to simulate module movement. Currently, the manipulator receives fixed instructions for module mobility from software simulations on Earth [2]. This research aims to enhance the system by developing an algorithm to automate module reconfiguration, enabling self-repair and self-assembly. Once implemented, this technology could automate space system assembly and platform construction in space, overcoming current limitations in the space industry.

1.2 Project Objectives and Specification

This project intends to enable autonomous assembly and reconfiguration of modular space systems by implementing a reconfiguration planning program made up of simple algorithms. This program, given the initial state and final state of a modular system, will generate a list of commands to be sent to a mobile manipulator to autonomously rearrange modules on a spacecraft or space platform. The planning program must account for physical constraints imposed by the mobile manipulator present on the modular system; therefore, this project will strive to explore methods of incorporating physical constraints into the planning process.

To achieve the research goal, the primary objective is to implement a functional planning program capable of autonomous module reconfiguration, which will be demonstrated through software simulation. If time allows, an additional goal is to physically demonstrate the planning program by integrating it with the available manipulator arm in the lab to reconfigure real modules.

To achieve the research objectives, the following sub-objectives have been identified:

1. Develop a reconfiguration planning program that generates module movement instructions for a mobile manipulator based on initial and final state configurations.
2. Enhance the reconfiguration planning program to integrate physical constraints imposed by the mobile manipulator.
3. Implement a display function to create reconfiguration slideshows or videos, allowing users to visualise the modular systems reconfiguration process.
4. Conduct systematic testing of the system with various inputs to analyse system performance during solution generation.
5. Demonstrate the system by integrating it with the laboratories robot arm to physically reconfigure real modules.

By pursuing these steps, the project aims to showcase the feasibility and effectiveness of the planning program for autonomous assembly and reconfiguration of modular space systems, potentially paving the way for practical applications in the space industry.

1.3 Report Structure

This document serves as a comprehensive report of the research and development carried out during the Autonomous Re-Configuration of Modular Spacecraft with Manipulator Arm project. The report encompasses the following key components:

1. **Literature Review and Research:** A thorough examination of the current state-of-the-art in modular reconfiguration, including a review of relevant literature and existing technologies in the field.

2. **Detailed Design Development:** Creation of a detailed design plan outlining the implementation strategy for the reconfiguration planning program, specifying key components and methodologies

3. **Implementation Description and Specification:** Description and Specifications of the final implemented design, detailing the development and optimisation.

4. **Design Analysis and Results:** Analysis of the implemented design, records of performance metrics, solution generation times, and failure rates obtained through testing and simulation.

5. **Discussion of Results:** Interpretation and discussion of the analysis results, evaluating their significance and implications within the broader context of the area of study.

6. **Project Management Approach:** Examination of the project management methodology employed throughout the project lifecycle, documenting the evolution of the project plan and strategic adjustments made to achieve project objectives.

7. **Recommendations for Further Work:** Identification of potential areas for future research and development to build upon the findings and achievements detailed in this report, suggesting methods for expanding and refining the implemented system.

2 Literature Review

This section builds, and expands, on material previously included in the project Initial Report (see Appendix J - Initial Report)

2.1 Overview of Modular Spacecraft

Modular spacecraft represent a design concept where the overall space system consists of interchangeable modules, each fulfilling specific functions such as propulsion, communication, power generation, or sensing. These standardised modules enable easy assembly to form a unified system, allowing for module movement or replacement to optimize craft efficiency and extend system lifespan. Adopting a modular design approach offers several advantages over traditional methods, including enhanced flexibility, adaptability, and simplified maintenance.

Modules feature standardised interfaces that govern physical and electronic interactions, facilitating seamless integration of modules with different purposes or manufacturers into the overall system architecture. While module sizes and shapes may vary across designs, standardisation principles ensure compatibility for integration. The scalability of modular space system architectures depends on the types and quantities of modules used, providing versatility and cost-effectiveness as the system can be tailored to suit specific mission requirements without necessitating a complete redesign.

2.2 State-of-the-art in Spacecraft Modularity and Autonomous Reconfiguration

This section explores existing cases of spacecraft modularity and reconfiguration technologies currently or previously in operation. Due to the challenges related to developing automated reconfiguration systems for space operations, there are limited existing cases of automated reconfiguration aside from the International Space Station (ISS). However, modular design principles have been integral in spacecraft development since the 1980s, notably with the introduction of the Multimission Modular Spacecraft (MMS).

2.2.1 Multi-mission Modular Spacecraft (MMS)

The Multi-mission Modular Spacecraft (MMS) was designed and deployed by NASA in the 1980s and 1990s [4] with the intention of decreasing space mission costs. Intended to be recoverable/serviceable by the Space Shuttle Orbiter [5], It is one of the first cases of modular designs seen in the space industry and has paved the way for future innovations.

The MMS consisted of a small number of immobile modules, with the most basic deployed MMS containing only modules for altitude control, communications and data handling, and the power subsystems module [4].

Figure 1: Artist rendering of the TOPEX/-Poseidon mission. Image from [3]

The MMS flew only six missions through its lifetime which was vastly different from the thirty-one expected in the 1970s [4], it suffered limitation in the form of electronic technologies rather than mechanical restraints. NASA's first Standard Spacecraft Computer (NSSC-1) [6] was developed to prevent requiring an entire redesign of onboard computers for each mission, requiring only a software redesign, though this was still a heavy burden affecting the MMS's mission flexibility. While no longer in operation as of 2006 [7], the system did show cost-savings in the range of 55% to 65% [4]. "The idea of a modular system serving many purposes was the pioneer of the leading systems within the space technology ecosystem today as it has left a lasting legacy" [4]. In the wake of the MMS's legacy, new design techniques were developed such as the Modular, Adaptive, Reconfigurable Systems (MARS) system-level architecture [4] that has built the foundation for modern space systems.

2.2.2 Modular Common Spacecraft Bus (MCSB)

The MCSB is a fast-development, low-cost, general purpose spacecraft platform consisting of a series of 4-5 modules stacked on top of each other, each serving separate functionality [9]. According to NASA, "the spacecraft is roughly one tenth the price of a conventional unmanned mission and could be used to land on the Moon, orbit Earth, or rendezvous with near-Earth objects." [10]

The MCSB system received the Popular Mechanics 2014 breakthrough Award for innovation in science and technology [11] and is proving to be at the forefront of existing modular space technologies, first deployed on the Lunar Atmosphere and Dust Environment Explorer (LADEE) mission in 2013 [12].

Figure 2: LADEE Bus Modules from the MCSB Architecture. Image from [8]

The MCSB system is an example of modularity being used to streamline and reduce costs of the initial development process of the craft, being able to carry up to 50kg of scientific equipment inside its payload module [9], though the end product is still a whole system that has limited in-operation service capabilities and is not capable of being reconfigured to adapt to mission requirements in-orbit.

2.2.3 International Space Station

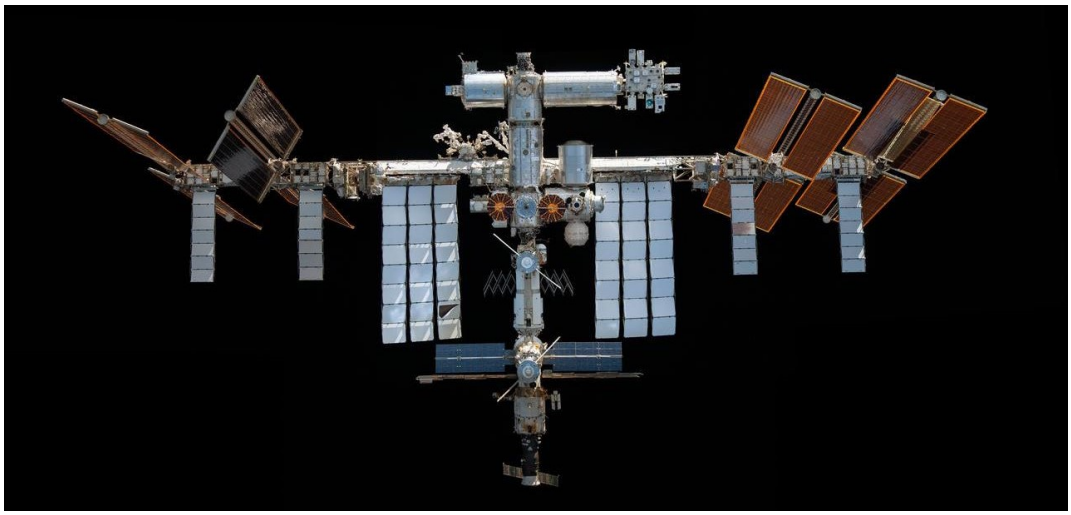

Figure 3: The ISS pictured from the SpaceX Crew Dragon (Dec. 8, 2021). Image from [13]

The International Space Station (ISS) seen in figure 3 is the largest space platform ever built, created with the purpose of performing microgravity and space environment experiments. First launched in 1998 [14] and expanded through the integration of additional modules and serviced by human occupants up until its planned de-orbit in 2031 [15], it is a monument to advancements in the space industry.

The ISS is capable of reconfiguration using a robotic arm and automated docking with human oversight [16] unlike previous cases, though unsupervised automated reconfiguration is yet to be attempted due to the consequences of failure.

Although the examples provided are not exhaustive, they encompass significant cases of modularity in the history of space exploration. Currently, automated spacecraft reconfiguration remains unimplemented in the industry. This project aims to contribute towards the future widespread adoption of automated modular reconfiguration by developing a system that can be compared with other emerging systems, aiding to identify techniques that offer the most substantial benefits. These techniques can then be utilised to create increasingly advanced reconfiguration systems for space applications.

2.3 Challenges and Limitations of Automated Reconfiguration in Space

The limited deployment of complex automated systems, like automated reconfiguration systems, in space is not due to a lack of interest, but rather stems from the formidable technical challenges and high-risk nature of space missions, which cannot afford failures due to their high cost and critical objectives.

Space systems must exhibit high reliability and operate effectively across a wide range of conditions. As system complexity increases, so does the number of potential failure points, making the validation, verification, and deployment of complex systems in the space industry a lengthy and costly process. Challenges that autonomous space systems face include:

- **Communication latency:** Delays in communications render real-time human intervention impossible, necessitating autonomous systems capable of operating independently without human oversight. Unlike terrestrial applications like self-driving cars that operate under human supervision, autonomous space systems must meet stringent autonomous reliability requirements.
- **Safety Requirements:** Systems will often be hosting valuable scientific equipment while operating in harsh, unpredictable environments with various hazards such as extreme temperature fluctuations, radiation, space debris, ice, and microgravity.
- **Limited Power Sources:** Autonomous systems rely on power sources that may not be constant or reliable. For instance, solar-powered crafts may experience power loss during eclipses or due to unexpected collisions with space debris. Autonomous systems must be capable of recovering from temporary power losses or have reliable backup power sources to prevent mission failure.
- **Isolation:** Unlike on Earth, space missions lack immediate external assistance or observation. Autonomous systems must possess robust sensing capabilities to self-diagnose issues, detect anomalies, and suspend standard operations when necessary to prevent further damage.

Overcoming these challenges demands cutting-edge technology, which has only recently become available, motivating research projects like this one. As computational power and materials sciences advance, we can expect a significant increase in autonomous systems within the space industry in the coming decades.

2.4 Emerging Advancements in Reconfiguration Technologies

2.4.1 MOSAR Project Outcomes

The MOSAR project has achieved several significant outcomes to date:

- Development of a standardized module framework utilizing the HOTDOCK adapter.
- Design and fabrication of a walking manipulator arm.
- Establishment of a related system architecture for remote control of the manipulator arm.
- Successful ground demonstration showcasing the manipulator arm's capabilities to move and connect modules.

At this stage, the MOSAR demonstrator could theoretically perform reconfiguration in orbit but currently requires manual transmission of reconfiguration instructions to the craft. Further work is needed to enable automated functionality, including:

- Automatic determination of a desired module configuration to meet mission requirements.
- Automated computation of manipulator instructions necessary to reconfigure the craft from one configuration to another.

The following review of automated reconfiguration literature will focus on identifying the best methods for the computation of manipulator instructions.

2.4.2 Automated Reconfiguration

Automatic planners, algorithms that find a solution for which sequence of operations must be accomplished to achieve a specified goal, have been an area of development attracting wide-spread interest since the earliest days of robotics. Currently there are many different types of automatic planning techniques available. They encompass a large set of algorithmic requirements which

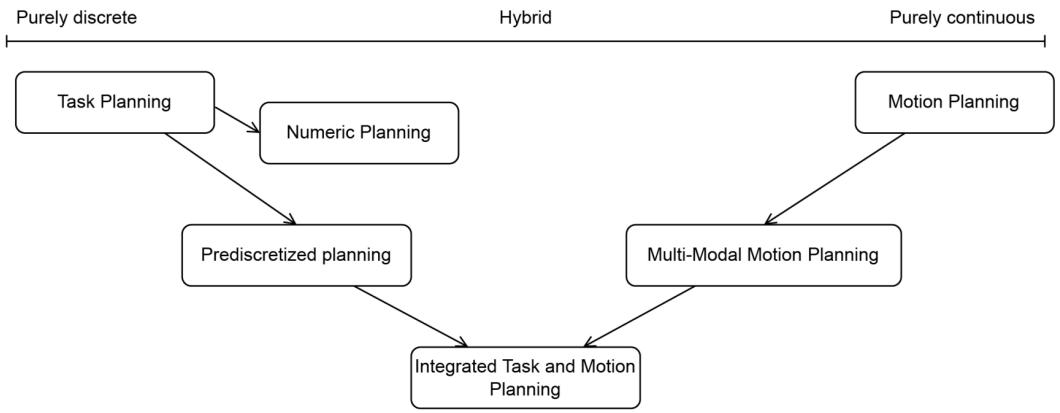

Figure 4: Taxonomy of automated planning approaches based on their search spaces' characteristics. Image from [18]

trend towards purely discrete or purely continuous search space characteristics. The development of "Hybrid" automated planning approaches with search space characteristics that are not purely discrete or continuous, especially Task and Motion Planning (TAMP) algorithms, represent an area of study of which solutions are considered the most computationally difficult in theory [17]. Consequently, the application of automated planning algorithms to robotic assembly of modular satellites is a very recent development in which little work has been published that implements automatic reconfiguration algorithms while fully considering the range of real-world physical restraints and limitations presented by usage of a mobile manipulator arm in a low-gravity environment.

2.4.3 Motion and Manipulation Planning

Motion Planning is finding solutions to move a robot "from one configuration to another configuration without colliding with the objects in the world" [18]. It involves searching for paths within the robots reach which is a continuous configuration space limited by dimensions represented by the joints of the robot. These collision-free paths are important for robot motion but do not by themselves allow the robot to interact with the world. Further planning must be implemented to allow manipulation of objects through manipulation planning (known as Multi-Modal Motion Planning).

Due to the increased complexity of the problem presented by manipulation planning, the problem is best broken down into a hybrid discrete-continuous search problem of "selecting a finite sequence of discrete action types (e.g. which objects to pick and place), continuous action parameters (such as object poses to place and grasps), and continuous motion paths" [18].

2.4.4 Task Planning

While Motion and Manipulation planning are seen as problems mainly within the robotics field, planning within large discrete domains such as in problems presented by task planning has been more deeply researched within the artificial intelligence (AI) community [19]. Task planning (also known as Action planning) referring to deducing a composition of symbolic actions to achieve a high-level goal (e.g. computing a sequence of actions required to stack boxes in a specified order). The discrete nature of the problem makes it particularly suitable for many machine learning techniques which have particularly advanced in recent years.

2.4.5 Task and Motion Planning

Current research in task and motion planning (TAMP) primarily aims to combine the robotics solutions for manipulation planning under physical constraints with the usually unrestricted AI approach to task planning. With the goal of deriving automated planning systems capable of reasoning symbolically with discrete "high-level" robotic action sets while geometrically taking into account continuous "low-level" robotic motion planning and restrictions. To date, several papers have developed algorithms for similar TAMP problems to the scenario of modular satellite reconfiguration that unfortunately are not compatible due to the method of module mobility, but act as a proof of concept that a solution is possible [20]–[22].

2.4.6 Related Work

The 2010 Intelligent Building Blocks for On-Orbit Satellite Servicing and Assembly (IBOSS) project [23] by DLR provided many advances in the area of satellite modularization with the development of standardised building blocks and interfaces [24]. Simple task planning techniques were implemented using Hierarchical task network (HTN) planning to produce high-level mobile arm instruction sets to then be verified through inverse kinematic checks and motion planning. This implementation solved the discrete and continuous planning problems separately, which simplified the problem however does not allow the separate systems to properly integrate. The system was not capable of efficiently solving more difficult tasks of identifying where solutions were not feasible.

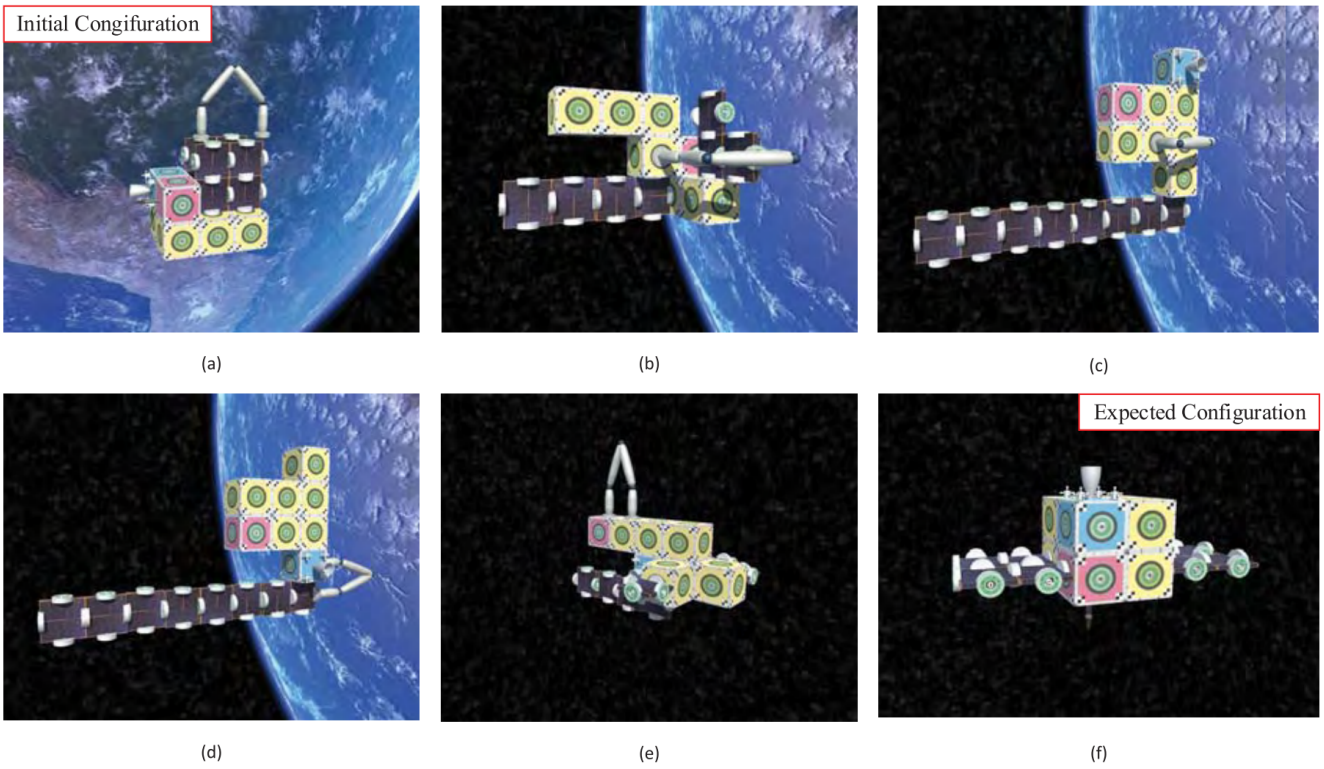

Figure 5: Melt-Grow algorithm simulation results. Image from [25]

Alternatively, another approach was taken here [25] through the implementation of the melt-grow algorithm [26]. The physical restraints of the robot were not including in the reconfiguration planning stage of the system, effectively reducing the problem to task planning. This reduces complexity though can only be achieved due to the behaviour of the melt-grow algorithm, which deconstructs (melts) the initial module configuration into chains of modules defined as the intermediate configuration, seen in configuration d in figure 5, before then reconstructing (growing) the modules into the expected configuration. The system then does not need to consider whether a move is possible for the mobile arm through manipulation planning as due to the algorithms inclusion of an intermediate state between the melting and growing operations, the algorithm essentially reconstructs the satellite instead of modifying the current state. All required moves are possible for the mobile manipulator and simply require motion planning. While proven to work, this method is shown to be highly inefficient for the mobile manipulator, especially as the number of modules increases in the system. Though, the paper [25] suggests this could be offset by the inclusion of additional manipulators which would consequently increase construction and operational costs.

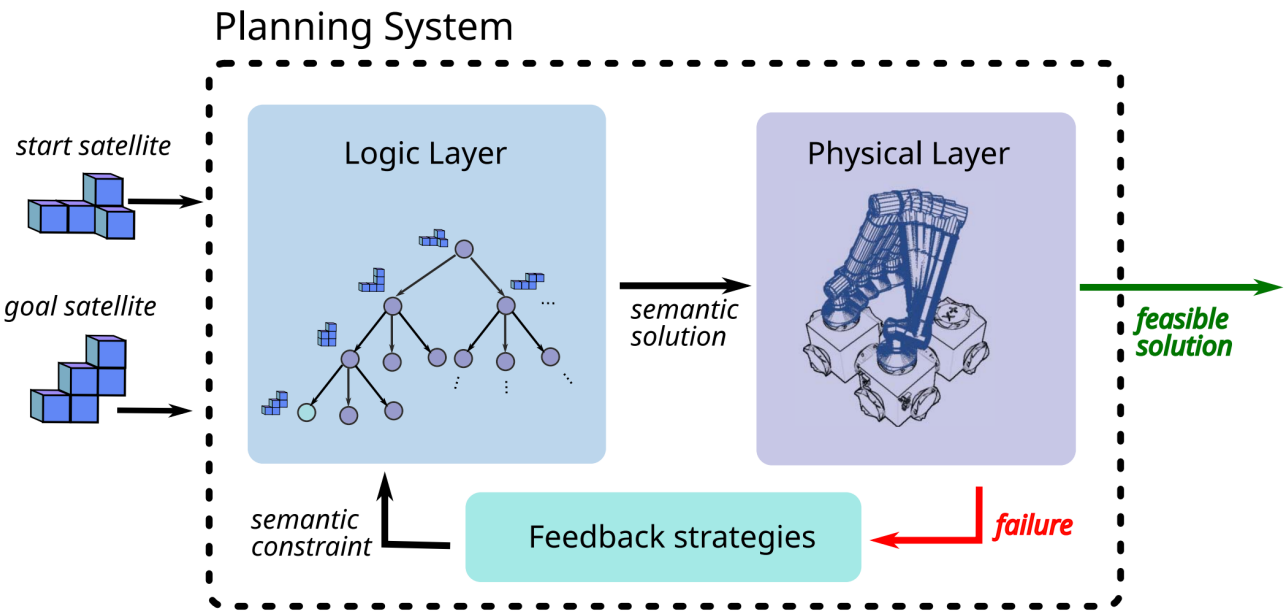

Figure 6: "Architecture of the autonomous robot planning system. The system receives as inputs the start and goal satellite configurations, and iterates between the logic and physical layer until a solution is found." Image and text from [27]

More recent research has taken inspiration from these previous works to propose a comprehensive Task and Motion Planning (TAMP) problem solver [27] to intrinsically include the robot constraints into the system. The system, seen in figure 6, includes a logic layer, a physical layer, and a feedback system. Where the logic layer acts as a task planner finding a semantic solution by considering the solution as a sequence of states, with module movements defining the transition between states. A graph is developed to represent the possible states where nodes are system states and edges represent module movements which are verified by the physical layer which provides manipulation planning results through the feedback system. Using this graph the shortest and hence most efficient set of operations to reconfigure the system into the desired state can be identified. The removal of the intermediate configuration present in the melt-grow algorithm improves the efficiency of the solution set of operations, especially as the number of modules in the system increases, requiring less movement from the mobile manipulator. The paper notes "the goal of this work was not to set a baseline for planning problems in terms of absolute times, but to demonstrate the usefulness of integrating feedback from the physical layer on the logic layer." [27], suggesting that there is an opportunity for further research into the components of the planning

system and the related feedback strategies to prepare the system for space applications.

2.5 Gaps and Opportunities

Modular reconfiguration defines a subclass of the generic planning problems usually addressed by TAMPs. Although research has previously demonstrated effective systems that can handle both symbolic and geometric reasoning, their application to robotic assembly and in particular robotic re-assembly is currently limited. There is additionally a distinct lack of discriminating modular blocks by type in existing algorithms which could potentially be easily implemented without a substantial hit to system performance.

The system proposed in figure [27] is promising due to the robustness of solutions and flexibility of the logic layer, however, there lacks the extensive performance testing required to recognise weaknesses and future improvements, identifying why this system could not be used in real-world application currently.

3 System Design and Development

3.1 Overview

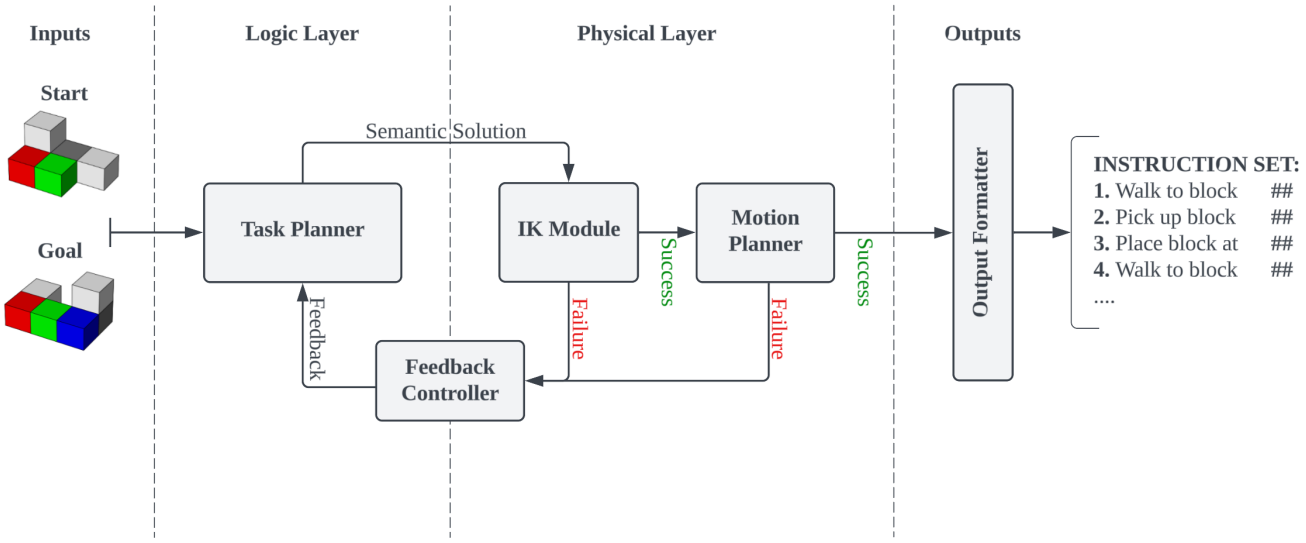

Figure 7: High-level System Design

The Reconfiguration Task and Motion Planner (TAMP) program is designed using Python, leveraging existing Python implementations [28] for controlling the robotic arm [29] available in the lab. The program inputs an initial and final state and outputs a list of instructions to reconfigure the initial state into the final state using a mobile manipulator.

The TAMP system comprises three main components: the logic layer, the physical layer, and the feedback controller. Each component has a distinct role, allowing the discrete and continuous aspects of the solution search to be managed separately while ensuring integration and communication between layers through the feedback controller.

- **Logic Layer:** Responsible for discrete task planning.
- **Physical Layer:** Manages continuous motion planning.
- **Feedback Controller:** Integrates the logic and physical layers, facilitating communication and iterative solution searching through feedback strategies.

The feedback strategies implemented by the feedback controller create a control loop behaviour, iteratively refining the solution. Once a feasible solution is found, it is formatted into the required instruction set by the output formatter.

3.2 Logic Layer

3.2.1 Overview

The logic layer of the TAMP program functions as a Task Planner, managing the discrete portion of the search to find semantic solutions. These solutions are sequences of state configurations, each differing by one module movement, that transform the initial state configuration into the goal state configuration.

While many contemporary task planners use machine learning techniques to find solutions, these are unsuitable for the space industry due to their black box behaviour. Instead, the TAMP program employs simple graph search techniques to ensure transparency and reliability.

3.2.2 Searching the Graph

To search the graph and find the path to the goal state configuration, two major search algorithms are considered: Depth-First Search (DFS) and Breadth-First Search (BFS).

- **Depth-First Search (DFS):** In DFS, the algorithm explores a path to its full depth before backtracking and trying alternative paths. This can be implemented using a state priority queue that sorts states based on their proximity to the goal state, enabling quick solutions. However, the resulting path may not be the most efficient for the mobile manipulator, as each state transition involves additional movement.
- **Breadth-First Search (BFS):** In contrast, BFS explores all states at one depth level before proceeding to the next level. It searches all states one step away from the starting state, then two steps away, and so on. Although BFS is much slower than DFS and less scalable to larger numbers of modules, it guarantees finding paths with the fewest transitions to the goal state, making it more suitable for efficient reconfiguration.

Given the requirement for solution efficiency over planner speed, the Task Planner implements the BFS algorithm. The pseudo-code for the Task Planner algorithm is shown in Algorithm 1.

Algorithm 1 FIND PATH

Input: $startState, goalState, maxBranches$

```
searchTree ← []
currentState ← startState
while currentState ≠ goalState do
    priorityQueue ← GenNewStates(currentState)
    for maxBranches do
        newState ← priorityQueue.pop(0)
        newState.parent ← currentState
        searchTree.add(newState)
    end for
    currentState ← searchTree.pop(0)
end while
return currentState.GetStatePath()
```

3.2.3 Generating States

To expand the graph, the task planner generates new states using the **'GenNewStates()'** function, referenced in the search algorithm pseudo-code (Algorithm 1). States are generated based on a set of rules that prioritize which modules to move, ensuring efficient state expansion. The priority rules for module movement are as follows:

1. Modules not yet in their final position
2. Modules adjacent to modules not yet in their final position
3. Remaining modules

These rules ensure that the task planner consistently generates new states while prioritizing the movement of modules that need to be repositioned first. This approach minimizes unnecessary movements and helps streamline the reconfiguration process. A priority queue that prioritizes

modules based on their distance from modules not yet in their final position was considered. This would allow for efficient repositioning of deeply embedded modules. However, this added complexity was deemed unnecessary for the current program's scope and would increase computation time for a relatively rare scenario. For larger structures, implementing such a priority queue could be beneficial to improve computation efficiency.

3.2.4 Trimming States

When handling inputs with large numbers of modules, the search tree expansion can quickly result in a vast number of states, consuming significant memory, and computation time. To expedite the search process, generated states are sorted into a priority queue, and only the highest priority states are added to the search graph. The remaining states are discarded, as illustrated in Figure 8. The states are prioritized based on their proximity to the desired goal configuration, using the following heuristics:

- Number of modules already in their final positions.
- Number of modules not in their final positions but occupying positions that are vacant in the goal state.
- Sum of the Euclidean distances of the module positions from their final positions in the goal state.

These heuristics measure how far a state is from the goal state and allow comparison between states to determine which is closer to the goal. States are first sorted using the number of modules in their final positions. In the event of a tie, the second heuristic is used. If there is still a tie, the computationally intensive third heuristic is applied.

By primarily using the first two heuristics, the planner reduces the frequency of expensive calculations, thereby speeding up the state comparison process and improving overall efficiency.

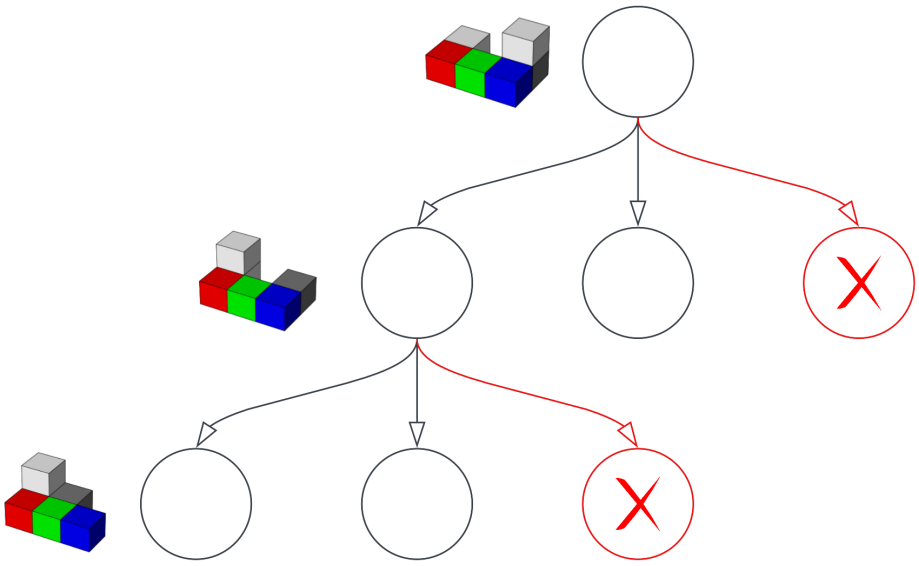

Figure 8: Selection of generated states demonstration

3.2.5 Physical Layer Feedback

When a semantic solution is identified, it is passed to the physical layer for verification. If the physical layer encounters a failure, the transition causing the failure is pruned from the search tree, and all subsequent transitions on that branch are removed. The search then resumes without the failing transition.

An alternative approach would involve performing physical layer checks for each move during state generation. However, physical layer calculations are significantly more computationally intensive compared to those in the logic layer. Thus, it is more efficient to focus on verifying only the transitions within the semantic solution, even if it means spending more time searching for these solutions. This approach balances computational load by leveraging the relatively quicker logic layer to identify potential solutions and reserving the intensive checks for validation.

3.3 Physical Layer

3.3.1 Overview

The physical layer is primarily composed of an inverse kinematics verifier and a motion planner. Its main function is to verify the physical feasibility of transitions in the semantic solution proposed by the logic layer. It processes a semantic solution and returns either a success or a failure.

- **Inverse Kinematics Verifier:** This module verifies whether the poses for grabbing and placing a module are possible from the current base position. If either pose is not feasible, the module attempts to find a base position that allows both poses. If both poses are feasible from the same base position, the verifier permits the semantic solution to proceed to the motion planner. If it fails to find such a position, it triggers an inverse kinematics failure and discards the semantic solution.

- **Motion Planner:** Upon receiving a transition, the motion planner determines a path from the start pose to the end pose that avoids collisions between the arm, the grabbed module, and other modules. If no collision-free path is found, the motion planner returns a motion planning failure, and discards the semantic solution.

3.3.2 Inverse Kinematics Verifier

Inverse kinematics (IK) determines the joint angles required to position a mobile manipulator's end-effector at a desired location and orientation. There are two primary methods for solving IK [30]:

- **Analytical Approach:** This method involves deriving a mathematical solution specific to each unique manipulator arm. It is complex because it requires detailed analysis and formulation. However, once the formulas are derived, calculations are extremely fast, making it suitable for frequent computations.

- **Pseudoinverse Jacobian Method**: This iterative method guesses the required joint angles and adjusts them incrementally to achieve the target position and orientation. It can be applied to any arm configuration without needing unique formulations but is computationally intensive.

Given the need for frequent IK calculations, the analytical approach is preferred. The mobile manipulator is represented using Denavit-Hartenberg (DH)[31] parameters, which facilitates the derivation of analytical formulae for each joint. The formulae allow for efficient computation of IK, ensuring that the system can quickly verify and adjust the manipulator's poses.

3.3.3 Manipulator Base Location Planning

The mobile manipulator present in the MOSAR [1] project can traverse the modular system. If the IK module fails to connect to a module from a position, moving the arm closer to the module that failed the inverse kinematic check could result in a successful solution like in figure 9. When the inverse kinematics module fails a check, it attempts to move the base to another available surface in between the 2 movement points. As our available arm is stationary, our implementation will only include base location planning if time permits.

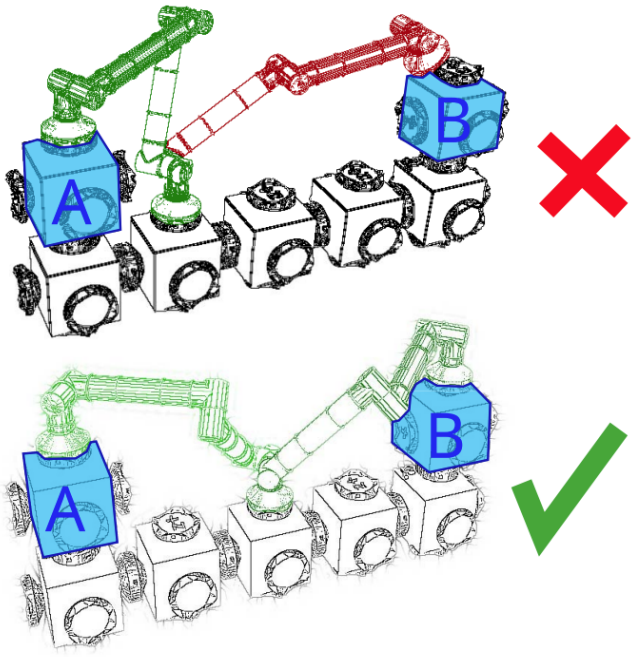

Figure 9: "Inverse kinematic checks performed to verify if it is possible to translate a module from position A to position B. In the first case (above) this is not possible, but it is solved by changing the intermediate position in the second case (bottom)." Image and text from [27]

3.3.4 Motion Planning

Originally, the plan was to implement a motion planner using the Rapidly-exploring Random Tree (RRT) algorithm used here [32] to generate collision-free motion paths for the robot arm. However, due to time constraints, a simpler approach is adopted.

In the lab scenario, where modules are reconfigured on a platform by a stationary robot arm, motion planning is simplified. We can assume that a module can be picked up if there are no modules above it and if it is within reach according to the inverse kinematic solution. Modules are then moved between positions by lifting them to the arm's maximum z-limit, moving above the new position, and lowering the module into place.

Additional physical constraints are applied through feedback:

- Modules cannot be moved to negative-z values due to the platform.
- Modules must be placed on top of the platform or another module due to gravity.

These constraints will induce physical layer failures, leading to an expected increase in the time taken to generate results compared to operation in space.

3.3.5 Failure Feedback

The physical layer reports various types of failures upon detecting conflicts, including:

- **Out of reach:** The starting and final positions of the module are unreachable for the current mobile manipulator.
- **No Base Location:** Although the starting and final positions of the module are within reach, there is no suitable base position on the module configuration to reach both points.
- **Collision:** There is no available path to move the module without colliding with another module.

3.4 Feedback Strategies

Without feedback strategies, the system can find a step-by-step solution to reconfigure modules into the desired goal configuration and verify whether the mobile manipulator can execute the semantic solution. If the mobile manipulator cannot execute the solution, the system fails in its current state.

Feedback Strategies establish a control loop that facilitate communication between the logic and physical layers, enabling them to collaborate towards finding solutions that meet their respective goals. While various feedback strategies were considered, the project focuses on implementing those related to final output validation rather than computation time optimization. However, strategies for both aspects were explored in case of early project completion or for future work beyond the project's scope.

3.4.1 Semantic Solution Verification

The primary feedback strategy implemented in this project involves simple verification of semantic solutions. Failed solutions are returned to the logic layer, and the problematic transition is removed from the search tree, along with all associated states. The logic layer then continues the search. There is concern that early failures in the physical layer may significantly reduce the search space, potentially hindering the discovery of reasonable solutions. Further testing and research are needed to address this issue and refine the feedback strategy.

3.4.2 Failure Memory

Although implementing a failure memory, like the one developed in previous work [27], was considered, it was deemed beyond the project's scope due to its primary focus on time performance. The failure memory utilises a machine learning algorithm to predict physical layer failures based on past data, optimizing the system's runtime by reducing failures. Incorporating a failure memory would provide the system with scalability over time with the collection of failure data, enabling it to generate solutions for larger systems in a reasonable timeframe. Though currently it is unsuitable for space applications due to the reliance on black box machine learning algorithms.

4 System Implementation and Specifications

4.1 Hardware Specifications

4.1.1 Processing Hardware

The current software implementation requires a simple processor but is constrained by its speed. The primary hardware limitation is the available RAM, with a recommended range of 2 to 4 GB for the planner, depending on the size of the input state configuration.

During the project, system timing results were obtained using a general-purpose desktop computer with the following specifications:

Operating System:	Windows 10 Pro 64-bit (10.0, Build 19045)
Processor:	AMD Ryzen 7 3700X 8-Core (16 CPUs) clocked at approximately 3.6 GHz
GPU:	NVIDIA GeForce RTX 2070 Super with 8GB VRAM
RAM:	3 x 8 GB DDR4 Memory clocked at 3200 MHz
Memory:	1 TB M.2 SSD with read speeds of 7,300 MB/s and write speeds of 540 MB/s

4.1.2 Mobile Manipulator

The mobile manipulator available in the lab is the Automata EVA, further details can be seen in Appendix D - Automata EVA Technical Specifications. The manipulator was not physically used during the project due to time constraints. Though, the arm specifications were used for simulation so the current state of the reconfiguration planner can be integrated with the arm in a future project.

4.2 Software Specifications

4.2.1 Software Architecture Overview

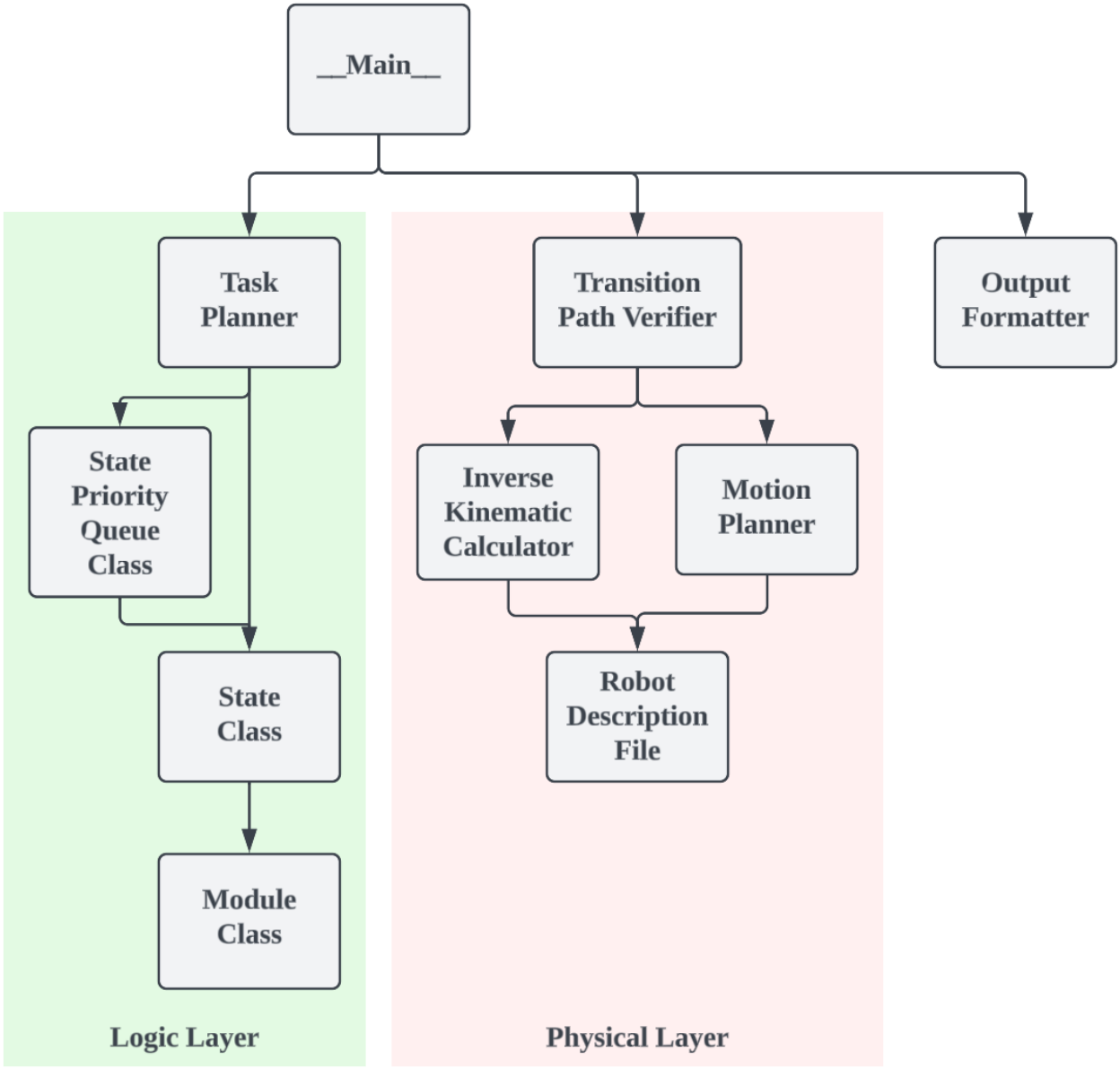

Figure 10: High-level System Implemented Design

The modular structure of the final software implementation is illustrated in Figure 10. The main file integrates the Logic Layer and Physical Layer, facilitating communication and applying feedback strategies between them. Once a solution is found, the Output Formatter is used to display the instructions to users (seen in Appendix B - 5 Module Test) and to create animations of state reconfiguration transitions for visual analysis of the process.

4.2.2 Logic Layer - Overview

The Logic Layer consists of a Task Planner and its related methods, and classes to implement States, Modules and State Priority Queues

4.2.3 Logic Layer - Task Planner

The Task Planner begins by verifying the start and goal states have the same number and composition of modules; And generates states utilising the State Priority Queue and State Classes. It returns an array of states representing the transitions required to reconfigure the start state into the goal state.

The Planner consists of two main methods, **"FindPath()"** and **"GenNewStates()"**. The **"FindPath()"** method implements the search algorithm shown in Algorithm 1, while **"GenNewStates()"** expands the tree based on the heuristics specified in the design section 3.2.3, as can be seen in Algorithm 2. After generating states, "FindPath()" records their parent states, enabling the program to trace the transition path from the starting state to the goal state.

Algorithm 2 GEN NEW STATES

Input: $state, goalState$

$stateQueue \leftarrow newStateQueue(goalState)$
$from \leftarrow state.getNonFinalModules()$
$to \leftarrow state.getAvailablePositions()$
$stateQueue.push(state.GenerateMoves(from, to))$
if stateQueue.empty() **then**
  $from \leftarrow state.getAdjacentModules(b)$
  $stateQueue.push(state.GenerateMoves(from, to))$
end if
if stateQueue.empty() **then**
  $from \leftarrow state.getModules()$
  $stateQueue.push(state.GenerateMoves(from, to))$
end if
return $stateQueue$

4.2.4 Logic Layer - State Priority Queue Class

The State Priority Queue class maintains a sorted list of states using a simple array data structure. States are ordered based on their proximity to the goal state, as determined by the heuristics detailed in design section 3.2.4. When a state is inserted, a binary search algorithm [33] is used to find the appropriate position in the queue to maintain the queue's priority order. Initially, a linear search algorithm was used for simplicity, but during optimization, the binary search algorithm was implemented, significantly reducing the overall insertion time.

4.2.5 Logic Layer - State Class

The State Class represents a state configuration and stores module positions. It includes methods for state comparisons, measurements, validation, and visualization. Modules are stored in a dictionary data structure, where keys represent module positions and values are module objects.

Initially, a position matrix was used for its simplicity in testing and modifying logic, which sped up development. However, it was later replaced by a dictionary for optimisation as unlike a position matrix, dictionaries do not store unnecessary zero values so increase memory efficiency of the system.

The class includes a verification function to ensure all modules in the state are connected. Originally, an out-of-the-box labelling algorithm from the scikit-image python package [34] was used with the position matrix to verify connectivity. If the algorithm could only label one cluster of objects, then all modules in the state were connected. After switching to a dictionary, a new search algorithm was implemented for state verification (shown in Algorithm 3).

Several functions are implemented to measure the number of modules in final positions, the number of modules in free positions and the Euclidean distance between all modules in non-final positions and their final positions. Since these measurements are often requested multiple times for the same state, the calculated values are saved within the state after the initial computation and updated only if the goal state changes.

Algorithm 3 VERIFY STATE

Input: *state*

```
foundList ← []
searchList ← [state.getFirstModule()]
while searchList.length ≠ 0 do
    module ← searchList.pop()
    for neighbour ∈ module.getNeighbours() do
        if neighbour ∉ foundList then
            if neighbour ∉ searchList then
                searchList.push(neighbour)
            end if
        end if
    end for
end while
return foundList.length() ≡ state.numModules()
```

For visualising reconfigurations and aiding in-depth testing and analysis, the State Class includes a display function. This function translates the dictionary into a position matrix, which is then displayed as a 3D matrix using Matplotlib [35]. Configurations, such as the one shown in figure 11, can then be visualised. Additionally, this function is used to create reconfiguration videos, enabling clear visualisation of the system's output.

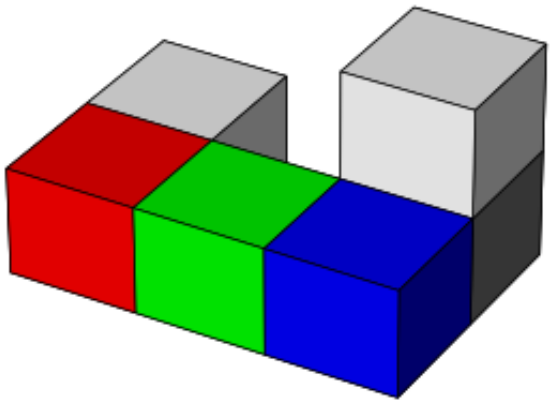

Figure 11: State Visualisation

As shown in Algorithm 2, the State Class includes a function called **'GenerateMoves()'** for generating a list of states for mass movement of modules. This function takes two lists: one of module positions and another of positions the modules can move to. For each module, the function validates the state without the moving module to ensure the state remains intact during movement. It then moves the module to each of the possible positions, adding the new state to a return list if the state is valid after the movement. This custom mass movement function optimizes state generation during search tree expansion.

4.2.6 Logic Layer - Module Class

The Module Class is a straightforward class that holds information about a module, primarily used for comparing modules through an **"Equals()"** function. This function can be modified to adjust which module properties determine equality. In its current implementation, colour is used to decide if two modules are equal. There is functionality to compare by module type or module identification number. However, comparing by colour simplifies analysis during testing, as it allows for visual differentiation of modules when displayed.

4.2.7 Physical Layer - Inverse Kinematics Verifier

As detailed in design section 3.3.2, the implemented Inverse Kinematics Verifier uses an analytical solution to calculate the manipulator joint angles required to position the end-effector at a specified location. Initially, an analytical formula was developed specifically for the Automata EVA arm (details in Appendix D - Automata EVA Technical Specifications) available in the lab. This limited the compatibility of the reconfiguration planner to only that specific manipulator.

To increase compatibility, a Unified Robotics Description Format (URDF) file was created for the Automata EVA, as shown in Appendix E - Automata EVA URDF File. The IKPy package [36] was then used to generate an analytical solution for use by the Inverse Kinematics Verifier and Motion Planner. Users can now update the reconfiguration planner to work with different mobile manipulators by simply replacing or modifying the URDF file.

The inverse kinematics verifier is used to ensure that the start and final position of each module movement in the reconfiguration semantic solution is reachable by the mobile manipulator. In the case of a module being out of reach, the verifier returns the state transition that caused the failure. Otherwise, the semantic solution is sent to the motion planner for further processing.

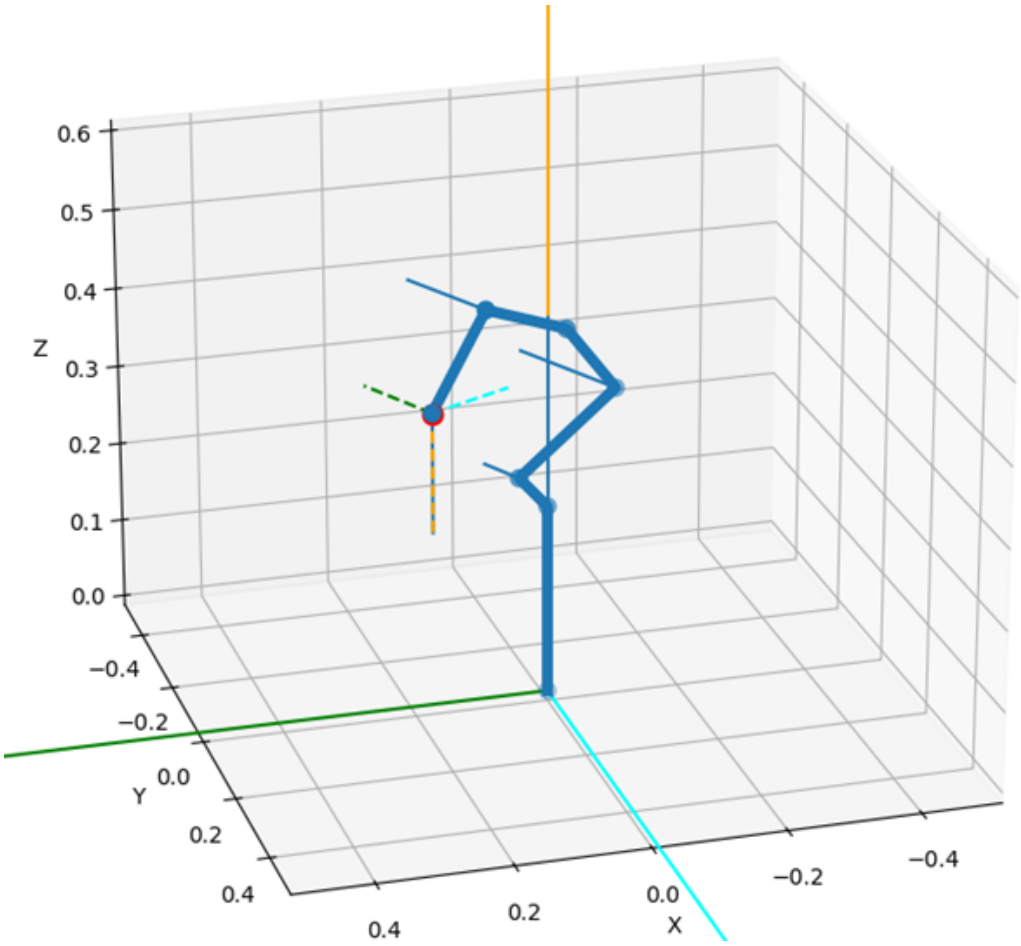

Figure 12: Automata EVA Kinematics Visualisation

4.2.8 Physical Layer - Robot Description File

A URDF file is used to define the mechanical structure, dimensions, joint configurations, and physical constraints of the mobile manipulator the physical layer is simulating to verify the logic layers semantic solution. URDF files are an XML-based file format that is widely used in robotics [37] to describe robots to software systems. The file describes a robot as a collection of links and joints that can articulate around each other according to specified constraints. URDF files are also modular meaning they can include other URDF files, aiding in the design of particularly complex robots. This for example means that a user can develop a URDF file for an arm end-effector

and simply include it in the already existing arm file to attach it to the arm. URDF files also allow for the visualization of the defined arm joints, as seen in figure 12 which can be overlaid on top of our module state display to visualise mobile manipulator pose on the modular space system. Additionally available online packages such as urdf-loader [38] can display the visual meshes described in the URDF file to view the mobile manipulator in more detail such as seen in figure 13.

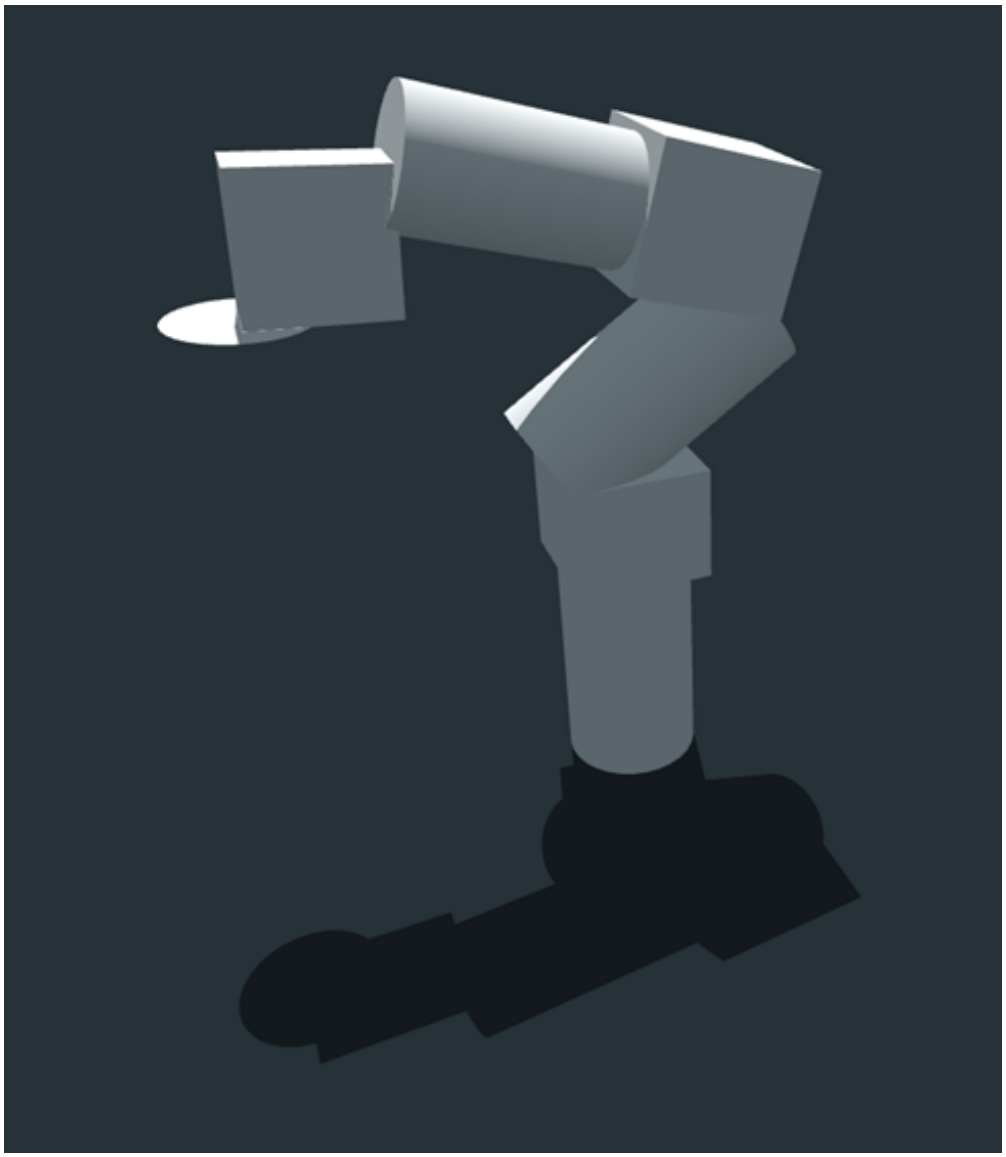

Figure 13: Automata EVA URDF Model Visualisation created through urdf-loader [38]

4.2.9 Physical Layer - Motion Planner

Due to time constraints during the project, instead of implementing an advanced motion planner, a simple motion planner was implemented that makes assumptions based on the lab environment surrounding the mobile manipulator available in the lab. Physical constraints of the robot arm and the environment are applied to the system during module movements using 2 basic rules:

1. Modules can only be picked up by the mobile arm if no blocks are above them.
2. Modules can only be placed at a supported position (above another module and on the ground) and cannot be placed at negative z values (below the ground)

The combination of these two rules applies the physical constraints of gravity and the presence of the ground. If either rule is broken, the planner returns the state transition causing the failure. Otherwise, it generates and returns an instruction set in the form of an array containing each manipulator action sequentially required to perform the state reconfiguration. The available instructions the motion planner can generate can be seen in Figure 14.

Instruction	**Information**	**Action**
CONNECT	None	Signal the arms end-effector to grab/connect
DISCONNECT	None	Signal to the arms end-effector to drop/disconnect
MOVE TO	Position, Joint angles	Move the arm to position the end-effector at the supplied position by transitioning the arms joint angles to the supplied joint angles

Figure 14: Instructions available to the Physical Layer Motion Planner

4.3 Feedback Strategies

Currently, the system employs a basic feedback strategy. Upon detecting a failure in the physical layer, the system returns the state transition responsible for the failure. Subsequently, the branch of the tree originating from the failing state transition is removed, eliminating states where the failing state was as an ancestor. The pruned search tree is then passed back to the logic layer to pursue an alternative semantic solution.

4.4 Implementation Challenges

The first implementation iteration suffered from several initially unforeseen problems that required in-depth analysis and modifications to the system. These issues resulted in the system either being unable to find a solution, requiring more powerful hardware, or taking so long to find a solution that testing was not feasible.

4.4.1 Memory Usage

The implemented data structure relied on a dictionary with positions serving as keys, storing only essential information, and facilitating straightforward modifications to internal logic and computations. However, during development, when testing configurations with a larger number of modules, the system encountered memory issues, halting the search prematurely despite having ample memory (24 GB). Upon investigation, it was discovered that the in-built copy module [39] in Python not only replicated objects, but also copied referenced objects within the object, leading to a duplication of the entire search tree with each transition. This oversight stemmed from a lack of familiarity with Python's memory management mechanisms. Subsequently, the development of a custom duplication method for the class significantly reduced memory usage and improved the efficiency of the search algorithm by 90%, enhancing the capabilities of the logic layer.

4.4.2 Repeated Computations

During an analysis of the system's performance, the **"GenNewStates()"** function (Seen in Algorithm 2) emerged as one of the slowest processes. This function inserts states into a priority queue using a custom comparison function, which involves calculating multiple measurements for each state to determine priority. Initially, the linear search used to find the insertion index was suspected to be the bottleneck. However, after implementing a more efficient binary search algorithm, the expected improvement in insertion time was not fully realized.

Further analysis showed that when inserting a state, if there were many states in the queue, the inserting state would undergo many comparisons to find the insertion index required to maintain priority order. Each time the inserted state is compared against another state, it would calculate its own measurement values and ask the other state to also calculate its measurement values. As these measurement values are based on their module layout, composition, and the goal state, all of which does not change during runtime, there was no need to be repeating calculations.

To address this issue, we optimized the process by saving measurement values upon calculation for each state and recalculating them only when necessary or when the goal state changed. This approach, coupled with the implementation of a binary search, significantly reduced the overall number of calculations required during insertion, improving the efficiency of the system.

4.4.3 Optimising Python

The greatest overall roadblock to further success and experimentation simply came down to the speed of the system and identifying slow processes. Due to the nature of the python programming language, often a considerable time-wasting process was a simple function that unknowingly had large overheads involved. The most successful method of improving the speed of the system during development was writing custom functions that fully utilise the underlying C programming language python was built on. While there was no issue writing the custom utility functions, identifying where custom functions needed to be implemented took up most of the dedicated development time.

5 Testing and Results

5.1 Testing Method

To test the system, a generated set of module configurations were input into the system, and the total time spent in the logic layer and hardware layer were recorded separately for both the system with and without physical constraints applied. An average time is then calculated for configurations consisting of $4 - 9$ modules. The branching factor was also varied throughout the test to analyse the effect on total time, failure rate and number of moves returned in the final solution.

5.2 Performance Metric

To measure performance of the overall system, the testing methods measure:

- The number of failures encountered during the solution search with feedback strategies.
- The number of modules effect on failure count
- The number of search branches effect on failure count
- The number of search branches effect on number of moves in the solution.
- The number of search branches effect on search time.
- Total calculation time.

To quantify the performance of the implemented feedback strategy, the number of semantic solutions that fail in the physical layer is used. The physical layer is the most computationally demanding section of the overall system, so it should be used as little as possible. Time spent calculating results is recorded for comparison with other systems and to prove the systems capabilities for real-world use but is not considered a measure of project success.

5.3 Analysis of Results

		branches					
		1	2	3	4	5	6
# of modules	4	0.106s	0.11s	0.124s	0.149s	0.193s	0.254s
	5	DNF	DNF	0.115s	0.169s	0.262s	0.414s
	6	DNF	DNF	0.735s	2.176s	6.362s	18.974s
	7	DNF	DNF	0.345s	0.487s	0.684s	0.936s
	8	DNF	DNF	DNF	DNF	10.987s	42.703s
	9	DNF	DNF	DNF	DNF	DNF	47.393s

Figure 15: Average time per move to find a solution

Tests were conducted on a system with the hardware specifications shown in section 4.1, results can be seen in Appendix C - Test Results. As expected, as the number of modules in the input configuration rises, so does the number of physical layer failures during the solution search as fig 16 shows. Interestingly, the more modules in the state, the more branches the task planner search tree requires to completely avoid failing the search by reaching max recursions. Fig 15 shows a good example of this where failed tests are labelled DNF, and passed tests show the total time take to find a solution divided by the number of moves in the solution, giving an average time per generated move. The time it takes to find a solution seems to be almost completely irrelevant for analysis of the system, as it is dependent on the configuration of the state entered. For example, if a module must be moved out of its final position first to enable another module to be moved to its final position, it will take longer to find a solution than if all modules are already accessible.

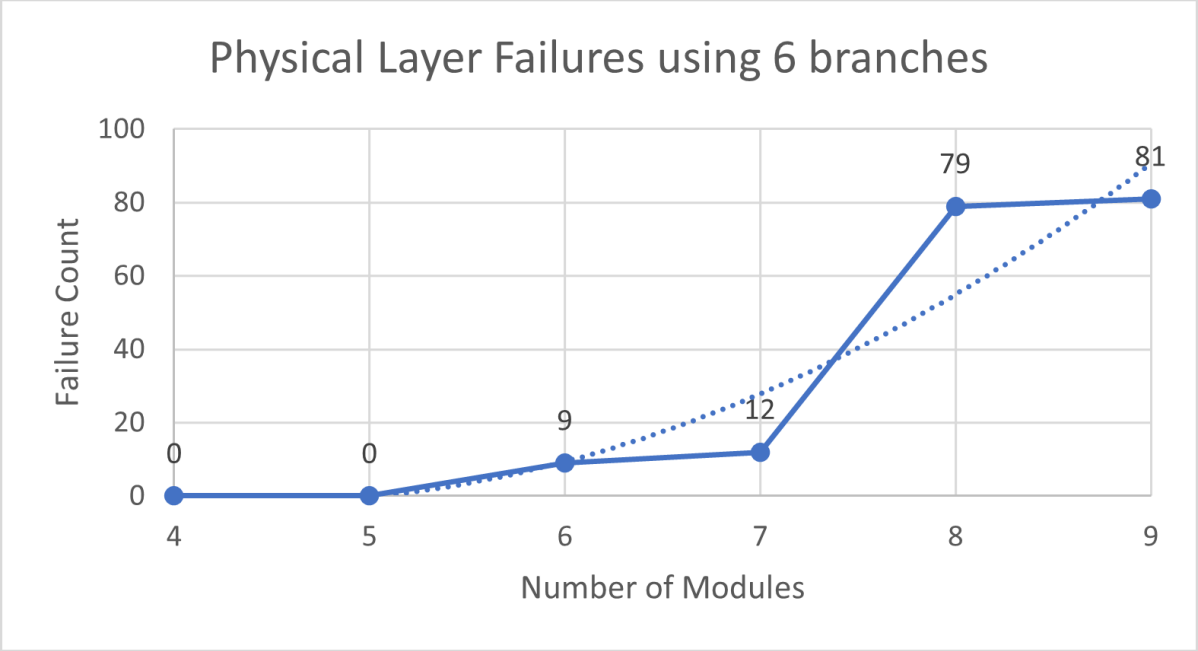

Figure 16: Failures vs Number of Modules for solutions found using 6 branches

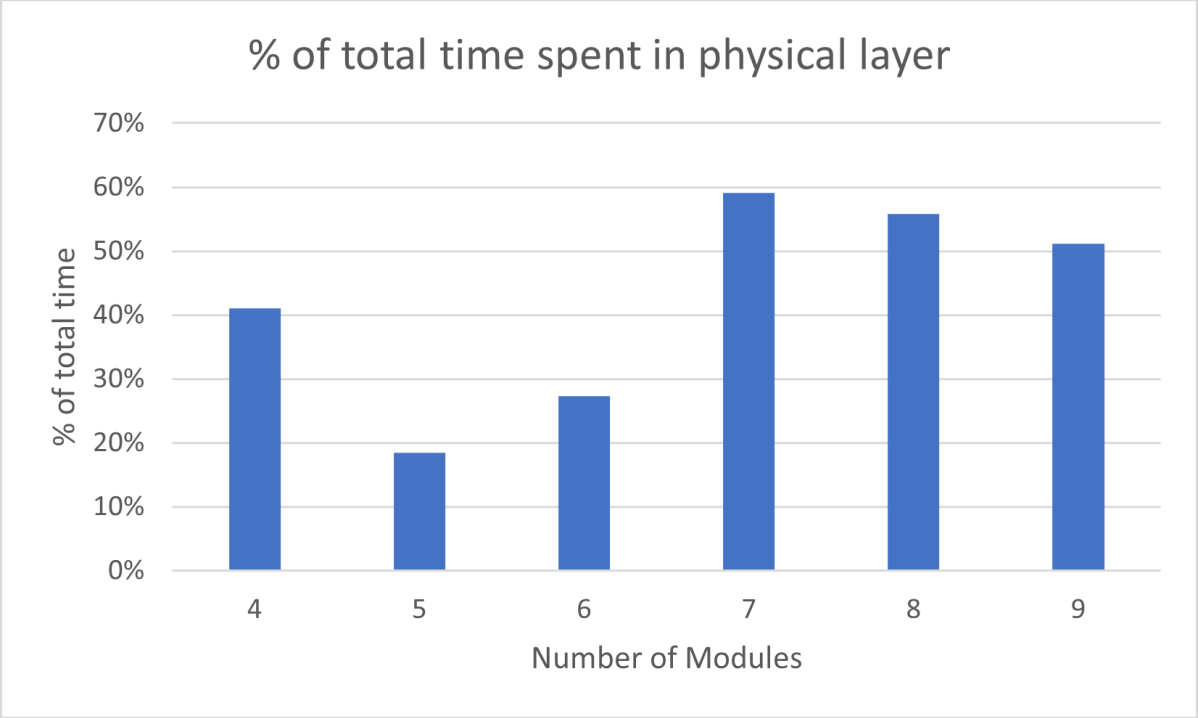

Figure 17: Percent of total time spent in the physical layer to find a solution (on average)

6 Discussion

6.1 Interpretation of results

The results indicate that while the system is almost guaranteed to find a solution if one exists, the efficiency and time required are highly dependent on the maximum number of branches in the logic layer's search tree. The optimal number of branches appears to be a function of the number of modules in the input state configuration, with larger configurations requiring more branches. This need arises due to the increased number of physical layer failures with higher module counts (Shown in Fig 16), necessitating more alternative paths to find a solution. Interestingly, the time spent in the physical layer does not increase according to a clear trend (Shown in Fig 17), suggesting that many failures occur at the inverse kinematics verifier stage, thus avoiding the more computationally expensive motion planner.

As the logic layer requires exponentially more time to find a solution with an increasing number of branches, and the physical layer experiences more failures as more semantic solutions are generated, the best path forward to improve the system includes:

- Optimising or enhancing the logic layer search algorithm to mitigate the exponential increase in search time.
- Improving feedback strategies from physical layer failures to provide more detailed information to the logic layer, enabling smarter trimming of the search tree. For example, when a state is trimmed from the search tree, if an equivalent state exists as the result of a different path, the trimmed branch could be appended to that state. This would prevent the loss of potentially valid paths along the trimmed branch.

6.2 Comparison to existing work

Solutions generate by the system implemented in this project are more efficient than solutions proposed by the melt-and-grow algorithm [25] used previously, however the melt-and-grow algorithm can handle far more modules.

The system used as a primary source of inspiration [27] tested for a different set of performance metrics, so is hard to compare against. However, when conducting the 5-module test also conducted in the study we received much faster results as seen in Fig 18. The timing comparison is slightly unfair as our system does not implement an advanced motion planner or generate instructions for mobile manipulator walking. Therefore, only the logic layer timing can be fairly compared which is highly in favour of our system, likely due to more time put towards optimisation and different overall project goals.

5 Module Test	**Our results**	**Previous System Results**
Logic Layer Time	0.008 s	0.21 s
Physical Layer Time	0.147 s	46.89 (±20.6) s
Total Time	0.154 s	47.10 (±20.6) s

Figure 18: Our results compared to the system developed in this paper [27] for a 5-Module test (Further test information in Appendix B - 5 Module Test)

6.3 Implications

At present, the developed reconfiguration plan serves as a proof of concept and gives a modular base for further development to improve on. As the system is comprised of multiple parts working separately, its possible for multiple later projects to develop the system in parallel, focusing on developing separate portions of the overall system increasing development time of a more capable system.

It was suggested during project demonstration to industry professionals that at its current state, the system does not consider enough factors to reliably operate unsupervised. Though could be implemented in the manufacturing or construction industry shortly under supervision to for example, stack and track containers in a warehouse. The production of a less capable system for industry could be key to raising the funds required for further development and eventual adoption in the space industry.

7 Planning and Time Management

7.1 Project Management Procedures

To streamline the design and development of the project, it followed a traditional engineering product development cycle consisting of 5 phases:

- **Initiation:** The definition of the problem and the projects goals, requirements, and risks. This phase was completed by the given description of the project and further questioning of the project supervisor.
- **Planning:** The definition of how to solve the problem by outlining the details and goals to meet the defined requirements. This phase was completed by the production of the initial report seen in Appendix J - Initial Report, the project plan seen in Figure 19, and a conceptual high-level product design seen in Figure 7.
- **Execution:** The working phase where the plan designed in the previous phase is put into action and the product is developed. This was completed according to the created project plan and was finished in its majority by the project demonstration day on the 29th of April 2024.
- **Controlling & Monitoring:** This phase runs alongside the execution phase and involves tracking progress and adjusting the workflow to remove potential roadblocks.
- **Closure:** Reflecting on the progress and results to officially end the project. This phase is conducted through analysis of project results, documentation of completed work and reflection of project success which is represented by this document.

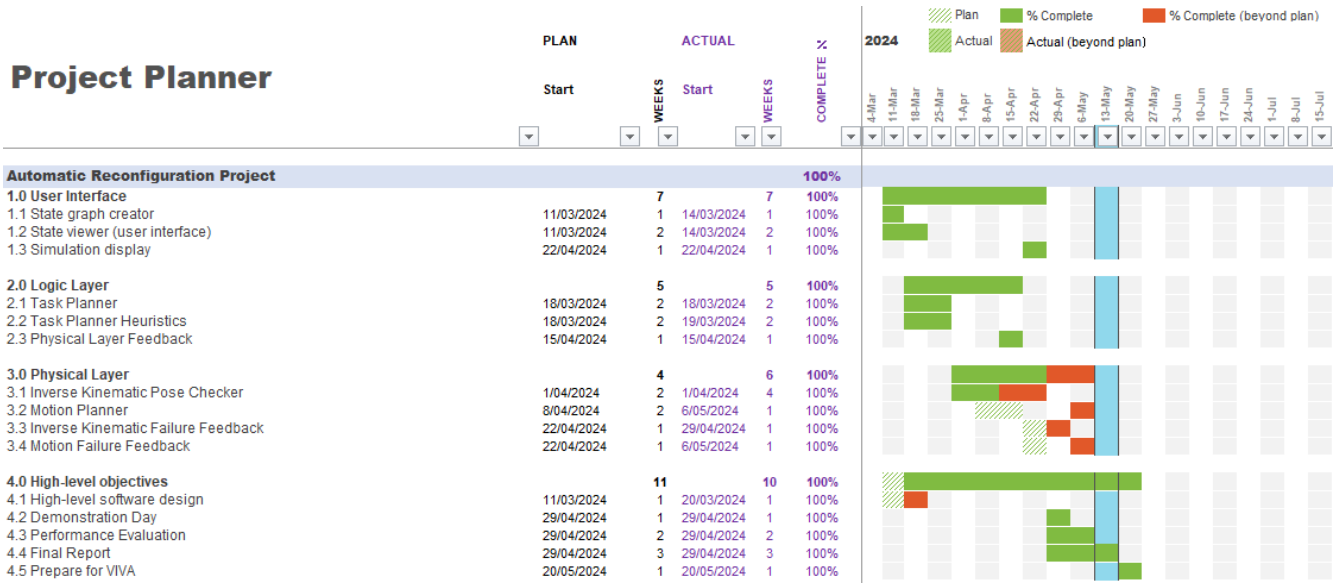

Figure 19: Project Plan after project completion. Task descriptions can be seen in Appendix J - Initial Report

7.2 Project Management Reflection

The project went according to plan through to the development of the physical layer. Due to unfamiliarity with robot kinematics, little in-depth design was created in the planning phase of the project with the assumption that with the knowledge of what each section of the physical layer needed to accomplish, figuring out how to accomplish it would not be a notable obstacle. This led to the physical layers' development taking far longer than expected, over-running its planned development time by a week despite completing the logic layer earlier than expected. Due to over-running the deadline, the project goal was instead completed by defining a simple physical layer rule to use for feedback such as "is the module at the top of the stack and hence, can be picked up in an environment with gravity by a stationary arm"; allowing the remainder of the project to be completed, making it possible to develop and analyse feedback strategies without sacrificing time to develop a mostly unused and complex simulation.

Despite the delay, the final product does match what was planned at the beginning of the project, and as such the goals of the project have been filled. This can be attributed to appropriate levels of slack in task timing guidelines and creatively making use of out-of-the-box implementations to decrease production time drastically and reduce complexity.

7.3 Risk Assessment

This section builds, and expands, on material previously included in the project Initial Report (see Appendix J - Initial Report).

ID	Risk Description	Impact	Risk Mitigation	Effectiveness
1	Missing or corrupted documents	High	Documents are backed up to a GitHub repository	Highly effective
2	Ambitions for project are too great for the project time limit	High	Setting appropriate scope expectations from the beginning of the project	Slightly effective: did not properly take into consideration prior knowledge
3	Illness or work unavailability	Medium	Record illness and provide proper explanation for missing work in final report. Decrease scope to provide meaningful results	Highly effective: Illness affected several weeks of the project; However, scope was reduced appropriately
4	Losing test results	Medium	Produce lab reports to document progress	Highly effective

Figure 20: Project Risk Register

7.4 Evolution of Project Plan

The project plan saw little modification over the project. During progress reviews during the project, if it was seen that a section of the project would overrun its deadline, alternative methods of reaching a functional overall system were found that involved sacrificing small features such as including module orientation and module connectivity direction. These features were still considered in the completed implementation allowing them to be designed and implemented with relative ease when the project is further developed in the future.

8 Conclusion

This paper details the development of a hybrid reconfiguration planning system through the completion of the following sub-objectives:

1. Development of a task planner to create high-level semantic solutions to state reconfiguration.
2. Development of a motion planner to impose robot capabilities, geometry, and physical restraints on the semantic solution, and implementation of feedback strategies to discard infeasible solutions and continue the search.
3. Development of state and state reconfiguration plan visualisation functions to create videos of reconfiguration simulations.
4. Conducting testing of the system through various inputs to analyse performance. We demonstrate that the system can produce efficient solutions and potentially can be integrated with the robot arm in the lab to complete sub-objective five. Though the implementation of enhanced feedback strategies is needed to improve solution generation time. The base high-level plan for the system has great potential for further development.

We demonstrate that the system can produce efficient solutions and potentially can be integrated with the robot arm in the lab to complete sub-objective 5. Though the implementation of enhanced feedback strategies is needed to improve generation time. The base high-level plan for the system has great potential for further development.

9 Further Work

This base system provides many opportunities for future work to enhance its capabilities. From minor changes such as support for the movement of multiple modules at once, multiple mobile manipulators or introducing module orientation and connection direction; To major changes like introducing a failure memory to predict physical layer failures, a purpose-built implementation utilising parallel programming, or the modification of the program to work in real-time so it can compensate for a non-stationary environment. To further identify areas of improvement, the next suggested development of the project would be the creation of a function to generate random goal states from a starting state, to be input into the system. This would allow the automation and conduction of mass testing to develop a data set for analysis.

10 References

[1] MOSAR. "About the project - mosar." Accessed 12-12-2023. (2020), [Online]. Available: https://www.h2020-mosar.eu/about/.

[2] MOSAR, "Detailed design document," 2020, Accessed 13-12-2023. [Online]. Available: https://www.h2020-mosar.eu/wp-content/uploads/2021/08/MOSAR-WP3-D3.6-SA_1.2.0-Detailed-Design-Document.pdf.

[3] J. P. Laboratory, "Topex/poseidon fact sheet," Accessed 03-02-2024. [Online]. Available: https://sealevel.jpl.nasa.gov/missions/topex-poseidon/summary/.

[4] J. Esper, "Modular, adaptive, reconfigurable systems: Technology for sustainable, reliable, effective, and affordable space exploration," 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:2408860.

[5] R. O. Bartlett, "Nasa standard multimission modular spacecraft for future space exploration," in *Proceedings of the Goddard Memorial Symposium, Washington, DC, United State, March 8–10, 1978*, ser. AAS PAPER 78-043, NASA, Goddard Space Flight Center Greenbelt, Md., United States, 1978.

[6] C. E. Trevathan, T. D. Taylor, R. G. Hartenstein, A. C. Merwarth, and W. N. Stewart, "Development and application of nasa's first standard spacecraft computer," *Commun. ACM*, vol. 27, no. 9, 902–913, 1984, ISSN: 0001-0782. DOI: 10.1145/358234.358252. [Online]. Available: https://doi.org/10.1145/358234.358252.

[7] E Hupp and E Moreaux, "Nasa's topex/poseidon oceanography mission ends," *Press Release*, pp. 06–001, 2006.

[8] B. Hine, S. Spremo, M. Turner, and R. Caffrey, "The lunar atmosphere and dust environment explorer mission," in *2010 IEEE Aerospace Conference*, 2010, pp. 1–9. DOI: 10.1109/AERO.2010.5446989.

[9] S. Tietz, J. H. Bell, and B. Hine, "Multi-mission suitability of the nasa ames modular common bus," in *AIAA and Utah State University 23rd Annual Conference*, 2009.

[10] W. M. NASA, "Common spacecraft bus for lunar explorer missions," 2008, Accessed 10-02-2024. [Online]. Available: https://lunarscience.arc.nasa.gov/articles/common-spacecraft-bus-for-lunar-explorer-missions/.

[11] R. H. Dwayne Brown, "Nasa lunar mission wins 2014 popular mechanics breakthrough award," 2014, Accessed 10-02-2024. [Online]. Available: https://www.nasa.gov/news-release/nasa-lunar-mission-wins-2014-popular-mechanics-breakthrough-award/.

[12] M. V. D'Ortenzio, J. L. Bresina, A. R. Crocker, *et al.*, "Operating ladee: Mission architecture, challenges, anomalies, and successes," in *2015 IEEE Aerospace Conference*, 2015, pp. 1–23. DOI: 10.1109/AERO.2015.7118961.

[13] NASA, *The station pictured from the spacex crew dragon jsc2021e064215_alt*, 2021. [Online]. Available: https://images.nasa.gov/details-jsc2021e064215_alt.

[14] S. K. Shaevich, "Results of five-years exploitation of the first iss element-fgb" zarya" module," in *54th International Astronautical Congress of the International Astronautical Federation, the International Academy of Astronautics, and the International Institute of Space Law*, 2003, T–1.

[15] E. Mahoney, "Nasa provides updated international space station transition plan," Accessed 12-12-2023, NASA, 2022. [Online]. Available: https://www.nasa.gov/humans-in-space/nasa-provides-updated-international-space-station-transition-plan/.

[16] M. A. Post and J. Austin, "Knowledge-based self-reconfiguration and self-aware demonstration for modular satellite assembly," in *10th International Workshop on Satellite Constellations & Formation Flying 2019*, York, 2019.

[17] A. Deshpande, L. P. Kaelbling, and T. Lozano-Pérez, "Decidability of semi-holonomic prehensile task and motion planning," in *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, K. Goldberg, P. Abbeel, K. Bekris, and L. Miller, Eds. Cham: Springer International Publishing, 2020, pp. 544–559, ISBN: 978-3-030-43089-4. DOI: 10.1007/978-3-030-43089-4_35. [Online]. Available: https://doi.org/10.1007/978-3-030-43089-4_35.

[18] Y. Huang, *Algorithmic planning for robotic assembly of building structures*, 2022. [Online]. Available: https://hdl.handle.net/1721.1/148285.

[19] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[20] S. Vassilvitskii, M. Yim, and J. Suh, "A complete, local and parallel reconfiguration algorithm for cube style modular robots," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, vol. 1, 2002, 117–122 vol.1. DOI: 10.1109/ROBOT.2002.1013348.

[21] D. Saldaña, B. Gabrich, M. Whitzer, *et al.*, "A decentralized algorithm for assembling structures with modular robots," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 2736–2743. DOI: 10.1109/IROS.2017.8206101.

[22] J. Paulos, N. Eckenstein, T. Tosun, *et al.*, "Automated self-assembly of large maritime structures by a team of robotic boats," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 3, pp. 958–968, 2015. DOI: 10.1109/TASE.2015.2416678.

[23] iBOSS. "Iboss background." Accessed 25-02-2024. (2023), [Online]. Available: https://www.iboss.space/background/.

[24] M. Kortmann, T. Schervan, H. Schmidt, S. Ruehl, J. Weise, and J. Kreisel, "Building block-based "iboss" approach: Fully modular systems with standard interface to enhance future satellites," Sep. 2015.

[25] Y. Zhang, W. Wang, J. Sun, H. Chang, and P. Huang, "A self-reconfiguration planning strategy for cellular satellites," *IEEE Access*, vol. 7, pp. 4516–4528, 2019. DOI: 10.1109/ACCESS.2018.2888588.

[26] D. Rus and M. Vona, "Self-reconfiguration planning with compressible unit modules," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 4, 1999, 2513–2520 vol.4. DOI: 10.1109/ROBOT.1999.773975.

[27] I. Rodríguez, A. S. Bauer, K. Nottensteiner, D. Leidner, G. Grunwald, and M. A. Roa, "Autonomous robot planning system for in-space assembly of reconfigurable structures," in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–17. DOI: 10.1109/AERO50100.2021.9438257.

[28] L. Brunner *et al.*, *Eva python sdk*, https://github.com/automata-tech/eva_python_sdk, 2019.

[29] Automata, "Eva technical specifications," 2019, Accessed 20-12-2023. [Online]. Available: https://automata.tech/wp-content/uploads/2019/09/Spec-Sheet-September2019.pdf.

[30] A. Sears-Collins. "The ultimate guide to inverse kinematics for 6dof robot arms." Accessed 20-04-2024. (2020), [Online]. Available: https://automaticaddison.com/the-ultimate-guide-to-inverse-kinematics-for-6dof-robot-arms/.

[31] A. Sears-Collins. "How to assign denavit-hartenberg frames to robotic arms." Accessed 20-04-2024. (2020), [Online]. Available: https://automaticaddison.com/how-to-assign-denavit-hartenberg-frames-to-robotic-arms/.

[32] P. Lehner and A. Albu-Schäffer, "The repetition roadmap for repetitive constrained motion planning," *IEEE Robotics and Automation Letters*, vol. 3, no. 4, pp. 3884–3891, 2018. DOI: 10.1109/LRA.2018.2856925.

[33] A. Lin, "Binary search algorithm," *WikiJournal of Science*, vol. 2, no. 1, pp. 1–13, 2019.

[34] S. van der Walt, J. L. Schönberger, J. Nunez-Iglesias, *et al.*, "Scikit-image: Image processing in Python," *PeerJ*, vol. 2, e453, Jun. 2014, ISSN: 2167-8359. DOI: 10.7717/peerj.453. [Online]. Available: https://doi.org/10.7717/peerj.453.

[35] J. D. Hunter, "Matplotlib: A 2d graphics environment," *Computing in Science & Engineering*, vol. 9, no. 3, pp. 90–95, 2007. DOI: 10.1109/MCSE.2007.55.

[36] P. Manceron, *Ikpy*, version v3.3.3, If you use this software, please cite it using the metadata from this file., May 2022. DOI: 10.5281/zenodo.6551158. [Online]. Available: https://doi.org/10.5281/zenodo.6551158.

[37] MathWorks. "Urdf primer." Accessed 10-04-2024. (2024), [Online]. Available: https://uk.mathworks.com/help/sm/ug/urdf-model-import.html.

[38] G. Johnson *et al.*, *Urdf-loaders*, https://github.com/gkjohnson/urdf-loaders, 2018.

[39] G. Van Rossum, *The Python Library Reference, release 3.12.3*. Python Software Foundation, 2024.

Appendix A - System Input/Output

A.1 System Inputs

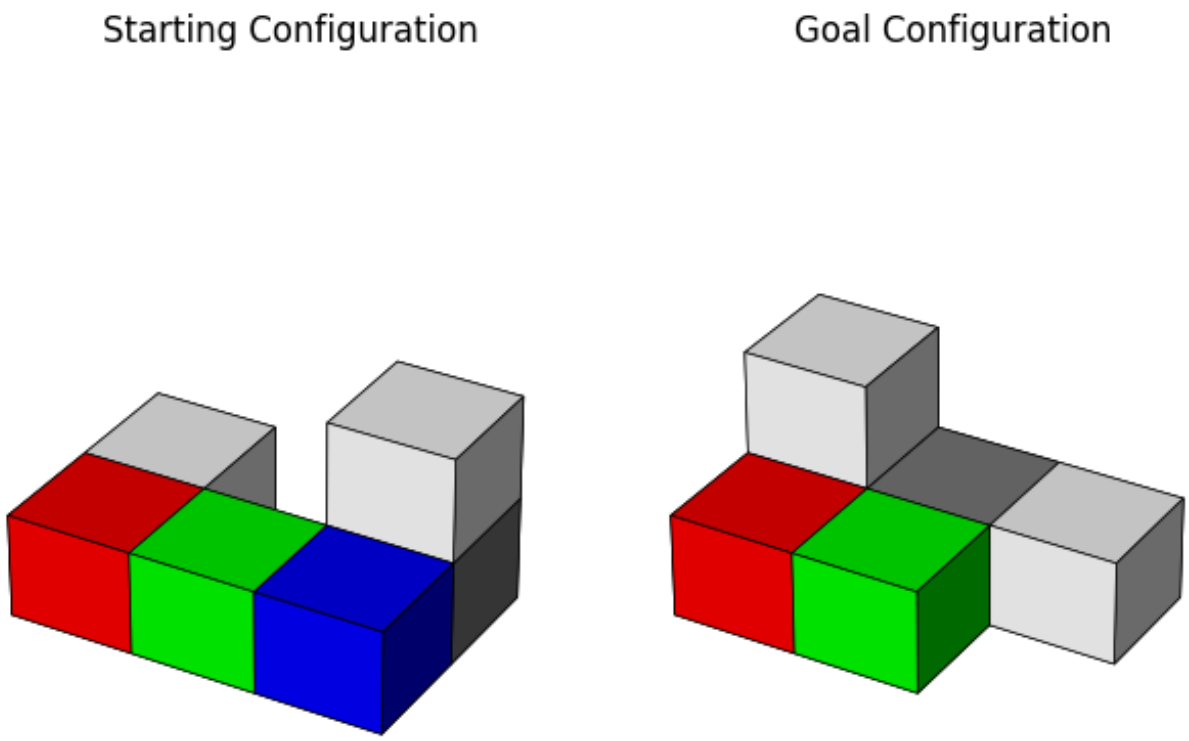

Figure 21: 6-Module Starting and Goal configurations for system input

A.2 System Output

```
Instruction Set
['START']
['MOVE_TO', (0.05, 1.05, 0.1), (0.0, -0.048, -1.664, -0.190, -3.169, -1.253, 0.0)]
['CONNECT']
['MOVE_TO', (1.05, 0.05, 1.1), (0.0, -1.52, -0.770, -0.190, 3.568, -2.070, 0.0)]
['DISCONNECT']
['MOVE_TO', (2.05, 1.05, 1.1), (0.0, -1.097, -1.171, -0.190, -2.075, -1.560, 0.0)]
['CONNECT']
['MOVE_TO', (0.05, 0.05, 1.1), (0.0, -0.786, 0.230, -0.580, -3.527, -2.705, 0.0)]
['DISCONNECT']
['MOVE_TO', (2.05, 1.05, 0.1), (0.0, -1.097, -1.615, -0.190, -0.598, -1.200, 0.0)]
['CONNECT']
['MOVE_TO', (1.05, 1.05, 0.1), (0.0, -0.785, -1.637, -0.190, -1.336, -1.242, 0.0)]
['DISCONNECT']
['MOVE_TO', (2.05, 0.05, 0.1), (0.0, -1.546, -1.620, -0.190, -5.366, -1.214, 0.0)]
['CONNECT']
['MOVE_TO', (0.05, 1.05, 0.1), (0.0, -0.048, -1.664, -0.190, -3.169, -1.253, 0.0)]
['DISCONNECT']
['MOVE_TO', (1.05, 0.05, 1.1), (0.0, -1.523, -0.769, -0.190, 3.568, -2.070, 0.0)]
['CONNECT']
['MOVE_TO', (2.05, 1.05, 0.1), (0.0, -1.097, -1.615, -0.190, -0.598, -1.200, 0.0)]
['DISCONNECT']
['MOVE_TO', (0.05, 0.05, 1.1), (0.0, -0.786, 0.230, -0.580, -3.527, -2.705, 0.0)]
['CONNECT']
['MOVE_TO', (0.05, 1.05, 1.1), (0.0, -0.0476, -0.769, -0.190, -4.074, -2.070, 0.0)]
['DISCONNECT']
['END']
```

Appendix B - 5 Module Test

B.1 System Inputs

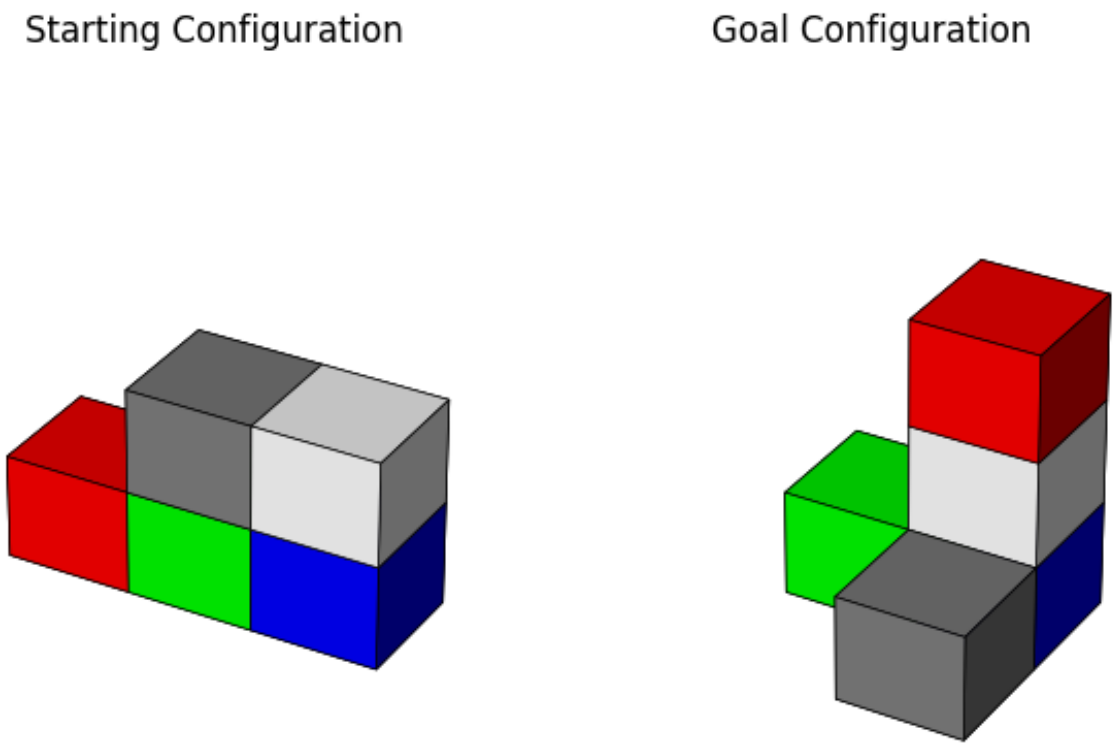

Figure 22: 5-Module Starting and Goal configurations for system input

B.2 System Output

```
----- Solution Found -----
COUNT: Branches           - 5
COUNT: moves              - 2
COUNT: Failures           - 0
TIME: Logic Layer time    - 0.008 s
TIME: Physical Layer time - 0.147 s
```

Appendix C - Test Results

Input		Without physical Constraints				With physical Constraints				
# of modules	branch factor	Number of Moves	Logic Layer Time (s)	Physical Layer Time (s)	Total Time (s)	Number of Moves	Failures	Logic Layer Time (s)	Physical Layer Time (s)	Total Time (s)
4	1	4	0.006001949	0.246054411	**0.252**	4	0	0.006002188	0.419092655	**0.425**
	2	4	0.024004936	0.241560459	**0.266**	4	0	0.024005413	0.417101145	**0.441**
	3	4	0.074015856	0.254906416	**0.329**	4	0	0.076019526	0.420815468	**0.497**
	4	4	0.183040142	0.245558977	**0.429**	4	0	0.176039457	0.421093702	**0.597**
	5	4	0.352078438	0.247054815	**0.599**	4	0	0.356149197	0.414094687	**0.770**
	6	4	0.608268499	0.244753838	**0.853**	4	0	0.5996387	0.418095112	**1.018**
5	1				DNF					DNF
	2				DNF					DNF
	3	4	0.154034615	0.175568581	**0.330**	4	0	0.157363892	0.304342747	**0.462**
	4	4	0.37908411	0.176585197	**0.556**	4	0	0.37908411	0.298068762	**0.677**
	5	4	0.752259493	0.176039219	**0.928**	4	0	0.74557209	0.30306673	**1.049**
	6	4	1.42871666	0.175089359	**1.604**	4	0	1.351318121	0.305649519	**1.657**
6	1	7	0.013002634	0.320630789	**0.334**					DNF
	2	6	0.214049816	0.273619652	**0.488**					DNF
	3	6	1.507008791	0.271930695	**1.779**	6	6	2.26717782	2.140664816	**4.408**
	4	5	3.056791782	0.225049734	**3.282**	6	9	9.87413168	3.179346561	**13.053**
	5	5	8.149960041	0.225049496	**8.375**	6	9	32.07893133	6.09358263	**38.173**
	6	5	17.59457445	0.228050709	**17.823**	6	9	82.75698733	31.08932686	**113.846**
7	1	3	0.005001545	0.117025375	**0.122**					DNF
	2	3	0.012003183	0.118026495	**0.130**					DNF
	3	3	0.028006315	0.117026329	**0.145**	4	6	0.154044151	1.227118969	**1.381**
	4	3	0.050011158	0.117026329	**0.167**	4	8	0.410094738	1.539860725	**1.950**
	5	3	0.076017618	0.118025541	**0.194**	4	10	0.853647947	1.881162643	**2.735**
	6	3	0.114040852	0.123012781	**0.237**	4	12	1.53081131	2.214050055	**3.745**
8	1	5	0.019365549	0.190842628	**0.210**					DNF
	2	5	0.094592333	0.190091848	**0.285**					DNF
	3	5	0.446454287	0.190597534	**0.637**					DNF
	4	5	1.357872963	0.194052696	**1.552**					DNF
	5	5	3.452106237	0.192105293	**3.644**	6	43	44.63452435	21.28689671	**65.921**
	6	5	7.406012297	0.191615105	**7.598**	6	79	113.3496633	142.8667283	**256.216**
9	1	5	0.020680904	0.189540625	**0.210**					DNF
	2	5	0.10995841	0.190983295	**0.714**					DNF
	3	5	0.522100687	0.192361832	**1.827**					DNF
	4	5	1.634381056	0.19306016	**4.322**					DNF
	5	5	4.12870121	0.193561554	**8.668**					DNF
	6	5	8.478348732	0.189818144	**8.668**	6	81	138.9033661	145.4561296	**284.359**

Figure 23: System test results

Appendix D - Automata EVA Technical Specifications

EVA
Technical Specifications

Automata

Eva Desktop Robot Arm

Degrees of Freedom: 6
Repeatability: ± 0.5 mm
Max Payload: 1.25 kg
Reach: 600 mm
Installation Position: Upright
Weight: 9.5 kg
Joint velocity limits: 120° / second
Toolpath speed limit: 750 mm / second
Power: 24 VDC @ 11.67A
Power Consumption: 280 W Peak
Footprint: 160 x 160 mm

Electronic Interfaces

4X: Analog Inputs
2X: Analog Outputs
6X: Digital Inputs
6X: Digital Outputs
1X: E-Stop inputs
1X: Ethernet
1X: Wifi Card
Tool Power: 24 VDC @ 1 A
Base I/O Power: 24 VDC @ 1.5 A

Joint Position Limits

A1: ±179°
A2: -155° / 70°
A3: 160° / 45°
A4: ±179°
A5: 5° / 10°
A6: ±179°

Operating Environment

Ingress Rating: IP20
Temperature: 5 - 40 °C
Cabling: Max 3 m for power, e-stop, tools
Max Humidity @ 40°C: 50%
Max Humidity @ 20°C: 90%

Programming

Type: Remote REST API
Interface: GUI through web browser
Communication: Ethernet or wifi
Deployment: Local (installed on robot)
Export/Import: Base64 / Javascript

Any questions?

Visit our website for more technical information

www.automata.tech

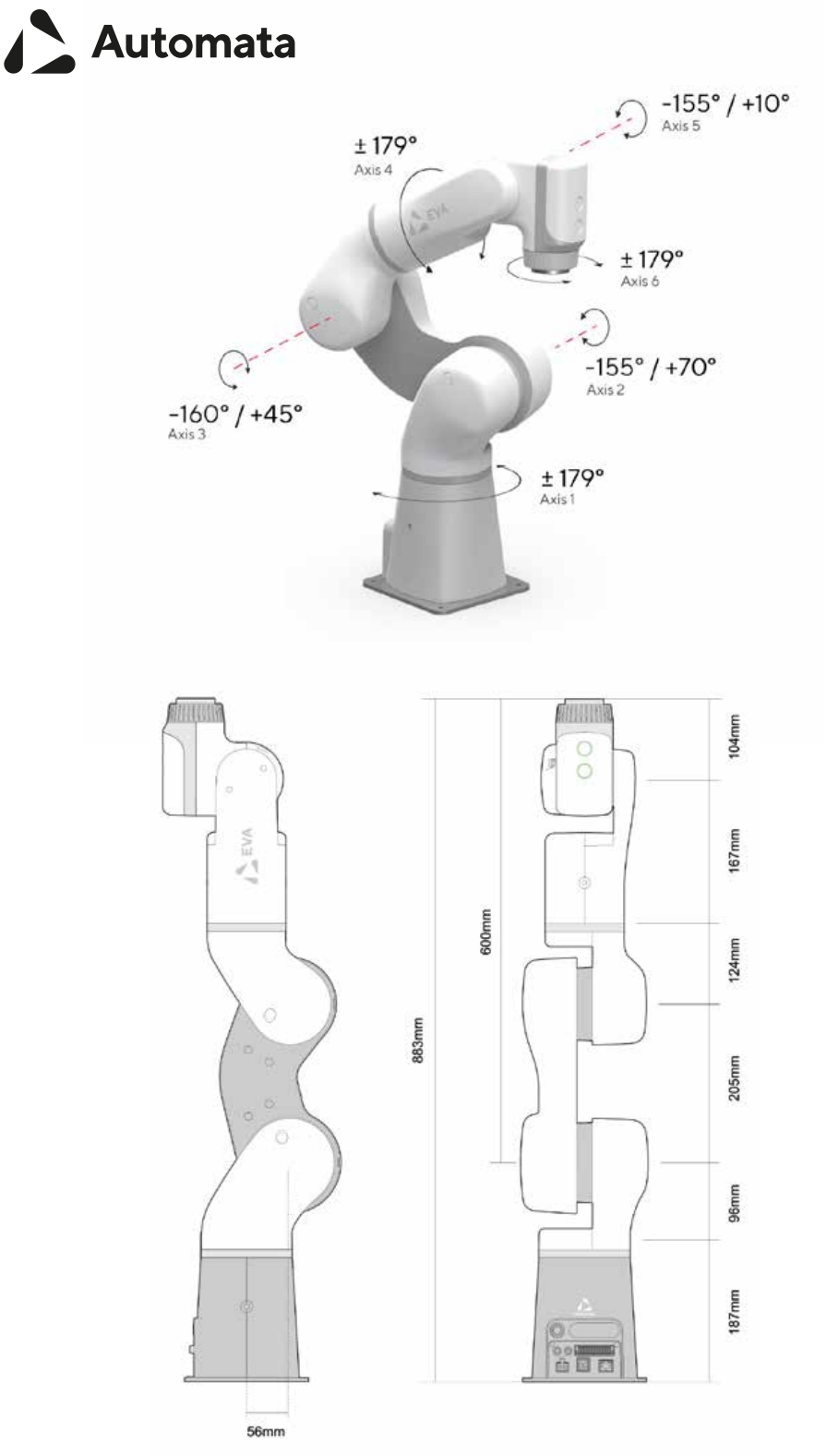

Figure 24: EVA Technical Specification [29]

Appendix E - Automata EVA URDF File

```
<robot name="robot_arm">
        <link name="base_link">
                <visual>
                        <origin xyz="0 0 0.0935" rpy="0 0 0"/>
                        <geometry>
                                <cylinder length="0.187" radius="0.056"/>
                        </geometry>
                </visual>
        </link>

        <link name="link_1">
                <visual>
                        <geometry>
                                <box size="0.112 0.112 0.096" />
                        </geometry>
                </visual>
        </link>

        <link name="link_2">
                <visual>
                <origin xyz="0 0 0.1025" rpy="0 0 0"/>
                        <geometry>
                                <cylinder length="0.205" radius="0.056"/>
                        </geometry>
                </visual>
        </link>

        <link name="link_3">
                <visual>
                        <origin xyz="0 -0.056 0.062" rpy="0 0 0" />
                        <geometry>
                                <box size="0.112 0.112 0.124" />
                        </geometry>
                </visual>
        </link>

        <link name="link_4">
                <visual>
                        <origin xyz="0 0 0.0835" rpy="0 0 0" />
                        <geometry>
                                <cylinder length="0.167" radius="0.0175"/>
                        </geometry>
                </visual>
        </link>

        <link name="link_5">
                <visual>
                        <origin xyz="0 -0.052 0.052" rpy="0 0 0" />
                        <geometry>
                                <box size="0.052 0.104 0.104" />
                        </geometry>
                </visual>
        </link>

```

```
55          <link name="end_effector">
56                  <visual>
57                          <origin xyz="0 0 0.0000005" rpy="0 0 0" />
58                          <geometry>
59                                  <cylinder length="0.000001" radius="0.05"/>
60                          </geometry>
61                  </visual>
62          </link>
63
64          <joint name="joint_1" type="revolute">
65                  <parent link="base_link"/>
66                  <child link="link_1"/>
67                  <origin xyz="0 0 0.235" rpy="0 0 0" />
68                  <axis xyz="0 0 1"/>
69          </joint>
70
71          <joint name="joint_2" type="revolute">
72                  <parent link="link_1"/>
73                  <child link="link_2"/>
74                  <origin xyz="0 0.056 0.048" rpy="0 0 0" />
75                  <axis xyz="1 0 0"/>
76                  <limit lower="-2.70526" upper="1.22173" />
77          </joint>
78
79          <joint name="joint_3" type="revolute">
80                  <parent link="link_2"/>
81                  <child link="link_3"/>
82                  <origin xyz="0 0 0.205" rpy="0 0 0" />
83                  <axis xyz="1 0 0"/>
84                  <limit lower="-2.79253" upper="0.785398" />
85          </joint>
86
87          <joint name="joint_4" type="revolute">
88                  <parent link="link_3"/>
89                  <child link="link_4"/>
90                  <origin xyz="0 -0.056 0.124" rpy="0 0 0" />
91                  <axis xyz="0 0 1"/>
92          </joint>
93
94          <joint name="joint_5" type="revolute">
95                  <parent link="link_4"/>
96                  <child link="link_5"/>
97                  <origin xyz="0 0 0.167" rpy="0 0 0" />
98                  <axis xyz="1 0 0"/>
99                  <limit lower="-2.70526" upper="0.174533" />
100         </joint>
101
102         <joint name="joint_6" type="revolute">
103                 <parent link="link_5"/>
104                 <child link="end_effector"/>
105                 <origin xyz="0 -0.104 0.104" rpy="0 0 0" />
106                 <axis xyz="0 0 1"/>
107                 <limit lower="-2.88" upper="2.1" />
108         </joint>
109 </robot>
```

Appendix F - TAMP code __main__.py

```
import time
import numpy as np
from matplotlib import pyplot as plt
from matplotlib.animation import FuncAnimation
from mpl_toolkits.mplot3d import Axes3D

from logic_layer.task_planner import Module, State, find_path, trim_state
from physical_layer.physical_layer import verify_pose, motion_planner
from test import *

# This program implements a simple Task and Motion Planner (TAMP) for the reconfiguration
# of modular block from a starting configuration to a goal configuration, under the physical
# constraints presented by a mobile manipulator rearranging modules on a desk


VIDEO_SAVE_LOCATION = __file__+"/../videos/"
VIDEO_FRAME_INTERVAL = 1000

STATIONARY_ARM_POSITION = [0.1, -0.3, 0]
MAX_PHYSICAL_LAYER_FAILURES = 500


# Create and save a video of a transition path
def create_reconfiguration_video(transition_path, label = "temp"):
    # find required frame size
    size = 0
    for state in transition_path:
        if state.size() > size:
            size = state.size()

    # Create figure
    fig = plt.figure(1)
    ax = plt.axes(111, projection = '3d')
    ax.set_title(label)
    ax.axis("off")

    # Create frames
    num_frames = len(transition_path)
    def update(frame):
        # Reset axis
        ax.clear()
        ax.set_title("Reconfiguration Video")
        ax.axis("off")

        # set the colors of each object
        current_state = transition_path[frame]
        colour_mask = current_state.create_colour_mask(size)

        container = ax.voxels(colour_mask,
                              facecolors = colour_mask,
                              edgecolors = 'k',
                              linewidth = 0.5)

        return container
```

```
55      # Save Video
56      ani = FuncAnimation(fig = fig, func = update,
57                          frames = num_frames, interval = VIDEO_FRAME_INTERVAL)
58      ani.save(filename = VIDEO_SAVE_LOCATION + label + "/video.html", writer = "html")
59
60
61  # Verify each start/end movement position in a transition path is reachable
62  def verify_inverse_kinematics(transition_path):
63      for state in transition_path:
64          # if not start state
65          if (state.birth_movement != 0):
66              # get start/end positions in metres
67              movement_decimetres = state.birth_movement
68              movement_metres = np.divide(movement_decimetres, 10) # Convert to metres
69
70              for pos in movement_metres:
71                  # verify arm can reach position
72                  if not verify_pose(STATIONARY_ARM_POSITION, pos):
73                      return state
74      return 0
75
76
77  # Generate a transition plan to reconfigure s_start into s_goal
78  def run_tamp_planner(s_start, s_goal):
79      logic_time = 0
80      physical_time = 0
81
82      # Find semantic solution
83      search_tree = None
84      failure_count = 0
85      while True:
86          ##### Logic Layer #####
87          t0 = time.time()
88          transition_path, search_tree = find_path(s_start, s_goal, search_tree)
89          logic_time += time.time() - t0
90
91          # If logic layer failed, no solution
92          if (transition_path == 0):
93              print_invalid_sol(s_start.get_num_modules(), logic_time, physical_time)
94              return 0, 0
95
96          ##### Physical Layer #####
97          t1 = time.time()
98
99          # verify inverse kinematics
100         failed_state = verify_inverse_kinematics(transition_path)
101
102         # If IK failed, remove failed state and continue search
103         if (failed_state != 0):
104             if failure_count > MAX_PHYSICAL_LAYER_FAILURES:
105                 print_invalid_sol(s_start.get_num_modules(), logic_time, physical_time)
106                 return 0, 0
107             trim_state(failed_state, search_tree)
108             failure_count += 1
109             physical_time += time.time() - t1
110             continue
```

```
111
112         # create motion plan
113         instruction_set, failed_state = motion_planner(transition_path)
114
115         # If motion plan failed, remove failed state and continue search
116         if (failed_state != 0):
117             if failure_count > MAX_PHYSICAL_LAYER_FAILURES:
118                 print_invalid_sol(s_start.get_num_modules(), logic_time, physical_time)
119                 return 0, 0
120             trim_state(failed_state, search_tree)
121             failure_count += 1
122             physical_time += time.time() - t1
123             continue
124         else:
125             # If motion plan successful
126             physical_time += time.time() - t1
127             break
128
129     print_valid_sol(str(len(transition_path) - 1), failure_count, logic_time, physical_time)
130     return instruction_set, transition_path
131
132
133 # Print Solution Found
134 def print_valid_sol(moves, failure_count, logic_t, physical_t):
135         print("----- Solution Found -----")
136         print("COUNT: moves              -", moves)
137         print("COUNT: Failures           -", failure_count)
138         print("TIME: Logic Layer time    -", logic_t, "s")
139         print("TIME: Physical Layer time -", physical_t, "s")
140
141
142 # Print No Solution Found
143 def print_invalid_sol(num_modules, logic_time, physical_time):
144         print("----- NO Solution Found -----")
145         print("COUNT: Modules            -", num_modules)
146         print("TIME: Logic Layer time    -", logic_time, "s")
147         print("TIME: Physical Layer time -", physical_time, "s")
148
149
150 def main():
151     # Get test configurations and run TAMP program
152     s_start, s_goal, label = six_mod_config()
153     instruction_set, transition_path = run_tamp_planner(s_start, s_goal)
154
155     # Display output instruction set
156     print("Instruction Set")
157     for i in instruction_set:
158         print(i)
159
160     # Save video
161     create_reconfiguration_video(transition_path, label + " - " +
162                                  str(len(transition_path) - 1) + " Moves")
163
164
165 if __name__ == "__main__":
166     main()
```

Appendix G - Logic Layer code task_planner.py

```
import math
import numpy as np
from matplotlib import pyplot as plt
from mpl_toolkits.mplot3d import Axes3D

from .utils import *


# This logic layer file details a simple Task Planner to generate semantic solutions
# for the reconfiguration of a start state to a goal state


MAXIMUM_BRANCHES = 5


# Generate priority queue of state transitions
def generate_states(state, s_goal):
    states_queue = StateQueue(s_goal)

    # Get available positions for modules to move to
    available_positions = state.get_available_positions()

    # Generate movement for each movable module in non_final position
    generated_moveset = state.generate_moves(state.get_non_final_positions(s_goal),
                                             available_positions)
    states_queue.push_multiple(generated_moveset)

    # If no new state generated, allow movement of adjacent final modules
    if states_queue.is_empty():
        adjacent_modules = []
        for module_pos in state.get_non_final_positions(s_goal):
            adjacent_modules = adjacent_modules + state.get_adjacent_modules(module_pos)
        adjacent_modules = np.unique(adjacent_modules, axis=0)

        # Generate movement for each adjacent module to modules in non_final position
        generated_moveset = state.generate_moves(adjacent_modules,
                                                 available_positions)
        states_queue.push_multiple(generated_moveset)

    # If no new state generated, allow movement of all modules
    if states_queue.is_empty():
        # Generate movement for all modules
        generated_moveset = state.generate_moves(state.get_module_positions(), available_positions)
        states_queue.push_multiple(generated_moveset)

    return states_queue


# Create a task plan to reconfigure state s_start into start s_goal
# If a search tree is input, planner continues search through search tree
def find_path(s_start, s_goal, search_tree = None):
    if search_tree == None:
        search_tree = []

```

```
55      # Confirm contents of each state match
56      if not do_contents_match(s_start, s_goal):
57          print("ERROR: Start and Goal state do not have an equal composition of modules")
58          return 0
59
60      # Set the recursion limit according to the input state
61      recursion_limit = set_recursion_limit(s_start)
62      recursion_counter = 1
63      s_current = s_start
64
65      # While goal state not found
66      while not s_current.equals(s_goal):
67          # generate state children
68          priority_queue_new = generate_states(s_current, s_goal)
69
70          # add children to list
71          for i in range(MAXIMUM_BRANCHES):
72              if (priority_queue_new.get_length()):
73                  s_new = priority_queue_new.pop()
74                  s_new.parent = s_current
75                  search_tree.append(s_new)
76
77          # Get next state
78          if (recursion_counter < recursion_limit):
79              s_current = search_tree.pop(0)
80              recursion_counter += 1
81          else:
82              print("MAX RECURSION COUNT REACHED! UNABLE TO FIND PATH.")
83              return 0, 0
84
85      # when goal reached, return transition plan and current search tree
86      return s_current.get_state_path(), search_tree
87
88
89  # Test whether states have equal module compositions
90  def do_contents_match(s_start, s_goal):
91      # Confirm number of modules match
92      if s_start.get_num_modules() != s_goal.get_num_modules():
93          return False
94
95      # Confirm composition of modules match
96      unmatched = np.asarray(s_goal.get_modules())
97      for m1 in np.asarray(s_start.get_modules()):
98          for m2 in unmatched:
99              # remove all module in s_goal from unmatched list
100             if Module.equals(m1, m2):
101                 unmatched = unmatched[unmatched != m2]
102
103     # If unmatched array is empty, states are made up of same modules
104     return not len(unmatched)
105
106
107 # Return a sensible task planner recursion limit for the number of modules in a state
108 # Returns enough recursions for a reconfiguration plan with (1.5 * number of modules) moves
109 def set_recursion_limit(state):
110     return math.pow(MAXIMUM_BRANCHES, len(state.get_modules()) * 1.5)
```

```
111 # Remove a state and all states generated from the removed state from the search tree
112 def trim_state(state, search_tree):
113     trimmed_search_tree = []
114     for s in search_tree:
115         if state not in s.get_state_path():
116             trimmed_search_tree.insert(0, trimmed_search_tree)
117     return trimmed_search_tree
118 
119 
120 # An implementation of a state priority queue
121 # where priority is defined by similarity to a goal state
122 class StateQueue:
123     def __init__(self, s_goal):
124         self.queue = []
125         self.s_goal = s_goal
126 
127 
128     def __str__(self):
129         s = "----- State Queue ----- "
130         s += "\nLength: " + str(self.get_length())
131         count = 1
132         for state in self.queue:
133             s += "\nItem" + str(count) + ":"
134             s += "\n\tFinal modules: "
135             s += str(State.get_num_modules_in_final_position(state, self.s_goal))
136             s += "\n\tFree modules: "
137             s += str(State.get_num_modules_in_free_position(state, self.s_goal))
138             s += "\n\tDistance modules: "
139             s += str(State.get_distance_from_completion(state, self.s_goal))
140             count += 1
141         return s
142 
143 
144     # Return number of states in the queue
145     def get_length(self):
146         return len(self.queue)
147 
148 
149     # Return True if queue is empty, else False
150     def is_empty(self):
151         return len(self.queue) == 0
152 
153     # Remove and return a state from the front of the queue
154     def pop(self):
155         return self.queue.pop(0)
156 
157 
158     # Insert a state into the queue at an index that maintains priority order
159     def push(self, state):
160         self.queue.insert(self.binary_search(state), state)
161 
162 
163     # Insert an array of states to the queue
164     def push_multiple(self, state_array):
165         for state in state_array:
166             self.push(state)
```

```
167     # Find a states position in the queue according to priority order
168     def binary_search(self, state):
169         if len(self.queue) == 0:
170             return 0
171 
172         # Use binary search to find state position in queue
173         high = len(self.queue) - 1
174         low = 0
175 
176         while (low <= high):
177             mid = math.floor((high + low) / 2)
178             comparison_val = self.comparator(state, self.queue[mid])
179             if ( comparison_val == 0):
180                 return mid
181             elif ( comparison_val > 0 ):
182                 # if inserting state higher priority, move search to left of queue
183                 high = mid - 1
184             else:
185                 # if inserting state lower priority, move search to right of queue
186                 low = mid + 1
187         return low
188 
189 
190     # Return
191     # POSITIVE if state_1 priority is HIGHER than state_2
192     # ZERO     if state_1 priority is EQUAL to state_2
193     # NEGATIVE if state_1 priority is LOWER than state_2
194     def comparator(self, state_1, state_2):
195         # if state_1 has higher number of modules in final position, return positive else negative
196         s1_finalist_num = State.get_num_modules_in_final_position(state_1, self.s_goal)
197         s2_finalist_num = State.get_num_modules_in_final_position(state_2, self.s_goal)
198         if (s1_finalist_num != s2_finalist_num):
199             return s1_finalist_num - s2_finalist_num
200 
201         # if equal, return state with highest number of modules
202         # not in final position, but are in free position
203         # if state_1 has higher number of modules not in final pos but in free pos, return positive
204         s1_free_num = State.get_num_modules_in_free_position(state_1, self.s_goal)
205         s2_free_num = State.get_num_modules_in_free_position(state_2, self.s_goal)
206         if (s1_free_num != s2_free_num):
207             return s1_free_num - s2_free_num
208 
209         # if equal finalists and free modules, return state with lowest
210         # sum distance between free modules and final locations
211         # if state_1 distance is lower, return positive
212         s1_dist = State.get_distance_from_completion(state_1, self.s_goal)
213         s2_dist = State.get_distance_from_completion(state_2, self.s_goal)
214         if (s1_dist != s2_dist):
215             return s2_dist - s1_dist
216 
217         # Return equal priority
218         return 0
219 
220 
221 
222 
```

```
# An implementation of a state configuration and associated functions
class State:
    def __init__(self):
        self.parent = 0
        self.birth_movement = 0
        self.modules_dict = dict()

        # values for state queue comparison
        self.comparison_goal_state = None
        self.finalist_val = None
        self.free_val = None
        self.goal_distance = None


    # Get number of modules in the state
    def get_num_modules(self):
        return len(self.get_modules())


    # Return a copy of the current state
    def duplicate(self):
        duplicate = State()
        duplicate.modules_dict = self.modules_dict.copy()
        return duplicate


    # Set a goal state for similarity measurement calculations
    def set_goal(self, goal_state):
        self.comparison_goal_state = goal_state
        self.finalist_val = None
        self.free_val = None
        self.goal_distance = None


    # Reset saved similarity measurements
    def reset_saved_values(self):
        self.finalist_val = None
        self.free_val = None
        self.goal_distance = None


    # Insert a module into the state at position
    def insert(self, position, module):
        position = tuple(position)
        if position not in self.modules_dict:
            self.modules_dict[position] = module
        self.reset_saved_values()


    # Remove module from state from position
    def remove_module(self, position):
        self.reset_saved_values()
        return self.modules_dict.pop(position)
```

```
279     # Move a module from start_pos to end_pos, if movement is valid
280     # Returns a copy of current state with module in new position
281     def move_module(self, start_pos, end_pos):
282         state = self.duplicate()
283         mod = state.remove_module(start_pos)
284         t1 = state.is_connected()
285         state.insert(end_pos, mod)
286         t2 = state.is_connected()
287         return state if (t1 and t2) else 0
288 
289 
290     # Return true if 2 states are equal
291     def equals(self, compared_state):
292         for pos in self.get_module_positions():
293             if ((pos not in compared_state.modules_dict) or
294                 (not Module.equals(self.modules_dict[pos], compared_state.modules_dict[pos]))):
295                 return False
296         return True
297 
298 
299     # Get size of the state (maximum 3D length)
300     def size(self):
301         return np.max(list(self.modules_dict.keys())) + 1
302 
303 
304     # Return a position matrix of the current state
305     def to_position_matrix(self, min_size = 0):
306         positions = list(self.modules_dict.keys())
307         max_val, min_val = max_min(positions)
308 
309         # shift module positions to move negative positions into positive space
310         adjustment_val = abs(min_val) if min_val < 0 else 0
311 
312         # find required size of matrix
313         max_val += 1
314         size = max_val if max_val > min_size else min_size
315         size += adjustment_val
316 
317         # Copy dictionary to position matrix
318         position_matrix = np.zeros((size, size, size), dtype = 'object')
319         for pos in positions:
320             pos_adjusted = increment_tuple(pos, adjustment_val)
321             position_matrix[pos_adjusted] = self.modules_dict[pos]
322         return position_matrix
323 
324 
325     # Create a colour mask for state visualisation
326     def create_colour_mask(self, min_size = 0):
327         colour_mask = self.to_position_matrix(min_size)
328 
329         # Replace each module in matrix with the module colour
330         it = np.nditer(colour_mask, flags=['multi_index', 'refs_ok'])
331         for x in it:
332             if x.item() != 0:
333                 colour_mask[it.multi_index] = colour_mask[it.multi_index].colour
334         return colour_mask
```

```
335     # Create and display a figure of the current state
336     # Argument label is the displayed figure title
337     def display(self, label = ""):
338         # Get colour mask
339         colour_mask = self.create_colour_mask()
340 
341         # plot state
342         plt.figure()
343         ax = plt.axes(projection = '3d')
344         ax.set_title(label)
345         ax.axis("off")
346 
347         # Display mask
348         container = ax.voxels(colour_mask, facecolors = colour_mask,
349                               edgecolors = 'k', linewidth = 0.5)
350 
351         plt.show(block=True)
352         return container
353 
354 
355     # Validate whether all modules in a state are connected
356     # Returns True if all modules are connected, else False
357     def is_connected(self):
358         processed_modules = []
359         discovered_modules = []
360 
361         # Add first module in dictionary to discovered modules
362         discovered_modules.append(list(self.modules_dict.keys())[0])
363 
364         # Process all discovered modules
365         while len(discovered_modules):
366             mod = discovered_modules.pop()
367             # Get neighbours
368             adjacent_modules = self.get_adjacent_modules(mod)
369 
370             # If neighbour has not been discovered yet, add to discovered modules
371             for adj_mod in adjacent_modules:
372                 if (adj_mod not in processed_modules) and (adj_mod not in discovered_modules):
373                     discovered_modules.append(adj_mod)
374             processed_modules.append(mod)
375 
376         # if number of discovered modules matches number of module in the state, return True
377         return len(processed_modules) == len(self.modules_dict)
378 
379 
380     # Verify a module can be connected to
381     # Return true if module has an exposed face, else False
382     def is_module_removable(self, pos):
383         # if an adjacent position doesn't contain a module
384         for i in State.get_adjacent_positions(pos):
385             if i not in self.modules_dict:
386                 # module has free face to connect to
387                 return True
388         return False
389 
390 
```

```
391     # Return state transition history
392     def get_state_path(self):
393         task_plan_list = [self]
394 
395         while task_plan_list[0].parent != 0:
396             task_plan_list.insert(0, task_plan_list[0].parent)
397 
398         return task_plan_list
399 
400 
401     # Return array of modules in the state
402     def get_modules(self):
403         return list(self.modules_dict.values())
404 
405     # Return array of module positions in the state
406     def get_module_positions(self):
407         return list(self.modules_dict.keys())
408 
409 
410     # Get number of modules in the current state that match positions in the goal state
411     def get_num_modules_in_final_position(self, goal_state):
412         # if value already computed, return
413         if self.comparison_goal_state == goal_state and self.finalist_val != None:
414             return self.finalist_val
415 
416         if self.comparison_goal_state != goal_state:
417             self.set_goal(goal_state)
418 
419         count = 0
420 
421         for position in self.get_module_positions():
422             # if position is occupied in both states
423             if position in goal_state.modules_dict:
424                 #if position is occupied with equal modules
425                 if Module.equals(self.modules_dict[position], goal_state.modules_dict[position]):
426                     count += 1
427 
428         self.finalist_val = count
429         return self.finalist_val
430 
431 
432     # Get number of modules in the current state that do not match positions in the goal state
433     def get_non_final_positions(self, goal_state):
434         results = []
435 
436         # if position not occupied or
437         # if modules don't match
438         for position in self.get_module_positions():
439             if position not in goal_state.modules_dict:
440                 results.append(position)
441             else:
442                 if not Module.equals(self.modules_dict[position],
443                                      goal_state.modules_dict[position]):
444                     results.append(position)
445 
446         return results
```

```
    # Get number of modules in the current state that do not match positions in the goal state,
    # and are in positions that are not occupied in the goal state
    def get_num_modules_in_free_position(self, goal_state):
        # if value already computed, return
        if self.comparison_goal_state == goal_state and self.free_val != None:
            return self.free_val

        if self.comparison_goal_state != goal_state:
            self.set_goal(goal_state)

        count = 0
        for position in self.get_module_positions():
            if position not in goal_state.modules_dict:
                count += 1

        self.free_val = count
        return self.free_val


    # Get euclidean distance of all non-final modules to their final positions
    def get_distance_from_completion(self, goal_state):
        # if value already computed, return
        if self.comparison_goal_state == goal_state and self.goal_distance != None:
            return self.goal_distance

        if self.comparison_goal_state != goal_state:
            self.set_goal(goal_state)

        # get modules positions not in final pos
        # get empty final module locations
        non_final_module_positions = self.get_non_final_positions(goal_state)
        empty_final_positions = []

        # if position not occupied or
        # if modules don't match
        for position in goal_state.get_module_positions():
            if not ((position in self.modules_dict) and
                    (Module.equals(goal_state.modules_dict[position],
                                   self.modules_dict[position]))):
                empty_final_positions.append(position)


        # for each module not in final pos, get sum of distances to empty final locations
        dist = 0
        for p1 in non_final_module_positions:
            for p2 in empty_final_positions:
                dist += np.linalg.norm(np.asarray(p1) - np.asarray(p2))

        self.goal_distance = dist
        return self.goal_distance
```

```
503     # Get available positions in the state that can be connected to
504     def get_available_positions(self):
505         available_positions = []
506 
507         # iterate through modules and add all free positions to list
508         for module_pos in self.get_module_positions():
509             for adjacent_pos in State.get_adjacent_positions(module_pos):
510                 if adjacent_pos not in self.modules_dict:
511                     available_positions.append(adjacent_pos)
512 
513         # Remove duplicates and return positions list
514         return np.unique(available_positions, axis=0)
515 
516 
517     # Get adjacent modules to module in position pos
518     def get_adjacent_modules(self, pos):
519         adjacent_modules = []
520         for adjacent_pos in State.get_adjacent_positions(pos):
521             if adjacent_pos in self.modules_dict:
522                 adjacent_modules.append(adjacent_pos)
523         return adjacent_modules
524 
525 
526     # Get adjacent positions to position
527     def get_adjacent_positions(position):
528         # get adjacent positions
529         adjacent_positions = []
530         for i in range(0,3):
531             adjacent_positions.append(increment_tuple_val(position, i, 1))
532             adjacent_positions.append(increment_tuple_val(position, i, -1))
533         return adjacent_positions
534 
535 
536     # Generate a new state for each valid movement of each module in module_positions to
537     # each movement_position, Return generated states as a list
538     def generate_moves(self, module_positions, movement_positions):
539         moveset = []
540         for module_pos in module_positions:
541             # If module has an available connection point to be grabbed by a manipulator
542             if (self.is_module_removable(module_pos)):
543                 # if state valid without module
544                 tmp = self.duplicate()
545                 mod = tmp.remove_module(tuple(module_pos))
546 
547                 if tmp.is_connected():
548                     # For each available position, generate a valid state with the possible movement
549                     for move_pos in movement_positions:
550                         return_state = tmp.duplicate()
551                         return_state.insert(tuple(move_pos), mod)
552                         if return_state.is_connected():
553                             return_state.birth_movement = [module_pos, move_pos]
554                             moveset.append(return_state)
555         return moveset
556 
557 
558 
```

```
559 # An implementation of a basic Module object
560 class Module:
561     id_number = 0
562     module_type = 'basic'
563 
564     # Initialize module with identifying colour
565     def __init__(self, colour):
566         self.colour = colour
567 
568 
569     def __str__(self):
570         s = "Module - " + str(self.id_number) + ":"
571         s += "\n\tColour: " + str(self.colour)
572         s += "\n\tType: " + str(self.module_type)
573         return s
574 
575 
576     # Return true if 2 modules are of equal colour
577     def equals(mod_1, mod_2):
578         return (mod_1 != 0 and
579                 mod_2 != 0 and
580                 mod_1.colour == mod_2.colour)
581 
582     # Set module colour
583     def set_colour(self, colour):
584         self.colour = colour
585 
586 
```

Appendix H - Logic Layer code utils.py

```
1  import numpy as np
2
3
4  # A library of useful utility functions used to increase the speed of basic operations
5
6
7  # Return a position tuple with the value of pos + increment
8  def increment_tuple(pos, increment):
9      return tuple([x + increment for x in pos])
10
11
12 # Return a position tuple with value at index incremented
13 def increment_tuple_val(pos, index, increment):
14     a = list(pos)
15     a[index] += increment
16     return tuple(a)
17
18
19 # Return the maximum and minimum of an array
20 # Will accept nested arrays
21 def max_min(array):
22     min = 0
23     max = 0
24     for n in np.asarray(array).flatten():
25         if n > max:
26             max = n
27         if n < min:
28             min = n
29     return max, min
```

Appendix I - Physical Layer code motion_planner.py

```python
import ikpy.chain as ik
import ikpy.utils.plot as plot_utils
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from logic_layer.task_planner import State


# This physical layer file details a simple inverse kinematics verifier and motion planner for the
# verification of Logic Layer semantic solutions and the generation of mobile manipulator
# instructions sets


# Import mobile manipulator specifications
my_chain = ik.Chain.from_urdf_file("physical_layer/arm_urdf.urdf",
                                   active_links_mask=[False, True, True, True, True, True, True])
HOME_POSITION_ANGLES = [0, 0, 1, -2.5, 0, -1.6, 0]
END_POSITION_ERROR_MARGIN = 0.0015 # 1.5 mm


#verify if a position is reachable from an arm positioned at arm_position
def verify_pose(arm_position, target_position):
    target_position -= arm_position             # arm base offset
    target_position += [0.05, 0.05, 0.1]        # target middle of block top face

    # orientate end-effector downwards
    target_orientation = [0, 0, -1]
    orientation_mode = "Z"

    # Get joint angles to place end-effector at target_position at the target_orientation
    ik = my_chain.inverse_kinematics(target_position, target_orientation,
                                     initial_position=HOME_POSITION_ANGLES,
                                     orientation_mode=orientation_mode)

    # Verify joint angles with forward kinematics
    computed_position = my_chain.forward_kinematics(ik)

    # If end-effector within error margin of target position, return true
    if is_valid_position(computed_position[:3, 3], target_position):
        return True
    else:
        return False


# Get joint angles to place end-effector at target_position orientated downwards
def get_ik(target_position):
    # point down
    target_orientation = [0, 0, -1]
    orientation_mode = "Z"

    # Get joint angles
    return my_chain.inverse_kinematics(target_position, target_orientation,
                                       initial_position=HOME_POSITION_ANGLES,
                                       orientation_mode=orientation_mode)
```

```
55  # Return true if position is within error range of target position, else False
56  def is_valid_position(position, target_position):
57      error_margin = END_POSITION_ERROR_MARGIN
58      for i in range(0,3):
59          if (position[i] < target_position[i] - error_margin) or (
60              position[i] > target_position[i] + error_margin):
61              return False
62      return True
63
64
65  # Display mobile manipulator pose
66  def display_pose(inverse_kinematics, target_position):
67      fig, ax = plot_utils.init_3d_figure()
68      fig.set_figheight(9)
69      fig.set_figwidth(13)
70      my_chain.plot(inverse_kinematics, ax, target=target_position)
71      plt.xlim(-0.5, 0.5)
72      plt.ylim(-0.5, 0.5)
73      ax.set_zlim(0, 0.6)
74      plt.ion()
75      plt.show()
76
77
78  # Verify transition path movements adhere to physical constraints
79  # If they do, Return list of instructions for the mobile manipulator
80  def motion_planner(transition_path):
81      # confirm all movements possible
82      for state in transition_path:
83          if (state.birth_movement != 0):
84              if not movement_possible(state):
85                  return 0, state
86
87
88      # develop movement plan
89      # instructions entered according to key skills:
90      # ["CONNECT"]
91      # ["DISCONNECT"]
92      # ["MOVE_TO", target_pos, joint_angles_array]
93      instruction_set = [["START"]]
94      for state in transition_path:
95          if (state.birth_movement != 0):
96              instruction_set.append(generate_move_inst(state.birth_movement[0]))
97              instruction_set.append(["CONNECT"])
98
99              instruction_set.append(generate_move_inst(state.birth_movement[1]))
100             instruction_set.append(["DISCONNECT"])
101     instruction_set.append(["END"])
102
103     return instruction_set, 0
104
105
106 # Return move to instruction for position
107 def generate_move_inst(position):
108     # correct module position to top face of module
109     target_position = np.asarray(position) + [0.05, 0.05, 0.1]
110     return ["MOVE_TO", tuple(target_position), tuple(get_ik(target_position))]
```

```
111 # Verify movement adheres to physical constraints
112 def movement_possible(state):
113     movement = state.birth_movement
114     # verify module can be lifted (no blocks above module)
115     for key in state.get_module_positions():
116         if (key[0] == movement[0][0]) and (key[1] == movement[0][1]) and (key[2] > movement[0][2]):
117             return False
118 
119     # verify module placement is above surface
120     if movement[1][2] < 0:
121         return False
122 
123     # verify module placement is supported position
124     # (a surface exists directly below placement position)
125     supporting_position = tuple(movement[1] - [0,0,1])
126     if (movement[1][2] != 0):
127         if not (supporting_position in state.get_module_positions()):
128             return False
129 
130     return True
```

Appendix J - Initial Report

SCHOOL OF PHYSICS,
ENGINEERING AND TECHNOLOGY

Autonomous Re-Configuration of Modular Spacecraft with Manipulator Arm

Author: Connall Shurey

Supervisors: Prof. Mark A. Post,
Prof. Kanapathippillai Cumanan

4th Year Project Initial Report for
MEng in Electronic and Computer Engineering

March 6, 2024

CONTENTS

I. STATEMENT OF ETHICS

After consideration of the University's code of practice and principles for good ethical governance no ethical issues were identified in this project.

II. Introduction

A. Background and Context

Space systems have rapidly developed in recent years, with a global drive to increase commercial availability. Current commercial systems designed under the limitation of mass and launch costs, are traditionally highly customized whole systems which consequently have very limited or no maintenance and repair capabilities. The number of ageing satellites is rapidly increasing and upon reaching end of life, are discarded through atmospheric deconstruction methods if possible, or left in orbit contributing to space debris build-up.

Technology to circumvent these conditions is not currently available and as such, the HORIZON 2020 EU-funded MOdular Spacecraft Assembly and Reconfiguration (MOSAR) project was launched to develop novel technologies that would allow standardising satellites and components [1]. The modularisation and standardisation of space systems will benefit the European space industry by facilitating mass production of standard components and therefore decreasing assembly costs, reducing time between customer orders and commissioning in space, and allowing repair and upgrading of components directly in-orbit.

MOSAR primarily aims to produce on-orbit modular and reconfigurable satellites. At present the project has developed a demonstrator for re-configuring cubic modules to simulate the movement of modules through the use of a mobile robotic manipulator. Currently fixed instructions facilitating module mobility are sent to the manipulator from a software simulation on earth [2], this research project aims to further develop the capabilities of the system by developing an algorithm to automate the module reconfiguration process, facilitating self-repair and self-assembly. Following development, this technology has the potential to facilitate the automated assembly of space systems and platforms directly in space, expanding the limitations currently imposed on the space industry.

B. Review Objectives and Questions

This literature review aims to identify gaps and opportunities in current reconfiguration technologies and produce a plan to undertake the associated research over the following semester.

To guide this literature review, we will aim to answer the following research questions:

- What is the current state-of-the-art of modular spacecraft autonomy in the real-world?
- What are the challenges to improving automated reconfiguration in space?
- What research is being conducted into automated reconfiguration?
- What gaps and opportunities can be identified in current research?

III. Project Specification

This project intends to introduce the capability of autonomous modular assembly and reconfiguration of a spacecraft by implementing a planning program of simple algorithms that, given the initial state and final state of a modular craft as parameters, can produce a list of commands to send to a mobile manipulator to autonomously rearrange modules on a spacecraft or space platform in operation.

To accomplish the research goal, a reconfiguration planning program must be implemented in software. Which can then be evaluated and demonstrated in simulation if time permits. For measuring project success, the efficiency, error rate, robustness and potential for future advancement of the overall system will be considered.

Due to the strict guidelines required to acquire licensing for use in space, robustness will be an especially important measure of success as no matter the speed and efficiency of the program, if the manipulator arm ever receives instructions which jeopardise the system, spacecraft could be destroyed and in the worst-case scenario, lives lost. It is preferred for the planning program to return an error and be unable to provide a list of commands than to return commands which are not confirmed to be safe and accurate.

IV. Overview of Modular Spacecraft

Modular spacecraft are a design concept where the overall space system is composed of interchangeable modules / components, where each module is designed to serve a specific function such as propulsion, communication, power generation, or sensing. These modules are standardized allowing them to be easily connected to form a singular system, where modules can be moved or replaced to improve craft efficiency during operation and extend the overall lifetime of the system. Taking a modular approach to design offers several advantages over traditional designs such as flexibility, adaptability, and ease of maintenance.

Modules are equipped with standardised interfaces which define how modules physically and electronically interact, enabling modules with different purposes or made from different manufacturers to seamlessly integrate into the overall system architecture. The size and shape of modules can vary in different modular designs however standardisation principles allow these components to be integrated with other modules regardless. The size of the final space system architecture is only limited by the type and number of modules it is comprised of, providing scalability which enhances the spacecrafts versatility and cost-effectiveness, as the system can be tailored to meet specific needs of different missions without requiring a complete redesign.

V. State-of-the-art in Spacecraft Modularity and Autonomous Reconfiguration

The following section explores existing cases of spacecraft modularity and reconfiguration technologies currently or previously in operation. Due to the challenges related to developing automated reconfiguration systems for operation in space, there are no relevant existing cases of automated reconfiguration other than the International Space Station (ISS), however the use of modular design principles has been present in the space industry for development of spacecraft from as early as the 1980s with the development of the MMS.

A. Multi-mission Modular Spacecraft (MMS)

The Multi-mission Modular Spacecraft (MMS) was designed and deployed by NASA in the 1980s and 1990s [4] with the intention of decreasing space mission costs. Intended to be recoverable/serviceable by the Space Shuttle Orbiter [5], It is one of the first cases of modular designs seen in the space industry and has paved the way for future innovations.

Fig. 1. Artist rendering of the TOPEX/Poseidon mission. Image from [3]

The MMS consisted of a small number of immobile modules, with the most basic deployed MMS containing only modules for altitude control, communications and data handling, and the power subsystems module [4].

The MMS flew only six missions through its lifetime which was vastly different from the thirty-one expected in the 1970s [4], it suffered limitation in the form of electronic technologies rather than mechanical restraints. NASA's first Standard Spacecraft Computer (NSSC-1) [6] was developed to prevent requiring an entire redesign of onboard computers for each mission, requiring only a software redesign, though this was still a heavy burden affecting the MMS's mission flexibility. While no longer in operation as of 2006 [7], the system did show cost-savings in the range of 55% to 65% [4]. "The idea of a modular system serving many purposes was the pioneer of the leading systems within the space technology ecosystem today as it has left a lasting legacy" [4]. In the wake of the MMS's legacy, new design techniques were developed such as the Modular, Adaptive, Reconfigurable Systems (MARS) system-level architecture [4] that has built the foundation for modern space systems.

B. Modular Common Spacecraft Bus (MCSB)

The MCSB is a fast-development, low-cost, general purpose spacecraft platform consisting of a series of 4-5 modules stacked on top of each other, each serving separate functionality [9]. According to NASA, "the spacecraft is roughly one tenth the price of a conventional unmanned mission and could be used to land on the Moon, orbit Earth, or rendezvous with near-Earth objects." [10]

The MCSB system received the Popular Mechanics 2014 breakthrough Award for innovation in science and technology [11] and is proving to be at the forefront of existing modular space technologies, first deployed on the Lunar Atmosphere and Dust Environment Explorer (LADEE) mission in 2013 [12].

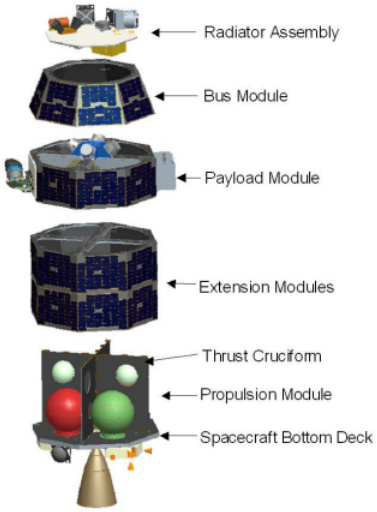

Fig. 2. LADEE Bus Modules from the MCSB Architecture. Image from [8]

The MCSB system is an example of modularity being used to streamline and reduce costs of the initial development process of the craft, being able to carry up to 50kg of scientific equipment inside its payload module [9], though the end product is still a whole system that has limited in-operation service capabilities and is not capable of being reconfigured to adapt to mission requirements in-orbit.

C. International Space Station

The International Space Station (ISS) seen in figure 3 is the largest space platform ever built, created with the purpose of performing micro-gravity and space environment experiments. First launched in 1998[14] and expanded through the integration of additional modules and serviced by human occupants up until its planned de-orbit in 2031 [15], it is a monument to advancements in the space industry.

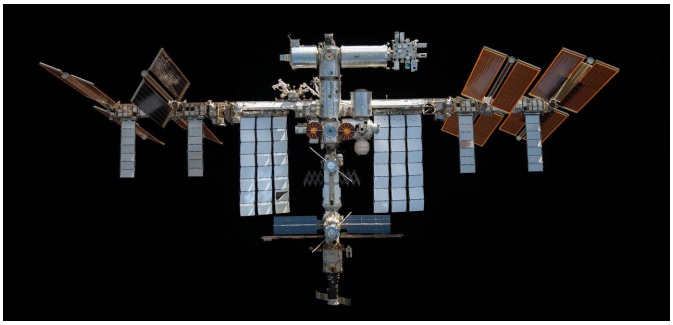

Fig. 3. The ISS pictured from the SpaceX Crew Dragon (Dec. 8, 2021). Image from [13]

The ISS is capable of reconfiguration using a robotic arm and automated docking with human oversight [16] unlike previous cases, though unsupervised automated reconfiguration is yet to be attempted due to the consequences of failure.

VI. Challenges and Limitations of Automated Reconfiguration in Space

The lack of deployment of complex automated systems such as automated reconfiguration systems in space is not due to a lack of interest, but instead due to the difficulty of the technical challenges presented by such systems and the risk introduced to extremely expensive and often critical missions that cannot afford failure.

Space systems must be reliable and work in a wide range of conditions. The more complex a system is, the more likely it can go wrong, which makes the validation, verification, and deployment of such systems in the space industry a lengthy and expensive process. Challenges that autonomous space systems face include:

- **Communication latency** – delays in communications from systems make it impossible for human controllers to react to unexpected situations in real-time, meaning any autonomous system must be capable of performing completely without human intervention. Simply having an autonomous system that is allowed to operate under human observation such as a self-driving car does not meet the reliability requirements for space applications.
- **Safety Requirements** – Systems will often be hosting expensive scientific equipment while operating in harsh, unpredictable environments where various hazards are present such as extreme temperature differences, radiation, space debris, ice and lack of gravity.
- **Limited Power Sources** – autonomous systems require power which depending on a crafts power source is not always guaranteed. For example, a craft relying on solar power may lose power during eclipses or due to unexpected collisions. Autonomous systems must be capable of recovering from temporary power loses or have reliable backup power sources to prevent mission failure.
- **Isolation** – unlike on land, it is usually not possible for a craft to quickly receive help or be viewed by an external observer. An autonomous system must have the sensing capabilities to self-diagnose problems or detect anomalies and halt standard operation, otherwise the system could cause further damage to itself.

Overcoming these challenges requires a level of technology that has only become available in recent years leading to the undertaking of research projects such as this one. It can be expected to see the number of autonomous systems present in the space industry drastically increase over the next few decades as computational power and materials sciences continue to advance.

VII. Emerging Advancements in Reconfiguration Technologies

A. MOSAR Outcomes

Up till now the MOSAR project has produced several major outcomes:

- A standardised module framework making use of the HOTDOCK adapter.
- Design and fabrication of a walking manipulator arm.
- Related system architecture to control the arm remotely.
- Successful ground demonstration of the manipulator arms capabilities to move and connect modules.

At this stage, an in-orbit implementation of the MOSAR demonstrator would be capable of reconfiguration functionality, though requires reconfiguration instructions to be manually sent to the craft. Further work is required for automated functionality such as:

- Automatically find a desired module configuration for the craft to meet mission requirements.
- Automatically compute a set of manipulator instructions required to reconfigure the craft from one configuration to another.

The following review of literature will be focused on identifying the best method to perform the latter.

B. Automated Reconfiguration

Automatic planners, algorithms that find a solution for which sequence of operations must be accomplished to achieve a specified goal, have been an area of development attracting wide-spread interest since the earliest days of robotics. Currently there are many different types of automatic planning techniques available. They encompass a large set of algorithmic requirements which trend towards purely discrete or purely continuous search space characteristics. The development of "Hybrid" automated planning approaches with search space characteristics that are not purely discrete or continuous, especially Task and Motion Planning (TAMP) algorithms, represent an area of study of which solutions are considered the most computationally difficult in theory [17]. Consequently, the application of automated planning algorithms to robotic assembly of modular satellites is a very recent development in which little work has been published that implements automatic reconfiguration algorithms while fully considering the range of real-world physical restraints and limitations presented by usage of a mobile manipulator arm in a low-gravity environment.

Motion Planning is finding solutions to move a robot "from one configuration to another configuration without colliding with the objects in the world" [18]. It involves searching for paths within the robots reach which is a continuous configuration space limited by dimensions represented by the joints of

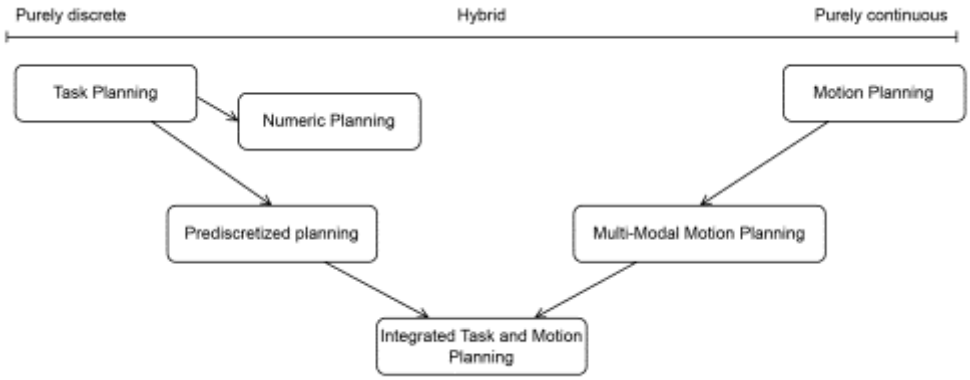

Fig. 4. Taxonomy of automated planning approaches based on their search spaces' characteristics. Image from [18]

the robot. These collision-free paths are important for robot motion but do not by themselves allow the robot to interact with the world. Further planning must be implemented to allow manipulation of objects through manipulation planning (known as Multi-Modal Motion Planning).

Due to the increased complexity of the problem presented by manipulation planning, the problem is best broken down into a hybrid discrete-continuous search problem of "selecting a finite sequence of discrete action types (e.g. which objects to pick and place), continuous action parameters (such as object poses to place and grasps), and continuous motion paths" [18].

While Motion and Manipulation planning are seen as problems mainly within the robotics field, planning within large discrete domains such as in problems presented by task planning has been more deeply researched within the artificial intelligence (AI) community [19]. Task planning (also known as Action planning) referring to deducing a composition of symbolic actions to achieve a high-level goal (e.g. computing a sequence of actions required to stack boxes in a specified order). The discrete nature of the problem makes it particularly suitable for many machine learning techniques which have particularly advanced in recent years.

Current research in task and motion planning (TAMP) primarily aims to combine the robotics solutions for manipulation planning under physical constraints with the usually unrestricted AI approach to task planning. With the goal of deriving automated planning systems capable of reasoning symbolically with discrete "high-level" robotic action sets while geometrically taking into account continuous "low-level" robotic motion planning and restrictions. To date, several papers have developed algorithms for similar TAMP problems to the scenario of modular satellite reconfiguration that unfortunately are not compatible due to the method of module mobility, but act as a proof of concept that a solution is possible [20]–[22].

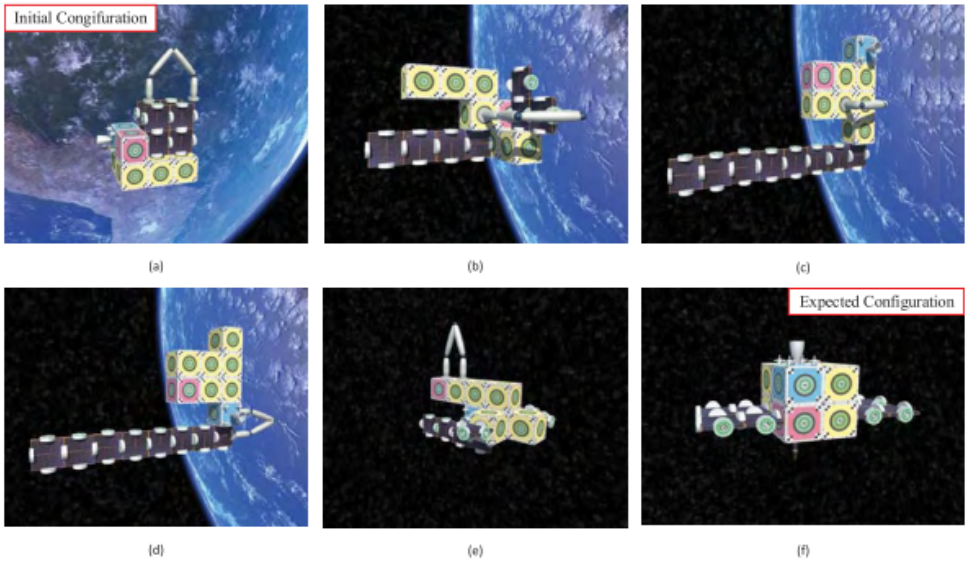

Fig. 5. Melt-Grow algorithm simulation results. Image from [25]

The 2010 Intelligent Building Blocks for On-Orbit Satellite Servicing and Assembly (IBOSS) project [23] by DLR provided many advances in the area of satellite modularization with the development of standardised building blocks and interfaces [24]. Simple task planning techniques were implemented using Hierarchical task network (HTN) planning to produce high-level mobile arm instruction sets to then be verified through inverse kinematic checks and motion planning. This implementation solved the discrete and continuous planning problems separately, which simplified the problem however does not allow the separate systems to properly integrate. The system was not capable of efficiently solving more difficult tasks of identifying were solutions where not feasible.

Alternatively, another approach was taken here [25] through the implementation of the melt-grow algorithm [26]. The physical restraints of the robot were not including in the reconfiguration planning stage of the system, effectively reducing the problem to task planning. This reduces complexity though can only be achieved due to the behaviour of the melt-grow algorithm, which deconstructs (melts) the initial module configuration into chains of modules defined as the intermediate configuration, seen in configuration d in figure 5, before then reconstructing (growing) the modules into the expected configuration. The system then does not need to consider whether a move is possible for the mobile arm through manipulation planning as due to the algorithms inclusion of an intermediate state between the melting and growing operations, the algorithm essentially reconstructs the satellite instead of modifying the current state, all required moves are possible for the mobile manipulator and simply require motion planning. While proven to work, this method is shown to be highly inefficient for the

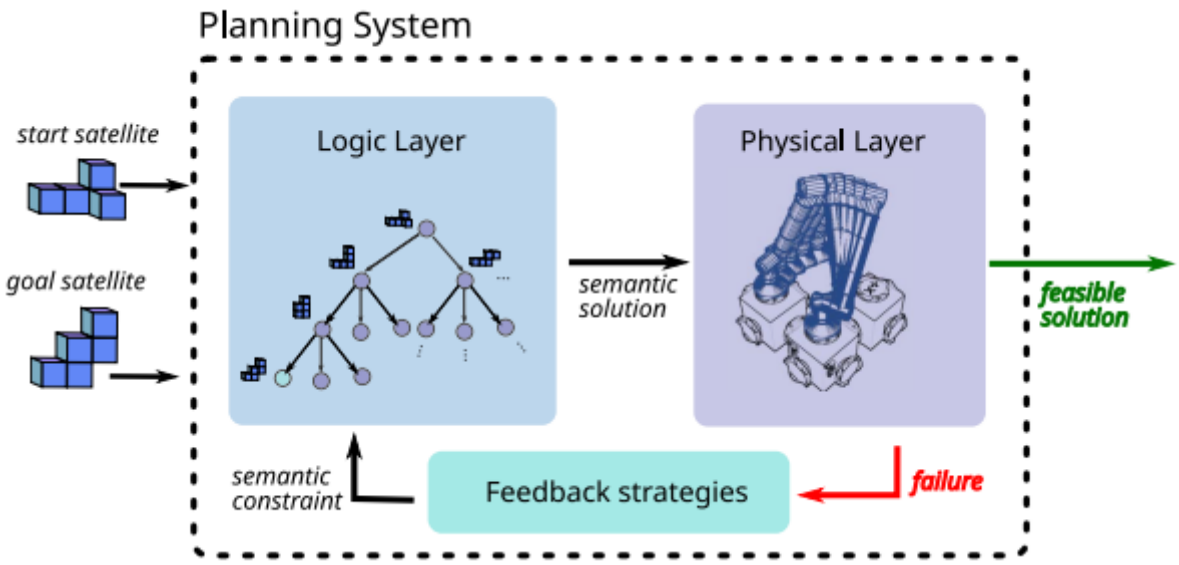

Fig. 6. "Architecture of the autonomous robot planning system. The system receives as inputs the start and goal satellite configurations, and iterates between the logic and physical layer until a solution is found." Image and text from [27]

mobile manipulator, especially as the number of modules increases in the system. Though, the paper [25] suggests this could be offset by the inclusion of additional manipulators which would consequently increase construction and operational costs.

More recent research has taken inspiration from these previous works to propose a comprehensive Task and Motion Planning (TAMP) problem solver [27] to intrinsically include the robot constraints into the system. The system, seen in figure 6, includes a logic layer, a physical layer, and a feedback system. Where the logic layer acts as a task planner finding a semantic solution by considering the solution as a sequence of states, with module movements defining the transition between states. A graph is developed to represent the possible states where nodes are system states and edges represent module movements which are verified by the physical layer which provides manipulation planning results through the feedback system. Using this graph the shortest and hence most efficient set of operations to reconfigure the system into the desired state can be identified. The removal of the intermediate configuration present in the melt-grow algorithm improves the efficiency of the solution set of operations, especially as the number of modules in the system increases, requiring less movement from the mobile manipulator.

The paper notes "the goal of this work was not to set a baseline for planning problems in terms of absolute times, but to demonstrate the usefulness of integrating feedback from the physical layer on the logic layer." [27], suggesting that there is an opportunity for further research into the components of the planning system and the related feedback strategies to prepare the system for space applications.

VIII. Gaps and Opportunities

Modular reconfiguration defines a subclass of the generic planning problems usually addressed by TAMP. Although research has previously demonstrated effective systems that can handle both symbolic and geometric reasoning, their application to robotic assembly and in particular robotic re-assembly is currently limited. There is additionally a distinct lack of discriminating modular blocks by type in existing algorithms which could potentially be easily implemented without a substantial hit to system performance.

The system proposed in figure [27] is promising due to the robustness of solutions and flexibility of the logic layer, however, there lacks the extensive performance testing required to recognise weaknesses and future improvements, identifying why this system could not be used in real-world application currently.

IX. Research Approach

The approach to this research project will be to replicate the system proposed in [27] to meet the project specification using an implementation that can display the system results in simulation. Then the system will be subjected to a range of performance tests to evaluate how modifications to feedback strategies affect overall performance and identify what roadblocks exist preventing space application in its current state.

As there is not enough time to build two completely different systems and compare them, the project takes an identifiably robust system and searches for improvements and failure points in the functionality.

X. Research Schedule

The schedule the project will follow can be seen in figure 7. Task descriptions can be seen below:

1.0 **User Interface**

- 1.1 **State graph creator** – Create a program that can be used to input modular state configurations and represent the state configuration appropriately in software.
- 1.2 **State viewer (user interface)** – Implement a user interface that can display, and allow the manipulation of, a state configuration to allow human input.
- 1.3 **Simulation display** – A simulation displaying the movement of the mobile manipulator as it reconfigures the craft from one state to another. This task will be highly time dependent as it pushes the boundary of the project scope.

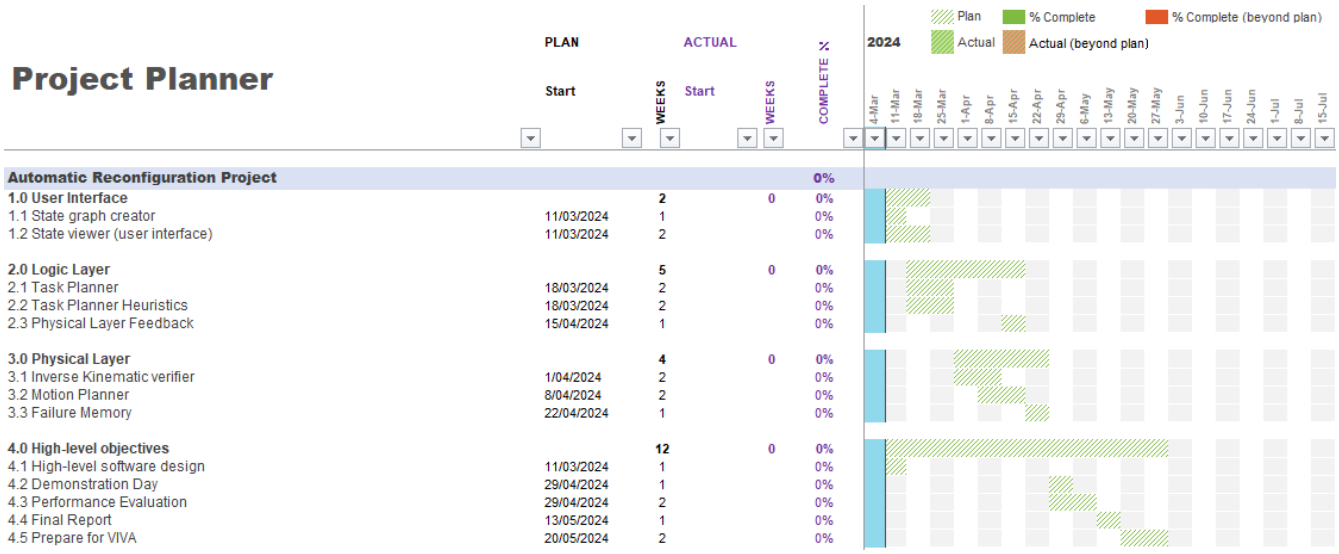

Fig. 7. A Project Planner displaying the overall project schedule

2.0 Logic Layer

2.1 **Task Planner** – Create a decision tree planner to compute basic semantic solutions.

2.2 **Task Planner Heuristics** – Add heuristics to the task planner to increase efficiency.

2.3 **Physical Layer Feedback** – Join the logic and physical layer together through feedback to create a control loop to produce verified solutions.

3.0 Physical Layer

3.1 **Inverse Kinematic Verifier** – Create a function that takes a module movement as an argument, and uses inverse kinematics to verify whether the movement is feasible for the manipulator.

3.2 **Motion Planner** – Produce a motion planner for the mobile manipulator so arm movement can later be displayed in simulation.

3.3 **Failure Memory** – Implement a machine learning algorithm that is trained from simulated movements, to create a failure predictor. The probability of failure produced by the algorithm can then be used for task planning optimisation.

4.0 High-level Objectives

4.1 **High-level software design** – Basic high-level software design to follow during the project.

4.2 **Demonstration Day** – Demonstration of the project to supervisors and industry professionals.

4.3 **Performance Evaluation** – Conduct performance testing on the algorithm under various conditions and with modified feedback strategies.

4.4 **Final Report** – Assemble collected information into a final report.

4.5 **Prepare for VIVA** – Prepare a 15 minute presentation for final project presentation and VIVA.

A. Risk Register

ID	Risk Description	Impact	Risk Probability	Mitigation of Risk
1	Missing or corrupted documents	High	Medium	Documents are backed up to a GitHub repository
2	Ambitions for project are too great for the project time limit	High	High	Setting appropriate scope expectations from the beginning of the project
3	Illness or work unavailability	High	Medium	Record illness and provide proper explanation for missing work in final report. Decrease scope to provide meaningful results
4	Losing test results	Medium	Medium	Produce lab reports to document progress

XI. Conclusion

This literature review has highlighted the newest advances within a small subset of planning algorithms relevant to automated reconfiguration of modular satellites. The implementation of modular reconfiguration systems to space applications is a relatively new field, and this research project intends to show how interconnected systems handling symbolic and geometric reasoning in tandem can appropriately find solutions and identify where they are not feasible. Additionally, this research intends to identify the shortcomings of existing theoretical systems and provide areas for future research and development.

XII. References

[1] MOSAR, "About the project - mosar," 2020, Accessed 12-12-2023. [Online]. Available: https://www.h2020-mosar.eu/about/.

[2] MOSAR, "Detailed design document," 2020, Accessed 13-12-2023. [Online]. Available: https://www.h2020-mosar.eu/wp-content/uploads/2021/08/MOSAR-WP3-D3.6-SA_1.2.0-Detailed-Design-Document.pdf.

[3] J. P. Laboratory, "Topex/poseidon fact sheet," Accessed 03-02-2024. [Online]. Available: https://sealevel.jpl.nasa.gov/missions/topex-poseidon/summary/.

[4] J. Esper, "Modular, adaptive, reconfigurable systems: Technology for sustainable, reliable, effective, and affordable space exploration," 2005. [Online]. Available: https://api.semanticscholar.org/CorpusID:2408860.

[5] R. O. Bartlett, "Nasa standard multimission modular spacecraft for future space exploration," in *Proceedings of the Goddard Memorial Symposium, Washington, DC, United State, March 8–10, 1978*, ser. AAS PAPER 78-043, NASA, Goddard Space Flight Center Greenbelt, Md., United States, 1978.

[6] C. E. Trevathan, T. D. Taylor, R. G. Hartenstein, A. C. Merwarth, and W. N. Stewart, "Development and application of nasa's first standard spacecraft computer," *Commun. ACM*, vol. 27, no. 9, 902–913, 1984, ISSN: 0001-0782. DOI: 10.1145/358234.358252. [Online]. Available: https://doi.org/10.1145/358234.358252.

[7] E Hupp and E Moreaux, "Nasa's topex/poseidon oceanography mission ends," *Press Release*, pp. 06–001, 2006.

[8] B. Hine, S. Spremo, M. Turner, and R. Caffrey, "The lunar atmosphere and dust environment explorer mission," in *2010 IEEE Aerospace Conference*, 2010, pp. 1–9. DOI: 10.1109/AERO.2010.5446989.

[9] S. Tietz, J. H. Bell, and B. Hine, "Multi-mission suitability of the nasa ames modular common bus," in *AIAA and Utah State University 23rd Annual Conference*, 2009.

[10] W. M. NASA, "Common spacecraft bus for lunar explorer missions," 2008, Accessed 10-02-2024. [Online]. Available: https://lunarscience.arc.nasa.gov/articles/common-spacecraft-bus-for-lunar-explorer-missions/.

[11] R. H. Dwayne Brown, "Nasa lunar mission wins 2014 popular mechanics breakthrough award," 2014, Accessed 10-02-2024. [Online]. Available: https://www.nasa.gov/news-release/nasa-lunar-mission-wins-2014-popular-mechanics-breakthrough-award/.

[12] M. V. D'Ortenzio, J. L. Bresina, A. R. Crocker, *et al.*, "Operating ladee: Mission architecture, challenges, anomalies, and successes," in *2015 IEEE Aerospace Conference*, 2015, pp. 1–23. DOI: 10.1109/AERO.2015.7118961.

[13] NASA, *The station pictured from the spacex crew dragon jsc2021e064215_alt*, 2021. [Online]. Available: https://images.nasa.gov/details-jsc2021e064215_alt.

[14] S. K. Shaevich, "Results of five-years exploitation of the first iss element-fgb" zarya" module," in *54th International Astronautical Congress of the International Astronautical Federation, the International Academy of Astronautics, and the International Institute of Space Law*, 2003, T–1.

[15] E. Mahoney, "Nasa provides updated international space station transition plan," Accessed 12-12-2023, NASA, 2022. [Online]. Available: https://www.nasa.gov/humans-in-space/nasa-provides-updated-international-space-station-transition-plan/.

[16] M. A. Post and J. Austin, "Knowledge-based self-reconfiguration and self-aware demonstration for modular satellite assembly," in *10th International Workshop on Satellite Constellations & Formation Flying 2019*, York, 2019.

[17] A. Deshpande, L. P. Kaelbling, and T. Lozano-Pérez, "Decidability of semi-holonomic prehensile task and motion planning," in *Algorithmic Foundations of Robotics XII: Proceedings of the Twelfth Workshop on the Algorithmic Foundations of Robotics*, K. Goldberg, P. Abbeel, K. Bekris, and L. Miller, Eds. Cham: Springer International Publishing, 2020, pp. 544–559, ISBN: 978-3-030-43089-4. DOI: 10.1007/978-3-030-43089-4_35. [Online]. Available: https://doi.org/10.1007/978-3-030-43089-4_35.

[18] Y. Huang, *Algorithmic planning for robotic assembly of building structures*, 2023.

[19] M. Ghallab, D. Nau, and P. Traverso, *Automated Planning: theory and practice*. Elsevier, 2004.

[20] S. Vassilvitskii, M. Yim, and J. Suh, "A complete, local and parallel reconfiguration algorithm for cube style modular robots," in *Proceedings 2002 IEEE International Conference on Robotics and Automation (Cat. No.02CH37292)*, vol. 1, 2002, 117–122 vol.1. DOI: 10.1109/ROBOT.2002.1013348.

[21] D. Saldaña, B. Gabrich, M. Whitzer, *et al.*, "A decentralized algorithm for assembling structures with modular robots," in *2017 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2017, pp. 2736–2743. DOI: 10.1109/IROS.2017.8206101.

[22] J. Paulos, N. Eckenstein, T. Tosun, *et al.*, "Automated self-assembly of large maritime structures by a team of robotic boats," *IEEE Transactions on Automation Science and Engineering*, vol. 12, no. 3, pp. 958–968, 2015. DOI: 10.1109/TASE.2015.2416678.

[23] iBOSS, "Iboss background," 2023, Accessed 25-02-2024. [Online]. Available: https://www.iboss.space/background/.

[24] M. Kortmann, T. Schervan, H. Schmidt, S. Ruehl, J. Weise, and J. Kreisel, "Building block-based "iboss" approach: Fully modular systems with standard interface to enhance future satellites," Sep. 2015.

[25] Y. Zhang, W. Wang, J. Sun, H. Chang, and P. Huang, "A self-reconfiguration planning strategy for cellular satellites," *IEEE Access*, vol. 7, pp. 4516–4528, 2019. DOI: 10.1109/ACCESS.2018.2888588.

[26] D. Rus and M. Vona, "Self-reconfiguration planning with compressible unit modules," in *Proceedings 1999 IEEE International Conference on Robotics and Automation (Cat. No.99CH36288C)*, vol. 4, 1999, 2513–2520 vol.4. DOI: 10.1109/ROBOT.1999.773975.

[27] I. Rodríguez, A. S. Bauer, K. Nottensteiner, D. Leidner, G. Grunwald, and M. A. Roa, "Autonomous robot planning system for in-space assembly of reconfigurable structures," in *2021 IEEE Aerospace Conference (50100)*, 2021, pp. 1–17. DOI: 10.1109/AERO50100.2021.9438257.